

KDE- und Qt-Programmierung

Linux Specials

Burkhard Lehner

KDE- und Qt-Programmierung

GUI-Entwicklung für Linux
2., aktualisierte und erweiterte Auflage

Bitte beachten Sie: Der originalen Printversion liegt eine CD-ROM bei.
In der vorliegenden elektronischen Version ist die Lieferung einer CD-ROM nicht enthalten.
Alle Hinweise und alle Verweise auf die CD-ROM sind ungültig.



ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

**Ein Titelsatz für diese Publikation ist
bei Der Deutschen Bibliothek erhältlich**

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht. Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt. Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen. Trotzdem können Fehler nicht vollständig ausgeschlossen werden. Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen. Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien. Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

Falls alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden, sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:

Dieses Buch wurde auf chlorfrei gebleichtem Papier gedruckt.

Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltfreundlichem und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

04 03 02 01

ISBN 3-8273-1753-3

© 2001 Addison-Wesley Verlag,

ein Imprint der Pearson Education Company Deutschland GmbH

Martin-Kollar-Straße 10–12, D-81829 München/Germany

Alle Rechte vorbehalten

Einbandgestaltung: Hommer Design Production, Haar bei München

Lektorat: Susanne Spitzer, susanne.spitzer@gmx.net

Korrektur: Friederike Daenecke, Zülpich

Herstellung: TYPisch Müller, Archevia, Italien, typmy@freefast.it

Satz: reemers publishing services gmbh, www.reemers.de

Druck und Verarbeitung: Druckerei Kösel, Kempten

Printed in Germany

Inhaltsverzeichnis

Vorwort	IX
1 Was ist KDE? Was ist Qt?	1
1.1 Das KDE-Projekt	1
1.2 Die Qt-Bibliothek	2
1.3 Vergleich mit ähnlichen Projekten	4
1.4 Informationsquellen	6
1.5 Lizenzbestimmungen	9
2 Erste Schritte	13
2.1 Benötigte Programme und Pakete	13
2.2 Das erste Qt-Programm	15
2.3 Das erste KDE-Programm	25
2.4 Was fehlt noch zu einer KDE-Applikation?	41
3 Grundkonzepte der Programmierung in KDE und Qt	43
3.1 Die Basisklasse – <i>QObject</i>	44
3.2 Die Fensterklasse – <i>QWidget</i>	76
3.3 Grundstruktur einer Applikation	91
3.4 Hintergrund: Event-Verarbeitung	112
3.5 Das Hauptfenster	115
3.6 Anordnung von GUI-Elementen in einem Fenster	192
3.7 Überblick über die GUI-Elemente von Qt und KDE	219
3.8 Der Dialogentwurf	248
4 Weiterführende Konzepte der Programmierung in KDE und Qt	271
4.1 Farben unter Qt	273
4.2 Zeichnen von Grafikprimitiven	290
4.3 Teilbilder – <i>QImage</i> und <i>QPixmap</i>	327
4.4 Entwurf eigener Widget-Klassen	347
4.5 Flimmerfreie Darstellung	376
4.6 Klassendokumentation mit <i>doxygen</i>	382
4.7 Grundklassen für Datenstrukturen	396
4.8 Der Unicode-Standard	423

4.9	Mehrsprachige Anwendungen und Internationalisierung	441
4.10	Konfigurationsdateien	448
4.11	Online-Hilfe	455
4.12	Timer-Programmierung, Datum und Uhrzeit	460
4.13	Blockierungsfreie Programme	471
4.14	Audio-Ausgabe	494
4.15	Die Zwischenablage und Drag&Drop	497
4.16	Session-Management	507
4.17	Drucken mit Qt	513
4.18	Dateizugriffe	516
4.19	Netzwerkprogrammierung	526
4.20	Interprozesskommunikation mit DCOP	553
4.21	Komponenten-Programmierung mit KParts	575
4.22	Programme von Qt 1.x auf Qt 2.x portieren	581
4.23	Programme von KDE 1.x auf KDE 2.x portieren	588
5	Hilfsmittel für die Programmerstellung	595
5.1	tmake	595
5.2	automake und autoconf	598
5.3	kapptemplate und kappgen	603
5.4	Qt Designer	608
5.5	KDevelop	612
6	Ausgewählte, kommentierte Klassenreferenz	615
6.1	KDE-Klassen in kdeui, kdecop, kfile und kio	615
6.2	Qt-Klassen	626
A	Lösungen zu den Übungsaufgaben	707
A.1	Lösungen zu Kapitel 2.2	707
A.2	Lösungen zu Kapitel 3.1	711
A.3	Lösungen zu Kapitel 3.6	716
A.4	Lösungen zu Kapitel 4.1	723
A.5	Lösungen zu Kapitel 4.2	726
A.6	Lösungen zu Kapitel 4.10	728
A.7	Lösungen zu Kapitel 4.12	743

B	Die Lizenzen von Qt und KDE	745
B.1	Qt Free Edition License für Versionen vor Qt 2.0	745
B.2	QPL – Qt Public License – ab Qt 2.0	747
B.3	GPL – General Public License	749
B.4	LGPL – Library General Public License	755
C	Die KDE-Standardfarbpalette	765
	Stichwortverzeichnis	767

Vorwort

In den letzten Jahren hat das freie Betriebssystem Linux eine enorme Verbreitung gefunden. Durch seine hohe Stabilität, die gute Netzwerkunterstützung und die Kompatibilität mit anderen Unix-Betriebssystemen hat sich Linux im Server-Bereich sehr schnell etabliert. Für Privatanwender hatte Linux dagegen lange Zeit das Image eines Bastler-Betriebssystems, da eine einheitliche Oberfläche für alle Applikationen fehlte. Jedes Programm war anders zu bedienen. Dadurch wurde die Einarbeitungszeit unnötig verlängert.

Es gab eine Reihe von Ansätzen, dieses Problem zu beheben. Den mit Abstand größten Erfolg hatte jedoch das 1996 gegründete KDE-Projekt. Zielsetzung des Projekts ist es, dem Privatanwender eine große Anzahl leistungsfähiger und intuitiver Programme mit einheitlichem Erscheinungsbild zur Verfügung zu stellen. Neben dem Window-Manager *kwin*, dem Dateimanager *Konqueror* und dem Office-Paket *KOffice* enthält das KDE-Projekt eine Vielzahl kleiner Applikationen. Alle Programme haben ein einheitliches, modernes Aussehen und lassen sich intuitiv bedienen.

Obwohl das KDE-Projekt zum größten Teil auf Linux-Systemen entwickelt und getestet wird, sind nahezu alle KDE-Programme auch auf sehr vielen anderen Unix-Betriebssystemen lauffähig, zum Beispiel unter HP-UX, SunOS, Solaris und IRIX. Das KDE-Projekt umfasst eine Menge an Know-how, wie Anwendungen plattformunabhängig geschrieben werden können. Bei vielen Linux-Distributionen ist KDE inzwischen zum Standard-Desktop geworden und auch auf vielen anderen Unix-Rechnern ist KDE bereits installiert.

Die grafische Oberfläche aller KDE-Programme wird mit Bedienelementen der Bibliothek Qt gebildet, die von der norwegischen Firma Trolltech entwickelt und gewartet wird. Qt nutzt das Klassenkonzept von C++, wodurch die Erweiterbarkeit und die Modularität ohne großen Aufwand erreicht werden können. Für die Entwicklung neuer KDE-Applikationen ist das Grundwissen über den Aufbau und die Anwendung dieser Bibliothek unerlässlich.

Meine ersten Erfahrungen mit KDE machte ich bereits 1997, als ich auf der Suche nach einem guten grafischen Client für das Talk-Protokoll über das Programm *KTalk* stolperte. Das Programm war noch nicht perfekt, sah aber schon deutlich besser aus als die meisten der anderen Programme mit diesem Funktionsumfang. Da das Programm im Quelltext vorlag, änderte ich zunächst ein paar kleine Dinge. Innerhalb weniger Tage arbeitete ich mich in die Materie der Qt-Bibliothek ein – da es damals noch keine Bücher zu diesem Thema gab, benutzte ich die exzellente Klassenreferenz – und war fasziniert von den Möglichkeiten. Bisher hatte ich unter Linux fast ausschließlich Programme geschrieben, die ohne grafi-

sche Benutzeroberfläche auskommen mussten. KDE und Qt boten mir nun die Möglichkeit, all meine Ideen auf einfache und elegante Art zu verwirklichen, mit professionellen Ergebnissen.

Schnell wurde mein Hobby zum Nebenjob. Ich hielt Vorträge und Schulungen zu den Themen Qt und KDE. Dabei traf ich viele Leute, die aus den unterschiedlichsten Gründen an diesem Thema interessiert waren. Viele Hobby-Programmierer haben mit KDE und Qt nun die Möglichkeit, ihre bisher spartanischen Programme mit einer professionellen Oberfläche zu versehen, und das auf einfache Art. Andere sind vom KDE-Projekt so fasziniert, dass sie ihre Programme in das Projekt einbinden lassen oder an anderen Stellen am Projekt mitwirken. Aber auch professionelle Programmierer orientieren sich um, da die Integration in die Benutzeroberfläche auf KDE-Systemen viel besser funktioniert. Viele Firmen und Entwicklungsabteilungen stellen auch ganz gezielt ihre Programme auf Qt um, so dass die Programme sowohl auf Unix-Systemen als auch unter Microsoft Windows laufen. So schaffen sie sich einen zweiten Absatzmarkt und arbeiten außerdem zukunftsicher, unabhängig davon, wie sich die Marktanteile der Betriebssysteme entwickeln.

Zusammen mit meiner Lektorin Susanne Spitzer entwickelte ich das Konzept für dieses Buch, das sich an alle Programmierer wendet, die ihre Programme in Zukunft auf Basis der Qt- und KDE-Bibliotheken entwickeln wollen. Dabei werden nur Grundkenntnisse in der Programmiersprache C++ vorausgesetzt. Das Buch kann sowohl von Einsteigern in die Programmierung von grafischen Benutzeroberflächen als auch von fortgeschrittenen Entwicklern benutzt werden.

Kapitel 1, *Was ist KDE? Was ist Qt?*, gibt einen kurzen Überblick über die Entwicklung der KDE- und Qt-Bibliotheken und über die lizenzrechtlichen Bestimmungen, die beachtet werden müssen.

Kapitel 2, *Erste Schritte*, wendet sich vor allem an Anfänger und erläutert in zwei einfachen Beispielen die Eigenschaften und Strukturen von KDE- und Qt-Programmen. Diese Beispiele können bereits als Grundlage für erste eigene Experimente dienen.

In Kapitel 3, *Grundkonzepte der Programmierung in KDE und Qt*, werden die wichtigsten Elemente, aus denen ein KDE-Programm besteht, detailliert vorgestellt. Auch dieses Kapitel richtet sich vor allem an Einsteiger in die KDE-Programmierung, enthält aber auch für erfahrene Programmierer eine Reihe von Lösungsansätzen und Hintergrundinformationen, die bei der Arbeit an einer Applikation hilfreich sein können. Mit dem Wissen aus diesem Kapitel können bereits vollständige Programme mit intuitiven Benutzerdialogen entwickelt werden.

Spezielle Problemstellungen, die beim Erstellen einer Applikation auftauchen können, werden in Kapitel 4, *Weiterführende Konzepte der Programmierung in KDE und Qt*, aufgegriffen. Es werden verschiedene Lösungsmöglichkeiten detailliert beschrieben und miteinander verglichen. Unter anderem wird an Beispielen

erläutert, wie eigene Anzeige- und Bedienelemente entwickelt und benutzt werden können, wie man Drag&Drop realisieren kann, wie man Timer nutzen und Zeitintervalle messen und wie man eine Blockierung des Programms bei Systemaufrufen verhindern kann. Das Kapitel richtet sich an Programmierer, die bereits einige Erfahrungen beim Erstellen von KDE-Programmen gesammelt haben. Es ist auch als Nachschlagewerk bei auftretenden Problemen geeignet.

Kapitel 5, *Hilfsmittel für die Programmerstellung*, gibt einen kurzen Überblick über die wichtigsten Hilfsprogramme, die dem Programmierer für die Entwicklung von KDE-Programmen dem Programmierer zur Verfügung stehen. Diese Programme können die Entwicklungszeit einer Anwendung drastisch verkürzen. Vor Beginn eines größeren Projekts ist es daher sehr anzuraten, sich mit diesen Programmen auseinanderzusetzen. Sie vereinfachen die Erstellung und Verwaltung von Makefiles und erlauben eine sehr einfache und schnelle Entwicklung von Bildschirmdialogen.

In Kapitel 6, *Ausgewählte, kommentierte Klassenreferenz*, werden alle wichtigen Klassen aus den KDE- und Qt-Bibliotheken kurz beschrieben und an vielen Stellen mit Beispielen erläutert. Es dient daher als Nachschlagewerk und als Ergänzung zu den Online-Referenzen der Qt- und KDE-Bibliotheken.

Viele Abschnitte des Buchs sind mit Übungsaufgaben ausgestattet, an denen der Leser sein erworbenes Wissen testen kann. Ausführliche Lösungen zu allen Übungsaufgaben finden Sie in Anhang A.

In dieser zweiten, aktualisierten und erweiterten Auflage des Buchs wird die Qt-Bibliothek in der Version 2.2 sowie die KDE-Oberfläche in der Version 2.0 besprochen. Neuere Versionen der Bibliotheken sind in der Regel kompatibel, sofern die Hauptversionsnummer weiterhin 2 ist. Sie können die hier beschriebenen Programme meist ohne Änderungen kompilieren und starten. In seltenen Fällen sind kleine Änderungen nötig. Beachten Sie in diesem Fall die Dokumentation zu den Qt- und KDE-Paketen.

Auf der CD-ROM, die dem Buch beiliegt, finden Sie neben den Beispielprogrammen aus diesem Buch auch die aktuellen Versionen des KDE-Projekts und der Qt-Bibliotheken inklusive Klassendokumentation und Tutorial. Weiterhin sind alle Hilfsprogramme aus Kapitel 5 in ihrer aktuellsten Version enthalten sowie weitere interessante Programme und Dokumentationen. Versäumen Sie also nicht, einen Blick auf den Inhalt der CD-ROM zu werfen.

Ich möchte an dieser Stelle meiner Lektorin Susanne Spitzer für ihre Geduld danken, die sie aufbringen musste, wenn ich mal wieder mit meinen Terminen im Verzug war oder die Reihenfolge der Kapitel umgestellt habe. Ebenso geht ein herzlicher Dank an die Firma Trolltech für ihre Unterstützung bei der Entwicklung des Buchs.

Ich hoffe, dass dieses Buch den Entwicklern von KDE- und Qt-Applikationen hilfreich sein wird. Ich gebe hierin mein Wissen weiter, dass ich im Laufe meiner Experimente mit KDE und Qt gesammelt habe. Vielleicht sind auch Sie anschließend so begeistert von diesem Projekt, dass Sie daran mitarbeiten und helfen, es noch besser zu machen.

Burkhard Lehner

Kaiserslautern im Dezember 2000

1 Was ist KDE? Was ist Qt?

In diesem Kapitel wollen wir uns zunächst die Zielsetzung des KDE-Projekts anschauen und uns die Entwicklung dieses Projekts und der zugrunde liegenden Qt-Bibliothek vor Augen führen.

1.1 Das KDE-Projekt

KDE ist ein sich sehr rasant entwickelndes Projekt. Seine Entwickler haben es sich zum Ziel gesetzt, eine einheitliche Oberfläche mit vielen Applikationen auf möglichst vielen Unix-Systemen zur Verfügung zu stellen. Der Grundstein für das KDE-Projekt wurde 1996 gelegt.

Die Abkürzung *KDE* steht für *K Desktop Environment*, was übersetzt etwa *K-Oberflächen-Umgebung* heißt. Welche Bedeutung der Buchstabe *K* hat, ist nicht bekannt. Womöglich war er als Kontrast zur kommerziellen Oberfläche *CDE* gedacht, vielleicht hat sich aber auch einer der Gründer in diesem Buchstaben verewigt. Dieses Geheimnis wird aber gut gehütet.

Während die grafische Oberfläche Windows von Microsoft in der PC-Welt eine enorme Verbreitung gefunden hat, konnte sich auf Unix-Systemen eine einheitliche Oberfläche bisher nicht durchsetzen. Die Vielfältigkeit der Oberflächen macht es aber einem Einsteiger schwer, sich in die Bedienung eines Programms einzuarbeiten. Das Fehlen eines Standards und die unterschiedlichen Vorlieben, die jeder Programmierer in seine Arbeit hat einfließen lassen, verlängern die Einarbeitungszeit.

Das KDE-Projekt versucht nun, einen solchen Standard auch in der Unix-Welt durchzusetzen. Das KDE-Projekt besteht dabei aus einer Vielzahl kleiner und größerer Programme. Alle Programme sind mit einer grafischen Benutzeroberfläche ausgestattet, die eine Reihe von festgelegten Bedingungen erfüllen muss. Diese Regeln orientieren sich dabei natürlich stark an bereits existierenden Standards – insbesondere auch an Microsoft Windows –, um dem Anwender den Umstieg von anderen Oberflächen auf KDE zu erleichtern. Die ersten zentralen Bestandteile des KDE-Projekts waren der Window-Manager *kwm* (seit KDE 2.0 heißt diese Komponente *kwin*) und der Dateimanager *kfm* (der nun den Namen *Konqueror* trägt). Beide sind sehr intuitiv und komfortabel zu bedienen. Im Laufe der Zeit kamen immer mehr Programme zum Projekt hinzu, von einfachen Texteditoren und Taschenrechnern über Werkzeuge für die Systemverwaltung bis hin zur Office-Anwendung. Natürlich sind inzwischen auch viele Spiele vorhanden.

Welches der KDE-Programme ein Anwender nutzen will, bleibt völlig ihm überlassen. Er kann jeden beliebigen Window-Manager benutzen, ohne dass die anderen KDE-Programme wesentlich beeinträchtigt würden. Auch *Konqueror*

muss nicht benutzt werden. Das Ziel des KDE-Projekts ist es natürlich, alle wichtigen Programme in das KDE-Projekt einzubinden. Dazu werden zum Beispiel Unix-Kommandos, die auf der Textebene arbeiten, mit einer grafischen Benutzeroberfläche versehen, um ihre Bedienung zu vereinfachen. Optionen lassen sich in einem übersichtlich angeordneten Fenster gerade für Anfänger und Umsteiger viel leichter einstellen als über Kommandozeilenparameter.

KDE ist ein freies Projekt, das keiner Firma untersteht. Wie auch beim Linux-Projekt kann sich jeder interessierte Programmierer am KDE-Projekt beteiligen. Die Arbeit ist natürlich unentgeltlich, da mit KDE keine Einnahmen erzielt werden. Über hundert Entwickler aus der ganzen Welt arbeiten bereits am KDE-Projekt mit. Wenn Sie eine gute Idee für ein weiteres KDE-Programm haben, so sind Sie herzlich eingeladen, sich an der Entwicklung zu beteiligen. Auch wenn Sie nicht programmieren wollen, können Sie das KDE-Projekt unterstützen: Sie können Dokumentationen für die Online-Hilfe schreiben oder übersetzen, sich an der Gestaltung der KDE-Homepage beteiligen oder sich auch einfach als Tester auf die Suche nach Fehlern begeben.

1.2 Die Qt-Bibliothek

Als Grundlage für die Entwicklung des KDE-Projekts dient die Grafikbibliothek *Qt* der norwegischen Firma *Trolltech*. Diese Firma wurde 1994 von einigen Entwicklern gegründet; die Arbeit an Qt begann allerdings schon 1992.

Qt ist eine Bibliothek für die Programmiersprache C++. Sie nutzt intensiv das Konzept der Klassenvererbung, wodurch die Bibliothek eine übersichtliche und einheitliche Schnittstelle bekommt. Auch die Erweiterung durch neue Klassen ist sehr einfach möglich.

Qt enthält Klassen für fast alle gängigen grafischen Eingabeelemente, zum Beispiel für Buttons, Auswahlboxen, Eingabezeilen und Menüleisten, sowie Klassen zur Anordnung dieser Elemente in einem Fenster. Weiterhin bietet Qt einfache Möglichkeiten, auf den Bildschirm zu zeichnen, von einfachen Punkten und Linien bis zu komplexen Vielecken und Text. Weitere Klassen sind für einen plattformunabhängigen Zugriff auf Dateien, das Ansteuern eines Druckers oder die Erzeugung elementarer Datenstrukturen zuständig.

Es gibt Qt in zwei Versionen: eine Version für X-Server, die auf den meisten gängigen Unix-Systemen lauffähig ist, und eine Version für Microsoft Windows (ab Windows 95). So lassen sich mit der Qt-Bibliothek Programme entwickeln, die sowohl unter Windows als auch unter den meisten Unix-Systemen lauffähig sind. Dazu muss der Quelltext des Programms meist nur auf dem entsprechenden System kompiliert werden. Für die Entwicklung freier Software bietet die Firma Trolltech die Möglichkeit, die Qt-Version für X-Server kostenlos zu nutzen.

Wollen Sie Programme für Windows entwickeln oder Ihre Programme verkaufen, brauchen Sie eine kostenpflichtige Lizenz. Nähere Informationen entnehmen Sie bitte dem Kapitel 1.5, *Lizenzbestimmungen*.

Beachten Sie, dass das KDE-Projekt ausschließlich für Unix-Betriebssysteme entwickelt wird. Auch wenn es auf der Qt-Bibliothek aufbaut, enthält es sehr viele spezielle Konzepte, die nur in einer Unix-Umgebung lauffähig sind. Wenn Sie also Programme entwickeln wollen, die auch unter Windows eingesetzt werden sollen, müssen Sie sich auf die Qt-Bibliothek beschränken und können die speziellen KDE-Klassen nicht benutzen.

Abbildung 1.1 zeigt die Abhängigkeiten zwischen den Bibliotheken, die ein KDE-Programm benutzt. Ein Pfeil bedeutet dabei, dass die Bibliothek am Ende des Pfeils von der am Anfang benutzt wird.

Eine KDE-Applikation greift auf die Klassen der KDE- und der Qt-Bibliothek zurück. Die KDE-Klassen selbst benutzen zum Teil die Qt-Klassen. Qt benutzt zum Zeichnen der Elemente und zum Zugriff auf den X-Server die Funktionen der XLib-Bibliothek. Diese stellt eine einfache Aufrufchnittstelle zum X-Server zur Verfügung, der dann die Befehle ausführt und die Grafik anzeigt. Der Programmierer der Applikation braucht dabei keine Kenntnisse von der Programmierung eines X-Servers zu besitzen. Er muss auch nicht die Aufrufchnittstelle der XLib-Bibliothek kennen. Es reicht in nahezu allen Fällen aus, dass er die Klassen der KDE- und Qt-Bibliotheken beherrscht.

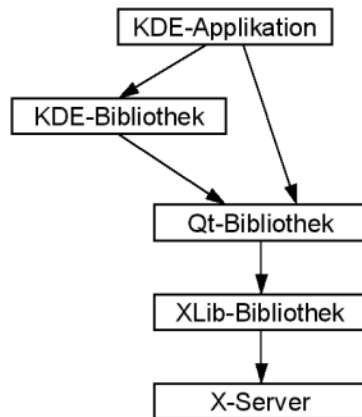


Abbildung 1-1 Abhängigkeiten zwischen den Bibliotheken

1.3 Vergleich mit ähnlichen Projekten

Unix-Systeme benutzen für die Darstellung von grafischen Benutzeroberflächen fast ausschließlich *X-Server*. Ein X-Server ist ein eigener Prozess, der die alleinige Kontrolle über den Bildschirm, die Tastatur und die Maus besitzt. Programme, die etwas auf dem Bildschirm darstellen wollen, müssen zunächst eine Verbindung zum X-Server aufbauen und dann ein oder mehrere so genannte Fenster öffnen – rechteckige Bildschirmbereiche, die vom X-Server verwaltet werden. Das Programm kann nur innerhalb der ihm zugeordneten Fenster zeichnen. Dazu muss es alle Zeichenbefehle an den X-Server schicken, der dann die Darstellung übernimmt. Die Befehle, die der X-Server ausführen kann, sind sehr mächtig: Sie reichen von der Auswahl einer Farbe und vom Setzen eines Punktes bis zum Füllen beliebig komplexer Vielecke mit frei definierbaren Mustern. Moderne Grafikkarten unterstützen den X-Server, indem sie bereits viele der Befehle im Grafikprozessor ausführen und so den Hauptprozessor für andere Aufgaben freihalten. Aber auch ältere Grafikkarten ohne Beschleunigerfunktion können mit der vollen X-Server-Funktionalität genutzt werden, indem der X-Server alle Berechnungen selbst ausführt und die Pixel auf dem Bildschirm setzt.

Durch die Trennung von ausgeführtem Programm und X-Server kann man diese Prozesse auch auf getrennten Rechnern laufen lassen, sofern diese über das Internet (oder ein internes Netzwerk mit TCP/IP-Unterstützung) verbunden sind. Auf diese Weise kann man zum Beispiel ein Programm auf einem Rechner auf der anderen Seite der Erde starten und sich die Fenster auf den eigenen Rechner schicken lassen. Durch die Mächtigkeit der X-Server-Befehle ist die Datenmenge, die übertragen werden muss, oft sehr gering. Auch über ein langsames Netz ist damit ein Arbeiten gut möglich.

Eine andere Anwendung von getrennten Rechnern für Programm und X-Server ist die Bedienung mehrerer Bildschirme durch einen Großrechner. Jeder Bildschirm – man nennt ihn in diesem Fall *X-Terminal* – enthält dabei einen kleinen Computer, auf dem nur der X-Server und das Netzwerkprotokoll laufen. Jeder Benutzer, der sich an ein X-Terminal setzt und sich in den Großrechner einloggt, bekommt die Ausgabe automatisch an den X-Server seines Bildschirms geschickt. Anders als in den meisten Fällen ist hier der Server ein kleiner, preisgünstiger und spezialisierter Rechner, der in der Regel nur mit Bildschirm, Tastatur und Maus ausgestattet ist, während der Client ein teurer, schneller Großrechner mit riesigen Festplatten- und Hauptspeicherkapazitäten ist.

X-Server haben sich in der Unix-Welt als Standard durchgesetzt. Nahezu jedes Unix-Betriebssystem wird bereits mit einem X-Server ausgeliefert. Unter Linux beispielsweise hat sich der X-Server *XFree86* etabliert, aber auch andere X-Server sind verfügbar. Die Schnittstelle zum X-Server ist standardisiert. Ein Programm kann also unabhängig vom Rechnertyp, auf dem es läuft, und vom Rechnertyp,

auf dem der X-Server läuft, geschrieben sein. Die Programmierung eines X-Servers ist jedoch sehr kompliziert und so aufwendig, dass selbst eine winzige Applikation ein langes und unübersichtliches Programm benötigt.

Keinem Programmierer, der eine Applikation mit grafischer Benutzeroberfläche entwickeln möchte, kann zugemutet werden, 80% oder mehr der Entwicklungsarbeit in die Implementierung von Schaltflächen, Auswahlboxen, Eingabefeldern und ähnlichen Elementen zu stecken. Für eine schnelle Entwicklung ist eine Bibliothek mit Benutzerelementen für eine grafische Oberfläche unverzichtbar. An einer solchen Bibliothek mangelte es in der Unix-Welt lange.

Ein Ansatz für eine einheitliche Bibliothek ist *Motif*. Für professionelle Anwendungen hat sich Motif verbreitet. Im Bereich der freien Software konnte Motif allerdings keinen Fuß fassen, da die Lizenzbestimmungen von Motif verlangen, dass jeder Anwender eines Motif-Programms die Motif-Bibliothek gegen eine Gebühr erwirbt. Insbesondere Linux-Benutzer haben aber selten Interesse daran, Geld für eine zusätzliche Bibliothek zu investieren. Außerdem braucht der Entwickler eines auf Motif basierenden Programms eine Lizenz, um seine Programme zu testen. Qt bietet für die Entwicklung kommerzieller Programme eine Lizenz auf Entwicklerbasis, bei der nur der Programmierer eine einmalige Lizenzgebühr bezahlt. Der Anwender kann die Bibliothek kostenlos nutzen, braucht also nur für das Programm selbst zu zahlen (sofern es nicht kostenlos ist). Nähere Informationen hierzu finden Sie in Kapitel 1.5, *Lizenzbestimmungen*.

Eine kostenlose Bibliothek mit GUI-Elementen ist die *XForms*-Bibliothek, die auf verschiedene Unix-Betriebssysteme portiert worden ist. Eine Reihe von freien Programmen benutzt diese Bibliothek. Durchsetzen konnte sie sich allerdings bisher nicht. Kritik wurde oft am Aussehen der Bedienelemente geübt. Da XForms vollständig in C geschrieben ist, also keine Klassenkonzepte nutzt, ist die Erstellung und Einbindung eigener Elemente aufwendig. Ein großer Vorteil der Bibliothek ist allerdings das Programm *FDesigner*, ein so genannter GUI-Builder, mit dem man Fenster und Bildschirmmenüs ganz einfach am Bildschirm zusammenstellen kann, ohne Quelltext schreiben zu müssen. FDesigner erstellt den Quelltext automatisch, der anschließend nur noch durch die gewünschte Funktionalität ergänzt werden muss. Ähnliche Programme gibt es auch für Qt. Bisher unübertroffen ist dabei zur Zeit das Programm *Qt Designer*, das von Trolltech selbst entwickelt wurde. Mit diesem Programm ist es sehr einfach, innerhalb kürzester Zeit auch komplexe Bildschirmdialoge zu erstellen. Eine detailliertere Beschreibung dieses Programms finden Sie in Kapitel 5.4, *Qt Designer*.

Das *GNOME*-Projekt hat ebenso wie KDE zum Ziel, alle wichtigen Programme in einer GNOME-Version zur Verfügung zu stellen, um eine einheitliche Bedienung zu ermöglichen. Es basiert auf der GUI-Bibliothek *GTK*, die in der Programmiersprache C geschrieben ist. Diese Bibliothek ist freie Software. Sie kann also kostenlos benutzt werden, um kommerzielle Programme zu entwickeln. Auch eine

C++-Version dieser Bibliothek existiert bereits. Sie heißt *GTK++*. Der Disput zwischen den Anhängern des KDE- und des GNOME-Projekts kochte zum Teil recht hoch. Schon einige Male wurde das eine oder das andere Projekt totgesagt, bisher halten sich beide Projekte aber noch sehr gut. Die Entwickler selbst betonen immer wieder, dass es sich nicht um einen Konkurrenzkampf handele. Jeder Anwender kann beide Projekte auf seinem Rechner installiert haben und auch gleichzeitig sowohl KDE- als auch GNOME-Programme benutzen. Es gibt inzwischen auch projektübergreifende Standards, so dass die Zusammenarbeit zwischen den Projekten verbessert wird.

Ein Kritikpunkt, der dem KDE-Projekt immer wieder vorgeworfen wurde, ist die Benutzung der Qt-Bibliothek. Qt war zu Anfang des KDE-Projekts keine freie Software im engeren Sinne, was vielen Linux- und BSD-Puristen ein Dorn im Auge war. Neuere Versionen von Qt wurden aber schrittweise unter immer lockereren Lizenzen ausgegeben, und seit Qt 2.2 ist die Unix-Version der Qt-Bibliothek unter den Lizenzbestimmungen der *General Public License (GPL)* erhältlich. Dieser Vorwurf ist damit endgültig entkräftet. Nähere Informationen hierzu finden Sie in Kapitel 1.5, *Lizenzbestimmungen*.

1.4 Informationsquellen

Alle Informationsquellen, die im Folgenden beschrieben werden, sind weitgehend englischsprachig. Mit den Grundkenntnissen aus dem Schulenglisch – zusammen mit den ohnehin englischen Fachausdrücken – sollte es jedoch kein Problem sein, diese Informationen zu nutzen.

Um weitere Informationen über Qt zu erhalten, eignet sich am besten die Homepage der Firma Trolltech unter <http://www.trolltech.com/>. Hier finden Sie allgemeine Informationen über die Firma und über die Qt-Bibliothek und in einer speziellen Rubrik für Entwickler auch die neueste Version von Qt, die vollständige Klassendokumentation, ein ausführliches Tutorial und Links auf viele Seiten von Projekten, die auf Qt basieren. Dort finden Sie auch eine aktuelle Preisliste für den Erwerb einer Professional Edition der Qt-Bibliothek.

Außerdem können Sie sich auf der Homepage von Trolltech in eine der beiden Mailing-Listen qt-announce@trolltech.com und qt-interest@trolltech.com eintragen. Über [qt-announce](mailto:qt-announce@trolltech.com) erhalten Sie eine Informations-E-Mail, sobald eine neue Version von Qt geplant ist. Über [qt-interest](mailto:qt-interest@trolltech.com) können Sie Fragen an die Mailing-Liste senden, die von anderen Programmierern, die auch die Mailing-Liste lesen, beantwortet werden. Stellen Sie auf dieser Mailing-Liste nur Fragen, die Qt betreffen. Fragen nach KDE-spezifischen oder compiler-spezifischen Problemen sind auf dieser Liste nicht gern gesehen. Die Mailing-Listen sind übrigens englischsprachig, deutschsprachige E-Mails sind auf der Liste ebenfalls nicht gern gesehen.

Informationen, wie Sie eine der Mailing-Listen abonnieren, finden Sie auf der Trolltech-Homepage. Dort können Sie auch in einem Archiv alle Fragen und Antworten nachlesen, die bisher über diese Mailing-Listen verschickt wurden.

Wenn Sie einen Fehler in der Qt-Bibliothek entdeckt haben, können Sie ihn per E-Mail an die Adresse bugs@trolltech.com melden. Machen Sie dabei möglichst genaue Angaben über die benutzte Qt-Version und die Umstände, unter denen der Fehler auftritt.

Zum Herunterladen von Software können Sie neben der Homepage auch den FTP-Server der Firma Trolltech unter <ftp://ftp.trolltech.com> nutzen.

Informationen über das KDE-Projekt finden Sie in erster Linie auf dessen Homepage unter <http://www.kde.org/>. Von dieser Seite existiert auch eine deutschsprachige Version, die jedoch nicht immer aktuell ist. Hier finden Sie Informationen zur Installation und Anwendung von KDE, aber auch viele Informationen für Entwickler. Auf der Homepage wird auf viele weitere Seiten verwiesen, jedoch ist es manchmal nicht ganz leicht, in der Menge der Links die interessanten Seiten zu finden. Eine Seite mit speziellen Tipps für KDE-Entwickler finden Sie unter <http://developer.kde.org/>. Dort finden Sie neben vielen Tutorials und Dokumentationen auch Links zu anderen Seiten, die die Entwicklung mit KDE, Qt und C++ zum Thema haben. Eine weitere interessante Seite ist <http://www.ph.unimelb.edu.au/~ssk/kde/devel/>. Dort finden Sie viele Informationen gerade für Einsteiger.

Zwei weitere KDE-Seiten finden Sie unter <http://www.kde.com/> und <http://apps.kde.com/>. Dort finden Sie neben neuesten Nachrichten aus der KDE-Welt die neuesten KDE-Programme zum Herunterladen, Informationen für Entwickler und viele Links auf weitere Seiten, und alles ist sehr übersichtlich angeordnet.

Auch für das KDE-Projekt gibt es eine Reihe von Mailing-Listen, in die Sie sich eintragen können, um über aktuelle Änderungen informiert zu werden. Beachten Sie jedoch, dass über diese Listen durchaus bis zu hundert E-Mails am Tag verschickt werden. Die für Sie interessanten E-Mails herauszufiltern ist also nicht ganz leicht.

Die Mailing-Liste *kde* ist ein allgemeines Diskussionsforum zu KDE. Über *kde-announce* werden neue Versionen und Anwenderprogramme angekündigt. In *kde-user* können Anwender Fragen stellen und Probleme schildern, die von anderen Abonnenten der Liste beantwortet werden. *kde-devel* ist eine spezielle Seite für Entwickler von KDE-Programmen. Hier werden Probleme diskutiert und Lösungen erarbeitet. Speziell für die Entwicklung der Kern-Komponenten von KDE gibt es die Liste *kde-core-devel*. In *kde-licensing* wird über lizenzrechtliche Fragen diskutiert. *kde-look* ist ein Diskussionsforum für das äußere Erscheinungsbild von KDE. Über die Mailing-Liste *kde-i18n* wird schließlich die Übersetzerarbeit für KDE-Programme organisiert. Weitere Mailing-Listen zu Spezialthemen rund um KDE werden je nach Bedarf eingerichtet.

Informationen darüber, wie Sie diese Listen abonnieren (*subscribe*), finden Sie auf der Seite <http://www.kde.org/maillinglists.html>. Schreibberechtigt sind Sie nur für die Mailing-Listen *kde*, *kde-user*, *kde-licensing*, *kde-look*, *kde-devel* und *kde-i18n*. Um in die *kde-core-devel*-Liste schreiben zu können, müssen Sie sich zuerst bei Martin Konold als Entwickler anmelden. Schreiben Sie dazu einfach eine kurze E-Mail an konold@kde.org.

Alle alten Nachrichten der Mailing-Listen können Sie auch im Archiv unter <http://lists.kde.org/> anschauen.

Zum Herunterladen von Software können Sie neben der Homepage auch den FTP-Server <ftp://ftp.kde.org> benutzen. Da dieser FTP-Server aber meist stark ausgelastet ist, sollten Sie einen der Spiegel-Server benutzen, die auf der KDE-Homepage aufgelistet sind.

Wenn Sie ein fertiges Programm geschrieben haben und dieses der Benutzerwelt zur Verfügung stellen wollen, so können Sie Ihr Programmpaket auf den Server <ftp://upload.kde.org> hochladen. Dieser Server ist meist deutlich weniger ausgelastet.

Neben den offiziellen KDE- und Qt-Bibliotheken gibt es noch eine große Anzahl an Klassen, die von Programmierern aus aller Welt entwickelt und ins Internet gestellt wurden. Sie finden eine Liste mit Klassen zum Beispiel auf der Seite <http://www.ksource.org/> sowie unter <http://apps.kde.com/> im Abschnitt DEVELOPMENT – LIBRARIES/CLASSES.

Da die KDE- und Qt-Bibliotheken in C++ geschrieben sind, benötigen Sie zum Entwickeln eigener KDE-Programme grundlegende C++-Kenntnisse. Neben der Programmiersprache C müssen Sie auch die Konzepte der Objektorientierung von C++ kennen, insbesondere die Klassendefinition und die Vererbung. Kenntnisse über Templates, virtuelle Methoden und die Operatorüberladung sind auch sehr nützlich, aber nicht unbedingt erforderlich. Das Verlagsprogramm von Addison-Wesley enthält eine ganze Reihe guter Bücher zur Programmiersprache C++, zum Beispiel das Buch *Die C++-Programmierersprache* von Bjarne Stroustrup, dem Entwickler von C++ (ISBN 3-8273-1660-X), oder das Buch *Go To C++-Programmierung* von André Willms (ISBN 3-8273-1495-X). Den eigenen C++-Stil kann man mit den beiden Büchern *Effektiv C++ programmieren* (ISBN 3-8273-1305-8) und *Mehr Effektiv C++ programmieren* (ISBN 3-8273-1275-2), beide von Scott Meyers, verbessern. Speziell für die Entwicklung von Programmen unter Linux ist das Buch *Anwendungen entwickeln unter Linux* von Michael K. Johnson und Erik W. Troan (ISBN 3-8273-1449-6) sehr empfehlenswert.

1.5 Lizenzbestimmungen

Die Lizenzbestimmungen der KDE- und Qt-Bibliotheken haben in der Vergangenheit oft für Verwirrung gesorgt und einige Grundsatzdiskussionen ausgelöst. Hauptursache hierfür ist die Vermischung des kommerziellen Produkts Qt mit freier Software nach dem Verständnis der Free Software Foundation. Die Lizenzbestimmungen wurden oftmals sehr eng ausgelegt, und es kam so zu Streitigkeiten, die der Sache der freien Software nicht immer dienlich waren. Im Folgenden werden wir die Lizenzbestimmungen, die Qt und KDE zugrunde liegen, etwas genauer beleuchten.

Alle hier beschriebenen Lizenzen können Sie in Anhang B, *Die Lizenzen von Qt und KDE*, im Originalwortlaut nachlesen.

1.5.1 Die kommerzielle Qt-Lizenz

Die Qt-Bibliothek der Firma Trolltech ist ein kommerzielles Produkt. Die Lizenzierung erfolgt dabei auf der so genannten Entwickler-Basis: Der Programmierer, der Qt nutzen will, kauft eine Lizenz von Qt und kann damit beliebig viele Programme entwickeln und verkaufen, ohne dass weitere Gebühren fällig werden. Die Programme dürfen dabei sowohl statisch als auch dynamisch mit der Qt-Bibliothek gelinkt sein.

Die Enterprise-Edition der Qt-Bibliothek gibt es in zwei Versionen: eine Version für X-Window-Systeme (also X-Server, vor allem auf Unix-Systemen) und eine Version für Microsoft Windows. Sie kostet für einen einzelnen Entwickler jeweils ca. \$ 1.900, im DuoPack (X-Window-System und Microsoft Windows zusammen) ca. \$ 2.900.

Neben dieser vollständigen Version gibt es auch noch eine »Light«-Version – die so genannte Professional-Edition. In ihr fehlen einzelne Module – unter anderem die Unterstützung von OpenGL, von netzwerktransparentem Dateizugriff und von den Klassen zur Analyse von XML-Dateien. Die Professional-Edition kostet pro Entwickler ca. \$ 1.500 für ein System bzw. \$ 2.300 im DuoPack.

Bei Lizenzen für mehrere Programmierer wird ein Mengenrabatt gewährt. Genauere Preisinformationen finden Sie auf der Homepage der Firma Trolltech unter <http://www.trolltech.com/>. In der Lizenz sind eine einjährige Support-Möglichkeit sowie kostenlose Updates für ein Jahr enthalten.

1.5.2 Die freie Qt-Lizenz

Obwohl der Preis der kommerziellen Lizenz für den Umfang und die Qualität des Produkts Qt durchaus angemessen ist, werden Hobby-Programmierer natürlich davon abgeschreckt. Trolltech bietet daher die X-Window-Version für Unix für

die Entwicklung freier Software kostenlos an. Diese Version unterscheidet sich in nichts von der Enterprise-Edition, außer in den Rechten und Pflichten des Entwicklers. Sie enthält ebenfalls alle Module und Hilfsprogramme sowie den gesamten Quellcode der Bibliothek.

Diese Qt Free Edition kann kostenlos – ohne dass eine Anmeldung nötig wäre – von der Homepage der Firma Trolltech heruntergeladen werden. Sie ist aber auch bereits in vielen Linux-Distributionen enthalten. Das Paket enthält neben der eigentlichen Bibliothek auch die Header-Dateien, den vollständigen Quellcode, die Klassendokumentation sowie ein Tutorial mit Beispielprogrammen. Solange Sie keine kommerzielle Lizenz von Qt erworben haben, dürfen Sie mit Qt entwickelte Programme nur als freie Software vertreiben: Sie müssen die kostenlose, uneingeschränkte Weitergabe des Programms erlauben, und müssen – zumindest auf Anfrage – den Quelltext des Programms weitergeben.

Als Lizenzbedingungen für die Qt Free Edition gilt die QPL (Q Public License). Ab Qt 2.2 gilt optional auch die GPL (General Public License) der Free Software Foundation. Beide Lizenztexte haben ähnliche Aussagen. In Stichworten lassen sich die wichtigsten Punkte folgendermaßen zusammenfassen:

- Ein Programm oder eine Bibliothek, das Sie mit der Qt Free Edition entwickelt haben, müssen Sie kostenlos weitergeben und den Quellcode offen legen. Sie können es beispielsweise ebenfalls unter den Bedingungen der GPL veröffentlichen oder unter einer der vielen anderen Lizenzen für freie Software. Sie dürfen Ihre Software dabei sowohl statisch als auch dynamisch mit Qt linkern.
- Sie erhalten den Quellcode der Qt-Bibliothek kostenlos. Sie dürfen diesen Quelltext auch selbst verändern und den veränderten Quellcode an andere uneingeschränkt weitergeben. (Sie dürfen dabei allerdings nicht die Lizenzbestimmungen ändern.)
- Die Firma Trolltech übernimmt keine Schäden, die durch Fehler in der Qt-Bibliothek entstehen.

Für den eigenen Gebrauch und für die Veröffentlichung als freie Software brauchen Sie also keine Lizenz für die Enterprise- oder Professional-Edition zu kaufen. Sie brauchen auch keine Lizenz, wenn Sie ein Programm, das jemand anderes entwickelt hat und das die Qt-Bibliothek nutzt, verwenden wollen, sei es privat oder geschäftlich. Nur wenn Sie mit Ihrem selbst geschriebenen Programm viel Geld verdienen wollen oder Sie den Quellcode Ihres Programms (z.B. aufgrund von Firmengeheimnissen) nicht öffentlich preisgeben wollen, müssen Sie die Enterprise- oder Professional-Edition kaufen. Das betrifft aber in den meisten Fällen nur Firmen und professionelle Programmierer, für die die doch angemessenen Lizenzgebühren kein wirkliches Problem sind.

Wenn Sie übrigens Qt für Microsoft Windows benutzen wollen, so müssen Sie leider ebenfalls eine Lizenz der Professional- oder Enterprise-Edition kaufen, da diese Version nicht als Qt Free Edition herausgegeben wird. Diese Version ist aber ebenfalls vor allem für Firmen und professionelle Programmierer interessant, die Programme schreiben wollen, die auf verschiedenen Plattformen laufen, ohne dass viele Anpassungen vorgenommen werden müssen. (Als günstige Alternative können Sie auch einen X-Server unter Windows starten und die Qt Free Edition für X-Window benutzen. Dieser Umweg läuft aber nicht so stabil und effizient wie die Microsoft Windows-Version von Qt.)

Seit Qt 2.2 wird die Qt Free Edition mit den Lizenzbedingungen der QPL und der GPL herausgegeben. Der Entwickler kann sich aussuchen, welche der beiden Lizenzen er berücksichtigen will. Dadurch ist man zu den meisten Lizenzbestimmungen für freie Software »kompatibel«. Qt kann also problemlos in Programmen genutzt werden, die zum Beispiel unter die GPL-, die LGPL-, die BSD- oder die Artistic-Lizenz gestellt sind. (Vor Qt 2.2 war nur die QPL möglich, die zumindest für spitzfindige Juristen nicht »kompatibel« zur GPL ist. Wollte man Qt daher in einem GPL-Programm nutzen, musste man explizit den Zusatz hinzufügen, dass ein Linken mit der Qt-Bibliothek gestattet ist. Dieser Hinweis, den Sie vielleicht noch in alten Programmen finden, ist nun nicht mehr nötig.)

In Anhang B, *Die Lizenzen von Qt und KDE*, finden Sie den Originalwortlaut (in Englisch) der oben beschriebenen Lizenzbestimmungen. Weitere Informationen finden Sie ebenfalls auf der Homepage von Trolltech unter <http://www.trolltech.com/> sowie auf der Homepage der Free Software Foundation – den Erfindern der GPL – unter <http://www.fsf.org/>.

1.5.3 Die KDE-Lizenzbestimmungen

Das KDE-Projekt ist nicht kommerziell. Es ist freie Software, die kostenlos weitergegeben und genutzt werden kann. Es handelt sich also nicht um Shareware, bei der Sie nach einer Testphase das Produkt kaufen müssen. Der Quelltext von KDE ist öffentlich, kann also von jedem Interessierten benutzt und auch verändert werden, solange die Copyright-Vermerke unangetastet bleiben.

Die KDE-Bibliotheken stehen unter der *Library General Public License (LGPL)*, die von der Free Software Foundation erarbeitet wurde und die auch für die meisten Linux-Bibliotheken gilt. Diese Bibliotheken dürfen kostenlos benutzt und weitergegeben werden, auch zur Entwicklung kommerzieller Programme. Sie dürfen also Programme, die die KDE-Bibliotheken benutzen, zu einem beliebigen Preis verkaufen. Beachten Sie aber dabei, dass die KDE-Bibliotheken meist die Qt-Bibliothek voraussetzen, so dass Sie zumindest eine Qt Enterprise- oder Professional-Edition benötigen. Die KDE-Bibliotheken dürfen modifiziert werden. Allerdings müssen die Lizenzbedingungen auch für die modifizierten Bibliotheken gelten, und Copyright-Vermerke dürfen nicht verändert werden.

Für die Anwenderprogramme des KDE-Projekts gilt die *General Public License (GPL)*, die ebenfalls von der Free Software Foundation erarbeitet wurde. Die Programme dürfen daher kostenlos genutzt und weitergegeben werden. Sie dürfen modifiziert werden. Es ist auch erlaubt, Teile vom Quellcode für eigene Programme zu benutzen. Diese Programme müssen dann allerdings auch freie Software sein, dürfen also nicht kommerziell genutzt werden.

Wenn Sie selbst ein Programm für das KDE-Projekt schreiben wollen, sollten Sie es ebenfalls unter die GPL stellen. Wenn Sie eine neue Klasse entwickeln, die in die KDE-Bibliotheken mit aufgenommen werden soll, stellen Sie sie unter die LGPL. Das Programm bzw. die Klasse kann dann problemlos in das KDE-Projekt aufgenommen werden. Eine kommerzielle Nutzung Ihres Programms ist dann aber nicht mehr möglich.

Den genauen Wortlaut der Lizenzbestimmungen können Sie in Anhang B, *Die Lizenzen von Qt und KDE*, nachlesen.

2 Erste Schritte

In diesem Kapitel werden wir zwei kleine Programme erstellen, die die Mächtigkeit von KDE und Qt aufzeigen sollen und die als erste Grundlage für eigene Experimente dienen können. Diese beiden Programme werden in den Kapiteln 2.2, *Das erste Qt-Programm*, und 2.3, *Das erste KDE-Programm*, behandelt. In Kapitel 2.4, *Was fehlt noch zu einer KDE-Applikation?*, werden weitere Features aufgezeigt, die ein KDE-Programm bieten sollte. Zunächst wollen wir aber in Kapitel 2.1, *Benötigte Programme und Pakete*, die Software beschreiben, die Sie installieren müssen, um eigene Qt- und KDE-Programme entwickeln zu können.

2.1 Benötigte Programme und Pakete

Um KDE-Programme zu entwickeln, benötigen Sie auf Ihrem System einen C++-Compiler, die KDE- und Qt-Bibliotheken sowie die zugehörigen Header-Dateien und zum Ausprobieren natürlich einen laufenden X-Server.

Als C++-Compiler hat sich unter Linux der *gcc* etabliert. Aber auch alle anderen Compiler sind hier möglich. Weder die Qt-Bibliotheken noch die KDE-Bibliotheken stellen besondere Ansprüche an den unterstützten Sprachumfang. Einfache Templates und Namensräume beherrscht nahezu jeder C++-Compiler, Exceptions werden bisher nicht verwendet. Benutzen Sie möglichst eine aktuelle, stabile Compiler-Version.

Um die Quelltexte zu erstellen, benötigen Sie natürlich auch einen Editor. KDE-Programme stellen keine besonderen Ansprüche an den Editor, so dass Sie am besten weiterhin Ihren gewohnten Editor benutzen. Neben rudimentären Editoren, wie dem *vi* oder dem *aXe*, können Sie natürlich auch aufwendigere Editoren benutzen, die Sie beispielsweise durch Syntaxhervorhebung unterstützen, wie etwa den *Emacs* bzw. *XEmacs* oder den *nedit*. Eine integrierte Entwicklungsumgebung stellt zum Beispiel auch das Programm *KDevelop* dar, das neben einem Editor auch andere unterstützende Möglichkeiten bietet, die speziell bei der Entwicklung von KDE-Programmen die Arbeit sehr erleichtern können (siehe Kapitel 5.5, *KDevelop*).

Zum Ausprobieren der Programme eignet sich jeder beliebige X-Server, wie zum Beispiel der *XFree86*-Server, der vielen Linux-Distributionen beiliegt. An die Grafikauflösung werden keine besonderen Ansprüche gestellt, mindestens 800 x 600 Pixel und 256 Farben sind für ein gutes Arbeiten aber sicher sinnvoll.

Die Qt-Bibliothek für Unix-Betriebssysteme ist in den meisten neueren Linux-Distributionen bereits enthalten. Es ist jedoch wichtig, dass Sie das *Developers-Paket* installieren, denn nur dieses Paket enthält neben der kompilierten Bibliothek die Header-Dateien, die unverzichtbar sind, wenn Sie selbst KDE- oder Qt-

Programme kompilieren wollen. Außerdem enthält das Developers-Paket eine umfassende Klassenreferenz im HTML-Format, ein einleitendes Tutorial, einige Beispielprogramme sowie den vollständigen Quellcode der Qt-Bibliothek. Bei Bedarf können Sie die Bibliothek auch selbst kompilieren, wenn Sie spezielle Compiler-Optionen benutzen wollen. Falls Ihre Distribution das Developers-Paket nicht enthält, können Sie es auch von der Buch-CD installieren. Die aktuellste Version erhalten Sie im Internet auf der Homepage der Firma Troll Tech unter <http://www.trolltech.com/>.

Die derzeitige aktuelle Version der Qt-Bibliothek hat die Versionsnummer 2.2. Sie enthält eine Reihe von Ergänzungen zu den Vorgängerversionen 2.0 und 2.1 und ist weit gehend kompatibel zu diesen. Die aktuelle KDE-Version 2.0 lässt sich jedoch nur zusammen mit Qt 2.2 nutzen. Falls Sie eine neuere Version als KDE 2.0 verwenden wollen, so achten Sie darauf, dass Sie die dazu passende Qt-Version installieren.

Auch die KDE-Pakete sind inzwischen in den meisten neueren Linux-Distributionen enthalten. Um eigene Programme zu entwickeln, müssen Sie mindestens das Paket *kdelibs* installieren, das die KDE-Bibliotheken sowie deren Header-Dateien enthält. Weiterhin sollten Sie die Pakete *kdebase* und *kdesupport* installieren, um Ihre Programme in einer KDE-Umgebung (mit dem Window-Manager *KWin* und dem File-Manager *Konqueror*) zu testen. Falls diese Pakete nicht in Ihrer Distribution enthalten sein sollten, können Sie auch den Quellcode der Bibliotheken auf der Buch-CD verwenden und die Bibliotheken selbst kompilieren. Die aktuellste KDE-Version finden Sie auf der KDE-Homepage (<http://www.kde.org/>) oder auf dem KDE-ftp-Server (<ftp://ftp.kde.org>).

Sofern es bei der Installation der Bibliotheken nicht automatisch erfolgt ist, sollten Sie unbedingt zwei Umgebungsvariablen, *\$QTDIR* und *\$KDEDIR*, setzen, die auf die Unterverzeichnisse der Pakete verweisen. Insbesondere sollten sich im Verzeichnis *\$QTDIR/include* die Qt-Header-Dateien (wie zum Beispiel *qapplication.h* oder *qobject.h*) und in *\$KDEDIR/include* die KDE-Header-Dateien (wie zum Beispiel *kapp.h* oder *ktmainwindow.h*) befinden.

Weiterhin sollten sich die Bibliotheken *libqt*, *libkdeui*, *libkdecore*, *libkfile*, *libkfm* im Suchpfad des Linkers befinden. Dazu können Sie entweder diese Bibliotheken in das Verzeichnis */usr/lib* kopieren, oder (falls noch nicht geschehen) einen symbolischen Link auf die Bibliothek in diesem Verzeichnis anlegen. Falls Sie die Dateien nicht kopieren wollen, müssen Sie zusätzlich noch die Pfade mit den Bibliotheken in die Datei */etc/ld.so.conf* eintragen und dann das Programm *ldconfig* starten. Bei einer fertigen Distribution sind diese Schritte in der Regel schon erfolgt. Wenn Sie Qt und KDE selbst installieren, halten Sie sich am besten an die Installationsanweisungen in den *README*-Dateien.

Es gibt noch eine Reihe weiterer Programme, die Sie beim Erstellen einer KDE-Applikation unterstützen. Eine Auflistung der wichtigsten Programme finden Sie in Kapitel 5, *Hilfsmittel für die Programmerstellung*. Für unsere ersten Versuche werden wir jedoch den Quellcode von Hand erstellen und übersetzen.

Dieses Buch beschreibt die jeweils aktuellen Versionen Qt 2.2 und KDE 2.0. Wenn Sie ältere Programme an diese Versionen anpassen wollen, so bieten Ihnen das Kapitel 4.22, *Programme von Qt 1.x auf Qt 2.x portieren*, und Kapitel 4.23, *Programme von KDE 1.x auf KDE 2.x portieren*, viele Informationen dazu.

2.2 Das erste Qt-Programm

Als erstes Beispiel wollen wir das klassische Hello-World-Beispiel als Qt-Programm schreiben. Von den KDE-Bibliotheken machen wir dabei noch keinen Gebrauch.

2.2.1 Das Listing

Hier folgt das Listing des ersten Programms. Bereits die folgenden 15 Zeilen (davon drei Kommentarzeilen und eine Leerzeile) reichen aus.

```
// Das erste Programm
// KDE- und Qt-Programmierung
// Addison-Wesley Germany
#include <qapplication.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}
```

Geben Sie dieses Programm mit einem beliebigen Texteditor ein, und speichern Sie es unter dem Namen *hello-qt.cpp* ab.

2.2.2 Kompilieren des Programms unter Linux

Anschließend kompilieren Sie das Programm mit dem folgenden Befehl:

```
% g++ -o hello-qt -I$QTDIR/include -lqt hello-qt.cpp
```

Das Prozentzeichen am Anfang der Zeile ist nicht einzugeben, es dient hier nur als Zeichen dafür, dass diese Zeile in einer Kommandozeile einzugeben ist. Der

aufgerufene Compiler ist hier das Programm `g++`. Dabei handelt es sich bei den meisten Linux-Distributionen um ein Skript, das den `gcc` mit speziellen Parametern aufruft, um C++-Dateien zu übersetzen. Sollte `g++` auf Ihrem System nicht installiert sein oder wollen Sie einen anderen Compiler benutzen, so müssen Sie hier den entsprechenden Befehl benutzen. Beachten Sie, dass Sie unter Umständen andere Parameter angeben müssen.

Hier eine kurze Erläuterung der Kommandozeilenparameter im Einzelnen:

- `-o hello-qt` legt den Namen der ausführbaren Datei fest, die erzeugt werden soll.
- `-I$QTDIR/include` weist den Compiler an, auch das angegebene Verzeichnis nach benötigten Header-Dateien zu durchsuchen. In unserem Fall sind das konkret die Dateien `qapplication.h` und `qlabel.h`.
- `-lqt` gibt an, dass die ausführbare Datei auf die Bibliotheksdatei `libqt.so` zugreifen wird.
- `hello-qt.cpp` legt schließlich die zu kompilierende Quelldatei fest.

Diese Zeile sollte zunächst den C++-Compiler und anschließend den Linker aufrufen. Wenn alles geklappt hat und keine Fehlermeldung ausgegeben wurde, so wurde die ausführbare Datei `hello-qt` erzeugt. Trat jedoch ein Fehler auf, so lesen Sie unten im Abschnitt *Problembehandlung* weiter.

Starten Sie nun die ausführbare Datei. Geben Sie dazu auf der Kommandozeile

```
% hello-qt
```

ein (wiederum ohne das Prozentzeichen), oder klicken Sie auf das Dateisymbol im Dateimanager (*Konqueror*). Wenn auch das funktioniert, so sollte sich ein Fenster öffnen, das ähnlich wie in Abbildung 2.1 aussieht. Dieses Fenster können Sie mit der Maus verschieben oder in der Größe ändern. Je nach Breite des Fensters wird der Schriftzug dabei in einer oder zwei Zeilen dargestellt. Durch Klicken auf den Button oben rechts in der Titelleiste schließen Sie das Fenster und beenden dadurch das Programm. (Einige Window-Manager – zum Beispiel der *fwm* – bieten keinen CLOSE-Button in der Titelleiste. In diesem Fall klicken Sie auf den Fenstermenü-Button oben links in der Titelleiste und wählen im erscheinenden Menü den Punkt SCHLIESSEN bzw. CLOSE.)



Abbildung 2-1 Bildschirmausgabe des Programms `hello-qt` unter Linux

Wenn Ihr erstes Programm so weit funktioniert, können Sie in Kapitel 2.2.4, *Analyse des Programms*, erfahren, was die einzelnen Befehle bedeuten. Sollte es Schwierigkeiten gegeben haben, so lesen Sie den folgenden Abschnitt.

Problembehandlung

Wenn sich das Programm nicht auf Anhieb kompilieren und starten lässt, kann das viele Ursachen haben. Lassen Sie sich dadurch aber nicht verunsichern: Sobald Sie Ihr System einmal richtig eingerichtet haben, können Sie weitere Programme problemlos erstellen.

Versuchen Sie, den Fehler anhand der folgenden Liste einzukreisen und zu beheben.

- Der Compiler bricht mit der Fehlermeldung ab, dass die Datei *qapplication.h* oder *qlabel.h* nicht gefunden werden kann. Es folgen in der Regel noch weitere Warnungen und Fehlermeldungen.

Dieser Fehler tritt auf, wenn der Compiler die Header-Dateien der Qt-Bibliothek nicht finden konnte. Damit der Compiler weiß, wo er suchen muss, haben wir beim Aufruf den Parameter *-I\$QTDIR/include* angegeben.

Stellen Sie sicher, dass die Umgebungsvariable `QTDIR` korrekt gesetzt ist. Sie muss auf das Hauptverzeichnis des Qt-Pakets zeigen. Alternativ können Sie auch den vollständigen Pfad angeben, zum Beispiel *-I/usr/lib/qt/include*.

Im Unterverzeichnis *include* sollten die benötigten Dateien *qapplication.h* und *qlabel.h* liegen. Achten Sie unbedingt auf die Groß-/Kleinschreibung: Alle Dateinamen sollten vollständig kleingeschrieben sein, sowohl im *include*-Unterverzeichnis als auch im Listing. Oder sind unter Umständen die langen Dateinamen abgeschnitten worden, da Sie die Dateien zwischendurch auf einem alten DOS-System gespeichert hatten?

- Der Compiler meldet, dass die Bibliotheksdatei `qt` nicht gefunden werden konnte (z. B. »cannot open libqt«).

Diese Meldung kommt vom Linker (in der Regel das Programm *ld*), der vom Compiler aufgerufen wird. Sie besagt, dass die dynamische Bibliotheksdatei *libqt.so* (bzw. *libqt.a*, wenn Sie die Bibliothek statisch einbinden) nicht gefunden wurde. Diese Datei enthält den bereits kompilierten Code der Qt-Klassen. Kontrollieren Sie in diesem Fall, ob Sie alle Schritte zur Installation der Qt-Pakete korrekt durchgeführt haben. Sie sollten die Datei im Verzeichnis *\$QTDIR/lib* finden. Sie ist in der Regel ein Link auf eine Datei mit etwas präziserer Versionsangabe. Für Qt 2.2.0 lautet die Bibliotheksdatei beispielsweise *libqt.so.2.2.0*.

Falls die Datei vorhanden ist, der Compiler sie aber trotzdem nicht findet, können Sie zusätzlich noch den Suchpfad als Kommandozeilenparameter mit der Option `-L` angeben: `g++ -o hello-qt -I$QTDIR/include -L$QTDIR/lib -lqt hello-qt.cpp`

- Der Compiler meldet, dass einige Referenzen zu Klassen-Methoden nicht aufgelöst werden konnten (z. B. »Undefined reference to QApplication::QApplication (int &, char *)« oder eine ähnliche Meldung).

Diese Meldung kommt vom Linker, der vom Compiler automatisch aufgerufen wird. Das erkennen Sie in der Regel an der abschließenden Meldung »ld returned 1 exit status« oder einer ähnlichen Ausgabe. Sie besagt, dass der Maschinencode für einige Klassen-Methoden nicht in den eingebundenen Bibliotheken gefunden werden konnte. Eine mögliche Ursache ist, dass die Bibliotheksdatei *libqt.so* nicht gefunden wurde, der Compiler aber keine Fehlermeldung liefert. Einige Compiler ignorieren nämlich Bibliotheken, die nicht gefunden werden konnten. Sie erkennen das meist daran, dass die erste Meldung auf eine fehlende Methode »QApplication::QApplication (int &, char *)« hinweist. Eine andere mögliche Ursache ist, dass die Version der Bibliotheksdatei nicht mit der Version der Header-Dateien übereinstimmt. Das kann sehr leicht passieren, wenn noch eine andere Qt-Version im System installiert ist. In der Regel ist es dann der Konstruktor der Klasse *QLabel*, der nicht gefunden wird.

In beiden Fällen müssen Sie sicherstellen, dass sich die richtige Bibliotheksdatei im Dateisystem befindet. Versuchen Sie, das Verzeichnis der Datei zusätzlich als Option mit `-L$QTDIR/lib` festzulegen, wie es im vorhergehenden Punkt besprochen wurde.

- Das Kompilieren verlief fehlerfrei, und die ausführbare Datei *hello-qt* wurde erzeugt. Beim Versuch, das Programm zu starten, erscheint aber eine Fehlermeldung, dass eine Bibliothek nicht gefunden werden konnte. Besonders oft tritt dieser Fehler auf, wenn Sie die ausführbare Datei auf einem anderen Rechner oder unter einer anderen Linux-Distribution oder nach einem Betriebssystem-Update starten wollen.

Der Code aus dynamischen Bibliotheken wird nicht in die ausführbare Datei kopiert. Dieser Code wird erst beim Starten des Programms zusätzlich in den Speicher geladen. Dadurch bleibt die ausführbare Datei klein, und mehrere Programme können sich die gleichen Bibliotheken teilen. Dazu müssen die dynamischen Bibliotheken aber beim Start des Programms gefunden werden, und zwar in der gleichen Version, wie sie beim Linken benutzt wurden.

Mit dem Programm *ldd* können Sie testen, welche dynamischen Bibliotheken eine ausführbare Datei benötigt und welche dieser Bibliotheken gefunden wurden. Geben Sie zum Beispiel *ldd hello-qt* ein, um unser Beispielprogramm

zu untersuchen. Als Ergebnis erhalten Sie die Liste aller hinzugelinkten dynamischen Bibliotheken, und einen Verweis auf die tatsächlich benutzte Bibliotheksdatei, falls sie gefunden wurde. Wahrscheinlich fehlt die entsprechende Datei *libqt.so.2*, es kann aber auch an der Datei *libjpeg*, *libXext* oder *libX11* liegen. Wurde eine der Dateien nicht gefunden, so durchsuchen Sie das Dateisystem nach einer Datei mit diesem Namen (z. B. mit dem Programm *find*). Ist diese Datei vorhanden, können Sie sie entweder in eines der Bibliotheksverzeichnisse kopieren oder verschieben bzw. einen symbolischen Link mit demselben Namen in einem der Bibliotheksverzeichnisse anlegen (z. B. im Verzeichnis */usr/lib*, oder */usr/local/lib*). Alternativ können Sie das Verzeichnis, das die fehlende Bibliotheksdatei enthält, mit in den Suchpfad für Bibliotheken aufnehmen. In den meisten Linux-Distributionen haben Sie dazu zwei Möglichkeiten: Entweder tragen Sie das Verzeichnis in die Umgebungsvariable *LD_LIBRARY_PATH* ein, oder Sie tragen diesen Pfad in die Datei */etc/ld.so.conf* ein und führen anschließend (mit root-Rechten) das Programm *ldconfig* aus. Anschließend können Sie wiederum mit *ldd* prüfen, ob nun alle dynamischen Bibliotheken gefunden werden.

2.2.3 Kompilieren des Programms unter Microsoft Windows

Eine der Intentionen bei der Entwicklung von Qt war es, ein grafisches Benutzer-Toolkit zu schaffen, das sowohl unter Microsoft Windows als auch unter den meisten Unix-Betriebssystemen genutzt werden kann. Diese Besonderheit ist besonders für professionelle Entwickler interessant, da sie ein Programm schreiben können, das auf fast allen in Industrie und Wirtschaft genutzten Systemen lauffähig ist. Um das Programm auf ein anderes System zu portieren, reicht es meist schon aus, es neu zu kompilieren. Eine aufwendige Anpassung ist nicht nötig.

Auch unser Hello-World-Programm benutzt nur die Qt-Bibliothek und keine Klassen der KDE-Bibliothek. Daher ist es ohne weiteres auch auf Microsoft Windows kompilierbar. Dazu ist aber die Windows-Version von Qt nötig. Sie wird von Trolltech nicht kostenlos zur Verfügung gestellt. Um sie zu nutzen, müssen Sie also eine kostenpflichtige Lizenz erwerben. Die aktuellen Preise und Nutzungsbedingungen erfahren Sie auf der Homepage der Firma Trolltech unter <http://www.trolltech.com>.

Weiterhin benötigen Sie einen 32-Bit-Compiler für Microsoft Windows. Sie können beispielsweise den *FreeBCC 5.5* von Borland/Imprise benutzen, einen Kommandozeilen-C++-Compiler, den Sie kostenlos unter <http://www.borland.com> herunterladen können. Die Nutzung dieses Compilers wird im Folgenden näher beschrieben. Sie können aber auch andere Compiler benutzen, zum Beispiel eine der integrierten Entwicklungsumgebungen *Borland C++ Builder* (ab Version 4) oder *Microsoft Visual C++* (ab Version 5.0).

Um den *FreeBCC*-Compiler zu benutzen, müssen Sie diesen zunächst installieren, sofern das noch nicht geschehen ist. Dazu entpacken Sie das Compiler-Archiv und folgen anschließend der Installationsanleitung in der Datei *README.TXT*. Anschließend entpacken Sie das Qt-Archiv für Windows – die Datei lautet beispielsweise *qtwin211.zip*. Fügen Sie nun am besten die Verzeichnisse für Header-Dateien und Bibliotheken in die Konfigurationsdateien *bcc32.cfg* und *ilink32.cfg* ein, die Sie bei der Installation des Compilers im *bin*-Verzeichnis erzeugt haben. Der Inhalt der Dateien könnte beispielsweise folgendermaßen aussehen, eventuell mit angepassten Pfadangaben:

```
c:\borland\bcc55\bin\bcc32.cfg:
```

```
-I"c:\borland\bcc55\include"  
-I"c:\qt\include"  
-L"c:\borland\bcc55\lib"  
-L"c:\qt\lib"
```

```
c:\borland\bcc55\bin\ilink32.cfg:
```

```
-L"c:\borland\bcc55\lib"  
-L"c:\qt\lib"
```

Im nächsten Schritt kompilieren Sie nun die Qt-Bibliothek. Sie können Sie als statische (LIB) oder dynamische Bibliothek (DLL) erstellen lassen. Es scheint aber einige Probleme zu geben, wenn Sie mit der aktuellen BCC-Version 5.5 und einer Qt-Version ab Qt 2.1 eine dynamische Bibliothek erstellen wollen. Aktuelle Informationen dazu erhalten Sie auf der Homepage der Firma Trolltech (<http://www.trolltech.com>) im Developers-Teil im Abschnitt *Platform Notes*. Solange diese Probleme nicht behoben wurden, empfehle ich Ihnen die Erstellung von statisch gebundenen ausführbaren Dateien. Dadurch werden die ausführbaren Dateien allerdings mindestens 2 MByte groß (auch unser Hello-World-Programm). Auf der anderen Seite hat eine statisch gebundene Datei den Vorteil, dass sie ohne Probleme auf ein anderes Windows-System (95/98/NT/2000/ME) kopiert werden kann und dort ohne Installation der Qt-DLL lauffähig ist.

Um die Qt-Bibliothek zu erstellen, entpacken Sie den Inhalt der Datei *mkfiles\borland.zip* (bzw. *mkfiles\borland-dll.zip*, wenn Sie eine dynamische Bibliothek erstellen wollen) in das Qt-Verzeichnis. Nehmen Sie nun das *bin*-Verzeichnis des Compilers sowie das *bin*-Verzeichnis der Qt-Bibliothek in den Pfad auf. Wechseln Sie anschließend in das Verzeichnis *src*, und starten Sie hier das Programm *make*. Je nach Rechnerleistung dauert der Vorgang ca. zehn Minuten bis zu einer Stunde. Anschließend finden Sie im Verzeichnis *lib* die Datei *qt.lib* sowie eventuell die Datei *qt221.dll*. Zu Testzwecken können Sie jetzt die mitgelieferten Beispielprogramme kompilieren und testen. Wechseln Sie dazu in das Verzeichnis *examples*, und starten Sie erneut das Programm *make*. Erneut dauert es eine Weile, bis alle Programme erstellt worden sind. Die ausführbaren Dateien finden Sie in den Unterverzeichnissen.

Falls Sie die Beispielprogramme statisch gebunden haben, belegt jedes Beispiel mindestens 2 MByte. Bei etwa 50 Beispielprogrammen sind das über 100 MByte Speicher auf der Festplatte. Um nach dem Ausprobieren diesen Speicherplatz wieder freizugeben, starten Sie im Verzeichnis *examples* das Programm *make clean*.

Um nun das Programm *hello-qt.cpp* zu kompilieren und zu linkern, wechseln Sie zunächst in das Verzeichnis mit dieser Datei und geben danach nacheinander die folgenden beiden Zeilen ein:

```
% bcc32 -c hello-qt.cpp  
% ilink32 -aa hello-qt.obj c0w32.obj import32.lib cw32.lib qt.lib
```

Die Prozentzeichen am Anfang der Zeilen dienen nur als Hinweis, dass diese Befehle zum Beispiel in einer DOS-Box eingegeben werden müssen. Sie müssen diese Zeichen nicht eingeben. Der zweite Befehl ist in einer einzigen Zeile einzugeben. Das zweite Zeichen der Datei *c0w32.obj* ist die Ziffer »0«, nicht der Buchstabe »O«.

Nach diesen beiden Befehlen sollte sich im aktuellen Verzeichnis eine ausführbare Datei mit dem Namen *hello-qt.exe* befinden. Wenn Sie diese Datei ausführen lassen, sollte sich ein Fenster öffnen, das ähnlich wie in Abbildung 2.2 aussieht.



Abbildung 2-2 Bildschirmausgabe des Programms *hello-qt* unter Microsoft Windows

2.2.4 Analyse des Programms

Wir wollen den Aufbau unseres ersten Programms nur kurz anschauen, ohne zu sehr in die Details zu gehen. Eine genauere Beschreibung finden Sie in Kapitel 2.3, *Das erste KDE-Programm* sowie in Kapitel 3, *Grundkonzepte der Programmierung in KDE und Qt*.

Nach den ersten drei Kommentarzeilen folgen zwei Präprozessor Direktiven, die die Header-Dateien *qapplication.h* und *qlabel.h* einbinden:

```
#include <qapplication.h>  
#include <qlabel.h>
```

Diese Header-Dateien befinden sich im Verzeichnis *\$QTDIR/include* und enthalten die Klassendefinitionen für die Qt-Klassen *QApplication* bzw. *QLabel*, die weiter unten im Programm benutzt werden. Nahezu jede Qt-Klasse besitzt eine

eigene Header-Datei, die eingebunden werden muss, wenn die Klasse benutzt werden soll. Diese Header-Datei besitzt den gleichen Namen wie die Klasse, allerdings vollständig in Kleinbuchstaben. Alle Qt-Klassen beginnen mit dem Buchstaben *Q* und sind daher sehr leicht zu erkennen. KDE-Klassen – die wir hier jedoch noch nicht benutzt haben – beginnen analog mit dem Buchstaben *K*.

Im Gegensatz zu einigen anderen grafischen Bibliotheken fasst Qt die Klassendefinitionen nicht zu einer einzigen Header-Datei zusammen, sondern stellt für jede Klasse eine eigene Header-Datei zur Verfügung. Das erhöht zwar den Aufwand für den Programmierer, der für jede verwendete Klasse eine *include*-Direktive in den Quelltext einfügen muss, verkürzt aber die Kompilierzeit, da jede einzelne Header-Dateien nun sehr kurz ist.

Wie alle C- und C++-Programme braucht auch unser Programm die Funktion *main*, die beim Aufruf des Programms gestartet wird.

```
int main (int argc, char **argv)
{
    ...
}
```

Die *main*-Funktion hat die beiden Parameter *argc* und *argv*, in denen die Anzahl der Kommandozeilenparameter bzw. die Kommandozeilenparameter selbst gespeichert werden. Diese Variablen benötigt Qt, um einige der Parameter selbst zu interpretieren.

Die erste Zeile in der Funktion *main* erzeugt ein lokales Objekt der Klasse *QApplication* mit dem Variablennamen *app*:

```
QApplication app (argc, argv);
```

Dieses Objekt übernimmt einige Initialisierungen und interpretiert die Kommandozeilenparameter, die dem Konstruktor als Argumente übergeben werden.

Die nächste Zeile erzeugt ein Objekt der Klasse *QLabel* dynamisch auf dem Heap und legt einen Zeiger auf dieses Objekt in der Variablen *l* ab.

```
QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
```

Dieses Objekt dient dazu, auf dem Bildschirm ein Fenster mit einem Text anzuzeigen. Der Text, der angezeigt werden soll, wird dabei als erster Parameter des Konstruktors übergeben. Das Format des Strings ist dabei das so genannte RichText-Format, einer Untermenge des HTML-Formats. (Verwechseln Sie dieses RichText-Format von Qt nicht mit dem RTF-Dateiformat, mit dem viele Textverarbeitungsprogramme ihre Texte speichern.) Das so genannte Tag *<H1>* bewirkt, dass der folgende Text als Überschrift der obersten Stufe, also in größerer Schrift und fett dargestellt wird. Am Ende des Strings bewirkt das korrespondierende Tag

</H1>, dass das erste Tag abgeschlossen wird. Der zweite Parameter des Konstruktors ist ein Null-Zeiger. Er gibt an, in welches andere Fenster unser neues Fenster integriert werden soll. Da unser Fenster aber ein eigenständiges Fenster werden soll, benutzen wir hier als Kennzeichen dafür einen Null-Zeiger. Beachten Sie, dass aufgrund der strikteren Typprüfung in C++ für den Null-Zeiger nicht mehr wie in C das Makro *NULL* benutzt wird, da es auf einigen Compilern zu Warnungen führt. Stattdessen wird die Zahl 0 benutzt. Leider kann man so aber nicht mehr auf den ersten Blick unterscheiden, ob hier die Integer-Zahl 0 oder der Zeiger auf die Adresse 0 gemeint ist.

Unser Fenster wird allerdings zu diesem Zeitpunkt im Programmablauf noch nicht auf dem Bildschirm angezeigt. Standardmäßig sind alle Fenster zunächst »versteckt«. Um das Fenster »aufzudecken«, folgt die nächste Programmzeile:

```
l->show();
```

Der nächste Befehl erklärt unser soeben erzeugtes Fenster zum Hauptfenster unserer Applikation:

```
app.setMainWidget (l);
```

Die meisten Programme bestehen aus einem Hauptfenster, und bei Bedarf werden weitere Fenster geöffnet. Indem wir unser *QLabel*-Fenster zum Hauptfenster machen, erklären wir dem Qt-System gleichzeitig, dass das Programm beendet werden kann, sobald das Fenster geschlossen wird.

Es bleibt noch die letzte Zeile in der *main*-Funktion:

```
return app.exec();
```

Obwohl es zunächst den Anschein hat, als würde hier nur der Rückgabewert der *main*-Funktion zurückgegeben, enthält diese Zeile sehr viel mehr. Ihr zentraler Punkt ist nämlich der Aufruf der Methode *exec* des *QApplication*-Objekts. Diese Methode enthält die so genannte Haupt-Event-Schleife, eine Endlosschleife, in der das Programm auf Ereignisse (*Events*) wartet, auf die es reagieren muss. Typische Events sind beispielsweise das Drücken einer Maustaste, die Eingabe über die Tastatur, das Verschieben eines Fensters oder die Änderung seiner Größe. Auch das Schließen eines Fensters erzeugt ein solches Ereignis.

Diese Endlosschleife wird erst dann verlassen, wenn in unserem Fall das Hauptfenster des Programms geschlossen wird. Damit wird die Methode *exec* beendet, und der Kontrollfluss kehrt zur *main*-Funktion zurück. Die Methode *exec* liefert dabei einen Integer-Wert als Rückgabe zurück, der als Ergebnis der *main*-Funktion benutzt werden kann: Wurde das Programm korrekt beendet, ist dieser Wert 0, in jedem anderen Fall wird ein Fehlerindex zurückgegeben.

Damit ist die *main*-Funktion und somit auch das Programm beendet, könnte man denken. Aber das stimmt nicht ganz: Wo werden die erzeugten Objekte wieder gelöscht?

Das *QApplication*-Objekt ist eine lokale Variable der *main*-Funktion. Am Ende dieser Funktion wird automatisch das Objekt gelöscht, natürlich nicht, ohne vorher seinen Destruktor aufzurufen. In diesem Destruktor werden alle noch vorhandenen Fenster-Objekte automatisch mit *delete* gelöscht, in unserem Fall also das mit *new* erzeugte *QLabel*-Objekt. Somit wurde auch alles wieder korrekt aufgeräumt.

Wenn Sie lieber nach dem Grundsatz programmieren, dass Sie alles selbst wieder löschen, was Sie dynamisch angelegt haben, so können Sie das natürlich machen. Statt der letzten Zeile der *main*-Funktion könnten Sie zum Beispiel

```
int result = app.exec();
delete l;
return result;
```

schreiben. Das Ergebnis ist das gleiche.

In den folgenden Übungsaufgaben werden leichte Veränderungen am bisherigen Hello-World-Programm vorgenommen, die Ihnen die Bestandteile des Programms noch einmal verdeutlichen und Sie mit den Strukturen von Qt vertraut machen sollen. Im folgenden Kapitel 2.3, *Das erste KDE-Programm*, werden wir dann ein Programm kennen lernen, das auch von KDE-Klassen Gebrauch macht und nahezu alle Elemente enthält, die ein KDE-Programm besitzen sollte. Dort werden wir auch etwas genauer auf die Funktionsweise von Qt und KDE eingehen.

2.2.5 Übungsaufgaben

Übung 2.1

Schreiben Sie das Programm *hello-qt.cpp* so um, dass es, anstatt »Hallo, Welt!« auszugeben, Sie persönlich mit Namen begrüßt. Falls Sie sich mit dem HTML-Format auskennen, können Sie auch einmal versuchen, kompiliziertere Ausgaben – zum Beispiel mehrere Absätze Text mit Zwischenüberschriften, Gliederungspunkten oder Tabellen – zu erzielen. Beachten Sie aber, dass nur eine Untermenge aller HTML-Tags verstanden wird.

Übung 2.2

Welche Größe hat das Fenster beim Start des Programms? Wie klein und wie groß können Sie das Fenster mit der Maus machen? Was passiert dabei?

Übung 2.3

Wie wird der Text angezeigt, wenn Sie gar keine Tags verwenden? Achten Sie auch darauf, ob der Text weiterhin umbrochen wird oder nicht.

Fügen Sie anschließend zwischen die Zeilen

```
QLabel *l = new QLabel ("Hallo, Welt!", 0);
```

und

```
l->show();
```

eine der beiden folgenden bzw. beide folgenden Zeilen ein:

```
l->setFont (QFont ("Arial", 48));  
l->setAlignment (Qt::AlignCenter);
```

Welchen Effekt hat jede der Zeilen auf die Darstellung des Fensters? Wie wichtig ist die Reihenfolge bzw. die Position dieser Zeilen im Programm? Wie wird der Text nun dargestellt, wenn Sie wieder Tags benutzen?

Übung 2.4

Was geschieht, wenn Sie die Zeile `app.setMainWidget (l);` aus dem Programm entfernen und erneut kompilieren? Keine Panik: In den Lösungen zu den Übungsaufgaben erfahren Sie, wie Sie nun das Programm dennoch beenden können.

Übung 2.5

Erweitern Sie das Programm so, dass es zwei Fenster mit je einem Text öffnet. Können Sie mehrere Fenster zum »Hauptfenster« erklären? Wann wird nun das Programm beendet?

Erweitern Sie das Programm auch auf zehn Fenster.

2.3 Das erste KDE-Programm

Im letzten Kapitel haben wir ein Programm erstellt, das ausschließlich die Klassen der Qt-Bibliothek benutzt hat. Ein echtes KDE-Programm sollte aber an einigen Stellen stattdessen Klassen aus der KDE-Bibliothek benutzen, die für das typische KDE-Aussehen und -Verhalten sorgen. Außerdem sollte ein KDE-Programm eine Reihe von Richtlinien einhalten, die eine einheitliche Benutzung aller KDE-Programme gewährleisten. Diese Richtlinien sind natürlich nur dann einzuhalten, wenn sie für Ihr konkretes Programm sinnvoll sind.

Wir wollen nun eine Variante des Hello-World-Programms analysieren. Auch in diesem Programm wird nur ein kurzer Begrüßungstext in einem eigenen Fenster ausgegeben. Dieses Mal erhält unser Programm aber zusätzlich ein Menü. Es soll

nahezu alle wichtigen Eigenschaften besitzen, die von einem KDE-Programm gefordert werden:

- Das Programm besitzt ein Hauptfenster mit einer Menüleiste, in der Befehle abgelegt sind. In unserem Fall besitzt die Menüleiste allerdings nur zwei Gruppen: ein FILE-Menü und ein HELP-Menü. Das FILE-Menü enthält nur den Befehl QUIT, während das HELP-Menü die Befehle CONTENTS, WHAT'S THIS, BUG REPORT, ABOUT KHELLOWORLD und ABOUT KDE enthält.
- Das Programm besitzt bereits die Möglichkeit, eine Online-Hilfe darzustellen. Auch diese wird in unserem Beispiel sehr kurz ausfallen, denn das Programm erklärt sich fast von allein.
- Das Programm ist bereits darauf vorbereitet, an verschiedene Landessprachen angepasst zu werden. So kann der Anwender neben Englisch (das sich als Standard durchgesetzt hat) auch Deutsch oder eine beliebige andere Sprache auswählen, in die dann alle dargestellten Texte und Menübefehle übersetzt werden.

2.3.1 Das Programm *KHelloWorld*

Hier sehen Sie zunächst den vollständigen Programmtext unseres Programms:

```
// KHelloWorld

// Beispielprogramm für das Buch
// KDE- und Qt-Programmierung
// (c) 2000 Addison-Wesley Germany

#include <action.h>
#include <kapp.h>
#include <kcmdlineargs.h>
#include <klocale.h>
#include <kmainwindow.h>
#include <kmenubar.h>
#include <kstdaction.h>
#include <qlabel.h>
#include <qpopupmenu.h>

int main (int argc, char **argv)
{
    QString aboutText ("KDE- und Qt-Programmierung\n"
                      "(c) 2000 Addison-Wesley Germany");
    KCmdLineArgs::init (argc, argv, "khelloworld",
                      aboutText, "1.0");

    KApplication app;
```

```

KMainWindow *top = new KMainWindow ();

QPopupMenu *filePopup = new QPopupMenu (top);
KAction *quitAction;
quitAction = KStdAction::quit (&app, SLOT (quit()));
quitAction->plug (filePopup);

top->menuBar()->insertItem (i18n ("%&File"), filePopup);
top->menuBar()->insertSeparator();
top->menuBar()->insertItem (i18n ("%&Help"),
                           top->helpMenu());

QLabel *text = new QLabel (
    i18n ("<H1>Hello, World!</H1>"), top);
top->setCentralWidget (text);
top->show();

return app.exec();
}

```

Geben Sie dieses Programm mit einem Editor ein, und speichern Sie es unter dem Namen *khelloworld.cpp* ab.

Zum Kompilieren müssen wir nun außerdem den Pfad für die KDE-Header-Dateien sowie die KDE-Bibliotheken *kdeui* und *kdecore* angeben. *kdeui* enthält alle grafischen Elemente von KDE, während *kdecore* alle anderen Elemente (wie zum Beispiel interne Datenstrukturen oder die *KApplication*-Klasse) enthält. Die Reihenfolge der Bibliotheken ist für einige Linker wichtig: Eine Bibliothek, die von einer anderen abhängt, muss *vor* dieser stehen. Da zum Beispiel *kdeui* von *kdecore* und von *qt* abhängt, muss sie vor diesen stehen. Die Befehlszeile zum Kompilieren sieht dann zum Beispiel so aus:

```

% gcc khelloworld.cpp -o khelloworld -I$KDEDIR/include
-I$QTDIR/include -lkdeui -lkdecore -lqt

```

Geben Sie den Befehl ohne Zeilenwechsel ein. Wenn der Compiler eine der Header-Dateien nicht findet, so ist wahrscheinlich eine der Umgebungsvariablen *\$KDEDIR* oder *\$QTDIR* nicht richtig gesetzt. Wenn der Linker eine der angegebenen Bibliotheken nicht finden konnte, geben Sie die Verzeichnisse zusätzlich mit der Option *-L* an, in unserem Fall also mit den beiden zusätzlichen Optionen *-L\$KDEDIR/lib* *-L\$QTDIR/lib*.

Wenn der Compiler das Programm fehlerfrei übersetzt hat, hat er eine ausführbare Datei *khelloworld* erzeugt. Wenn Sie diese starten, erscheint auf dem Bildschirm ein Fenster wie in Abbildung 2.3.



Abbildung 2-3 Das Programm khelloworld

Das Fenster hat oben eine Menüzeile, in der zwei Befehle aufgeführt sind: FILE und HELP. Wenn Sie auf einen der Befehle klicken (oder die Tastenkombination `[Alt]+[F]` bzw. `[Alt]+[H]` drücken), geht ein Untermenü auf. Das Untermenü zum Befehl FILE umfasst nur den Befehl QUIT. Wenn Sie ihn auswählen (indem Sie ihn mit der Maus anklicken oder die Tastenkombination `[Alt]+[Q]` benutzen), wird das Programm beendet. Sie können das Programm übrigens auch jederzeit mit der Tastenkombination `[Strg]+[Q]` beenden, ohne dass das FILE-Menü geöffnet ist. Das Untermenü zu HELP enthält die Befehle CONTENTS..., WHAT'S THIS, REPORT BUG..., ABOUT KHELLOWORLD... und ABOUT KDE...

Wenn Sie den Befehl CONTENTS... wählen, erhalten Sie zur Zeit noch die allgemeine KDE-Hilfe. Wie man eine eigene Online-Hilfe einbindet, werden wir in Kapitel 2.3.4, *Die Online-Hilfe*, näher erläutern. Der Befehl WHAT'S THIS bewirkt zunächst nur, dass sich der Mauscursor ändert und nun ein Fragezeichen enthält. Mit diesem Cursor können Sie auf Elemente des Fensters klicken, um nähere Informationen zu erhalten. In unserem Fall bewirkt das aber noch nichts, da keines der Elemente mit zusätzlichen Informationen ausgestattet ist. Nähere Informationen hierzu erhalten Sie in Kapitel 4.11.2, *What's-This-Hilfe*.

Der Befehl BUG REPORT... öffnet ein neues Dialogfenster, mit dem eine E-Mail an den Entwickler des Programms (hier also uns selbst) geschickt werden kann.

Die Befehle ABOUT KHELLOWORLD... und ABOUT KDE... öffnen Informationsfenster, die in den Abbildungen 2.4 und 2.5 zu sehen sind.

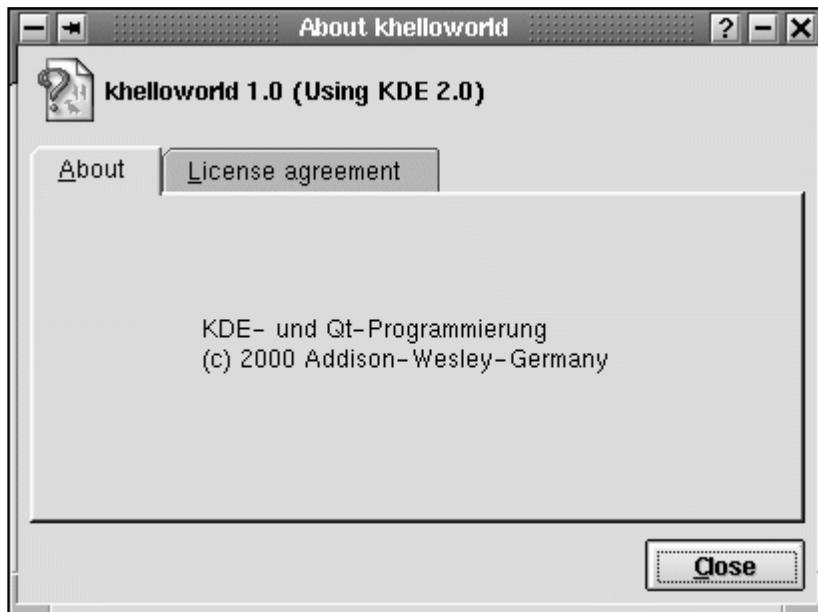


Abbildung 2-4 Das ABOUT KHELLOWORLD...-Fenster

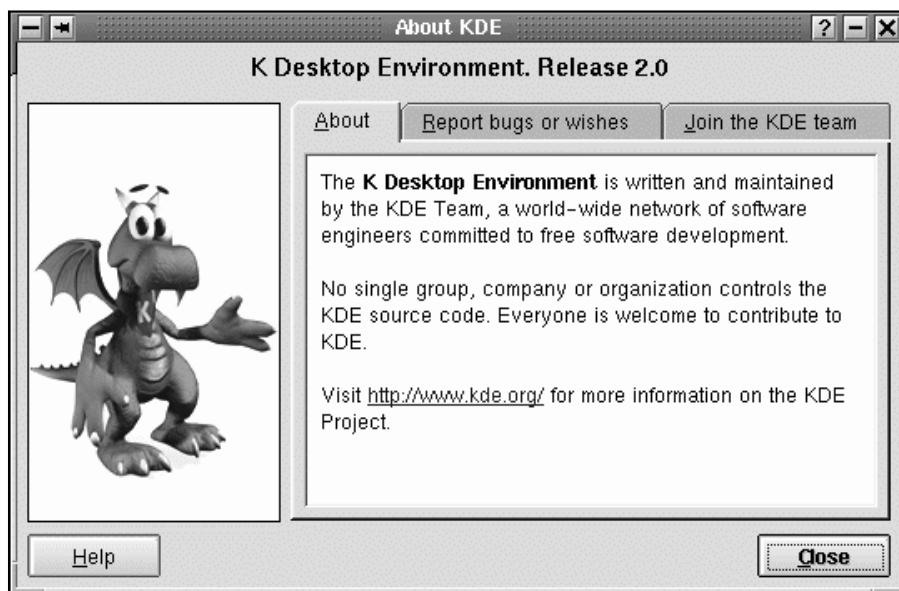


Abbildung 2-5 Das ABOUT KDE...-Fenster

Gehen wir das Listing des Programms nun schrittweise durch.

Die ersten Zeilen sind reine Kommentarzeilen. Jedes KDE-Programm sollte mindestens ein paar Zeilen über den Autor und den Zweck der Datei enthalten.

```
// KHelloWorld

// Beispielprogramm für das Buch
// KDE- und Qt-Programmierung
// (c) 2000 Addison-Wesley Germany
```

Die nächsten Zeilen binden die Klassendefinitionen für eine ganze Reihe von benötigten Klassen ein. Welche Klassen oder Funktionen das jeweils sind, steht hier als Kommentar hinter der Zeile:

```
#include <kaction.h>      // Klasse KAction
#include <kapp.h>          // Klasse KApplication
#include <kcmlineargs.h>  // Klasse KCmdLineArgs
#include <klocale.h>      // Funktion i18n
#include <kmainwindow.h>  // Klasse KMainWindow
#include <kmenubar.h>      // Klasse KMenuBar
#include <kstdaction.h>    // Klasse KStdAction
#include <qlabel.h>        // Klasse QLabel
#include <qpopupmenu.h>    // Klasse QPopupMenu
```

Wie man bereits am Namen erkennt, stammen die ersten sieben Header-Dateien aus der KDE-Bibliothek und die letzten beiden aus der Qt-Bibliothek. Welchen Zweck die einzelnen Klassen haben, schauen wir uns genauer an den Stellen an, an denen sie benutzt werden.

Anschließend beginnt das Hauptprogramm, die Funktion *main*, die beim Start des Programms aufgerufen wird. In den Parametern *argc* und *argv* bekommt die Funktion Anzahl und Text der Kommandozeilenparameter mitgeliefert.

```
int main (int argc, char **argv)
{
```

Als Erstes legen wir in einer Variablen vom Typ *QString* – eine Klasse von Qt, die Textstrings speichern kann – den Text ab, der beim Aufruf des Menüpunkts ABOUT KHELLOWORLD... ausgegeben werden soll.

```
QString aboutText ("KDE- und Qt-Programmierung\n"
                  "(c) 2000 Addison-Wesley Germany");
```

Als Nächstes benutzen wir die Klasse *KCmdLineArgs* der KDE-Bibliothek. Diese Klasse dient dazu, die Kommandozeilenparameter auszuwerten. Außerdem speichert sie Informationen über das Programm. Wir benutzen hier die statische Methode *init*. Ihr übergeben wir zunächst mit *argc* und *argv* die Kommandozei-

lenparameter. Als dritten Parameter geben wir den Namen des Programms an. (Dieser Name ist in der Regel identisch zum Dateinamen der ausführbaren Datei. Anhand dieses Namens werden weitere Dateien gesucht, die für unser Programm bestimmt sind, z. B. Icons, Übersetzungsdateien, Hilfedateien usw. Damit diese auch dann gefunden werden, wenn die ausführbare Datei umbenannt wird, legt man hier den Namen noch einmal explizit fest.) Der vierte und fünfte Parameter sind der Text für das Fenster in ABOUT KHELLOWORLD... sowie die Versionsnummer des Programms.

```
KCmdLineArgs::init (argc, argv, "khelloworld",  
                    aboutText, "1.0");
```

Welche Kommandozeilenparameter unser Programm zur Zeit versteht, können Sie ermitteln, wenn Sie folgenden Aufruf durchführen:

```
% ./khelloworld --help
```

Die Ausgabe, die dabei erscheint, wird ebenfalls von *KCmdLineArgs::init* erzeugt. Findet diese Methode nämlich den Parameter *--help*, so gibt sie den Hilfetext aus und beendet anschließend das Programm.

Ähnlich wie bei unserem ersten Qt-Programm erzeugen wir auch hier ein zentrales Objekt, dieses Mal aber nicht von der Klasse *QApplication*, sondern von *KApplication*. *KApplication* ist eine Unterklasse von *QApplication*, arbeitet also intern genauso, implementiert aber noch eine ganze Reihe von zusätzlichen Methoden. Da wir die Kommandozeilenparameter bereits analysieren ließen, brauchen wir sie hier nun nicht mehr im Konstruktor anzugeben. Die Parameter, die *KApplication* braucht, fragt es bei der Klasse *KCmdLineArgs* ab. Neben dem Verbindungsaufbau mit dem X-Server lädt der Konstruktor automatisch die Konfigurations- und Übersetzungsdateien für dieses Programm und setzt Einstellungen für Farben und Zeichensätze, die der Anwender zentral im *KDE Control Center* vorgenommen hat. Jedes KDE-Programm sollte unbedingt die Klasse *KApplication* und nicht die Klasse *QApplication* nutzen, da nur so ein einheitliches Aussehen und Verhalten aller KDE-Programme erreicht werden kann. (Beachten Sie, dass Sie hinter dem Konstruktor ohne Parameter hier kein Klammernpaar angeben dürfen, da diese Zeile dann eine ganz andere Bedeutung hätte. Beachten Sie außerdem, dass die Header-Datei für die Klasse *KApplication* inkonsequenterweise *kapp.h* heißt, nicht – wie man es erwarten sollte – *kapplication.h*.) In Kapitel 3.3, *Grundstruktur einer Applikation*, sind die Klassen *KApplication* und *QApplication* genauer beschrieben.

```
KApplication app;
```

Als Nächstes erzeugen wir das Fensterobjekt, das auf dem Bildschirm angezeigt werden soll. Es ist ein Objekt der Klasse *KMainWindow*. Wir erzeugen das Objekt dynamisch auf dem Heap und speichern die Adresse für spätere Zugriffe in der Zeigervariablen *top*.

```
KMainWindow *top = new KMainWindow ();
```

Dieses Objekt stellt später das gesamte Programmfenster auf dem Bildschirm dar (zu diesem Zeitpunkt ist das Fenster noch versteckt). Dabei verwaltet es die Menüzeile (die zur Zeit noch keine Einträge enthält) und kann auch eine oder mehrere Werkzeugleisten oder eine Statuszeile verwalten.

Die nächsten Zeilen dienen dazu, die Menüzeile zu erzeugen und mit Befehlen zu füllen. Als Erstes erzeugen wir dazu das Menüfenster, das sich öffnet, wenn man den Befehl `FILE` auswählt. Dabei handelt es sich um ein so genanntes Popup-Menü, also ein Menü, das auf Knopfdruck aufspringt. Erzeugt wird dieses Menü mit einem Objekt der Klasse *QPopupMenu*, das wir ebenfalls mit *new* auf dem Heap anlegen. Im Konstruktor von *QPopupMenu* geben wir unser Programmfenster *top* als so genanntes Vaterobjekt an. Dadurch wird das Objekt automatisch mit *delete* gelöscht, wenn das Fenster freigegeben wird.

```
QPopupMenu *filePopup = new QPopupMenu (top);
```

Für jeden Eintrag, den man in das Menü einträgt, erzeugt man in der Regel ein Aktionsobjekt. Dieses Objekt übernimmt dann die Ausführung des Befehls. Wir legen dazu zunächst einmal eine Zeigervariable namens *quitAction* an, die die Aktion speichern soll, die dem Menüpunkt `QUIT` zugeordnet wird. Die Klasse für die Aktionsobjekte heißt *KAction*.

```
KAction *quitAction;
```

Da nahezu jedes KDE-Programm den Menüpunkt `QUIT` benötigt, gibt es in der KDE-Bibliothek eine Klasse *KStdAction*, die alle wichtigen Standardaktionen erzeugen kann. Zu einer Standardaktion gehört der Name im Menü (inklusive der Kennzeichnung des Buchstabens, der unterstrichen werden soll), der Kurzbefehl (in diesem Fall `[Strg]+[Q]`) sowie ein spezielles Icon (hier ein Kreis mit einem senkrechten Strich – die internationale Kennzeichnung für einen Ein/Aus-Schalter). Um das Aktionsobjekt erzeugen zu lassen, rufen wir hier einfach die statische Methode *quit* der Klasse *KStdAction* auf. Sie liefert die Adresse des Aktionsobjekts zurück, die wir hier in unserer Variablen *quitAction* speichern. Als Parameter müssen wir der Methode *quit* mitteilen, welche Slot-Methode von welchem Objekt aufgerufen werden soll, wenn diese Aktion aufgerufen wird. In unserem Fall ist es die Methode *quit* unseres *KApplication*-Objekts.

```
quitAction = KStdAction::quit (&app, SLOT (quit()));
```

Nun fügen wir die Aktion noch in das Popup-Menü für den Menüpunkt `FILE` ein:

```
quitAction->plug (filePopup);
```

Wir haben bisher zwar schon das Popup-Menü für FILE erstellt, es aber noch nicht in die Menüleiste eingebaut. Das passiert mit der nächsten Zeile:

```
top->menuBar()->insertItem (i18n ("%File"), filePopup);
```

Wir rufen dazu zunächst einmal die Methode *menuBar* von unserem Hauptfenster *top* auf. Diese Methode liefert einen Zeiger auf das Menüleistenobjekt (Klasse *KMenuBar*) zurück. Von diesem Objekt rufen wir dann die Methode *insertItem* auf. Mit dieser Methode fügen wir ein Untermenü ein. Der erste Parameter ist dabei die Bezeichnung des Untermenüs. Es trägt den Namen FILE. Das Kaufmanns-Und (&) legt dabei fest, dass der darauf folgende Buchstabe (also das **F**) zusammen mit der Taste **Alt** diesen Menüpunkt aktiviert. (Die Funktion *i18n*, die den String umschließt, hat hier noch keine besondere Bedeutung. Sie übersetzt den eingeschlossenen Text, falls eine andere Sprache als Englisch eingestellt ist.) Der zweite Parameter für *insertItem* ist ein Zeiger auf das Popup-Menü für diesen Menüpunkt.

In die Menüleiste fügen wir als Nächstes noch mit *insertSeparator* einen Abstandshalter ein. Ist als Stil *Windows* eingestellt, so ist der Abstandshalter wirkungslos (siehe Abbildung 2.3). Im *Motif*-Stil dagegen bewirkt der Abstandshalter, dass alle Menüpunkte davor (also FILE) linksbündig und alle danach (also HELP) rechtsbündig innerhalb der Leiste stehen.

```
top->menuBar()->insertSeparator();
```

Als zweiten Punkt fügen wir das Hilfe-Menü in die Menüleiste ein. Der Menüeintrag lautet dabei HELP (mit **Alt+H** als aktivierendem Tastaturbefehl). Das Popup-Menü lassen wir hier wieder automatisch erzeugen: Unsere Hauptfensterklasse *KMainWindow* enthält dafür die Methode *helpMenu*, die ein *QPopupMenu*-Objekt erzeugt und einen Zeiger darauf zurückliefert. Dieses Menü enthält automatisch die oben beschriebenen fünf Einträge.

```
top->menuBar()->insertItem (i18n ("%Help"),  
                           top->helpMenu());
```

Nachdem unser Menü jetzt vollständig aufgebaut ist, können wir uns um den eigentlichen Anzeigebereich unseres Programms kümmern. Schließlich soll unser Programm ja auch etwas zeigen. Wir erzeugen dazu ein Fenster-Objekt der Klasse *QLabel*, ganz analog wie in unserem ersten Qt-Programm. In diesem Fall geben wir im Konstruktor als zweiten Parameter aber nicht den Wert 0 an, sondern unser Hauptfenster *top*. Dadurch erreichen wir, dass das Textobjekt nicht ein eigenes Fenster erzeugt, sondern als Unterfenster in unserem Hauptfenster erscheint.

```
QLabel *text = new QLabel (  
    i18n ("<H1>Hello, World!</H1>"), top);
```

Wir müssen dem Hauptfenster nun noch mitteilen, dass das Textobjekt das eigentliche Anzeigefenster darstellt. Es belegt damit automatisch den gesamten Platz unterhalb der Menüleiste.

```
top->setCentralWidget (text);
```

Noch immer wird nichts auf dem Bildschirm angezeigt, denn unser Hauptfenster ist noch versteckt. Wir müssen die Methode *show* aufrufen, um es sichtbar zu machen. Das enthaltene *QLabel*-Objekt wird automatisch ebenfalls sichtbar. Für dieses brauchen wir also nicht mehr *show* aufzurufen.

```
top->show();
```

Genau wie bei unserem ersten Qt-Programm starten wir nun die so genannte Haupt-Event-Schleife, indem wir die Methode *exec* von unserem *KApplication*-Objekt aufrufen. (*KApplication* ist eine Unterklasse der Klasse *QApplication* und erbt daher auch die Methode *exec*.)

```
return app.exec();
```

Das Programm wartet von nun an auf Tastatureingaben und Mausektionen vom Benutzer. Diese Schleife wird erst dann beendet, wenn die Slot-Methode *quit* unseres *KApplication*-Objekts aktiviert wurde. Und das passiert entweder dadurch, dass die Aktion *quitAction* aktiviert wird (durch Auswahl des Menüpunkts FILE-QUIT oder durch Strg+Q), oder dadurch, dass das Hauptfenster geschlossen wird (mit dem X-Button in der Titelleiste oder über das Fenstermenü).

Nach Beendigung der Schleife kehrt die Methode *exec* zum Aufrufer zurück und hat eine Zahl als Rückgabewert, die anzeigt, ob das Programm normal (Wert 0) oder aufgrund eines Fehlers (Wert ungleich 0) beendet wurde. Diesen Wert benutzen wir wieder als Rückgabewert der *main*-Funktion.

Damit ist die *main*-Funktion beendet. Die Variable *app* wird wieder freigegeben, was bewirkt, dass der Destruktor von *KApplication* aufgerufen wird. In diesem wird die Verbindung zum X-Server abgebaut und alle geöffneten Dateien werden geschlossen. Auch alle noch offenen Fenster auf dem Bildschirm werden gelöscht (also auch unser Hauptfensterobjekt *top* und als Folge davon das Popup-Menü *filePopup*).

Das Programm *KHelloWorld* ist nun zwar kompiliert und lässt sich starten, es sind jedoch noch einige Punkte zu ergänzen. *KHelloWorld* sollte sich im Verzeichnis der anderen ausführbaren KDE-Programme befinden, und es sollte direkt über die KDE-Menüleiste *Kicker* aufgerufen werden können. Wie Sie das erreichen, erfahren Sie im nächsten Abschnitt 2.3.2, *Einbinden in die KDE-Verzeichnisstruktur*. In den beiden darauf folgenden Abschnitten 2.3.3, *Landessprache: Deutsch*, und 2.3.4, *Die Online-Hilfe*, werden wir unser *KHelloWorld*-Programm mehrsprachig machen und mit einer Online-Hilfe versehen.

2.3.2 Einbinden in die KDE-Verzeichnisstruktur

Unser Programm *KHelloWorld* sollte sich im Suchpfad befinden, damit es aus jedem beliebigen Verzeichnis heraus gestartet werden kann. Wenn Sie das gesamte KDE-System auf Ihrem Rechner installiert haben, befinden sich normalerweise alle KDE-Applikationen im Verzeichnis *\$KDEDIR/bin*. Dorthin sollten Sie auch die Datei *khelloworld* kopieren, die der Compiler erzeugt hat. Je nachdem, wie das KDE-System auf Ihrem Rechner installiert ist, brauchen Sie eventuell Superuser-Rechte, um die Datei in das KDE-Verzeichnis zu kopieren. Falls Sie keine Superuser-Rechte erhalten können, müssen Sie die Datei in einem anderen Verzeichnis speichern und die Pfadvariable *\$PATH* gegebenenfalls anpassen.

Unser kleines Programm hält sich an die KDE-Namenskonvention, nach der KDE-Programme mit dem Buchstaben »k« anfangen sollten. Im Applikationsnamen – also so, wie das Programm in der Online-Hilfe genannt wird – werden dabei jeder Anfangsbuchstabe eines neuen Wortes sowie das »K« am Anfang großgeschrieben (*KHelloWorld*). Die ausführbare Datei selbst enthält dagegen nur Kleinbuchstaben (*khelloworld*).

Wenn KDE auf Ihrem System läuft, so werden Programme meist über das Startmenü in der Menüzeile (*Kicker*) am unteren Bildschirmrand gestartet. Um nun auch *KHelloWorld* in das Startmenü mit aufzunehmen, müssen Sie eine kleine Textdatei mit Informationen über das Programm erzeugen und in einem der Unterverzeichnisse im Verzeichnis *\$KDEDIR/share/applnk/* abspeichern. Je nachdem, in welchem Unterverzeichnis Sie die Datei ablegen, erscheint *KHelloWorld* in der entsprechenden Gruppe im Startmenü. Wir wollen, dass *KHelloWorld* in der Gruppe ANWENDUNGEN aufgeführt wird, und kopieren daher die Informationsdatei in das Verzeichnis *\$KDEDIR/share/applnk/Applications/*. Auch dazu sind eventuell Superuser-Rechte nötig. Falls Sie keine Superuser-Rechte erlangen können, können Sie diese Datei auch in das Verzeichnis *\$HOME/.kde/share/applnk/* oder eines der Unterverzeichnisse kopieren. In diesem Fall ist es dann allerdings nur bei Ihrem Login im Startmenü enthalten.

Die Informationsdatei für unser Programm hat folgende Form:

```
[Desktop Entry]
Type=Application
Exec=khelloworld
Name=Hello, World!
Name[de]=Hallo, Welt!
Comment=A simple Hello World example
Comment[de]=Ein einfaches Hallo-Welt-Beispiel
```

Erzeugen Sie also diese Datei mit einem beliebigen Texteditor, und speichern Sie sie unter dem Namen *\$KDEDIR/share/applnk/Applications/khelloworld.desktop* ab. Unser Programm erscheint dadurch automatisch im Menü im Unterpunkt

ANWENDUNGEN. Je nach eingestellter Landessprache erscheint dabei die englische oder die deutsche Beschreibung aus dem Punkt *Name* im Menü. (In den ersten Versionen von KDE 2.0 wird das Menü aufgrund eines Bugs nicht regelmäßig aktualisiert. Es kann daher passieren, dass unser Programm zunächst nicht im Menü erscheint. In der Regel hilft ein Ausloggen und Wiedereinloggen.)

Wenn Sie den Eintrag mit der Maus aus dem Menü auf die Kontrollleiste am unteren Bildschirmrand ziehen, können Sie unser Programm für den besonders schnellen Aufruf dort platzieren. Wenn Sie mit der Maus für eine kurze Zeit auf dem Eintrag verweilen, wird der Text aus *Comment* angezeigt, wieder je nach Landessprache auf englisch oder deutsch.

Unter *Exec* wird der Dateiname der ausführbaren Datei angegeben. Falls diese Datei nicht im Suchpfad ist, kann die genaue Pfadangabe hier ergänzt werden.

Statt die Datei von Hand einzugeben, können Sie sie auch vom Programm *KMenuEdit* erstellen lassen. Sie finden es im Startmenü unter PANEL MENU-CONFIGURE-MENU EDITOR. Wählen Sie in diesem Programm den Befehl NEW ITEM und geben Sie die entsprechenden Informationen in die Felder ein. Sie können mit diesem Programm aber nur die Bezeichnungen für die gerade eingestellte Landessprache festlegen.

2.3.3 Landessprache: Deutsch

Eine wichtige Forderung an alle KDE-Applikationen ist die Mehrsprachigkeit. Die Einarbeitungszeit ist viel kürzer, wenn der Anwender ein Programm nicht nur in englischer Sprache bedienen kann, sondern alle Bedienelemente auch in seiner Muttersprache angezeigt werden können. Die zu verwendende Sprache wird dabei im KDE-System zentral im Einstellungsmenü ausgewählt.

Der Programmierer muss sein Programm auf diese Möglichkeit vorbereiten. Jeder Text, der auf dem Bildschirm angezeigt werden soll, muss dabei der Funktion *i18n* übergeben werden, die ihn anhand einer Umsetzungstabelle in eine andere Sprache übersetzt. Dies haben wir in *khelloworld.cpp* bereits erledigt. Nun müssen wir aber auch noch eine Umsetzungstabelle erstellen, in der zu jedem englischen Text der übersetzte deutsche Text aufgeführt ist.

Mit dem folgenden Befehl extrahieren wir alle zu übersetzenden Texte aus dem Quellcode und speichern sie in einer Datei ab:

```
% xgettext -C -ki18n -x $KDEDIR/include/kde.pot  
khelloworld.cpp
```

Dieser Befehl muss in einer einzelnen Kommandozeile stehen. Das Programm *xgettext* filtert aus der Datei *khelloworld.cpp* alle konstanten String-Ausdrücke heraus, die hinter der Zeichenfolge *i18n* stehen, und speichert sie in der Datei

messages.po ab. Eine ganze Reihe von Ausdrücken, die in vielen Programmen benutzt werden, ist bereits in einer KDE-Umsetzungstabelle gespeichert und braucht nicht mehr übersetzt zu werden. Diese Ausdrücke sind in der Datei *\$KDEDIR/include/kde.pot* enthalten. Durch die Angabe `-x $KDEDIR/include/kde.pot` werden sie beim Erzeugen der Datei *messages.po* nicht berücksichtigt. (In unserem Fall sind es die Menütitel *&File* und *&Help*, die bereits enthalten sind.)

Auf einigen Systemen ist das Programm *xgettext* nicht installiert. Wenn das Programm *gettext* installiert ist, können Sie versuchen, mit diesem zu arbeiten. Falls das nicht zum Erfolg führt – *xgettext* und *gettext* sind nicht völlig kompatibel –, müssen Sie sich das *xgettext*-Programm von einer Linux-Distribution oder aus dem Internet besorgen und es installieren. Es befindet sich meist in einem Paket der GNU-Utilities. Vorsicht ist auch bei der Manual-Page für *xgettext* geboten. Auf vielen Systemen ist diese hoffnungslos veraltet. Um die Optionen von *xgettext* zu erhalten, benutzen Sie stattdessen besser *xgettext --help*.

Die von *xgettext* erzeugte Datei *messages.po* hat in unserem Beispiel folgenden Inhalt:

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR Free Software Foundation, Inc.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
"Project-Id-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2000-11-17 11:57+0100\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: ENCODING\n"

#: khelloworld.cpp:36
msgid "<H1>Hello, World!</H1>"
msgstr ""
```

Die Kommentarzeilen beginnen in dieser Datei mit dem »#«-Zeichen. Der Anfang der Datei sollte vom Programmierer mit Informationen gefüllt werden. Er dient hauptsächlich dazu, die wichtigsten Versionsinformationen zu protokollieren, wenn mehrere Übersetzer an einer Datei arbeiten. Die zu übersetzenden Ausdrücke stehen am Ende der Datei. Das ist in unserem Fall nur der Text *<H1>Hello, World!</H1>*, der angezeigt werden soll.

Übersetzen wir nun den Ausdruck ins Deutsche, und speichern wir die Datei anschließend wieder ab. Der Originaltext steht dabei jeweils hinter *msgid*, die Übersetzung hinter *msgstr*. (Hier ist nur die unteren drei Zeilen der Datei abgedruckt. Der obere Teil bleibt unverändert, bzw. dort können Sie die Informationen über Autor, Übersetzer und Datum eintragen.)

```
#: khelloworld.cpp:36
msgid "<H1>Hello, World!</H1>"
msgstr "<H1>Hallo, Welt!</H1>"
```

Diese Übersetzungstabelle muss nun noch in eine Form gebracht werden, in der sie leichter vom Computer eingelesen werden kann. Dazu benutzen wir das Programm *msgfmt*:

```
% msgfmt messages.po -o khelloworld.mo
```

Auf einigen Systemen heißt das Programm *gmsgfmt*. Falls beide Programme installiert sein sollten, versuchen Sie beide Möglichkeiten.

Dieser Aufruf von *msgfmt* erzeugt die Datei *khelloworld.mo*, die nun die Übersetzungsinformationen enthält. Diese Datei ist mit einem normalen Texteditor nicht mehr lesbar.

Die Datei *khelloworld.mo* wird nun in das Verzeichnis *\$KDEDIR/share/locale/de/LC_MESSAGES/* kopiert. Wenn anschließend *KHelloWorld* gestartet wird und die Landessprache Deutsch aktiviert ist, wird diese Umsetzungstabelle automatisch erkannt und benutzt. Ebenso können Sie natürlich auch für andere Sprachen Umsetzungstabellen erstellen und in das entsprechende Verzeichnis kopieren.

Die Bildschirmausgabe des Programms *KHelloWorld* auf Deutsch sehen Sie in Abbildung 2.6. Auch alle Menübefehle sind nun auf Deutsch, ebenso wie die Fenster für die Menüpunkte ÜBER KHELLOWORLD... und ÜBER KDE...



Abbildung 2-6 Das Hauptfenster in deutscher Sprache

Sie werden auf Probleme stoßen, wenn Sie versuchen, deutsche Umlaute in den übersetzten Texten zu verwenden. Diese werden nicht korrekt dargestellt. Der Grund dafür liegt darin, dass KDE die übersetzten Texte im Zeichensatzformat UTF-8 erwartet. In diesem Format werden die ASCII-Zeichen im Bereich 0 bis 127 normal dargestellt, alle anderen Zeichen des Unicode-Standards (und dazu gehören auch die deutschen Umlaute) werden dagegen als Kombinationen von Zeichen im Bereich 128 bis 255 dargestellt. Sie müssten die Übersetzungen daher mit einem Editor erstellen, der ebenfalls UTF-8-Dateien erzeugen kann. Das können aber die meisten Editoren bisher nicht.

Benutzen Sie stattdessen am besten das Programm *KBabel*, das speziell für die Übersetzung entwickelt wurde. (Es ist im Paket *kdesdk* enthalten, das sich auch auf der CD zum Buch befindet. Die aktuellste Version finden Sie beispielsweise auf dem KDE-FTP-Server. Dieses Programm ist meist nicht im normalen Distributionsverzeichnis enthalten, sondern nur in den aktuellen Snapshots, die Sie unter <ftp://ftp.kde.org/pub/kde/unstable/CVS/snapshots/current/> finden können.) Beachten Sie, dass Sie auch in diesem Programm vorher das Ausgabeformat UTF-8 wählen, unter EINSTELLUNGEN – PERSÖNLICHE EINSTELLUNGEN – SPEICHERN.

2.3.4 Die Online-Hilfe

Jedes KDE-Programm sollte eine Online-Hilfe bieten, mit der sich der Anwender einen genauen Überblick über die Fähigkeiten des Programms verschaffen kann. KDE bietet dabei bereits die Möglichkeit, Hilfe-Dateien im HTML-Format darzustellen. Im automatisch erzeugten HELP-Menü wird bereits im Menüpunkt CONTENTS das Programm *kdehelp* aufgerufen. *kdehelp* sucht dabei für unser Programm *KHelloWorld* nach der Datei `$KDEDIR/share/doc/HTML/<Sprache>/<Applikationsname>/index.html`. Wir wollen nun eine kurze HTML-Datei als Online-Hilfe erzeugen.

Wir erstellen also mit einem Texteditor folgende Datei und speichern sie unter `$KDEDIR/share/doc/HTML/de/khelloworld/index.html` ab. (Das Unterverzeichnis *khelloworld* muss dazu zunächst angelegt werden.)

```
<HTML>
<TITLE>
KHelloWorld
</TITLE>

<BODY>
<H1>KHelloWorld</H1>
KHelloWorld ist ein kleines Beispielprogramm aus dem Buch
KDE- und Qt-Programmierung vom Addison-Wesley-Verlag.<P>
Es hat:
<UL>
<LI>eine Menüzeile
```

```
<LI>eine englische und eine deutsche Übersetzung  
<LI>eine Online-Hilfe (diese hier)  
<LI>einen Eintrag im Kicker-Menü  
</UL>  
</BODY>  
</HTML>
```

Wenn als Landessprache Deutsch gewählt ist, wird nun beim Aufruf des Befehls INHALT ein Hilfefenster geöffnet (siehe Abbildung 2.7).



Abbildung 2-7 Das deutschsprachige Hilfefenster

Ebenso können Sie natürlich Online-Hilfe-Dokumente für andere Sprachen erstellen und im zugehörigen Verzeichnis ablegen. Sie können auch mehrere HTML-Dateien zu einer Sprache erstellen und Links zwischen den Dateien erzeugen. Legen Sie dazu alle Dateien zu einer Sprache in dem entsprechenden Verzeichnis ab. Die Datei *index.html* kann dann zum Beispiel ein Inhaltsverzeichnis enthalten, in dem Links zu den anderen Dateien führen.

Eine Einführung in die Sprache HTML kann hier aus Platzgründen nicht gegeben werden. Es gibt jedoch viele Dokumentationen zu HTML im Internet und viele Editoren, die direkt HTML-Code erzeugen können.

2.4 Was fehlt noch zu einer KDE-Applikation?

Unser Beispielprogramm aus Kapitel 2.2, *Das erste Qt-Programm*, war ein reines Qt-Programm. *KHelloWorld* aus Kapitel 2.3, *Das erste KDE-Programm*, ist dagegen eine vollständige KDE-Applikation. Es fehlt noch ein Icon – also ein kleines Bild –, das das Programm symbolisiert, zum Beispiel im Startmenü. Außerdem sollte das Programm zu einem Paket gepackt werden, mit dem auch Anwender mit nur geringen Unix-Kenntnissen die Installation vornehmen können.

Die meisten Programme haben natürlich eine aufwendigere grafische Oberfläche und eine viel größere Funktionalität. Wie man die vordefinierten Einstellungselemente benutzt, sie anordnet und miteinander verbindet, wird ausführlich im Kapitel 3, *Grundkonzepte der Programmierung in KDE und Qt*, beschrieben. Dort wird auch das Signal/Slot-Konzept beschrieben, das wir bereits benutzt haben.

Um eigene grafische Elemente zu entwickeln, die dem Anwender mehr Möglichkeiten als die vordefinierten anbieten, muss man sich in die tieferen Schichten von Qt einarbeiten, die das Zeichnen auf den Bildschirm und die direkte Verarbeitung von Ereignissen von Maus und Tastatur ermöglichen. Diese Methoden werden in Kapitel 4.2, *Zeichnen von Grafikprimitiven*, und Kapitel 4.4, *Entwurf eigener Widget-Klassen*, beschrieben. Kapitel 4 enthält weitere spezielle Lösungsvorschläge für Probleme, die bei der Entwicklung von Programmen immer wieder entstehen.

Einige Hilfsprogramme, die den Entwickler eines KDE-Programms unterstützen können, werden in Kapitel 5, *Hilfsmittel für die Programmerstellung*, beschrieben. Insbesondere bei komplexen, umfangreichen Applikationen können diese Hilfsprogramme die Entwicklungszeit enorm verkürzen.

3 Grundkonzepte der Programmierung in KDE und Qt

In diesem Kapitel werden wir systematisch und detailliert alle wichtigen Techniken sowie die Elemente, die zur Erstellung einer Applikation erforderlich sind, besprechen.

Wir beginnen dabei mit zwei Teilkapiteln, die eher theoretischer Natur sind. Dennoch sollten sie genau studiert werden, da die dort beschriebenen Techniken grundlegend für alle selbst geschriebenen Programme sind. Kapitel 3.1, *Die Basis-klasse – QObject*, führt in die zentrale Qt-Klasse ein, von der fast alle anderen Klassen abgeleitet sind. Die Klasse, die für die Darstellung eines Fensters benutzt wird, wird in Kapitel 3.2, *Die Fensterklasse – QWidget*, besprochen.

In Kapitel 3.3, *Grundstruktur einer Applikation*, werden wir den Aufbau eines vollständigen Programms betrachten. Darunter fällt insbesondere die Struktur des Hauptprogramms. Die KDE-Bibliotheken bieten dabei noch einige Möglichkeiten, die Kommandozeilenparameter zu analysieren oder zu garantieren, dass ein Programm nicht mehrmals gestartet wurde.

Kapitel 3.4, *Hintergrund: Event-Verarbeitung*, ist wieder theoretischer. Es gibt einen Überblick über die Vorgänge innerhalb von Qt, die bei der Interaktion zwischen Anwender und Applikation ablaufen.

In Kapitel 3.5, *Das Hauptfenster*, wird es dann richtig praktisch: Anhand eines kleinen Texteditors wird beschrieben, wie man ein Programm mit Menü- und Werkzeugleiste ausstattet und wie man die wichtigsten Menüpunkte implementiert. Das dort entwickelte Programm kann als Grundlage für fast jedes größere Projekt benutzt werden.

Für individuelle Programme muss der Programmierer meist eigene Fenster (Dialoge) entwerfen, in denen er GUI-Elemente (Buttons, Auswahlboxen und Eingabezeilen) zusammenstellt. Wie man deren Platzierung organisieren kann, wird in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*, erläutert.

Die wichtigsten bereits vorhandenen Elemente der Bibliotheken werden in Kapitel 3.7, *Überblick über die GUI-Elemente von Qt und KDE*, vorgestellt. Die meisten Programme kommen mit diesem Satz an Elementen bereits aus.

Hinweise und Tipps für den Entwurf eines eigenen Fensters werden schließlich in Kapitel 3.8, *Der Dialogentwurf*, gegeben.

Mit dem Wissen aus Kapitel 3 kann auch ein Anfänger bereits gute Programme erstellen. Speziellere Techniken für etwas erfahrenere Programmierer werden in Kapitel 4 vorgestellt. Mit ihnen können Applikationen noch individueller an die Ansprüche des Anwenders und an das zu lösende Problem angepasst werden.

Dort werden auch weiterführende Konzepte beschrieben – wie zum Beispiel die Erweiterung eines Programms auf mehrere Landessprachen –, die in keiner guten KDE-Applikation fehlen sollten.

3.1 Die Basisklasse – *QObject*

QObject ist die zentrale Klasse in der Qt-Klassenhierarchie und dient insbesondere als rudimentäre Grundlage für alle Elemente der grafischen Benutzeroberfläche (GUI).

Die Klasse bietet die Möglichkeit, Objektinstanzen in einer *Baumstruktur* hierarchisch anzuordnen, was in Kapitel 3.1.1, *Hierarchische Anordnung der Objektinstanzen von QObject*, genauer beschrieben wird. Zum einen kann man damit den Programmieraufwand für die Speicherverwaltung reduzieren, zum anderen wird dieses Feature zur Darstellung der GUI-Elemente durch den X-Server benutzt, um Instanzen der Klasse *QWidget*, die von *QObject* abgeleitet ist, entsprechend dieser Hierarchie darzustellen.

Außerdem sind in *QObject* Methoden implementiert, die einen sehr flexiblen und leistungsfähigen Nachrichtenaustausch zwischen Objekten dieser Klasse (oder einer ihrer Unterklassen) ermöglichen: das so genannte *Signal-Slot-Konzept*. Auch dieses Konzept wird besonders von den GUI-Elementen genutzt, um Aktionen des Benutzers an andere Teile des Programms weiterzugeben (so genannte *Callbacks*). Eine genaue Beschreibung folgt in Kapitel 3.1.2, *Das Signal-Slot-Konzept*.

Außerdem sind in *QObject* die grundlegenden Methoden zur Verarbeitung von *Events* abgelegt. *Events* befinden sich auf einer rudimentäreren Ebene als Signale und Slots. In *QObject* selbst sind nur zwei *Events* definiert: ein Event, der das Einfügen oder Entfernen von anderen *QObject*-Instanzen in den Hierarchiebaum meldet, und ein Event, der von einem frei programmierbaren Timer in regelmäßigen Abständen aktiviert werden kann. Eine große Anzahl weiterer *Events* wird in der Klasse *QWidget* eingeführt. Diese Klasse – von *QObject* abgeleitet – benutzt die Event-Methoden, um vom X-Server über Änderungen an der Maus, an der Tastatur, an der Fensteranordnung und so weiter informiert zu werden. Das Event-Konzept wird in Kapitel 3.4, *Hintergrund: Event-Verarbeitung* etwas genauer besprochen.

Das Signal-Slot-Konzept und die Informationen über die Vererbungsstruktur können nicht durch C++-Konzepte realisiert werden. Daher wurde ein zusätzlicher Präprozessor namens *moc* (Meta Object Compiler) entwickelt, der aus der Klassendefinition einer von *QObject* abgeleiteten Klasse eine zusätzliche Quellcode-Datei erzeugt, die ebenfalls kompiliert und zum ausführbaren Programm hinzugelinkt werden muss. Die genaue Vorgehensweise wird in Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*, besprochen.

Jede von *QObject* abgeleitete Klasse enthält außerdem ein so genanntes *Meta-Objekt*, das Informationen über die Klasse enthält (Klassenname, Vaterklasse, Signale und Slots, Properties). Diese Informationen werden in den meisten Programmen nicht benötigt, sie werden aber der Vollständigkeit halber in Kapitel 3.1.4, *Informationen über die Klassenstruktur*, bereits beschrieben.

3.1.1 Hierarchische Anordnung der Objektinstanzen von *QObject*

Der Konstruktor der Klasse *QObject* enthält zwei Parameter:

```
QObject (QObject *parent = 0, const char *name = 0)
```

Im ersten Parameter, *parent*, kann man einen Pointer auf ein anderes Objekt der Klasse *QObject* (oder eine ihrer Unterklassen) übergeben, das damit als Vaterknoten in der internen baumartigen Hierarchie festgelegt wird. In diesem Vaterknoten wird das neu erzeugte Objekt als weiterer Kindknoten registriert.

Benutzt man als Wert für den ersten Parameter den Null-Pointer – das ist auch der Default-Wert –, so hat das erzeugte Objekt keinen Vater und steht somit ganz oben in der Hierarchie.

Im zweiten Parameter, *name*, kann man dem Objekt noch einen Namen in Form eines nullterminierten Strings geben. Auf dem Heap wird dann neuer Speicher angelegt und der String kopiert. Benutzt man hier den Null-Pointer – auch hier ist das der Default-Wert für diesen Parameter –, so hat das Objekt keinen Namen.

Der Name hat im Moment noch keine spezielle Bedeutung. Er kann jedoch vom Programmierer ausgelesen und für eigene Zwecke benutzt werden. Er kann auch beim Debuggen wertvolle Hilfe leisten.

Der Vaterknoten und der Name des Objekts können *nur* im Konstruktor festgelegt und anschließend nicht mehr geändert werden. (Eine Ausnahme bilden die Unterklasse *QWidget* und die von ihr abgeleiteten Klassen, in denen man mit der Methode *QWidget::reparent* den Vaterknoten auch nachträglich ändern kann, siehe den Abschnitt *Methoden für den Modus des Widgets* in Kapitel 3.2.3, *Die wichtigsten Widget-Eigenschaften*.) Eine Baumstruktur kann daher nur von der Wurzel zu den Blättern aufgebaut werden, da bei der Erzeugung eines Objekts der Vater schon existieren muss.

Die folgende Zeile erzeugt eine vaterlose Instanz von *QObject* ohne Namen auf dem Heap:

```
QObject *obj1 = new QObject;
```

Zu diesem Objekt fügen wir nun ein Kindobjekt ohne Namen hinzu:

```
QObject *obj2 = new QObject (obj1);
```

Wir fügen noch ein weiteres Kind mit dem Namen »Ich bin das Zweitgeborene« hinzu:

```
QObject *obj3 = new QObject (obj1, "Ich bin das Zweitgeborene");
```

Ein weiteres Objekt wird nun erzeugt, das diesmal jedoch *obj3* als Vater hat:

```
QObject *obj4 = new QObject (obj3, "Ich bin das Enkelkind");
```

Wir erzeugen noch ein weiteres vaterloses Objekt, aber diesmal mit einem Namen:

```
QObject *obj5 = new QObject (0, "Ich bin vaterlos");
```

Die entstandene Struktur lässt sich grafisch wie in Abbildung 3.1 darstellen.

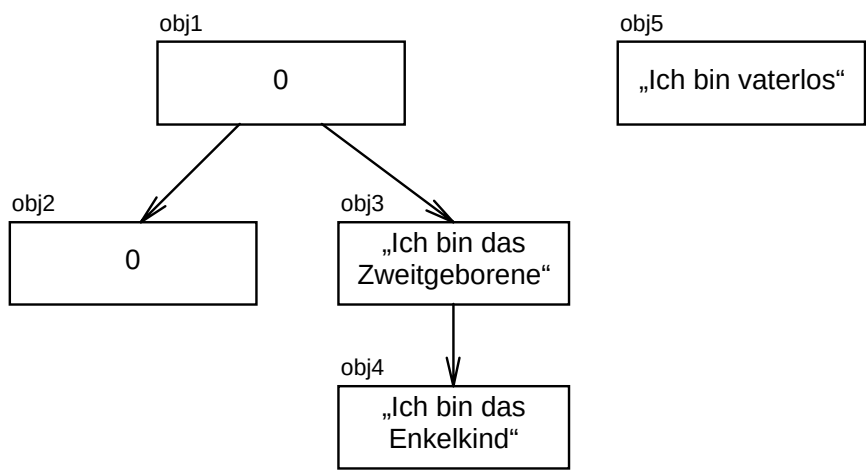


Abbildung 3-1 Baumstruktur mit QObject

Wird eines der erzeugten Objekte gelöscht, also der Destruktor aufgerufen, so werden automatisch zunächst alle Kindobjekte (und rekursiv auch deren Kinder) gelöscht, bevor das Objekt selbst freigegeben wird. Die Auswirkungen eines *delete*-Befehls auf eines der Objekte sind in Tabelle 3.1 aufgelistet.

Aufruf von	Löschung der Objekte
delete obj1;	obj1, obj2, obj3 und obj4
delete obj2;	obj2
delete obj3;	obj3 und obj4
delete obj4;	obj4
delete obj5;	obj5

Tabelle 3-1 Rekursives Löschen der Kindobjekte

Da zum Löschen der Kindobjekte *delete* benutzt wird, *müssen alle* Kindobjekte mit *new* auf dem Heap erzeugt worden sein. Erzeugt man stattdessen Instanzen in lokalen Variablen, so kommt es zu einem Laufzeitfehler!

Die Eigenschaft der rekursiven Löschung der Kindobjekte vereinfacht für den Programmierer die Speicherfreigabe. Sobald die Objekte eines Baumes nicht mehr benötigt werden, gibt der Programmierer einfach das Objekt an der Wurzel frei.

Als Faustregel kann man sich merken, dass Instanzen der Klasse *QObject* möglichst mit *new* auf dem Heap angelegt werden sollten, insbesondere dann, wenn sie ein Vater-Objekt haben. Bei Objekten mit einem Vater braucht man sich dann in der Regel nicht mehr um die Speicherfreigabe zu kümmern, da dies automatisch beim Löschen des Vaters geschieht.

Von dieser Faustregel gibt es zwei häufig benutzte Ausnahmen:

- Die Klasse *QApplication* (bzw. *KApplication* für KDE-Programme) ist ebenfalls von *QObject* abgeleitet. Man erzeugt aber von dieser Klasse grundsätzlich nur eine Instanz, die keinen Vater besitzt und die bis zum Ende des Programms existiert. Man kann diese Instanz also getrost in einer lokalen Variable in der *main*-Funktion speichern. Sie wird dann automatisch am Ende des Programms gelöscht. (Für Beispiele siehe Kapitel 2.2, *Das erste Qt-Programm*, sowie Kapitel 3.3, *Grundstruktur einer Applikation*.)
- Modale Dialogfenster, die durch Unterklassen von *QDialog* gebildet werden. Diese sind ebenfalls von *QObject* abgeleitet, blockieren aber das Programm so lange, bis der Anwender sie schließt. Sie können auch sinnvollerweise in lokalen Variablen abgelegt werden. Am Ende des Programmblocks werden sie dadurch automatisch gelöscht (siehe auch Kapitel 3.8, *Der Dialogentwurf*).

Die Klassen in diesen Ausnahmen werden später noch detailliert beschrieben; sie sollen hier nur schon einmal erwähnt werden.

Der Name, den das Objekt beim Aufruf des Konstruktors zugewiesen bekommen hat, kann mit der Methode *QObject::name()* ermittelt werden.

Zwei weitere Methoden der Klasse *QObject* ermöglichen das Abfragen der Hierarchie: Mit *QObject::parent()* kann man den Vaterknoten bestimmen (vaterlose Knoten liefern den Null-Zeiger zurück), mit *QObject::children()* erhält man eine Liste der Kinder der Instanz. Der Rückgabewert ist ein Zeiger auf die interne Liste und als *const* deklariert, so dass Sie die Liste nur auslesen, aber nicht verändern können. Der Datentyp der Liste ist *QObjectList*. Diese Klasse ist in der Header-Datei *qobjectlist.h* deklariert und ist eine Unterklasse der Template-Klasse *QList* <*QObject*>. Nähere Informationen zum Umgang mit *QList* finden Sie in Kapitel 4.7.2, *Container-Klassen*. Ein Beispiel für die Anwendung finden Sie in der Übungsaufgaben in Kapitel 3.1.5, *Übung 3.1*.

Für das vorangegangene Beispiel gilt:

```
obj3->name ();
```

liefert als Rückgabewert den String »Ich bin das Zweitgeborene«.

```
obj3->parent ();
```

liefert einen Zeiger auf die Instanz *obj1* zurück.

```
obj1->children ();
```

liefert einen Zeiger auf eine Liste vom Typ *QObjectList* mit zwei Elementen zurück: Die Elemente sind Zeiger auf die Instanzen *obj2* und *obj3*.

Zusammengefasst sollten Sie sich Folgendes merken:

- Die Klasse *QObject* hat im Konstruktor zwei Parameter, *parent* und *name*. Mit dem ersten Parameter kann man eine baumartige Hierarchie aufbauen, mit dem zweiten dem Objekt einen Namen zuweisen, den man frei benutzen kann.
- Mit den Methoden *QObject::parent()* und *QObject::children()* kann die Hierarchie und mit *QObject::name()* kann der Name des Objekts abgefragt werden.
- Der Vaterknoten und der Name können nur im Konstruktor festgelegt und danach nicht mehr geändert werden (Ausnahme: *QWidget*).
- Beim Löschen einer Instanz von *QObject* werden zunächst alle Kinder mit *delete* gelöscht (und rekursiv deren Kinder).
- Alle Instanzen von *QObject* – insbesondere solche mit Vaterknoten – sollten mit *new* auf dem Heap angelegt werden.
- Um die Speicherfreigabe einer Instanz mit Vaterknoten braucht man sich in der Regel nicht zu kümmern.

3.1.2 Das Signal-Slot-Konzept

Elemente einer grafischen Benutzeroberfläche – z.B. Buttons – müssen ihrer Umgebung mitteilen, dass eine Aktion des Benutzers stattgefunden hat. Dies geschieht in den meisten GUI-Bibliotheken durch so genannte *Callback-Funktionen*. Dabei trägt man im Objekt eine Funktion ein, die bei einer Aktion aufgerufen werden soll.

Callback und Signal-Slot – Gemeinsamkeiten und Unterschiede

Die Firma Troll Tech hat in ihrer GUI-Bibliothek Qt das Callback-Prinzip erweitert und das sehr mächtige Signal-Slot-Konzept entwickelt, das in der Klasse *QObject* angewandt wird. In jeder abgeleiteten Klasse von *QObject* kann man neue

Methoden als Signal oder Slot definieren. Signale übernehmen dabei die Aufgabe, eine Nachricht aus einem Objekt heraus abzusenden, und Slots dienen dazu, solche Nachrichten zu empfangen. Über die Methode *connect* kann man dann ein Signal mit einem Slot zur Laufzeit verbinden. Tabelle 3.2 vergleicht das herkömmliche Callback-Prinzip mit dem neuen Signal-Slot-Konzept.

herkömmliche Callbacks	Signal-Slot-Konzept
Callback-Funktion	Slot-Methode
Registrieren einer Callback-Funktion im GUI-Objekt x	Verbinden einer Signal-Methode des Objekts x mit einer Slot-Methode eines anderen Objekts durch <i>connect</i>
Aufrufen der Callback-Funktion	Versenden einer Nachricht durch Aufrufen der Signal-Methode

Tabelle 3-2 Gegenüberstellung von Callbacks und dem Signal-Slot-Konzept

Das Signal-Slot-Konzept ist jedoch weit mehr als nur eine Umbenennung der Begriffe. Die hohe Flexibilität wird durch folgende Eigenschaften erreicht:

- Jede Klasse kann eine beliebige Anzahl weiterer Signale und Slots definieren.
- Die Nachrichten, die über den Signal-Slot-Mechanismus verschickt werden, können eine beliebige Anzahl von Argumenten von beliebigem Typ haben.
- Ein Signal kann mit einer beliebigen Anzahl von Slots verbunden sein. Die Nachricht, die man durch dieses Signal schickt, wird an jeden angeschlossenen Slot weitergeleitet. (Die Aufrufreihenfolge ist dabei nicht festgelegt.) Ein Signal kann auch unverbunden bleiben. Eine Nachricht in diesem Signal hat dann keine Auswirkung.
- Ebenso kann ein Slot Nachrichten von mehreren Signalen von verschiedenen Objekten empfangen.
- Man kann jederzeit weitere Verbindungen zwischen Signalen und Slots einfügen oder bestehende Verbindungen löschen.
- Wird ein Objekt der Klasse *QObject* gelöscht, so werden im Destruktor des Objekts alle bestehenden Verbindungen gelöscht, die von seinen Signalen ausgehen oder die zu seinen Slots führen. Somit ist es ausgeschlossen, dass Nachrichten an nicht mehr existierende Objekte geschickt werden, was einen Laufzeitfehler bewirken würde.

Die Nachteile des Signal-Slot-Konzepts sollen aber auch nicht verschwiegen werden:

- Da Signale und Slots keine C++-Konstrukte sind, muss zusätzlich C++-Code aus der Klassendefinition mit Hilfe des Programms *moc* (Meta Object Compiler) erzeugt und zusätzlich kompiliert werden. Die Benutzung des *moc* wird in Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*, erläutert.

- Nachrichten per Signal-Slot-Paar zu verschicken ist etwas langsamer als ein einfacher Funktionsaufruf, der bei dem herkömmlichen Callback erfolgt. Das Verschicken einer Nachricht über ein Signal ist aber so effizient programmiert, dass der Unterschied kaum messbar ist. Etwas mehr Aufwand ist nötig, um ein Signal mit einem Slot zu verbinden, aber auch dieser kann in der Regel vernachlässigt werden.
- In der Regel ist es nötig, dass eine neue Klasse von *QObject* abgeleitet wird, um Slots selbst definieren zu können. Das ist aufwendiger, als nur eine neue Callback-Funktion zu schreiben. Da ein guter C++-Programmierer aber in der Regel ohnehin alles in Klassen und Methoden definiert, bedeuten die zusätzlichen Slots kaum Aufwand.

Erzeugen – Verbinden – Vergessen

Zusammen mit der hierarchischen Anordnung von *QObject*-Instanzen ermöglicht es das Signal-Slot-Konzept, viele Objekte nach dem Dreisatz *Erzeugen – Verbinden – Vergessen* zu behandeln. Nachdem sie als Kindobjekt eines anderen Objekts mit *new* erzeugt wurden und ihre Signale und Slots mit anderen Objekten verbunden wurden, arbeiten sie völlig selbstständig. Sie erhalten Meldungen von anderen Objekten über ihre Slots oder über eintreffende Events (siehe Kapitel 3.4, *Hintergrund: Event-Verarbeitung*) und schicken daraufhin selbst Signale an andere angeschlossene Objekte. Gelöscht werden sie automatisch zusammen mit ihrem Vater-Objekt. Es ist daher oftmals nicht nötig, einen Zeiger auf das Objekt aufzubewahren. Man kann das Objekt also getrost »vergessen«.

Deklaration von Signal- und Slot-Methoden

Signale und Slots werden wie normale Methoden in der Klassendefinition einer Klasse deklariert. Sie müssen den Rückgabetypp *void* haben, können aber sonst beliebige Parameter besitzen. Für die Deklaration werden zusätzlich die Spezifizierungssymbole *signals* und *slots* eingeführt, die ganz ähnlich verwendet werden wie die Symbole *private*, *public* und *protected* von C++. Slots kann man dabei auch als *private slots*, *public slots* oder *protected slots* definieren. Die erste Zeile einer Klassendefinition muss das Makro *Q_OBJECT* enthalten (ohne abschließendes Semikolon). Über dieses Makro werden einige interne Strukturen in die Klasse eingefügt.

Der Aufbau einer Klasse, die von *QObject* abgeleitet ist, sieht dann so aus:

```
class NewClass : public QObject
{
    Q_OBJECT

public:
    // Konstruktoren und andere public-Methoden...
```

```
signals:
    void selected (int, const char *, QList <QObject>);

public slots:
    void resetValues (float *value, QString &identifizier);

private slots:
    void rearrangeObjects ();

private:
    // weitere private Methoden und Attribute
};
```

In dieser Klassendefinition wird nun ein Signal namens *selected* mit drei Parametern definiert, ein Slot namens *resetValues* mit zwei Parametern und ein weiterer Slot namens *rearrangeObjects* ohne Parameter. Der Code für Signal-Methoden wird vom *Meta Object Compiler (moc)* erzeugt, darf also nicht vom Programmierer eingetragen werden. (Nähere Informationen zum *moc* finden Sie in Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*.) Slots dagegen verhalten sich wie normale Methoden und können auch wie eine normale Methode aufgerufen werden, ohne dass eine Verbindung zu einem Signal nötig wäre. Daher muss ihr Code auch vom Programmierer entweder direkt in der Klassendefinition (als Inline-Code) oder separat davon geschrieben werden.

Wenn Sie eigene Signale und Slots deklarieren, sollten Sie auf möglichst treffende Namen achten. Ein Signal meldet eine Veränderung, daher sind oft Partizipien geeignete Bezeichnungen, wie *clicked*, *activated*, *moved*, *killed*, *enabled*, *exchanged* oder *stopped*. Auch Adjektive, die den neuen Zustand beschreiben, sind oft passend: *full*, *empty* oder *on*. Veranschaulichen wir das noch einmal mit dem Beispiel einer Verkehrsampelsteuerung: Sie kann vier verschiedene Zustände annehmen: rot, gelb, grün und rot-gelb. Sie kann eine Änderung zum Beispiel mit einem Signal *changed* (Partizip) nach außen melden, das als Parameter den neuen Zustand übergibt, sei es als String, als Aufzählungstyp oder als Bitkombination der einzelnen Farben. Alternativ kann sie auch vier Signale definieren –, alle ohne Parameter – die melden, wenn ein Zustand anfängt. Diese Signale lauten dann *red*, *yellow*, *green* und *redYellow* (Adjektive). Substantive sind meist ungeeignet, da sie meist die Bezeichnung für einen Zustand oder eine Eigenschaft sind. So eignen sich die Bezeichnungen *status* oder *value* besser für Methoden, die den aktuellen Wert zurückgeben, aber nicht für Signale. Vermeiden Sie auch Redundanzen in der Bezeichnung (*buttonClicked* für ein Button-Objekt) und wenig sagende Zusätze (*statusChanged*).

Slot-Methoden führen eine Aktion aus (als Reaktion auf ein verbundenes Signal). Wählen Sie für die Bezeichnung eines Slots daher ein Verb, in der Regel als Imperativ, also als Befehl. *clear*, *move*, *set*, *reset*, *exchange*, *kill*, *copy* und *print* sind etwa solche Bezeichnungen. Oftmals kann man einen Zusatz anhängen, um die

Bedeutung klarer zu machen: *setText*, *setImage*, *turnLeft*, *selectAll* oder *goToLine*. Widerstehen Sie der Versuchung, einen Slot genauso zu nennen wie das Signal, mit dem Sie ihn verbinden wollen. *clicked*, *buttonCancelClicked* oder *reactOnCancelButton* sind ungeeignete Slot-Namen. Wählen Sie stattdessen *cancel*, denn das ist die Aktion, die ausgeführt werden soll. Vermeiden Sie auch Namen wie *slotCancel* oder *slotButton*.

Verbinden von Signal und Slot

Um ein Signal mit einem Slot zu verbinden, benutzt man die statische Methode *connect* der Klasse *QObject*. Diese Methode benötigt vier Parameter:

```
QObject::connect (const QObject *sender,
                  const char *signal,
                  const QObject *receiver,
                  const char *member)
```

sender und *signal* spezifizieren dabei das Objekt und seine Signal-Methode, die Nachrichten aussenden soll; und *receiver* und *member* definieren ein Objekt und seine Slot-Methode, die diese Nachrichten empfangen soll. *sender* und *receiver* sind Zeiger auf das entsprechende Objekt. *signal* und *member* sind Strings, die die Namen und Parameter des zu benutzenden Signals bzw. Slots angeben. Die Umsetzung einer Methode in einen String wird dabei eigentlich immer von zwei Makros, *SIGNAL* und *SLOT*, vorgenommen. Diesen Makros übergibt man als Parameter den Namen der Signal- bzw. Slot-Methode inklusive der Liste der Parametertypen. Ein Aufruf von *connect* sieht dann zum Beispiel wie folgt aus, wenn das Signal *send* von Objekt *obj1* mit dem Slot *receive* von Objekt *obj2* verbunden werden soll. Beide Methoden sollen dabei zwei Parameter haben, den ersten vom Typ *int* und den zweiten vom Typ *QString* *:

```
QObject::connect (&obj1, SIGNAL (send (int, QString*)),
                  &obj2, SLOT (receive (int, QString*)));
```

Innerhalb einer Methode in einer von *QObject* abgeleiteten Klasse kann man natürlich *QObject::* weglassen.

Achtung: Während des Kompilierens werden noch keinerlei Überprüfungen vorgenommen – weder, ob es überhaupt Signal- oder Slot-Methoden mit den entsprechenden Namen oder Parametern gibt, noch, ob Signal und Slot miteinander kompatibel sind. Eine Fehlermeldung gibt es erst zur Laufzeit, wobei das Programm nicht abgebrochen wird. Gerade für Programmierer, die mit dem Signal-Slot-Konzept noch nicht so vertraut sind, ist es daher wichtig, diese Fehlermeldungen auf *stderr* zu beachten.

signal muss eine definierte Signal-Methode enthalten, *member* kann sowohl eine Slot- als auch eine Signal-Methode enthalten. Ist *member* ein Signal, so wird bei der Aktivierung von *signal* diese Nachricht durch *member* weitergeleitet und erreicht somit auch alle Slots, die mit *member* verbunden sind.

Ein Signal kann mit einem Slot (oder einem anderen Signal) nur verbunden werden, wenn die Parameterlisten von beiden kompatibel sind. Das ist nur dann der Fall, wenn der empfangende Slot genauso viele oder weniger Parameter hat wie das sendende Signal und die Typen der Parameter und ihre Reihenfolge übereinstimmen. Überzählige Parameter am Ende des Signals werden ignoriert. Das Signal *send* von *obj1* aus dem obigen Beispiel kann also nur mit einem Slot mit den Parametern (*int*, *QString* *), mit dem Parameter (*int*) oder mit einer leeren Parameterliste () verbunden werden. Alles andere führt zu einer Fehlermeldung zur Laufzeit.

Neben der oben angegebenen statischen Form der *connect*-Methode kann auch eine andere Variante benutzt werden:

```
QObject::connect (const QObject *sender,  
                  const char *signal,  
                  const char *member)
```

Diese Methode ist nicht statisch, kann also zum Beispiel innerhalb einer Objekt-Methode aufgerufen werden. Sie ruft die statische Variante von *connect* mit dem Wert *this* für den Parameter *receiver* auf. Diese kürzere Variante wird besonders in Konstruktoren oft verwendet, um erzeugte Objekte mit eigenen Slots zu verbinden.

Nachdem die Verbindung etabliert wurde, kann ein Signal gesendet werden. Dies geschieht, indem die Signal-Methode mit den gewünschten Parameterwerten aufgerufen wird. Um deutlich zu machen, dass es sich um eine Nachricht handelt, die über eine Signal-Slot-Verbindung verschickt wird, kann man das Wort *emit* dem Methodenaufruf voranstellen:

```
emit send (10, &s);
```

Es macht keinen Unterschied, ob Sie zum Senden *emit* vor den Aufruf des Signals stellen oder nicht. *emit* ist ein Makro, das durch einen leeren Text ersetzt wird.

Obwohl Verbindungen automatisch gelöst werden, wenn eines der beiden Objekte gelöscht wird, kann es manchmal nötig sein, eine Verbindung explizit zu lösen. Dazu kann die Methode *disconnect* verwendet werden, die die gleichen Parameter wie *connect* benötigt:

```
QObject::disconnect (const QObject *sender,  
                     const char *signal,  
                     const QObject *receiver,  
                     const char *member)
```

Der erste Parameter der *disconnect*-Methode, *sender*, muss angegeben sein. Die anderen Parameter können zum Teil oder vollständig durch einen Null-Zeiger ersetzt werden, der dann als Wildcard-Symbol für eine beliebige Signal-Methode,

ein beliebiges Empfängerobjekt bzw. eine beliebige Slot-Methode steht. Alle passenden Verbindungen werden dann gelöst. Auch von *disconnect* gibt es zwei nichtstatische Varianten:

```
disconnect (const char *signal=0,  
           const QObject*receiver=0,  
           const char *member=0)
```

```
disconnect (const QObject *receiver,  
           const char *member=0)
```

Eigenschaften von Signalen und Slots

Hier sind noch ein paar Eigenschaften von Signal- und Slot-Methoden aufgelistet, die das Konzept ein wenig erläutern:

- Die verschickten Nachrichten werden nicht gespeichert oder verzögert. Der Aufruf einer Signal-Methode bewirkt unmittelbar, dass die verbundene Slot-Methode mit den entsprechenden Parametern aufgerufen wird. Sind mehrere Slot-Methoden mit der Signal-Methode verbunden, so werden diese nacheinander ausgeführt, wobei die Reihenfolge nicht festgelegt ist.
- Slot-Methoden sollen Reaktionen auf eine Nachricht ausführen. Sie müssen daher vom Programmierer mit Quellcode versehen werden. Was dort getan wird, bleibt ganz dem Programmierer überlassen. Er kann innerhalb der Slot-Methode auch weitere Nachrichten abschicken (die dann unmittelbar bearbeitet werden, bevor die Slot-Methode zurückkehrt); er kann Signal-Slot-Verbindungen erzeugen oder lösen usw.
- Die Signal-Methode ruft alle angeschlossenen Slot-Methoden auf. Da sonst keine weitere Aktion stattfinden muss, wird der Code für die Signal-Methode vom *moc* automatisch erzeugt. Wenn Sie versuchen, Code für eine Signal-Methode selbst zu schreiben, gibt es spätestens beim Linken eine Fehlermeldung, da nun zwei verschiedene Codestücke für die gleiche Methode vorliegen.
- In der Slot-Methode kann mit Hilfe der Methode *sender()* abgefragt werden, von welchem Objekt das gesendete Signal kam. Sie liefert einen Zeiger auf das Objekt zurück. Durch welche Signalmethode diese Nachricht allerdings verschickt wurde, lässt sich nicht ermitteln. Wurde die Slot-Methode direkt aufgerufen und nicht von einem Signal aktiviert, ist die Rückgabe von *sender* undefiniert. Sie ist also in jedem Fall mit Vorsicht zu genießen.
- Signale und Slots können beliebig viele Parameter von beliebigem Typ haben. Damit sinnvolle Verbindungen entstehen, können Signale nur mit solchen Slots verbunden werden, bei denen die Parameter passen. Das heißt, das Signal muss mindestens so viele Parameter haben wie der Slot, der mit ihm verbunden werden soll, und die Typen der Parameter und ihre Reihenfolge

müssen übereinstimmen. Dabei darf das Signal am Ende der Parameterliste zusätzliche Parameter enthalten, die bei der Aktivierung des Slots ignoriert werden. Wichtig ist, dass die Parameter exakt den gleichen Typ haben müssen. Hat das Signal einen Parameter vom Typ *const int **, so muss auch der entsprechende Parameter im Slot den gleichen Typ haben. Der Typ *int ** reicht hier nicht aus. Um möglichst universell und wiederverwertbar zu sein, sollten die Parametertypen möglichst allgemein gehalten sein. Ein Type-Cast, zum Beispiel von *char* auf *int* oder von einer spezielleren auf eine allgemeinere Klasse, wird nicht vorgenommen.

- Die Überprüfung, ob Signale und Slots kompatibel sind, wird erst zur Laufzeit vorgenommen. Sind sie es nicht, kommt die Verbindung nicht zustande, und eine Fehlermeldung wird auf *stderr* ausgegeben; das Programm läuft aber weiter. Eine solche Fehlermeldung zeigt also immer, dass bereits beim Programmieren ein falsches Signal oder ein falscher Slot benutzt wurde. Der Grund dafür kann in nicht passenden Parametertypen liegen oder kann auch ein einfacher Tippfehler beim Namen der Signal- oder Slot-Methode oder bei den Parametertypen sein. Beachten Sie also beim Testen solche Fehlermeldungen unbedingt. Sie sind nicht nur unschön, sie sind auch ein eindeutiges Zeichen für einen Fehler im Programm.
- Signale und Slots haben keinen Rückgabewert, es handelt sich immer um *void*-Methoden. Ein Slot kann nur über den Umweg über einen Referenzparameter Werte an die Methode zurückliefern, die das Signal aktiviert hat. Beachten Sie aber, dass mehrere Slots mit dem Signal verbunden sein können: Der Referenzparameter wird dann von jedem aktivierten Slot geändert.
- Signale und Slots zeichnen sich durch ihren Namen *und* die Zahl und Typen der Parameter aus. Es ist also möglich, verschiedene Signale und Slots mit dem gleichen Namen zu definieren, solange die Parameter verschieden sind.
- Signal- und Slot-Methoden können keine *statischen* Methoden sein. Slot-Methoden können jedoch als *const* oder *inline* deklariert werden.
- Signale und Slots werden vererbt. Ist eine Klasse B von Klasse A abgeleitet, so besitzt auch ein Objekt der Klasse B die Signale und Slots, die in Klasse A deklariert sind, und kann sie mit anderen Objekten verbinden. (Wenn Sie in der Dokumentation zu einer Klasse ein sinnvolles Signal oder einen Slot vermissen, so schauen Sie auch in der Dokumentation der Vaterklassen nach, ob es sich nicht dort findet.)
- Slots können als virtuelle Methoden deklariert werden (*virtual*). Sie können dann in einer abgeleiteten Klasse überschrieben – also mit anderem Code gefüllt – werden. Dabei sind sie dann nicht als Slot-Methode, sondern als normale Methode zu deklarieren (sonst gäbe es den Slot doppelt, einmal vererbt vom Vater und dann noch einmal deklariert). Auch Signale können theore-

tisch virtuell sein, nur macht dies wenig Sinn, da der Code des Signals automatisch generiert wird, ein Überschreiben der Signal-Methode also keinen Unterschied bewirkt.

- Default-Parameter sind für Signale und Slots nicht erlaubt.
- In der Regel verbindet man die Signale und Slots eines Objekts unmittelbar nach seiner Erzeugung. Diese Verbindungen bleiben während der kompletten Lebensdauer des Objekts erhalten, ein explizites Auflösen ist also meist nicht nötig. Dennoch können Sie Verbindungen beliebig mit *connect* erzeugen und mit *disconnect* wieder entfernen, um den Nachrichtenfluss zu steuern.
- Kurzfristig kann das Aussenden von Signalen durch einen Aufruf der Methode *blockSignals (true)* unterbunden werden. Alle danach ausgesendeten Signale dieses Objekts werden ignoriert. Mit *blockSignals (false)* kann man die Blockierung wieder aufheben. Die Methode *signalsBlocked* liefert den aktuellen Zustand der Blockierung zurück.
- Man kann sich informieren lassen, wenn eine Verbindung zu einem Signal aufgebaut oder gelöst wird, indem man die virtuellen Methoden *connectNotify* und *disconnectNotify* überschreibt. Das kann unter Umständen von Vorteil sein, wenn die Aktivierung eines Signals erst eine aufwendige Berechnung erfordern sollte. In dem Fall kann man einfach mitzählen, wie viele Verbindungen zum Signal es gibt, und die Berechnungen nur dann durchführen, wenn es mindestens eine Verbindung gibt. In der Regel ist es aber aufwendiger, die Zahl der Verbindungen zu ermitteln, als ein nicht verbundenes Signal zu aktivieren.
- Eine Signal-Methode ist bereits in *QObject* definiert: *destroyed ()* wird ohne Parameter im Destruktor des Objekts ausgesendet, unmittelbar bevor das Objekt gelöscht wird. Mit diesem Signal kann man zum Beispiel eine Liste von *QObject*-Instanzen aktuell halten, auch wenn eines der Objekte (von außen) gelöscht wird. Mit der Methode *sender* kann ermittelt werden, welches Objekt gerade zerstört wird.

Da das Signal-Slot-Konzept von enormer Bedeutung ist, wollen wir es anhand einer Reihe von Beispielen genauer erläutern.

Beispiel 1: *QPushButton*

Die Klasse *QPushButton* stellt das GUI-Element der Schaltfläche in der Qt-Bibliothek dar. Ein Objekt dieser Klasse zeichnet einen beschrifteten, nach vorn herausgehobenen Knopf, den man mit der linken Maustaste anklicken kann. Die entsprechenden Mausektionen bekommt das Objekt über Events mitgeteilt, die die Event-Methode des Objekts aktivieren. In der Klasse *QPushButton* sind nun vier verschiedene Signale definiert, die aktiviert werden, wenn der Benutzer eine entsprechende Aktion ausgeführt hat: Das Signal *pressed ()* wird aktiviert, wenn

die Maustaste innerhalb des Knopfes gedrückt wurde, das Signal *released()*, wenn die Maustaste innerhalb des Knopfes losgelassen wurde. Das Signal *clicked()* wird nur dann aktiviert, wenn die Maustaste zuerst innerhalb des Knopfes gedrückt und dann dort auch wieder losgelassen wurde. Außerdem gibt es das Signal *toggled(bool)*, das aktiviert wird, wenn es sich um einen so genannten Toggle-Button handelt, bei dem man durch einen Mausklick zwischen den Zuständen »ein« und »aus« wählen kann. Während die ersten drei Signale keine Parameter besitzen, hat das vierte Signal einen Parameter vom Typ *bool*, der *true* ist, falls der Button eingeschaltet ist, oder *false*, falls er ausgeschaltet ist.

In den meisten Anwendungsfällen verbindet man nur eines der vier Signale des *QPushButton*-Objekts (meist *clicked()*) mit einem Slot eines eigenen Objekts. Die anderen drei Signale werden zwar ebenfalls aktiviert, wenn eine entsprechende Aktion stattgefunden hat, da sie aber unverbunden sind, hat das keine Auswirkung.

Die ersten drei Signale können nur mit Slots verbunden werden, die keine Parameter besitzen. Das vierte Signal kann mit einem Slot ohne Parameter oder mit einem Slot mit einem einzigen Parameter vom Typ *bool* verbunden werden.

Beispiel 2: Verbindung zwischen *QPushButton* und *QLabel*

Nach aller Theorie nun wieder ein ganz praktisches Beispiel: Wir wollen zwei Fenster auf den Bildschirm bringen: Das eine enthält ein *QLabel*-Objekt mit einem Text, das andere ein *QPushButton*-Objekt mit der Aufschrift *CLEAR*. Beim Druck auf den Button soll der Text im *QLabel*-Objekt verschwinden.

Praktischerweise hat *QLabel* einen Slot namens *clear* ohne Parameter, der genau das macht. Diesen verbinden wir mit dem Signal *clicked()* des Buttons, und damit sind wir schon fertig.

Hier folgt der Code der Quellcodedatei. Er entspricht weitgehend dem Code aus unserem ersten Beispiel aus Kapitel 2.2, *Das erste Qt-Programm*. Speichern Sie ihn wieder unter dem Dateinamen *hello-qt.cpp* ab. Die hinzugekommenen Teile sind fett gedruckt.

```
#include <qapplication.h>
#include <qlabel.h>
#include <qpushbutton.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    QPushButton *b = new QPushButton ("Clear", 0);
    b->show();
```

```
QObject::connect (b, SIGNAL (clicked()),
                  1, SLOT (clear()));

app.setMainWidget (1);
return app.exec();
}
```

Kompilieren und linken Sie dieses Programm wie in Kapitel 2.2.2, *Kompilieren des Programms unter Linux*, bzw. Kapitel 2.2.3, *Kompilieren des Programms unter Microsoft Windows*, beschrieben, und starten Sie es. Abbildung 3.2 zeigt die beiden sich öffnenden Fenster nebeneinander.



Abbildung 3-2 *QLabel und QPushButton*

In der Zeile mit dem Befehl *connect* mussten wir die Klasse *QObject* angeben, da wir uns außerhalb einer Methode einer von *QObject* abgeleiteten Klasse befinden. Das gleiche können Sie auch erreichen, wenn Sie für das *QLabel*-Objekt die Methode *connect* aufrufen:

```
1->connect (b, SIGNAL (clicked()),
           SLOT (clear()));
```

Beachten Sie, dass wir den dritten Parameter (Empfängerobjekt) in diesem Fall weglassen: Dort wird automatisch als Empfänger das Objekt benutzt, für das wir die Methode *connect* aufrufen.

Beispiel 3: Verbindung zwischen *QListBox* und *QLabel*

Nachdem wir in Beispiel 2 nur parameterlose Signale und Slots miteinander verbunden haben, benutzen wir nun solche mit einem Parameter. Wir wählen dazu als ein Objekt die Klasse *QListBox* aus. Sie kann eine Reihe von Strings enthalten, aus denen der Anwender durch Anklicken einen auswählen kann. Der zugehörige String wird dann per Signal *selected* (*const QString &*) an alle angeschlossenen Slots verschickt. (Nähere Informationen zur Klasse *QString* finden Sie in Kapitel 4.7.4, *Die String-Klassen – QString und QString*.) Praktischerweise hat *QLabel* auch einen Slot, um einen solchen String entgegenzunehmen, nämlich *QLabel::setText* (*const QString &*). (Beachten Sie, dass wir die beiden nur deshalb verbinden kön-

nen, weil sie exakt die gleichen Parametertypen haben. Wäre der Parameter in einer der beiden Klassen beispielsweise vom Typ *QString* statt *const QString &*, so wären sie inkompatibel.)

```
#include <qapplication.h>
#include <qlabel.h>
#include <qlistbox.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    QListBox *b = new QListBox (0);
    b->insertItem ("George");
    b->insertItem ("Paul");
    b->insertItem ("Ringo");
    b->insertItem ("John");
    b->show();

    QObject::connect (b, SIGNAL (selected (const QString &)),
                      l, SLOT (setText (const QString &)));

    app.setMainWidget (l);
    return app.exec();
}
```

Jedes Mal wenn Sie in der Liste einen Namen mit einem Doppelklick auswählen, erscheint er auch im *QLabel*-Objekt (siehe Abbildung 3.3).

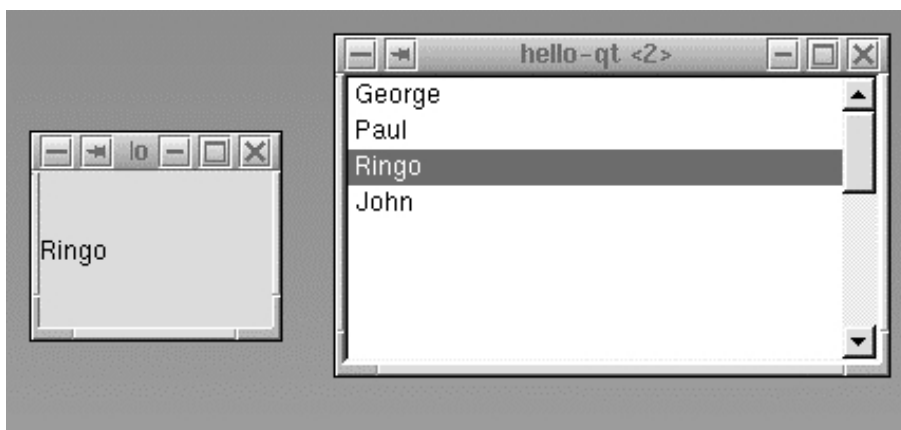


Abbildung 3-3 QListBox und QLabel

Wenn Sie einen Blick in die Online-Referenz der Qt-Bibliothek zur Klasse *QListBox* werfen, sehen Sie, dass *QListBox* noch viele andere Signale sendet. Wenn Sie beispielsweise das Signal *highlighted* (*const QString &*) verwenden, so genügt ein einfacher Mausklick zur Auswahl.

Es gibt auch ein Signal *selected* (*int*), das nicht den Text, sondern die Position in der Liste liefert. Dieses lässt sich natürlich nicht mit dem Slot *QLabel::setText* (*const QString &*) verbinden, da die Parameter nicht passen. Aber *QLabel* hat einen Slot namens *setNum* (*int*), und den können wir benutzen. Ändern Sie den *connect*-Befehl um in:

```
QObject::connect (b, SIGNAL (selected (int)),  
                 1, SLOT (setNum (int)));
```

Nun wird bei jeder Auswahl eines Namens die Listenposition als Text im *QLabel*-Objekt dargestellt (siehe Abbildung 3.4). Die erste Listenposition (*George*) hat dabei die Nummer 0, wie in der Computerwelt üblich.

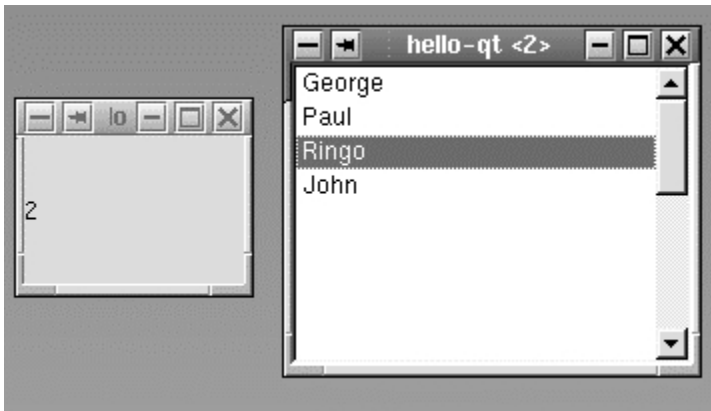


Abbildung 3-4 Verbindung von *selected* (*int*) und *setNum* (*int*)

Beispiel 4: Eine selbst definierte Klasse

GUI-Elemente haben meist eine kleine Zahl von einfachen Signalen, die sie als Reaktion auf eine Benutzeraktion oder eine andere Änderung innerhalb des Objekts senden können. Die Reaktion, die auf ein solches Signal erfolgen soll, ist aber meist sehr spezifisch und von der Applikation abhängig. Die Reaktion auf eine Schaltfläche »Beenden« ist sicher eine andere als eine Reaktion auf »Speichern«. Daher müssen die meisten Slots vom Programmierer selbst entworfen werden. Dazu ist es nötig, eine eigene Klasse zu definieren, die von der Klasse *QObject* abgeleitet ist.

Wir wollen hier eine eigene Klasse entwerfen, die mitzählt, wie oft ein Button angeklickt wurde. Wir definieren dazu eine eigene Klasse *Counter*. Die Header-Datei *counter.h* könnte zum Beispiel so aussehen:

```
#ifndef _COUNTER_H_
#define _COUNTER_H_

#include <qobject.h>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter (QObject *parent=0, const char *name=0);
    ~Counter ();

public slots:
    void countUp ();

private:
    int n;
};

#endif
```

Bei dieser Klassendefinition ist es wichtig, dass die neue Klasse von *QObject* (oder einer abgeleiteten Klasse) abgeleitet ist, dass die erste Zeile in der Klassendefinition das Makro *Q_OBJECT* ist (ohne Semikolon danach) und dass wir eine Slot-Methode definiert haben. Der Rückgabetypp ist *void*, wie für alle Signale und Slots vorgeschrieben, und der Slot besitzt keine Parameter. Dieser Slot ist als *public* deklariert, d.h. er kann auch von außerhalb der Klasse direkt aufgerufen werden.

Der Code für diese Klasse in der Datei *counter.cpp* könnte so aussehen:

```
#include "counter.h"
#include <iostream.h>

Counter::Counter (QObject *parent, const char *name)
    : QObject (parent, name), n (0)
{

}

Counter::~~Counter ()
{

}

void Counter::countUp ()
{
    n++;
    cout << "Anzahl Aktivierungen " << n << endl;
}
```

Der Konstruktor reicht zum einen die Parameter *parent* und *name* an den *QObject*-Konstruktor weiter und initialisiert außerdem *n* mit dem Wert 0. Der Destruktor macht nichts, außer den Destruktor von *QObject* automatisch aufzurufen. Der Slot *countUp* erhöht bei jeder Aktivierung *n* um eins und gibt den Wert von *n* auf dem Bildschirm aus.

Wir wollen nun unser neues Objekt zusammen mit einem *QPushButton*-Objekt benutzen, so dass jedes Mal der Slot *countUp* aufgerufen wird, wenn der Button angeklickt wird. Dazu brauchen wir ein Hauptprogramm, das die Objekte erzeugt und verbindet. Es entspricht in etwa dem Beispiel aus Kapitel 2.2, *Das erste Qt-Programm* (die Unterschiede sind wieder fett gedruckt). Wir legen es in der Datei *main.cpp* ab:

```
#include <qapplication.h>
#include <qpushbutton.h>
#include "counter.h"

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QPushButton *b = new QPushButton ("Click", 0);
    b->show();

    Counter *c = new Counter (b);

    QObject::connect (b, SIGNAL (clicked()),
                     c, SLOT (countUp()));

    app.setMainWidget (b);
    return app.exec();
}
```

Beachten Sie die Unterschiede zu den bisherigen Beispielen:

- Die Header-Datei *counter.h* wurde von uns selbst geschrieben. Der Compiler soll sie also nicht in den Verzeichnissen für die üblichen Header-Dateien suchen, sondern im aktuellen Verzeichnis. Daher binden wir diese Datei mit »*counter.h*« und nicht mit <*counter.h*> ein.
- Unser erzeugtes *Counter*-Objekt erhält als Vater-Objekt den Button *b*. Somit wird es automatisch gelöscht, sobald *b* gelöscht wird. (*b* wird automatisch beim Freigeben des *QApplication*-Objekts gelöscht, wie es mit allen vaterlosen *QWidget*-Objekten geschieht. Für »normale« von *QObject* abgeleitete Objekte gilt dies nicht. Hätten wir als Vater-Objekt für unser *Counter*-Objekt 0 gewählt, so würde es nicht mehr freigegeben; ein Speicherleck wäre entstanden, zwar ein harmloses, aber ein unschönes.)

- *Counter* ist direkt von *QObject* abgeleitet. Es handelt sich also nicht um ein Widget, das auf dem Bildschirm angezeigt wird. Dementsprechend besitzt es auch keine Methode *show*, die wir aufrufen müssten.

Durch *connect* sind nun die beiden Objekte miteinander verbunden. Die zu benutzenden Signal- bzw. Slot-Methoden müssen inklusive der leeren Klammern angegeben werden.

Um dieses Beispielprogramm zu kompilieren und zu linkern, sind einige Schritte mehr nötig als bisher:

Zunächst müssen wir unsere Klassendeklaration durch den *Meta Object Compiler* (*moc*) schicken, der eine weitere Code-Datei daraus erzeugt. (Einzelheiten werden in Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*, besprochen.)

```
% moc counter.h -o counter.moc.cpp
```

Diese Zeile erzeugt eine weitere Datei namens *counter.moc.cpp*. Anschließend müssen alle drei Code-Dateien kompiliert und schließlich zusammengelinkt werden:

```
% g++ -c counter.moc.cpp -I$QTDIR/include
% g++ -c counter.cpp -I$QTDIR/include
% g++ -c main.cpp -I$QTDIR/include
```

```
% g++ -o counter main.o counter.o counter.moc.o -lqt
```

In der letzten Zeile müssen Sie eventuell wieder den Pfad der Qt-Bibliothek angeben (*-L\$QTDIR/lib*). Anschließend können Sie die ausführbare Datei *counter* starten. Starten Sie sie aus einem Terminalfenster heraus, damit Sie die Ausgaben von *cout* sehen.

Bei vielen Compilern lassen sich die letzten vier Zeilen auch in einer einzigen Zeile schreiben. (Voraussetzung hierbei ist, dass sich im aktuellen Verzeichnis nur die drei Code-Dateien *counter.moc.cpp*, *counter.cpp* und *main.cpp* befinden.)

```
% g++ -o counter *.cpp -I$QTDIR/include -lqt
```

Beispiel 5: Eine von einem *QWidget*-Objekt abgeleitete Klasse

Dieses Mal wollen wir die Anzahl der Tastendrucke in einem Fenster darstellen. Wir benutzen dazu die altbekannte Klasse *QLabel* und erweitern sie (durch Ableiten einer neuen Klasse) um die Fähigkeit, einen Zähler hochzuzählen.

Hier folgen ohne weiteren Kommentar die drei Dateien (wiederum sind die Änderungen fett gedruckt.)

Datei *counter.h*:

```
#ifndef _COUNTER_H_
#define _COUNTER_H_

#include <qlabel.h>

class Counter : public QLabel
{
    Q_OBJECT

public:
    Counter (QWidget *parent=0, const char *name=0);
    ~Counter ();

public slots:
    void countUp ();

private:
    int n;
};

#endif
```

Die Datei *counter.cpp*:

```
#include "counter.h"

Counter::Counter (QWidget *parent, const char *name)
    : QLabel (parent, name), n (0)
{
    setNum (0);
}

Counter::~~Counter ()
{}

void Counter::countUp ()
{
    n++;
    setNum (n);
}
```

Die Datei *main.cpp*:

```
#include <qapplication.h>
#include <qpushbutton.h>
#include "counter.h"
```

```
int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QPushButton *b = new QPushButton ("Click", 0);
    b->show();

    Counter *c = new Counter (0);
    c->show();

    QObject::connect (b, SIGNAL (clicked()),
                     c, SLOT (countUp()));

    app.setMainWidget (b);
    return app.exec();
}
```

Sie kompilieren und linken dieses Beispiel genau wie das vorhergehende.

Hier noch einmal kurz die Unterschiede, auf die Sie achten müssen:

- *Counter* ist nun von *QLabel* abgeleitet. *QLabel* ist seinerseits von *QFrame*, das wiederum von *QWidget* und das wiederum von *QObject* abgeleitet. Somit ist auch *Counter* indirekt von *QObject* abgeleitet, so dass wir eigene Signale und Slots definieren können. *Counter* verbindet also die alte Funktionalität von *QLabel*, Zahlen und Texte darzustellen, mit der Fähigkeit, über einen Slot einen internen Zähler zu erhöhen.
- Das Vater-Objekt für *QWidget*-Klassen (und davon abgeleitete Klassen) muss vom Typ *QWidget* sein. Daher müssen wir auch in unserer Klasse den Konstruktor so anpassen, dass wir als *parent*-Parameter nur *QWidget* akzeptieren. (Beachten Sie die Klassenvererbung: Einem *QObject*-Parameter kann man auch ein *QWidget*-Objekt übergeben, aber nicht umgekehrt.)
- Wir benutzen den Slot *setNum* in diesem Fall wie eine normale Methode. Da wir sie von *QLabel* geerbt haben, können wir sie einfach aufrufen.
- Im Hauptprogramm benutzen wir als Vaterparameter für unser *Counter*-Objekt den Null-Zeiger. Hier setzen wir nicht mehr den Button *b* ein, denn für Widgets (wie es *Counter* ist) hat der Vater eine besondere Bedeutung: Ein Widget wird immer innerhalb des Vater-Widgets dargestellt. Probieren Sie es aus: Setzen Sie als Vater wieder *b* ein: Nun erscheint der *Counter* nicht mehr in einem eigenen Fenster, sondern auf dem Button.
- Wie üblich rufen wir für unseren Counter die Methode *show* auf, damit er überhaupt auf dem Bildschirm angezeigt wird.

Abbildung 3.5 zeigt unsere neue Widget-Klasse in Aktion.

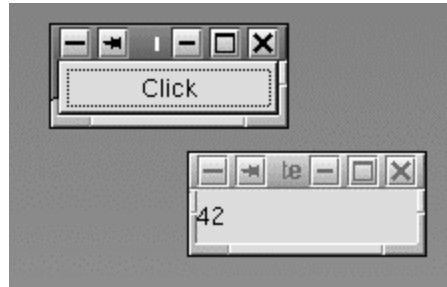


Abbildung 3-5 Die neue Counter-Klasse

Beispiel 6: Ein typisches Dialogfenster

Im Folgenden betrachten wir ein theoretisches Beispiel. Auch wenn Sie hier nichts selbst eintippen und ausprobieren können, sollten Sie das Beispiel genau studieren.

Selbst definierte Dialogfenster in Qt und KDE bestehen in der Regel aus einer selbst definierten Klasse vom Typ *QWidget*, die mehrere GUI-Elemente als Kindobjekte enthält. In der selbst definierten Klasse definiert man dann eine Reihe von Slots, die mit den Signalen der GUI-Elemente verbunden werden und die gewünschten Reaktionen ausführen. Ohne auf die Details der GUI-Elemente oder der Klasse *QWidget* eingehen zu wollen, folgt hier ein Beispiel, wie die Signal- und Slot-Definition aussehen kann. Da alle GUI-Elemente und die Klasse *QWidget* von *QObject* abgeleitet sind, können sie Signale und Slots benutzen.

```
class MyDialog : public QWidget
{
    Q_OBJECT
public:
    MyDialog (QWidget *parent=0, const char *name=0);
    ~MyDialog ();

signals:
    void OKWasPressed();
    void printMessage(const char*);

private slots:
    void printB2Message();
    void printB3Message ();
}
```

In dieser Klassendefinition wird das Signal *OKWasPressed* ohne Parameter definiert sowie das Signal *printMessage*, das einen Parameter vom Typ *const char** besitzt. Die beiden Slots *printB2Message* und *printB2Message* nehmen die Signale entgegen, die zwei weitere Buttons erzeugen. Sie sind *private*, da sie nur innerhalb der Klasse bekannt sein müssen. Eine Aktivierung von außen ist nicht nötig.

Der Konstruktor der neuen Klasse erzeugt nun drei Buttons und verbindet die Signale und Slots entsprechend:

```
MyDialog::MyDialog (QWidget *parent, const char *name)
: QWidget (parent, name)
{
    QPushButton *b1 = new QPushButton ("OK", this);
    connect (b1, SIGNAL (clicked ()),
            this, SIGNAL (OKWasPressed ()));
    QPushButton *b2 = new QPushButton ("Button2", this);
    connect (b2, SIGNAL (clicked ()),
            this, SLOT (printB2Message ()));
    QPushButton *b3 = new QPushButton ("Button3", this);
    connect (b3, SIGNAL (clicked ()),
            this, SLOT (printB3Message ()));
}
```

Direkt nach der Erzeugung der GUI-Elemente wird hier der entsprechende *connect*-Befehl ausgeführt. Beachten Sie, dass das Signal *clicked* vom OK-Button direkt an das Signal *OKWasPressed* weitergeleitet wird. Man hätte auch einen weiteren privaten Slot definieren können, in dessen Code das Signal *OKWasPressed* aktiviert worden wäre. Diese Schreibweise ist jedoch kürzer und übersichtlicher.

Beachten Sie auch, dass die *QPushButton*-Objekte auf dem Heap angelegt werden und dass für ihre Speicherung nur eine lokale Zeigervariable benutzt wird. Es ist nicht nötig, sich die Zeiger auf die Button-Objekte zu merken, nachdem der Konstruktor ausgeführt worden ist, denn durch die Verbindung mit *connect* ist die Funktionalität der Buttons bereits realisiert, und dadurch, dass sie Kindobjekte vom Objekt der Klasse *MyClass* sind, werden sie auch automatisch gelöscht, wenn das Dialogfenster gelöscht wird. Daher muss der Destruktor auch keine weiteren Aktionen ausführen, weder die Buttons löschen noch die Signal-Slot-Verbindungen aufheben:

```
MyClass::~MyClass () {}
```

Der Code für die Slots *printB2Message* und *printB3Message* kann zum Beispiel so aussehen:

```
MyClass::printB2Message ()
{
    emit printMessage ("Button 2 wurde angeklickt!\n");
}
MyClass::printB3Message ()
{
    emit printMessage ("Button 3 wurde angeklickt!\n");
    emit printMessage ("Eine weitere Meldung!\n");
}
```

Diese beiden Slots aktivieren das Signal *printMessage*, das einen Parameter verlangt. In diesem Fall werden einfach die konstanten Zeichenketten benutzt. Der zweite Slot aktiviert das Signal dabei sogar zweimal. Das hat natürlich erst dann eine Auswirkung, wenn das Signal *printMessage* des Objekts wiederum mit einem Slot verbunden ist, der darauf reagiert.

Beachten Sie, dass es nicht möglich ist, einen Wert für den Parameter in *connect* anzugeben und das Signal *clicked* von Button 2 beispielsweise direkt mit dem Signal *printMessage* zu verbinden. Auch wenn Sie nur einen festen Wert an das Signal weitergeben wollen, müssen Sie hier den Umweg über einen selbst definierten Slot benutzen.

Beispiel 7: Sammeln mehrerer Signale in einem Slot

Das folgende theoretische Beispiel ist schon etwas für fortgeschrittene Leser. Sie sollten es erst durcharbeiten, wenn Sie das Signal-Slot-Konzept wirklich verstanden haben.

Oftmals kommt es vor, dass eine Reihe von Signalen verschiedener Objekte ähnliche Auswirkungen haben soll. So kann man zum Beispiel fünf Buttons definieren, die einen Zähler um 1, 2, 3, 4 bzw. 5 erhöhen, je nachdem, welcher Button angeklickt wurde. Eine unelegante und unflexible Möglichkeit wäre es nun, fünf Slots zu definieren und jede Schaltfläche mit einem Slot zu verbinden.

Alternativ kann man eine neue Button-Klasse definieren, der man im Konstruktor eine Nummer mitgeben kann, die im Objekt gespeichert wird. Eine Aktivierung des Buttons ruft dann einen Slot auf, der seinerseits ein neues Signal mit der gespeicherten Zahl als Parameter aufruft. Dieses Signal kann dann bei allen Instanzen mit einem Slot verbunden werden, der den Wert als Parameter entgegennimmt.

Eine dritte Alternative, die ohne Definition einer neuen Button-Klasse auskommt, wollen wir hier vorstellen. Wir benutzen dazu die Methode *sender()*, die einen Zeiger auf das Objekt zurückliefert, das das Signal ausgesendet hat. Dieser Zeiger wird mit einer Liste von Zeigern auf die Buttons verglichen, um zu ermitteln, welche Schaltfläche angeklickt worden ist. Die Klassendefinition sieht wie folgt aus:

```
class MultiCounter : public QObject
{
    Q_OBJECT
public:
    MultiCounter (QObject *parent=0, const char *name=0);
    ~MultiCounter () {}
private slots:
    void countUp();
private:
```



```
    QList <QObject> list;
    int n;
};

MultiCounter::MultiCounter
    (QObject *parent, const char *name)
    : QObject (parent, name), list (), n (0)
{
    for (int i = 1; i <= 5; i++) {
        QString s;
        s.setNum (i);
        QPushButton *b = new QPushButton (s, 0, 0);
        connect (b, SIGNAL (clicked ()),
                this, SLOT (countUp ()));
        // auch möglich (da "this" implizit):
        // connect (b, SIGNAL (clicked ()),
        //          SLOT (countUp ()));
        list.append (b);
    }
}

MultiCounter::countUp ()
{
    QObject *button = sender ();
    int i = list.findRef (button);
    if (i == -1)
        qDebug ("Couldn't find the button!");
    else
        n += index + 1;
}
```

Im Konstruktor werden nun fünf Buttons erzeugt, die alle mit dem Slot *countUp* verbunden werden. In *list* werden die Zeiger auf die Buttons gespeichert. (*QList* ist eine Template-Klasse, die in Kapitel 4.7.2, *Container-Klassen*, näher beschrieben wird.) In *countUp* wird nun der Button, von dem das Signal kam, mit *sender* bestimmt und in *list* gesucht. Der Index, den *find* zurückliefert, liegt zwischen 0 und 4. Daher muss beim Erhöhen von *n* noch zusätzlich 1 addiert werden.

Auf genau diese Weise arbeitet übrigens die Klasse *QSignalMapper*, die auch zur Lösung dieses Problems benutzt werden kann. Verschiedene Objekte können durch Signale mit dem Slot *map* eines Objekts der Klasse *QSignalMapper* verbunden werden. Mit der Methode *setMapping* kann eine Zuordnung zwischen Sendereobjekten und einer Zahl oder einem String vorgenommen werden. Sendet nun eines der Objekte ein Signal, wird der zugeordnete Wert (Zahl oder String) gesucht und durch Aktivierung des Signals *mapped* weitergegeben.

3.1.3 Selbst definierte Klassen von *QObject* ableiten

Um in einem Programm das Signal-Slot-Konzept nutzen zu können, ist es fast immer unumgänglich, eine eigene Klasse zu definieren, die von *QObject* oder einer Unterklasse abgeleitet ist. Wir haben bereits in Kapitel 3.1.2, *Das Signal-Slot-Konzept*, in den Beispielen 4 bis 7 eigene Klassen von *QObject* abgeleitet. In dieser Klasse kann man neue Signale und Slots definieren. Ihre neue Klasse muss *QObject* oder eine von ihr abgeleitete Klasse als Basisklasse besitzen. Die erste Zeile der Klassendefinition muss das Makro *Q_OBJECT* enthalten – *ohne* abschließendes Semikolon! Mit diesem Makro werden einige interne Methoden in das Objekt eingefügt, wie zum Beispiel das Meta-Objekt, das die Verwaltung der Signale und Slots übernimmt. Außerdem muss aus der Datei mit der Klassendefinition (meist eine Header-Datei) mit Hilfe des Meta Object Compilers (*moc*) eine zusätzliche Source-Datei erzeugt werden, die beim Kompilieren oder beim Linken mit eingebunden werden muss.

Beachten Sie bei der Definition von Klassen, die von *QObject* abgeleitet sind, zwei wichtige Einschränkungen:

- Bei Mehrfachvererbung darf nur genau eine der Basisklassen *QObject* bzw. eine von *QObject* abgeleitete Klasse sein. Sie können also nicht zwei von *QObject* abgeleitete Klassen durch Mehrfachvererbung miteinander verschmelzen.
- Die erste Basisklasse bei einer Mehrfachvererbung muss die von *QObject* abgeleitete Klasse sein. Erst danach dürfen die anderen Klassen angegeben werden.

Nehmen wir an, Sie hätten eine eigene Klasse *MyClass* geschrieben, die von *QObject* abgeleitet ist. Die Deklaration der Klasse speichern Sie in einer Header-Datei namens *myclass.h*, den Code für die Methoden in der Code-Datei *myclass.cpp*.

Die Header-Datei *myclass.h* sieht beispielsweise so aus:

```
#ifndef _MYCLASS_H_
#define _MYCLASS_H_

#include <qobject.h>

class MyClass: public QObject
{
    Q_OBJECT

public:
    MyClass (QObject *parent=0, const char *name=0);
    ~MyClass ();

signals:
    void moved ();
```

```

    // es folgen weitere Methoden, Signale, Slots, ...
};
#endif

```

Die Code-Datei *myclass.cpp* hat etwa folgendes Aussehen:

```

#include "myclass.h"

MyClass::MyClass (QObject *parent, const char *name)
    : QObject (parent, name)
{
    // Weiterer Konstruktorcode
}

MyClass::~MyClass ()
{
    // Destruktorcode (oft leer)
}

// Code für weitere Methoden und Slots, aber
// nicht für Signale!

```

Wir müssen zur Klassendeklaration in *myclass.h* noch die zugehörige *moc*-Datei erzeugen:

```
% moc myclass.h -o myclass.moc.cpp
```

Mit dieser Zeile auf der Unix-Kommandoebene wird aus den Informationen der Klassendefinition eine Datei mit zusätzlichen Methoden für die Klasse *MyClass* erzeugt, die in der Datei *myclass.moc.cpp* abgelegt wird. Wie Sie diese Datei nennen, ist natürlich Ihnen überlassen, es sollte jedoch aus dem Dateinamen hervorgehen, dass die Datei von *moc* erzeugt worden ist. Immer wenn Sie nun zusätzliche Signale oder Slots in die Definition von *MyClass* einfügen, müssen Sie *moc* erneut ausführen.

Die Datei *myclass.moc.cpp* muss nun noch in das fertige Programm eingebunden werden. Kompilieren Sie es dazu in einem eigenen Schritt:

```
% g++ -c myclass.moc.cpp -I$QTDIR/include
```

Dadurch erzeugen Sie die Datei *myclass.moc.o*.

Nachdem Sie alle anderen Code-Dateien ebenfalls kompiliert haben, linkern Sie alle Objektdateien (auch *myclass.moc.o*) zu einer ausführbaren Datei zusammen:

```
% g++ -o myclass myclass.o myclass.moc.o main.o -lqt
```

Falls Sie vergessen haben, auch die Objekt-Datei der *moc*-Datei einzulinken, erhalten Sie vom Linker in der Regel viele Fehlermeldungen der Art:

```
undefined reference to "MyClass::... virtual table"
```

Bei solchen Fehlern sollten Sie also einmal schauen, ob alle Klassen, die von *QObject* abgeleitet sind, auch mit *moc* bearbeitet wurden und ob alle von *moc* erzeugten Dateien kompiliert und hinzugelinkt wurden.

Es empfiehlt sich, bei etwas größeren Projekten unbedingt ein Makefile zu erstellen, um die Kompilierschritte automatisch ausführen zu lassen. Insbesondere wenn man viele Änderungen am Programm macht, ist es viel zu aufwendig, jedes Mal die Aufrufe von Hand einzutippen.

Wenn Sie das Makefile selbst erstellen, können Sie zum Beispiel folgendermaßen die Abhängigkeit zwischen der Klassendeklaration in *myclass.h* und der daraus erzeugten Datei *myclass.moc.cpp* festlegen:

```
myclass.moc.cpp: myclass.h
    moc myclass.h -o myclass.moc.cpp
```

Ebenso müssen Sie natürlich die Schritte zum Kompilieren und Einbinden der *moc*-Datei mit in das Makefile aufnehmen.

Wenn Sie Ihr Programm mit *tmake* oder mit *automake* / *autoconf* kompilieren lassen, übernehmen diese Tools die Aufgabe für Sie, die *moc*-Dateien zu verwalten und sie bei Bedarf durch einen Aufruf des *moc* neu zu erzeugen (siehe Kapitel 5.1, *tmake*, Kapitel 5.2, *automake* und *autoconf*, und Kapitel 5.5, *KDevelop*). Ich empfehle Ihnen sehr, sich in diese Tools einzuarbeiten. Der Zeitaufwand, den das kostet, beträgt nur ein Bruchteil der Zeit, die Sie sparen werden.

3.1.4 Informationen über die Klassenstruktur

Jede Instanz der Klasse *QObject* oder einer abgeleiteten Klasse besitzt eine Datenstruktur namens *MetaObject*, in der Informationen über die Signale und Slots, aber auch über die Klasse selbst vorhanden sind. Diese Informationen benötigt man für die meisten Programme nicht. Sie sind hier nur der Vollständigkeit halber aufgeführt. Sie können dieses Teilkapitel überspringen und bei Bedarf darauf zurückkommen.

Mit der Methode *className()* kann man sich den Namen der Klasse als C-String (also als Array von *char*-Elementen mit abschließendem »\0«-Zeichen) zurückgeben lassen. Für eine Instanz von *QObject* liefert die Methode den String *QObject* zurück, für die abgeleitete Klasse *QTimer* entsprechend *QTimer*. Die Methode *isA(const char* cn)* vergleicht den übergebenen Klassennamen *cn* mit dem Namen der Klasse und gibt bei Gleichheit den booleschen Wert *true* zurück. Anstatt also zum Beispiel

```
if (strcmp(obj->className(), "QFrame") == 0)
```

zu schreiben, kann man einfacher schreiben:

```
if (obj->isA("QFrame"))
```

Über die Vererbungsstruktur kann man sich mit der Methode *inherits* (*const char *cn*) informieren. Sie liefert *true* zurück, wenn die Klasse des Objekts direkt oder in mehreren Stufen von der Klasse mit dem Namen *cn* abgeleitet ist oder es selbst ein Objekt der Klasse *cn* ist. Das funktioniert natürlich nur, wenn sowohl das Objekt als auch der Klassenname *cn* von *QObject* abgeleitet sind.

Da die zentrale Klasse für Bildschirmobjekte *QWidget* ist, ist die Abfrage, ob ein Objekt von *QWidget* abgeleitet ist, besonders wichtig. Daher gibt es für dieses Problem eine eigene, effizientere Methode: *isWidgetType()* liefert *true* zurück, wenn das Objekt eine Instanz der Klasse *QWidget* oder einer abgeleiteten Klasse ist.

Zu Debugging-Zwecken gibt es noch zwei weitere Methoden: *dumpObjectInfo* und *dumpObjectTree*. Ein Aufruf einer der Methoden gibt Informationen über das Objekt (Name, Klasse, Signal-Slot-Verbindungen, Kindobjekte, Vater-Objekt) auf *stdout* aus.

Weiterhin kann man in der Deklaration der Klasse so genannte *Properties* definieren. Eine Property ist eine Eigenschaft des Objekts, die in der Regel in einer Attributvariablen gespeichert wird und auf die man mit Methoden lesend oder schreibend zugreifen kann.

Betrachten wir als Beispiel die Klasse *QLabel*, die wir bereits im Anfangsbeispiel in Kapitel 2 häufiger benutzt haben. Sie stellt einen Text auf dem Bildschirm dar. Dieser Text ist eine Eigenschaft des Objekts, die mit der Methode *setText* geändert und mit der Methode *text* ausgelesen werden kann. Wie der Text intern gespeichert wird, ist zunächst einmal unwichtig.

In der Deklaration der Klasse *QLabel* wird nun diese Eigenschaft als Property deklariert. Werfen Sie dazu einen Blick in die Datei *qlabel.h* im Verzeichnis *\$QTDIR/include*. Dort finden Sie folgende Zeile:

```
Q_PROPERTY( QString text READ text WRITE setText )
```

Beachten Sie, dass diese Zeile keine Methoden oder Attribute automatisch anlegt. Die Methoden *text* und *setText* und auch das private Attribut, das den Text speichert (in unserem Fall heißt es *ltext*) müssen trotzdem erstellt werden. *Q_PROPERTY* ist sogar in der Tat ein Makro, das den in Klammern eingeschlossenen Ausdruck ignoriert. Diese Zeile ist ausschließlich für den *moc* relevant.

Was bringt es nun aber, eine solche Property zu deklarieren? Wir können uns über die definierten Properties eines Objekts informieren, die Properties auslesen und schreiben, ohne zu wissen, welche Klasse das Objekt genau hat. (Wir müssen natürlich wissen, dass es eine von *QObject* abgeleitete Klasse ist.)

Nehmen wir an, wir hätten einen Zeiger *obj* auf ein Objekt, und die Klasse wäre uns unbekannt. Dann liefert uns folgende Zeile eine Liste aller in dieser Klasse definierten Properties:

```
QStringList properties = obj->metaObject()->propertyNames();
```

Wir können auch Properties des Objekts auslesen oder verändern. Hat unser Objekt beispielsweise eine Property *text* (wie zum Beispiel *QLabel*), so kann man diese mit folgender Zeile ändern:

```
obj->setProperty ("text", "Neuer Text");
```

Zum Lesen und Schreiben von Properties wird der »Universaldatentyp« *QVariant* benutzt (siehe Kapitel 4.7.5, *Flexibler Datentyp – QVariant*).

Wo liegt nun aber das Einsatzgebiet des Property-Systems? Vor allen Dingen in Programmen wie *Qt Designer* (siehe Kapitel 5.4, *Qt Designer*), die einen Entwurf von GUI-Objekten auf dem Bildschirm erlauben. In einem Feld können die Properties eines Objekts tabellarisch aufgelistet werden – mit der Möglichkeit, die Werte zu verändern –, ohne dass das Programm für jede Klasse eine Liste der Properties speichern müsste. Die Objekte selbst geben Auskunft über ihre Properties. Außerhalb dieses Einsatzgebietes wird das Property-System aber kaum genutzt.

Weitere Informationen zum Property-System finden Sie in der Online-Referenz der Qt-Bibliothek.

3.1.5 Übungsaufgaben

Übung 3.1 – Hierarchische Anordnung

Schreiben Sie ein Programmfragment, das die Hierarchie einer Firma aus Abbildung 3.6 mit *QObject* darstellt.

Auf dieser Firmenhierarchie wollen wir eine Reihe von Operationen ausführen:

- Wie können Sie mit Hilfe des Textstreams *cout* den Namen von *obj6* ausgeben?
- Wie geben Sie den Namen der direkten Vorgesetzten von *obj8* aus?
- Schreiben Sie eine Funktion, die zu einem Objekt den höchsten Vorgesetzten findet (in unserem Beispiel also immer Herrn Direktor Klöbner).
- Schreiben Sie eine Funktion, die zu einem Objekt alle direkten Untergebenen ausgibt.
- Schreiben Sie eine Funktion (am einfachsten rekursiv), die zu einem Objekt *alle* Untergebenen (direkte und indirekte) ausgibt.

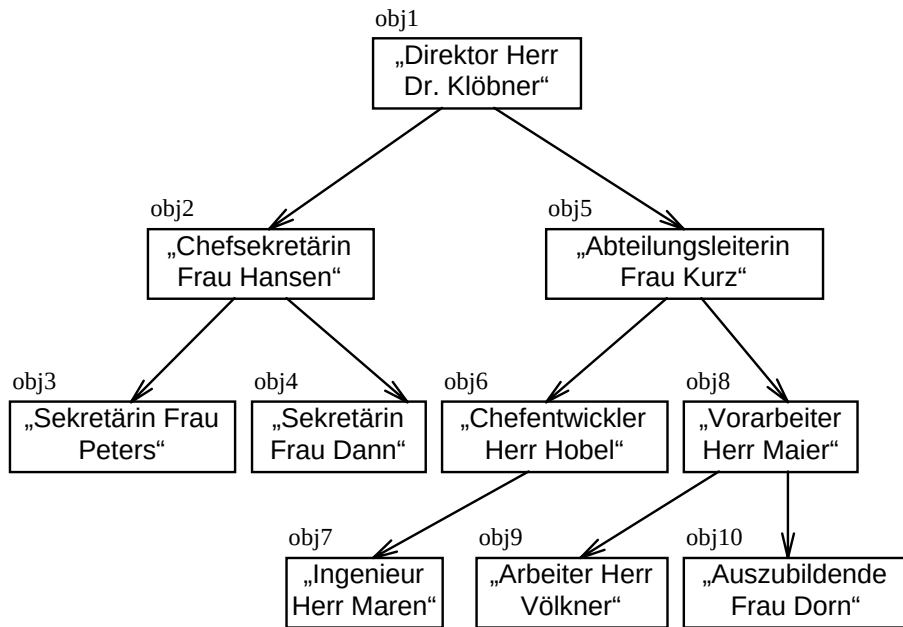


Abbildung 3-6 Hierarchiestruktur einer Firma

- f) Wie kann man die Objekte *obj6* und *obj7* gleichzeitig aus der Hierarchie entfernen?
- g) Gibt es eine einfache Möglichkeit, die Hierarchie zu aktualisieren, wenn Frau Hansen in den wohlverdienten Ruhestand geht und nun Frau Peters die Aufgaben der Chefsekretärin übernimmt?

Übung 3.2 – Signal-Slot-Konzept

Was passiert, wenn man in Kapitel 3.1.2, *Das Signal-Slot-Konzept*, in Beispiel 4 den *connect*-Befehl im Hauptprogramm zweimal hintereinander ausführt? Was passiert, wenn man anschließend die *disconnect*-Methode aufruft?

Übung 3.3 – Signal-Slot-Konzept

Was passiert bei einem Aufruf der folgenden Zeile?

```
connect (&obj1, SIGNAL (send (int, QString *)),
        &obj1, SIGNAL (send (int, QString *)));
```

Übung 3.4 – Signal-Slot-Konzept

Auf wie viele verschiedene Arten kann man eine Signal-Slot-Verbindung wieder löschen, vorausgesetzt, es ist die einzige Verbindung des Objekts *sender*?

Übung 3.5 – Signal-Slot-Konzept

Kann es Sinn machen, eine Slot-Methode als *inline* zu deklarieren?

Übung 3.6 – Informationen über die Klassenstruktur

In der Klassenhierarchie sind *QWidget* und *QTimer* von *QObject* abgeleitet. *QFrame* ist seinerseits von *QWidget* abgeleitet. Folgende Zeigervariablen seien definiert und zeigen auf Objekte des entsprechenden Typs:

```
QObject *object;  
QTimer *timer;  
QWidget *widget;  
QFrame *frame;
```

Welche Werte haben demnach die folgenden Ausdrücke?

```
object->isA ("QFrame")  
timer->isA ("QTimer")  
frame->isA ("QWidget")  
widget->isA ("QWidget")  
  
frame->inherits ("QWidget")  
frame->inherits ("QObject")  
widget->inherits ("QFrame")  
timer->inherits ("QTimer")  
timer->inherits ("QObject")  
  
timer->isWidgetType()  
frame->isWidgetType()
```

3.2 Die Fensterklasse – *QWidget*

Die Darstellung auf einem X-Terminal geschieht grundsätzlich in Fenstern, also in rechteckigen Bildschirmbereichen mit bestimmter Position und Größe, die sich überlappen können. Diese Fenster des X-Servers werden in Qt durch die Klasse *QWidget* erzeugt. Jedes dieser so genannten *Widgets* ist ein Objekt der Klasse *QWidget*.

Widgets können hierarchisch angeordnet werden. Diese Anordnung geschieht über den Hierarchiemechanismus der Klasse *QObject* (siehe Kapitel 3.1, *Die Basis-Klasse – QObject*). Da *QWidget* von *QObject* abgeleitet ist, kann man im Konstruktor einen Parameter *parent* angeben. Dadurch kann man eine baumartige Hierarchie darstellen. Jedes Widget wird innerhalb seines Vater-Widgets gezeichnet, und seine Position ist relativ zur oberen linken Ecke des Vater-Widgets angegeben. Der Typ des Parameters *parent* ist *QWidget*; ein Widget kann also nur ein anderes Widget als Vater haben, andere Objekte der Klasse *QObject* sind hier nicht erlaubt.

3.2.1 Toplevel-Widgets

Ein Widget ohne Vater (also mit *parent* = 0) ist ein so genanntes Toplevel-Widget. Ein solches Toplevel-Widget ist das, was ein Anwender in der Regel als Fenster bezeichnet: ein rechteckiger Bildschirmbereich, dem vom Window-Manager eine Dekoration mitgegeben wird, also ein Rahmen, an dem man die Größe des Fensters ändern kann, eine Titelzeile mit einem Namen und mehreren Buttons zum Maximieren, Minimieren und Schließen des Fensters. Eine Zeile zur Erzeugung eines Toplevel-Widgets kann beispielsweise so aussehen:

```
QWidget *myTopWidget = new QWidget ();
```

Da der Default-Wert für den *parent*-Parameter 0 ist, braucht nichts angegeben zu werden.

Toplevel-Widgets werden zunächst nicht angezeigt. Man muss die Methode *show* aufrufen, um sie anzeigen zu lassen. Mit der Methode *hide* kann man sie wieder verschwinden lassen. Sie werden dabei aber nicht gelöscht, sondern lediglich nicht mehr angezeigt. Ein erneuter Aufruf von *show* macht sie wieder sichtbar.

Ein Minimalprogramm (*widget.cpp*) zur Erzeugung eines einzelnen Toplevel-Widgets sieht beispielsweise so aus:

```
#include <qapplication.h>
#include <qwidget.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    QWidget *w = new QWidget ();
    app.setMainWidget (w);

    w->resize (300, 100);
    w->setCaption ("QWidget Example");

    w->show ();

    return app.exec ();
}
```

Kompilieren und linken Sie das Programm wie üblich, also mit dem Befehl:

```
% g++ -o widget -I$QTDIR/include -lqt widget.cpp
```

(Eventuell müssen Sie wieder die Position der Qt-Bibliothek mit *-L\$QTDIR/lib* zusätzlich angeben.) Rufen Sie anschließend die Datei *widget* auf. Das Ergebnis ähnelt dem Widget in Abbildung 3.7, hängt aber auch etwas vom verwendeten X-Server und Window-Manager ab.

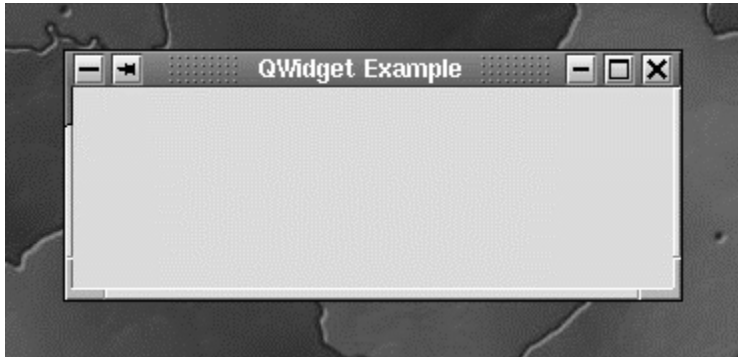


Abbildung 3-7 Ein einfaches QWidget-Objekt

Die Klasse `QWidget` ist in `QWidget.h` definiert, daher sollte diese Header-Datei eingebunden werden. Meist ist dies jedoch nicht nötig, da `QWidget.h` in sehr vielen anderen Header-Dateien eingebunden ist, zum Beispiel auch in `QApplication.h`.

Bevor ein Widget erzeugt werden kann, muss ein Objekt der Klasse `QApplication` erzeugt werden. Dieses Objekt baut eine Verbindung zum X-Server auf und übernimmt die Kontrolle über alle Widgets (siehe Kapitel 3.3, *Grundstruktur einer Applikation*).

Anschließend wird ein `QWidget`-Objekt auf dem Heap angelegt. Es wird jedoch noch nicht sofort angezeigt. Mit der Methode `resize` wird die Größe des Widgets auf 300 x 100 Pixel festgelegt. Diese Größe gilt für den inneren Widget-Bereich. Die Fensterdekoration des Window-Managers kommt also noch zusätzlich hinzu. Mit der Methode `setCaption` kann die Bezeichnung des Widgets festgelegt werden. Diese Bezeichnung wird nur bei Toplevel-Widgets benutzt und legt die Fensterbezeichnung fest, die der Window-Manager in der Titelseile anzeigt. Die Fensterbezeichnung ist nicht das gleiche wie der Objektname, der im Konstruktor angegeben werden kann (und der von `QObject` geerbt wird).

Erst nach dem Aufruf der Methode `show` öffnet sich das Fenster auf dem Bildschirm. Dieses Fenster kann bereits in der Größe geändert und verschoben, maximiert, minimiert und geschlossen werden. Beim Schließen wird die `hide`-Methode aufgerufen, so dass das Fenster von Bildschirm verschwindet. Es wird jedoch zunächst nicht gelöscht. Da es aber als Hauptfenster in `app` eingetragen ist (`app.setMainWidget(w)`), wird die Hauptschleife in `app.exec()` beendet. Daraufhin wird zunächst das Objekt `app` gelöscht und als Folge davon auch unser Widget `w`.

3.2.2 Unter-Widgets

Widgets, in deren Konstruktor als *parent* ein anderes Widget angegeben ist, werden nur innerhalb dieses Widgets dargestellt. Sie bekommen vom Window-

Manager keine Dekoration, also keinen Rahmen und keine Titelzeile. Ihre Position wird relativ zur oberen linken Ecke des Vater-Widgets angegeben, und wenn das Vater-Widget bewegt wird, bewegt sich das untergeordnete Widget mit.

Ein einfaches Beispiel soll das veranschaulichen. In unserem Beispiel von oben fügen wir jetzt zwei Unter-Widgets, *sub1* und *sub2*, in das Toplevel-Widget *w* ein. Dazu brauchen wir im Konstruktor nur *w* als *parent* für die beiden neuen Objekte eintragen. Da Unter-Widgets keine Dekoration bekommen, also nur aus einem grauen Rechteck bestehen, könnte man die Unter-Widgets im derzeitigen Zustand gar nicht erkennen. Daher benutzen wir hier Objekte der Klasse *QFrame*, einer von *QWidget* abgeleiteten Klasse. Diese Klasse zeichnet einen Rahmen an die äußere Kante (aber innerhalb des Widgets), so dass wir die Position der Widgets auch sehen.

Unser Beispielprogramm sieht nun so aus (neue Zeilen sind fett gedruckt):

```
#include <qapplication.h>
#include <qwidget.h>
#include <qframe.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    QWidget *w = new QWidget ();
    app.setMainWidget (w);
    w->resize (300, 100);
    w->setCaption ("QWidget Example");

    QFrame *sub1 = new QFrame (w);
    QFrame *sub2 = new QFrame (w);
    sub1->setFrameStyle (QFrame::Box | QFrame::Plain);
    sub2->setFrameStyle (QFrame::Box | QFrame::Plain);
    sub1->setGeometry (40, 20, 150, 50);
    sub2->setGeometry (130, 40, 150, 40);

    w->show ();
    return app.exec ();
}
```

Das neue Programmstück erzeugt die beiden Unter-Widgets. Durch die Angabe des Vaters *w* im Konstruktor werden sie Unter-Widgets von *w*. Die nächsten beiden Zeilen legen die Rahmenart fest. Die letzten beiden Zeilen legen die Position und Größe der Unter-Widgets fest. Die ersten beiden Parameter geben die Position der oberen linken Ecke an (relativ zum Vater-Widget), die letzten beiden Parameter die Breite und Höhe des Widgets. Statt des einen *setGeometry*-Befehls hätte man auch zwei Befehle benutzen können:

```
sub1->move (40, 20);  
sub1->resize (150, 150);
```

Das Ergebnis dieses Programms ist in Abbildung 3.8 dargestellt.

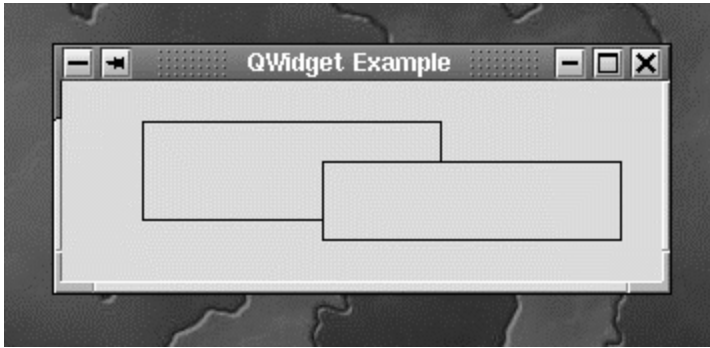


Abbildung 3-8 Zwei Unter-Widgets

Man erkennt, dass sich die beiden Unter-Widgets überlappen, wobei das zweite Widget, *sub2*, das erste, *sub1*, teilweise überdeckt. Es gilt ganz allgemein, dass das zuletzt eingefügte Widget die älteren Widgets überdeckt. Widgets werden beim Einfügen also immer »oben auf die anderen Widgets« gelegt. Diese Reihenfolge kann man jedoch verändern. Wenn wir vor die Zeile *w->show()*; folgende Zeile einfügen

```
sub1->raise ();
```

ergibt sich das Bild aus Abbildung 3.9.

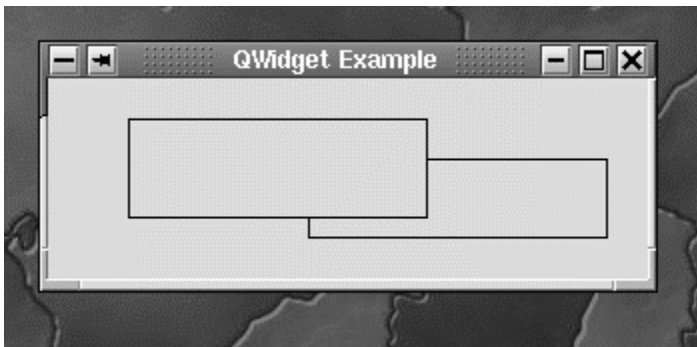


Abbildung 3-9 Vertauschung der Reihenfolge mit *raise*

Hier ist das Widget *sub1* wieder ganz oben auf die anderen Widgets gelegt worden. Umgekehrt kann man mit der Methode *lower* ein Widget ganz nach unten legen.

Dialoge sind in KDE und Qt fast immer so aufgebaut, dass ein `Toplevel-Widget` ein Fenster erzeugt, in das viele andere Widgets mit vordefiniertem Verhalten (zum Beispiel Buttons, Eingabefelder, Listboxen usw.) als Unter-Widgets eingefügt werden. Das Dialogfenster in Abbildung 3.10 besteht zum Beispiel aus einem `Toplevel-Widget`, einem Objekt der Klasse `QMultiLineEdit` sowie zwei Objekten der Klasse `QPushButton`.



Abbildung 3-10 Dialogfenster mit drei Unter-Widgets

`QMultiLineEdit` und `QPushButton` sind Klassen, die von `QWidget` abgeleitet sind, die also die Funktionalität der Widgets enthalten. Das folgende Programm könnte beispielsweise dieses Fenster erstellen:

```
#include <qapplication.h>
#include <qwidget.h>
#include <qpushbutton.h>
#include <qmultilineedit.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    // Toplevel-Widget anlegen, Größe setzen
    QWidget *messageWindow = new QWidget ();
    app.setMainWidget (messageWindow);
    messageWindow->resizeSize (220, 150);

    // Drei Unter-Widgets anlegen, Größe setzen
    QMultiLineEdit *messages =
        new QMultiLineEdit (messageWindow);
    messages->setGeometry (10, 10, 200, 100);
    QPushButton *clear =
        new QPushButton ("Clear", messageWindow);
    clear->setGeometry (10, 120, 95, 20);
    QPushButton *hide =
```

```

        new QPushButton ("Hide", messageWindow);
hide->setGeometry (115, 120, 95, 20);

// noch ein paar Eigenschaften festlegen
messageWindow->setCaption ("einfacher Dialog");
messages->setReadOnly (true);
// ersten Eintrag in das Fenster einfügen
messages->append ("Initialisierung abgeschlossen\n");

// Toplevel-Widget anzeigen
messageWindow->show ();

return app.exec ();
}

```

Ein Nachteil dieses Programms besteht darin, dass die Position der Unter-Widgets mit absoluten Koordinaten festgelegt wird. Zum einen bedeutet dies viel Aufwand für den Programmierer, diese Zahlen zu bestimmen, zum anderen ist das Fenster auf diese Weise sehr unflexibel; es passt sich zum Beispiel nicht der Größe des Gesamtfensters an. Wie Sie Probleme, die daraus entstehen, vermeiden können, wird in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*, besprochen.

Die Bibliotheken von KDE und Qt enthalten eine Vielzahl von fertigen Widgets. Eine Auflistung befindet sich im Kapitel 3.7, *Überblick über die GUI-Elemente von Qt und KDE*. Wie man Dialoge aus diesen Widgets zusammenstellt, wird in Kapitel 3.8, *Der Dialogentwurf* beschrieben. Eine Anleitung für den Entwurf eigener Widgets, die Bedienelemente realisieren sollen, gibt Kapitel 4.4, *Entwurf eigener Widget-Klassen*.

3.2.3 Die wichtigsten Widget-Eigenschaften

Die Klasse *QWidget* definiert eine sehr große Anzahl von Methoden, mit denen das Verhalten der Widgets gesteuert werden kann. Für viele der Eigenschaften existiert ein Paar von Methoden: Eine Methode setzt die Einstellung auf einen neuen Wert (die Methoden beginnen meist mit »set«, z.B. *setMinimumSize*, *setUpdatesEnabled*), während eine andere mit fast identischem Namen die aktuelle Einstellung ausliest (bei diesen Methoden entfällt meist das »set« oder sie beginnen mit »is«, wenn es sich um einen booleschen Wert handelt, z.B. *minimumSize*, *isUpdatesEnabled*). Diese Paarbildung von Methoden ist übrigens bei allen Klassen von Qt verbreitet und wird sehr konsequent durchgehalten. Dadurch kann man sich die Methodennamen einer Klasse leichter merken.

Methoden für den Modus des Widgets

Ein Widget kann mit der Methode *show* dargestellt und mit *hide* wieder versteckt werden. Ist ein Widget versteckt, so werden auch seine Unter-Widgets (und

ebenso deren Unter-Widgets usw.) nicht dargestellt. Standardmäßig sind Toplevel-Widgets nach dem Erzeugen versteckt, Unter-Widgets (also Widgets, die ein Vater-Widget haben) werden dargestellt. Erzeugt man also ein Dialogfenster aus einem Toplevel-Widget mit vielen Unter-Widgets als Bedienelemente, so erscheinen beim Aufruf von *show* für das Toplevel-Widget auch alle Unter-Widgets. Ruft man anschließend *hide* für das Toplevel-Widget auf, so werden auch alle Bedienelemente versteckt. Einzelne Bedienelemente kann man verstecken, indem man die *hide*-Methode des Unter-Widgets aufruft.

Ob ein Widget versteckt oder dargestellt wird, kann man mit der Methode *isVisible* erfragen. Sie liefert *true* zurück, wenn das Widget und alle Vater-Widgets bis zum Toplevel-Widget nicht mit *hide* versteckt sind. Das Fenster könnte trotzdem unsichtbar sein, wenn es von einem anderen Fenster verdeckt wird oder wenn seine Koordinaten außerhalb des Fensters des Vater-Widgets liegen. Ist das Toplevel-Fenster minimiert oder liegt es auf einem anderen virtuellen Desktop als dem aktuellen, so liefert *isVisible* ebenfalls *false*.

Wie bereits oben gezeigt wurde, kann man die Reihenfolge von Unter-Widgets mit den Methoden *raise* und *lower* ändern. Dadurch ändert sich die Reihenfolge beim gegenseitigen Überdecken. Angewandt auf ein Toplevel-Widget bewirkt *raise*, dass das Fenster über allen anderen Fenstern dargestellt wird, diese also verdeckt, während *lower* das Fenster unter die anderen Fenster schiebt.

Mit der Methode *close* kann ein Fenster geschlossen werden. *close* schließt das Fenster jedoch nicht sofort, sondern sendet einen *close*-Event an das Widget (siehe auch Kapitel 3.4, *Hintergrund: Event-Verarbeitung*). Wie darauf zu reagieren ist, kann ein Widget festlegen, indem es die virtuelle Methode *closeEvent* überschreibt. Dort kann das Fenster den Aufruf zum Schließen abschmettern. Die Default-Implementierung von *QWidget* akzeptiert aber den Event. Daraufhin wird das Fenster mit *hide* versteckt, aber nicht gelöscht. Ruft man allerdings *close* (*true*) auf, wird das Widget nach dem Schließen mit *delete* gelöscht (sofern der *close*-Event nicht abgeschmettert wurde).

Ein Widget ist nach der Erzeugung automatisch *enabled*, d.h. es kann Mausklicks und Tastatureingaben entgegennehmen. Mit *setEnabled* (*false*) kann man das Widget in den Zustand *disabled* versetzen. Viele Widgets verändern in diesem Zustand auch ihr Aussehen. Ein Button wird beispielsweise im Zustand *disabled* in grauer Schrift dargestellt – als Zeichen dafür, dass er zur Zeit nicht aktiviert werden kann. Mit *setEnabled* (*true*) kann man das Widget wieder aktivieren. Ganz ähnlich wie bei *show* und *hide* ist ein Widget auch dann automatisch *disabled*, wenn sein Vater-Widget (oder dessen Vater-Widget usw.) *disabled* ist. *isEnabled* liefert *true* zurück, wenn das Widget und alle Vater-Widgets im Zustand *enabled* sind.

Wie bereits in Kapitel 3.1.1, *Hierarchische Anordnung der Objektinstanzen von QObject*, erwähnt wurde, besitzt *QWidget* (ebenso wie alle abgeleiteten Klassen) die Möglichkeit, das Vater-Widget neu festzulegen. Bei allen anderen Klassen, die von *QObject* abgeleitet sind (inklusive *QObject* selbst), ist das nicht möglich. Man benutzt dazu die Methode *reparent*, die als Parameter den neuen Vater (bzw. 0, wenn es ein *Toplevel-Widget* werden soll) sowie neue Dekorationsflags (siehe *WFlags* im Abschnitt *Eigenschaften von Toplevel-Widgets* weiter unten) und eine neue Position innerhalb des neuen Vaters bekommt. Mit dieser Methode kann man ein Widget mitsamt seinen Unter-Widgets im Widget-Baum verschieben, ohne dass sich die Größe oder der Inhalt des Widgets ändert. Auf diese Weise funktionieren zum Beispiel auch die verschiebbaren Menüleisten und Werkzeugleisten in KDE-Programmen. Sie sind normalerweise Unter-Widgets des Hauptfensters, können aber zu *Toplevel-Widgets* werden, wenn man sie aus dem Fenster herauszieht, und können wieder Unter-Widgets werden, wenn sie in das Fenster zurückkehren. Dennoch sollte es eine absolute Ausnahme bleiben, dass Widgets mit *reparent* in der Hierarchie versetzt werden.

Fenstergröße und -position

Für das Rechnen mit Koordinaten stellt Qt eine Reihe von Klassen zur Verfügung, die häufig benutzt werden, auch als Parameter und Rückgabetypen in Methoden. Tabelle 3.3 zeigt die drei wichtigsten Klassen. Diese Klassen besitzen eine Vielzahl von Methoden und überladenen Operatoren zum Umgang mit den Koordinaten. Eine Liste der Koordinaten finden Sie in der Online-Referenz zur Qt-Bibliothek.

Klasse	Inhalt	Bedeutung
<i>QPoint</i>	2 <i>int</i> -Werte	Koordinaten eines Punktes
<i>QSize</i>	2 <i>int</i> -Werte	Größe eines rechteckigen Bereichs
<i>QRect</i>	4 <i>int</i> -Werte	Position und Größe eines rechteckigen Bereichs

Tabelle 3-3 Die wichtigsten Koordinaten-Klassen von Qt

Ein Widget repräsentiert immer einen rechteckigen Bildschirmbereich. Es gibt daher eine ganze Reihe von Methoden, mit denen man die Größe und Position dieses Bereichs abfragen und festlegen kann. Drei Methoden haben wir bereits weiter oben kennen gelernt: *move*, *resize* und *setGeometry*. Mit ihnen kann man die Position, die Größe oder beides festlegen. Angewendet auf *Toplevel-Widgets* ist zu beachten, dass die Widget-Größe *nicht* die Fensterdekoration mit einschließt. Die Position von *Toplevel-Widgets* muss nicht unbedingt der angegebenen Position entsprechen. Die geforderten Koordinaten sind nur ein Hinweis für den Window-Manager, sie sind für ihn jedoch nicht verbindlich.

Die Größe des Widgets kann man mit den Methoden *size*, *height* und *width* abfragen. Die Position ermittelt man mit den Methoden *pos*, *x* und *y*. Die Größe und Position in einer Methode liefern die beiden Methoden *frameGeometry* und

geometry zurück. Für Unter-Widgets sind diese beiden Methoden identisch; für Toplevel-Widgets liefert *geometry* die Größe und Position des Widgets *ohne* Dekoration, *frameGeometry* liefert dagegen die Werte *mit* Dekoration. (Eine genaue Beschreibung der zurückgegebenen Werte finden Sie in der Online-Referenz zur Qt-Bibliothek in der Datei *geometry.html*.)

Man kann Widgets auf eine minimale und maximale Breite bzw. Höhe beschränken. Dazu gibt es die Methoden *setMinimumSize* und *setMaximumSize*. Die Widget-Größe kann dann nur innerhalb dieser erlaubten Werte liegen. Standardmäßig ist die minimale Größe ein 0x0 Pixel großes Fenster, die maximale Größe 32.767x32.767 Pixel (also praktisch unbeschränkt). Ruft man *resize* oder *setGeometry* mit einer unerlaubten Größe auf, so wird die Größe zunächst auf ein erlaubtes Maß gesetzt (auf die minimale Breite oder Höhe, falls diese unterschritten wurde, oder auf die maximale Breite oder Höhe, falls diese überschritten wurde) und erst dann die Größenänderung durchgeführt. Für Toplevel-Widgets bedeutet dies insbesondere, dass das Fenster vom Benutzer am Rahmen nicht auf eine unerlaubte Größe verändert werden kann. So gewährleistet man, dass der Inhalt des Fensters immer lesbar bleibt. Mit der Methode *setFixedSize* wird die minimale und die maximale Größe auf den gleichen Wert festgelegt. Dadurch kann das Widget nur diese eine Größe haben, also nicht mehr verändert werden. Neben diesen Methoden gibt es auch noch die Methoden *setMinimumWidth*, *setMinimumHeight*, *setMaximumWidth*, *setMaximumHeight*, *setFixedWidth* und *setFixedHeight*, die jeweils nur eine Ausdehnung des Widgets beschränken. Die Beschränkung der anderen Ausdehnung bleibt unverändert. Mit den Methoden *minimumSize* und *maximumSize* sowie *minimumWidth*, *maximumWidth*, *minimumHeight* und *maximumHeight* kann man die Beschränkung auslesen.

Die Methode *childrenRect* berechnet das kleinste Rechteck, das alle Unter-Widgets enthält. Ist dieses Rechteck vollständig im Widget-Rechteck enthalten, sind alle Unter-Widgets sichtbar. *adjustSize* ändert die Größe des Widgets so, dass alle Unter-Widgets sichtbar sind.

Die Methode *sizeHint* sollte für jede Unterklasse von *QWidget* geeignet überschrieben sein. Sie liefert die bevorzugte Größe des Fensters zurück. Bei einem Button ist das zum Beispiel die Größe, bei der die Beschriftung vollständig lesbar ist. Die Default-Implementierung von *QWidget* liefert eine negative und daher ungültige Größe zurück, was bedeutet, dass das Widget eine beliebige Größe annehmen kann. Die Methode *sizePolicy* liefert dagegen ein *QSizePolicy*-Objekt zurück. In diesem sind Angaben darüber enthalten, wie sinnvoll eine Abweichung der Widget-Größe von *sizeHint* ist, ob das Widget beispielsweise kleiner sein darf, ohne unbenutzbar zu werden, oder ob es sinnvollerweise so groß wie möglich wird. Diese Methoden werden insbesondere beim automatischen Layout benutzt (siehe Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*), um einen Anhaltspunkt für eine geeignete Größe des Widgets zu bekommen.

Mit den Methoden *mapToParent* und *mapFromParent* kann man Fensterkoordinaten dieses Widgets in Koordinaten des Vater-Widgets umwandeln lassen bzw. umgekehrt. *mapToGlobal* und *mapFromGlobal* wandeln Fensterkoordinaten (bezogen auf die linke obere Ecke des Widgets) in Bildschirmkoordinaten (bezogen auf die linke obere Bildschirmecke) um.

Darstellung des Widgets

Ein Widget der Klasse *QWidget* stellt auf dem Bildschirm nur die Hintergrundfarbe dar. Die Unter-Widgets werden automatisch dargestellt, darum braucht sich ein Widget nicht zu kümmern. Wenn man also auf dem Bildschirm etwas darstellen will, muss man eine neue Klasse definieren, die *QWidget* als Basisklasse besitzt. In dieser muss man die virtuelle Methode *paintEvent* überschreiben und den Code einfügen, der das Zeichnen auf dem Bildschirm bewirkt. Genauer es dazu wird in Kapitel 3.4, *Hintergrund: Event-Verarbeitung*, Kapitel 4.2, *Zeichnen von Grafikprimitiven*, und Kapitel 4.4, *Entwurf eigener Widget-Klassen*, erläutert. Die Routine *paintEvent* wird immer aufgerufen, wenn das Widget oder ein Teil davon neu gezeichnet werden muss, z.B. weil das Fenster zum ersten Mal mit *show* angezeigt wird oder weil das Widget vorher verdeckt war und nun aufgedeckt wurde.

Man kann das Neuzeichnen des Widgets aber auch erzwingen, wenn sich zum Beispiel die Daten geändert haben, die dargestellt werden sollen.

repaint ruft die Methode *paintEvent* direkt auf, zeichnet also das Widget unmittelbar neu. Dabei kann man einen Ausschnitt angeben, der neu gezeichnet werden soll. Wird kein Ausschnitt angegeben, wird das ganze Widget neu gezeichnet. In einem weiteren Parameter, *erase*, kann man angeben, ob die neu zu zeichnende Fläche vorher mit der Hintergrundfarbe ausgefüllt werden soll. Ist das nicht nötig (weil *paintEvent* die ganze Fläche »bemalt«), kann man *erase* auf *false* setzen und so ein kurzes Flackern auf dem Bildschirm verhindern (siehe auch Kapitel 4.5, *Flimmerfreie Darstellung*). Vom Neuzeichnen sind die Unter-Widgets nicht betroffen, sie bleiben unverändert erhalten und werden auch nicht neu gezeichnet.

Im Gegensatz zu *repaint* führt ein Aufruf von *update* nicht unmittelbar die *paintEvent*-Methode aus. Stattdessen wird in die Qt-interne Event-Schlange ein Event eingefügt, der das Neuzeichnen durch *paintEvent* bewirkt, sobald die Kontrolle wieder in die Haupt-Event-Schleife gelangt (siehe Kapitel 3.4, *Hintergrund: Event-Verarbeitung*). Der Vorteil ist, dass mehrere *update*-Aufrufe von Qt automatisch zu einem Event zusammengefasst werden können, was den Aufwand des Neuzeichnens zum Teil erheblich reduzieren kann. Wenn Sie sich also nicht sicher sind, ob eventuell im weiteren Verlauf noch ein paar zusätzliche Befehle zum Neuzeichnen anfallen, benutzen Sie besser *update*. Auch bei *update* können Sie einen rechteckigen Bereich angeben, der gefüllt werden soll. Wenn Sie nichts

angeben, wird das ganze Widget neu gezeichnet. Der zu zeichnende Bereich wird in jedem Fall vorher mit der Hintergrundfarbe ausgefüllt. Auch hier bleiben die Unter-Widgets unverändert.

Aus Effizienzgründen kann es manchmal sinnvoll sein, das Neuzeichnen durch *repaint* oder *update* vorübergehend abzuschalten. Ein Widget der Klasse *QListBox* enthält beispielsweise eine Reihe von Einträgen. Beim Hinzufügen eines Eintrags wird automatisch ein *repaint* ausgelöst, um die Darstellung aktuell zu halten. Sollten jetzt Hunderte von Einträgen auf einen Schlag hinzugefügt werden, so würde jedes Mal das Widget neu gezeichnet, was sehr ineffizient wäre. Mit *setUpdatesEnabled (false)* kann man das Neuzeichnen unterbinden. Nach dem Einfügen der Einträge kann man dann mit *setUpdatesEnabled (true)* das Neuzeichnen wieder aktivieren und mit *repaint* ein Neuzeichnen erzwingen, um die Darstellung wieder auf den neuesten Stand zu bringen.

Der Hintergrund eines Widgets kann definiert werden. Die Default-Einstellung benutzt die Palettenfarbe *background* als Füllfarbe. Mit der Methode *setBackgroundMode* kann ein anderer Paletteneintrag gewählt werden. Mit *setBackgroundColor* kann eine beliebige Farbe – spezifiziert durch ein Objekt der Klasse *QColor* – als Hintergrundfarbe gewählt werden. Mit der Methode *setBackgroundPixmap* lässt sich schließlich sogar ein beliebiges Bild als Hintergrund verwenden. Wenn das Bild das Widget nicht ausfüllt, wird es entsprechend oft weiter rechts bzw. unten wiederholt; es wird also gekachelt. Abbildung 3.11 zeigt nochmals unser Widget aus dem Einführungsbeispiel der Unter-Widgets, wenn man die folgende Zeile in das Programm einfügt:

```
w->setBackgroundPixmap (QPixmap ("Circuit.jpg"));
```

(Außerdem müssen Sie die Header-Datei *qpixmap.h* einbinden.) Kopieren Sie dazu vorher die Datei *\$KDEDIR/share/wallpapers/Circuit.jpg* in das Verzeichnis mit der ausführbaren Datei. (Alternativ können Sie in der Zeile im Programm auch den absoluten Pfad zur Datei angeben.) Das Ergebnis sollte wie in Abbildung 3.11 aussehen. Falls Sie kein Hintergrundbild im Widget erhalten, kann das daran liegen, dass die Unterstützung des JPEG-Formats nicht in die Qt-Bibliothek eingebunden ist. Versuchen Sie stattdessen, eine XPM- oder PNG-Datei zu benutzen.

Jedes Widget hat außerdem noch einige Eigenschaften, die beim Zeichnen innerhalb des Widgets standardmäßig benutzt werden. Zum einen enthält jedes Widget eine Farbpalette, in der drei Tabellen mit je 14 Einträgen gespeichert sind. (Verwechseln Sie diese »Palette« nicht mit der Color-Lookup-Table, die bei Grafikkarten mit nur 256 darstellbaren Farben benutzt wird.) Diese Palette ist normalerweise für das ganze Programm einheitlich. Sollen einige Widgets allerdings in anderen Farben dargestellt werden – zum Beispiel ein Button mit roter Schrift –, so kann man ihnen mit *setPalette* eine eigene Palette zuordnen, die dann beim Zeichnen benutzt wird. Mit der Methode *setPalettePropagation (AllChildren)* kann

man dabei erreichen, dass die neue Palette auch in allen Unter-Widgets verwendet wird. Andere Werte für *setPalettePropagation* sind *NoChildren* (benutzt die Palette nur in diesem Widget; das ist die Default-Einstellung) und *SamePalette* (wendet die Palette auf alle Unter-Widgets an, die die gleiche Palette besitzen).

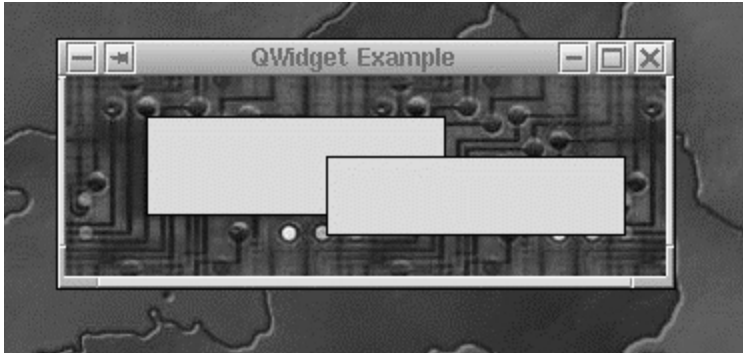


Abbildung 3-11 *setBackgroundPixmap (QPixmap (>Circuit.jpg<))*

Die aktuell eingestellte Palette kann man mit *palette* in Erfahrung bringen. Der Palettenbereich, der im aktuellen Zustand des Widgets für das Zeichnen verantwortlich ist (abhängig davon, ob das Widget gerade im Zustand *enabled* ist und ob es gerade den Tastaturfokus hat), kann mit *colorGroup* ermittelt werden.

Nähere Informationen über die Farbverwaltung von Qt und die Benutzung der Farbpalette finden Sie in Kapitel 4.1, *Farben unter Qt*.

Jedes Widget hat außerdem einen Standardzeichensatz, der zum Darstellen von Text benutzt wird. Dieser kann mit *font* ermittelt und mit *setFont* neu gesetzt werden. Bei *setFont* wird wie bei der Palette der Font an die Unter-Widgets weitergegeben, wenn man vorher *setFontPropagation (AllChildren)* aufruft. Die anderen möglichen Werte sind entsprechend *NoChildren* und *SameFont*.

Jedem Widget kann weiterhin ein Maus-Cursor zugeordnet werden, der benutzt wird, sobald der Mauszeiger das Widget-Fenster betritt. Der Default-Cursor der *QWidget*-Klasse ist der normale Pfeil nach oben links, einige Widgets haben aber spezielle Cursor. So hat beispielsweise ein Eingabefeld einen so genannten I-Beam-Cursor, der die Gestalt eines großen »I« hat. Mit der Methode *setCursor* kann man jedem Widget einen anderen Maus-Cursor zuordnen. Nähere Informationen über Cursor-Formen finden Sie in Kapitel 6, Klasse *QCursor*.

Tastaturfokus

Dialogfenster enthalten oft viele Eingabeelemente wie z.B. Buttons, Schieberegler oder Eingabezeilen. Dieses wird in Qt oft realisiert durch ein *QWidget*-Objekt, das die Eingabeelemente als Unter-Widgets enthält. Nur eines der Eingabeelemente

kann aber zu einem bestimmten Zeitpunkt die Eingaben von der Tastatur zugeordnet bekommen. Diesen Zustand nennt man den Tastaturfokus. Der Tastaturfokus kann mit der -Taste auf das nächste Eingabeelement geschoben werden oder mit  +  auf das vorhergehende. Ein Anklicken mit der Maus legt den Tastaturfokus in das angeklickte Eingabeelement. Die meisten Eingabeelemente ändern auch ihr Aussehen, wenn sie den Tastaturfokus erhalten. Ein Button stellt seine Beschriftung zum Beispiel mit einem gestrichelten Rechteck dar, eine Eingabezeile stellt einen blinkenden Cursor dar usw.

Ob ein Eingabeelement gerade den Tastaturfokus besitzt, kann mit der Methode *hasFocus* erfragt werden. Mit der Methode *setFocus* kann man den Fokus auch auf ein Element setzen.

Nicht alle Widgets benötigen den Tastaturfokus. Reine Anzeige-Widgets oder Widgets, die nur Unter-Widgets haben, reagieren nicht auf Tastatureingaben. Mit der Methode *setFocusPolicy* kann man festlegen, ob und auf welche Art ein Widget den Fokus bekommen soll. Mit *setFocusPolicy (QWidget::NoFocus)* erhält dieses Widget niemals den Fokus. Das ist auch die Default-Einstellung der Klasse *QWidget*. Mit *setFocusPolicy (QWidget::TabFocus)* kann das Widget durch die -Taste zum Fokus-Widget werden, mit *setFocusPolicy (QWidget::ClickFocus)* durch Klicken mit der Maus in das Widget und mit *setFocusPolicy (QWidget::StrongFocus)* sowohl durch die -Taste als auch durch die Maus. Die vordefinierten Eingabeelemente von Qt setzen einen geeigneten Wert in ihrem Konstruktor, meist *StrongFocus*. Dieser kann aber jederzeit nachträglich mit *setFocusPolicy* noch geändert werden. Mit *focusPolicy* kann man den eingestellten Wert ermitteln, und mit *isFocusEnabled* kann man abfragen, ob der Wert auf *NoFocus* eingestellt ist oder nicht.

Die Reihenfolge, in der die -Taste zwischen den Eingabeelementen wechselt, ist die Reihenfolge, in der die Unter-Widgets in das Haupt-Widget eingefügt werden. Hat ein Unter-Widget seinerseits weitere Unter-Widgets, werden diese zunächst in ihrer Reihenfolge durchlaufen, bevor das nächste Widget den Fokus bekommt. Diese Reihenfolge kann jedoch mit der Methode *setTabOrder* geändert werden. Diese Methode erhält zwei Widgets als Parameter und ordnet die Reihenfolge so um, dass das zweite Widget unmittelbar nach dem ersten folgt. Ein kleines Beispiel soll das verdeutlichen:

```
QWidget *w = new QWidget (); // Toplevel-Widget
QLineEdit *a = new QLineEdit (w);
QLineEdit *b = new QLineEdit (w);
QLineEdit *c = new QLineEdit (w);
QLineEdit *d = new QLineEdit (w);
```

Dieses Programmsegment erzeugt ein Toplevel-Widget mit vier Eingabezeilen als Unter-Widgets. Die Tabulator-Reihenfolge ist nun $a \rightarrow b \rightarrow c \rightarrow d \rightarrow a \rightarrow \dots$

Mit der Zeile

```
w->setTabOrder (b, d);
```

ändert sich die Reihenfolge in $a \rightarrow b \rightarrow d \rightarrow c \rightarrow a \rightarrow \dots$

Um eine vollständig neue Reihenfolge festzulegen, setzen Sie also am besten die Kette von vorn bis hinten. Die Reihenfolge $d \rightarrow b \rightarrow a \rightarrow c \rightarrow d \dots$ erhalten Sie zum Beispiel mit folgenden Zeilen:

```
w->setTabOrder (d, b);
w->setTabOrder (b, a);
w->setTabOrder (a, c);
// w->setTabOrder (c, d); kann entfallen, da automatisch
d->setFocus (); // setzt den Anfangsfokus auf d
```

Eigenschaften von Toplevel-Widgets

Toplevel-Widgets – also Widgets ohne Vater-Widget – können eine Reihe weiterer Eigenschaften haben. Diese Eigenschaften lassen sich zwar auch für Unter-Widgets setzen, haben aber nur bei Toplevel-Widgets Auswirkungen.

Der Text, der in der Titelseite erscheinen soll, kann mit der Methode *setCaption* gesetzt werden. In der Titelseite steht normalerweise der Name des Programms und eventuell der Dateiname eines gerade geöffneten Dokuments; es kann jedoch auch jeder andere Text sein.

Eine andere Eigenschaft, die eingestellt werden kann, ist die Art der Fensterdekoration, also des Rahmens und der Titelseite, die um das Widget vom Window-Manager gezeichnet werden. Diese Eigenschaft kann bereits im Konstruktor des Widgets im dritten Parameter, *WFlags*, festgelegt werden. Dieser Parameter ist standardmäßig 0, wodurch das Widget eine Titelseite mit Namen, Systemmenü und drei Buttons erhält sowie einen Rahmen zum Verändern der Größe. Diese Einstellung ist für fast alle Anwendungen die beste. Will man eine andere Dekoration, so kann man als dritten Parameter eine Oder-Kombination der Konstanten *WStyle_Customize* mit einigen der folgenden Konstanten benutzen:

- *WStyle_NormalBorder* (Rahmen zum Ändern der Größe), *WStyle_DialogBorder* (eine dünne Linie als Rahmen) oder *WStyle_NoBorder* (kein Rahmen)
- *WStyle_Title* erzeugt eine Titelleiste mit der Bezeichnung des Programms.
- *WStyle_SysMenu* erzeugt einen Button (meist oben links) für das Systemmenü.
- *WStyle_Minimize* erzeugt einen Button (meist den dritten von rechts) zum Minimieren des Fensters.
- *WStyle_Maximize* erzeugt einen Button (meist den zweiten von rechts) zum Vergrößern des Fensters auf die Bildschirmgröße.

- *WStyle_MinMax* ist die Abkürzung für *WStyle_Minimize* und *WStyle_Maximize*.
- *WStyle_Tool* erzeugt ein Fenster, das nur eine kurze Lebenszeit hat und es zum Beispiel für Popup-Menüs und die so genannten Tool-Tips (kleine Hilfetexte, die erscheinen, wenn die Maus auf einem Objekt stehen bleibt) eingesetzt wird.

Beachten Sie, dass diese Einstellungen nur Vorschläge für den Window-Manager darstellen. Es gibt unter Linux eine große Anzahl verschiedener Window-Manager, und jeder verfolgt eine eigene Strategie, um Fenster darzustellen. Das Ergebnis kann also auf verschiedenen Systemen unterschiedlich aussehen. Wenn Sie die Standardeinstellung benutzen, können Sie sicher sein, dass das Fenster auf jeden Fall vollständig bedienbar bleibt. Unter Microsoft Windows dagegen funktionieren alle möglichen Kombinationen.

3.3 Grundstruktur einer Applikation

Der Grundaufbau des Hauptprogramms einer Qt- bzw. KDE-Applikation ist in nahezu allen Fällen identisch. Eine zentrale Rolle spielt dabei die Klasse *QApplication* (für Qt-Applikationen) bzw. *KApplication* (eine Unterklasse von *QApplication*, für KDE-Applikationen).

Von dieser Klasse wird genau ein Objekt erzeugt. Dieses Objekt übernimmt die Initialisierung und Kommunikation mit dem X-Server (bzw. den Microsoft Windows-Bildschirmtreibern) und steuert den weiteren Ablauf des Programms. Alle Bildelemente (Widgets) dürfen erst erzeugt werden, nachdem dieses Objekt angelegt worden ist; ansonsten kann es zu einer Fehlermeldung oder sogar zu einem Programmabsturz kommen.

Üblicherweise ist dieses Objekt daher auch das erste Objekt, das man im Hauptprogramm in der Funktion *main* erzeugt.

3.3.1 Qt-Applikationen

Wenn Sie sich auf die Qt-Bibliothek beschränken wollen (oder müssen, z. B. um das Programm auch unter Microsoft Windows kompilieren zu können), so benutzen Sie die Klasse *QApplication*. Sie erledigt folgende Aufgaben innerhalb des Konstruktors:

- Sie stellt über eine Socketverbindung eine Verbindung zum X-Server her. Der X-Server kann dabei auch auf einem anderen Rechner laufen. Welcher Rechner benutzt wird, liest das *QApplication*-Objekt aus der Umgebungsvariablen *\$DISPLAY* aus, oder – falls vorhanden – aus dem Kommandozeilenparameter *-display*. Konnte keine Verbindung hergestellt werden, so wird das Programm mit einer Fehlermeldung beendet.

- Sie wählt eine geeignete Farbpalette aus, die im weiteren Programm benutzt wird. Dadurch nimmt Qt dem Programmierer viel Arbeit ab. Wer einmal versucht hat, ein Programm mit der XLib zu schreiben und alle möglichen Grafikmodi unterstützen wollte, wird bestätigen, dass die Bestimmung einer Farbpalette ein sehr aufwendiges Unternehmen ist. Nähere Informationen über die Farbverwaltung von Qt können Sie in Kapitel 4.1, *Farben in Qt*, nachlesen.
- Interne Datenstrukturen zur Verwaltung der Fenster auf dem Bildschirm werden angelegt und initialisiert.

Ein Programm kann immer nur ein Objekt des Typs *QApplication* enthalten. Falls Sie versuchen, ein zweites Element anzulegen, so wird eine Warnung ausgegeben und das neue Objekt nicht initialisiert.

Während des Programmlaufs ist *QApplication* das einzige Objekt, das alle Ereignisse von Tastatur und Maus entgegennimmt. Es verteilt die hereinkommenden Ereignisse an die Qt-Objekte, die sie betreffen. Dort werden dann die entsprechenden Aktionen als Reaktion darauf ausgeführt. Die Ereignisse – die so genannten Events – werden in einer Warteschlange im X-Server gespeichert. In der Haupt-Event-Schleife fragt das *QApplication*-Objekt diese Warteschlange ab. Es handelt sich dabei nicht um ein so genanntes *Busy-Waiting*, bei dem das Programm immer wieder nach neuen Events fragt. Wenn kein Event vorliegt, so wird das Programm so lange vom Betriebssystem gestoppt, bis ein neues Ereignis vorgefallen ist, das der X-Server in der Warteschlange abgelegt hat. So ist gewährleistet, dass das Programm keine Rechenzeit verbraucht, wenn nichts zu tun ist. Die Haupt-Event-Schleife ist in der Methode *QApplication::exec* enthalten. Beendet wird die Haupt-Event-Schleife (und damit diese Methode) erst, wenn der Slot *quit* des *QApplication*-Objekts aktiviert wurde. Dies kann natürlich nur innerhalb der Haupt-Event-Schleife selbst geschehen, zum Beispiel als Reaktion auf eine Maus- oder Tastaturaktion des Benutzers.

Ein typisches Beispiel für die Beendigung der Haupt-Event-Schleife ist die Verbindung eines Menüeintrags *EXIT* mit dem Slot *quit* des *QApplication*-Objekts. Innerhalb der Haupt-Event-Schleife wird dem *QApplication*-Objekt dann vom X-Server das Ereignis »Maus gedrückt bei den Koordinaten x/y« mitgeteilt. Anhand der Koordinaten wird dieses Ereignis dem Popup-Menü zugeordnet und an dieses Objekt weitergeleitet. Dort werden die Mauskoordinaten als Menüpunkt *EXIT* interpretiert. Ist dieser Menüpunkt nun mit dem Slot *quit* des *QApplication*-Objekts verbunden, wird dieser aktiviert. Diese Aktivierung wird zunächst nur vermerkt und hat keine weiteren Auswirkungen. Erst wenn das Mausereignis vollständig abgearbeitet worden ist und die Kontrolle wieder zurück zur Haupt-Event-Schleife gelangt, wird festgestellt, dass *quit* aktiviert wurde, und die Haupt-Event-Schleife wird verlassen.

Grundaufbau der main-Funktion

Der typische Grundaufbau einer Qt-Applikation sieht folgendermaßen aus (er ist meist in einer eigenen Datei *main.cpp* zu finden):

```
#include <qapplication.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    // ...
    // Hier folgen nun die Initialisierung und die Anzeige
    // von Fenstern.
    // ...

    return app.exec ();
}
```

Vergleichen Sie diese Struktur noch einmal mit unserem ersten Beispiel in Kapitel 2.2, *Das erste Qt-Programm*:

```
// Das erste Programm
// KDE- und Qt-Programmierung
// Addison-Wesley Germany
#include <qapplication.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}
```

Das *QApplication*-Objekt benötigt die Kommandozeilenparameter in den Variablen *argc* und *argv*, da es diese Argumente interpretiert und benutzt, um eine Verbindung zum X-Server aufzubauen. Die Argumente, die es erkannt hat, entfernt es aus den Variablen *argc* und *argv*.

Bevor das *QApplication*-Objekt nicht erzeugt worden ist, dürfen keine Fenster angelegt werden, da noch die Verbindung zum X-Server und einige Initialisierungen fehlen. Daher wird es in der Regel als allererste Aktion in der Funktion *main* erzeugt. Beachten Sie: Statische Variablen werden noch vor dem Aufruf von *main* angelegt, also noch ohne *QApplication*-Objekt! Sie können also keine Objekte der Qt-Bibliothek (insbesondere der Klasse *QObject* und der Unterklassen) als statische Variablen anlegen!

Damit Sie von jeder Stelle Ihres Programms aus einen einfachen Zugriff auf das *QApplication*-Objekt haben, ist eine globale Zeigervariable *qApp* definiert worden (in *qapplication.h*), die die Adresse des Objekts enthält. Häufig findet dieser Zeiger Anwendung, wenn Sie ein Signal mit dem Slot *quit* des *QApplication*-Objekts verbinden wollen:

```
QObject::connect (myPushButton, SIGNAL (clicked()),
                  qApp, SLOT (quit()));
```

Beenden des Programms

Wie wir bereits mehrfach erwähnt haben, wird die Haupt-Event-Schleife durch das Aktivieren des *quit*-Slots beendet. Es gibt nun mehrere Möglichkeiten, wie Sie diesen Slot in der Regel aktivieren:

1. Sie können ein Signal explizit mit diesem Slot verbinden. Hier benutzen wir als Beispiel ein *QPushButton*-Objekt, dessen Signal *clicked()* wir hier verwenden:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QPushButton *b = new QPushButton ("Quit", 0);
    b->show();
    QObject::connect (b, SIGNAL (clicked()),
                     qApp, SLOT (quit()));
    return app.exec();
}
```

Ein Mausklick auf den Button beendet nun das Programm. Aber Vorsicht: Wenn Sie nun das Button-Fenster schließen, läuft das Programm weiter, und Sie können das Programm nur noch mit *kill* beenden!

2. Sie können in einer Methode Ihres Programms den Slot *quit* auch als normale Methode benutzen und einfach aufrufen. Ändern Sie beispielsweise in Beispiel 4 aus Kapitel 3.1.2, *Das Signal-Slot-Konzept*, die Slot-Methode *Counter::countUp()* folgendermaßen ab:

```
void Counter::countUp ()
{
    n++;
    cout << "Anzahl Aktivierungen " << n << endl;
    if (n == 10)
        qApp->quit();
}
```

Nun wird das Programm automatisch beendet, wenn Sie zum zehnten Mal auf den Button geklickt haben.

3. Wenn Sie mit der Methode *QApplication::setMainWidget* ein Hauptfenster festgelegt haben, wird *quit* automatisch aufgerufen, wenn dieses Fenster geschlossen wird. Diese Variante haben wir in unserem Anfangsbeispiel in Kapitel 2.2, *Das erste Qt-Programm*, gewählt.
4. Wenn Sie mehrere Fenster geöffnet haben und keines von ihnen das Hauptfenster sein soll, so können Sie auch das Signal *QApplication::lastWindowClosed()* benutzen. Es wird in dem Moment aufgerufen, in dem das letzte noch sichtbare Fenster geschlossen wird. (Spätestens in diesem Moment sollte das Programm ohnehin beendet werden, denn der Anwender hat nun gar keine Interaktionsmöglichkeit mehr, um das Programm zu beenden.) Das folgende einfache Beispiel öffnet zwei Fenster. Das Programm wird beendet, nachdem beide Fenster geschlossen wurden:

```
#include <qapplication.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l1 = new QLabel ("<H1>Fenster Eins</H1>", 0);
    l1->show();
    QLabel *l2 = new QLabel ("<H1>Fenster Zwei</H1>", 0);
    l2->show();
    QObject::connect (qApp, SIGNAL (lastWindowClosed()),
                     qApp, SLOT (quit()));
    return app.exec();
}
```

Sie können natürlich mehrere der oben genannten Möglichkeiten gleichzeitig verwenden. So kann es im Grunde nie schaden, das Signal *lastWindowClosed* mit *quit* zu verbinden, auch wenn Sie von einer anderen Stelle des Programms aus explizit *quit* aufrufen.

3.3.2 KDE-Applikationen

Wenn Sie ein KDE-Programm schreiben wollen – also ein Programm, das auf einem KDE-System läuft und alle Möglichkeiten von KDE nutzen will –, so müssen Sie statt des *QApplication*-Objekts ein *KApplication*-Objekt benutzen. Die Klasse *KApplication* ist eine Unterklasse von *QApplication*. Sie bietet daher die gesamte Funktionalität von *QApplication* und zusätzlich folgende Möglichkeiten:

- Im Konstruktor werden automatisch die Konfigurationsdateien für das Programm geladen und ausgewertet (siehe Kapitel 4.10, *Konfigurationsdateien*) sowie die Übersetzungsdateien für die mehrsprachige Textdarstellung geladen (siehe Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*).
- Sie können mit Hilfe der Klasse *KAboutData* Informationen über das Programm (Name, Version, Beschreibung, Autoren, Homepage, Lizenzbestimmungen, E-Mail-Adresse für Fehlerbenachrichtigungen) festlegen, die beispielsweise beim Programmaufruf mit dem Kommandozeilenparameter *--help* oder im *HELP-ABOUT...*-Dialog ausgegeben werden.
- Sie haben die Möglichkeit, mit Hilfe der Klasse *KCmdLineArgs* die Kommandozeilenparameter sehr einfach und elegant nach eigenen Optionen durchsuchen zu lassen.

Es gelten natürlich zunächst einmal die gleichen Grundsätze für *KApplication* wie für *QApplication*, d.h. die erste Aktion im Programm *main* sollte darin bestehen, das *KApplication*-Objekt anzulegen. Andere Objekte der KDE- oder Qt-Bibliothek sollten vorher nicht angelegt werden. *KApplication* erbt von *QApplication* natürlich die Methode *exec* mit der Haupt-Event-Schleife, die auch hier mit *quit* beendet wird.

Auch bei *KApplication* können Sie theoretisch die globale Variable *qApp* benutzen. Allerdings ist sie ein Zeiger auf ein *QApplication*-Objekt. Sie können also über diesen Zeiger nur die Methoden von *QApplication* aufrufen, nicht die, die *KApplication* zusätzlich definiert (es sei denn, Sie setzen einen Type-Cast davor). Benutzen Sie hier stattdessen das Makro *kapp* (definiert in *kapp.h*). Es liefert die Adresse des *KApplication*-Objekts zurück.

Der einfachste Fall

Im einfachsten Fall (der aber nur für kleine Testprogramme benutzt werden sollte) können Sie folgendes Grundgerüst benutzen, das sich vom Grundgerüst eines reinen Qt-Programms kaum unterscheidet (die Unterschiede sind fett gedruckt):

```
#include <kapp.h>

int main (int argc, char **argv)
{
    KApplication app (argc, argv, "myApplication");

    // ...
    // Hier folgen die Initialisierung und die Anzeige
    // von Fenstern.
    // ...

    return app.exec ();
}
```

Beachten Sie folgende Unterschiede:

- Binden Sie die Header-Datei *kapp.h* ein. Aus historischen Gründen heißt sie inkonsequenterweise nicht *kapplication.h*.
- Erzeugen Sie als erste Aktion in *main* nun ein *KApplication*-Objekt. Diesem müssen Sie neben den Kommandozeilenparametern in *argc* und *argv* als dritten Parameter den Programmnamen übergeben. Dieser muss mit dem Namen der ausführbaren Datei identisch sein. (Aufgrund dieses Parameters werden die Dateinamen für die Konfigurationsdateien und einige andere Dateien festgelegt. Der Parameter muss hier angegeben werden, damit diese Konfigurationsdateien auch dann korrekt gefunden werden, wenn die ausführbare Datei – aus welchem Grund auch immer – umbenannt worden ist.)

Außerdem müssen Sie beim Kompilieren der Datei das Include-Verzeichnis der KDE-Header-Dateien zusätzlich zum Qt-Header-Verzeichnis angeben. Beim Linken müssen Sie zusätzlich zur Qt-Bibliothek die KDE-Bibliotheken *kdeui* und *kdecore* angeben.

Schreiben wir unser Beispielprogramm aus Kapitel 2.2, *Das erste Qt-Programm*, also einmal auf KDE um. Erstellen Sie mit einem Editor die Datei *hello-kde.cpp* mit folgendem Inhalt:

```
#include <kapp.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    KApplication app (argc, argv, "hello-kde");
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}
```

Kompilieren und linken Sie das Programm nun mit folgender Zeile (ohne den Zeilenumbruch):

```
% g++ -o hello-kde hello-kde.cpp -I$QTDIR/include
-I$KDEDIR/include -lkdecore -lkdeui -lqt
```

Falls Sie dieses Programm unter Microsoft Windows kompilieren wollen, so muss ich Sie leider enttäuschen: Die KDE-Bibliotheken sind leider nicht für Microsoft Windows vorhanden, da sie einigen sehr Unix-spezifischen Code enthalten. Daran wird sich wahrscheinlich auf absehbare Zeit auch nichts ändern. Wenn Sie also Ihr Programm auch unter Microsoft Windows zum Laufen bringen wollen, müssen Sie auf alle KDE-Erweiterungen verzichten. (Einzelne Klassen der KDE-

Bibliotheken lassen sich dennoch auch unter Microsoft Windows kompilieren und benutzen. Dazu sind aber oft Anpassungen nötig, die nur ein Experte schafft.)

Wenn Sie nun die ausführbare Datei *hello-kde* starten, sollte sich wiederum ein ähnliches Fenster öffnen wie in Kapitel 2.2. In der Regel wird es aber kleine Unterschiede geben. Abbildung 3.12 zeigt links das Qt-Programm (mit einfarbiger Fläche), rechts das KDE-Programm (mit leicht gemusterter Fläche). Die Unterschiede kommen daher, dass *KApplication* bei der Initialisierung einige KDE-globale Einstellungen aus den Konfigurationsdateien lädt und anwendet. *QApplication* lädt dagegen keine Konfigurationsdateien und benutzt nur die Default-Werte.



Abbildung 3-12 Das Beispielprogramm als Qt-Applikation (links) und als KDE-Applikation (rechts)

Nachdem wir nun diesen einfachsten Fall besprochen haben, will ich Ihnen diese Realisierung eines KDE-Programms auch gleich wieder ausreden. Echte KDE-Programme sollten sich an eines der beiden Grundgerüste halten, die in den nächsten beiden Abschnitten beschrieben werden.

Einige Informationen über das Programm

Die Klasse *KApplication* hat unserem Programm noch eine weitere nützliche Eigenschaft hinzugefügt, die wir bisher noch nicht bemerkt haben. Rufen Sie unser Programm *hello-kde* nun einmal mit dem Parameter *--help* auf:

```
% hello-kde --help
```

Das Ergebnis ist, dass nicht das Fenster mit dem Begrüßungstext erscheint, sondern dass Sie folgende Ausgabe in Ihrem Terminalfenster erhalten:

```
Usage: hello-kde [Qt-options] [KDE-options]
```

```
KDE Application
```

```
Generic options:
```

```
--help
```

```
Show help about options
```

```
--help-qt
```

```
Show Qt specific options
```

```
--help-kde           Show KDE specific options
--help-all          Show all options
--author             Show author information
-v, --version        Show version information
--license            Show license information
--                  End of options
```

Das Programm bietet dem Anwender also eine einfache Möglichkeit, sich über die Kommandozeilenparameter zu informieren. Allerdings sind die weiteren Informationen, die dabei angegeben werden, zum Teil recht dürftig: Unser Programm wird einfach nur als »KDE Application« beschrieben – eine Angabe, die auf sehr viele Programme zutrifft –, und auch die Lizenzbestimmungen und die Versionsnummer sind nicht angegeben.

Um diese Daten anzugeben, benutzen wir die Klasse *KCmdLineArgs*. Sie besitzt eine statische Methode *init*, mit der wir diese Informationen festlegen. Erweitern wir nun unser Programm entsprechend:

```
#include <kapp.h>
#include <kcmlineargs.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    KCmdLineArgs::init (argc, argv, "hello-kde",
        "The Hello-World program for KDE", "1.0");
    KApplication app;
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}
```

Wir übergeben die Kommandozeilenparameter jetzt an die Methode *init*, gefolgt vom Namen des Programms, einer kurzen Beschreibung des Programms und der Versionsnummer. Beachten Sie, dass wir nun unser *KApplication*-Objekt ohne Parameter anlegen. Alle notwendigen Informationen sind bereits in *KCmdLineArgs* gespeichert.

Achtung: Geben Sie beim Erzeugen des *KApplication*-Objekts *kein* leeres Klammernpaar an! Schreiben Sie also *nicht*:

```
KApplication app();
```

Diese Zeile hat für den C++-Compiler eine andere Bedeutung, als man zunächst vermuten würde. Sie definiert eine Funktion *app*, die keine Parameter hat und als Rückgabetypp ein *KApplication*-Objekt zurückgibt. Das ist natürlich nicht das, was

wir haben wollen. Diese Zeile führt so zu Fehlermeldungen, die nur schwer zu durchschauen sind. Also Vorsicht beim Anlegen eines Objekts, wenn wir dessen Konstruktor keine Parameter übergeben!

Nun haben wir unser Programm um eine Kurzbeschreibung und eine Versionsnummer erweitert. Diese Informationen werden übrigens auch im *Help-About...*-Dialog angezeigt, wenn Sie das automatische Hilfesystem in Ihr Programm aufnehmen. (Das hatten wir beispielsweise in Kapitel 2.3, *Das erste KDE-Programm*, gemacht.) Es fehlen noch Informationen über die Lizenzbestimmungen, den Autor, die Homepage und die E-Mail-Adresse. Wenn Sie auch diese Informationen zur Verfügung stellen wollen, benutzen Sie die Klasse *KAboutData*. Diese Klasse dient speziell zum Speichern von Informationen über ein Programm. Konstruieren Sie dazu ein Objekt dieser Klasse, und übergeben Sie es als dritten Parameter von *KCmdLineArgs::init*. Die weiteren Parameter entfallen, denn nun steckt diese Information im *KAboutData*-Objekt:

```
#include <kaboutdata.h>
#include <kapp.h>
#include <kcmdlineargs.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    KAboutData about
        ("hello-kde", "HelloWorld-KDE", "1.0",
         "The Hello-World program for KDE",
         KAboutData::License_GPL,
         "(c) 2000 Addison-Wesley-Germany",
         "http://www.addison-wesley.de",
         "Here is more text\neven with more lines",
         "Burkhard.Lehner@gmx.de");
    about.addAuthor ("Burkhard Lehner",
                    "Source and Testing",
                    "Burkhard.Lehner@gmx.de",
                    "http://www.burkhardlehner.de/");
    about.addCredit ("My mother",
                    "cooking coffee",
                    "mom@gmx.de",
                    "http://www.mother.de/");
    KCmdLineArgs::init (argc, argv, &about);
    KApplication app;

    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}
```


Wir legen also zunächst ein Objekt der Klasse *KAboutData* an, das wir bereits im Konstruktor mit vielen Informationen über unser Programm füttern, und zwar mit:

- dem Applikationsnamen (identisch zum Namen der ausführbaren Datei)
- dem Namen des Programms (mit Groß- und Kleinschreibung, wird übersetzt)
- der Versionsnummer
- der Kurzbeschreibung (wird übersetzt)
- den Lizenzbestimmungen (hier können Sie eine der Konstante des Aufzählungstyps in *KAboutData* wählen, z.B. auch *License_BSD* oder *License_QPL*)
- der Copyright-Meldung
- der Homepage des Programms
- einem längeren Text zur Beschreibung des Programms (wird übersetzt)
- einer E-Mail-Adresse für Fehlerbeschreibungen

Nur die ersten drei Parameter sind Pflicht, alle weiteren sind mit Default-Werten versehen. Sie können also (von hinten beginnend) die Parameter auch weglassen.

Beachten Sie, dass *KAboutData* eine Ausnahme von der Regel ist, dass vor dem *KApplication*-Objekt kein anderes Objekt erzeugt werden darf.

Anschließend können Sie mit der Methode *addAuthor* noch beliebig viele Leute als Autoren einfügen oder mit *addCredit* Personen in die Liste der Danksagungen aufnehmen. Dazu müssen Sie als ersten Parameter jeweils den Namen angeben. Die weiteren Parameter sind wieder optional und legen die Aufgabe im Projekt, die E-Mail-Adresse und die Homepage der Person fest. Falls Sie Ihr Programm unter einer ganz exotischen Lizenz veröffentlichen wollen, können Sie mit *setLicenseText* Ihre eigenen Lizenzvereinbarungen eintragen.

Nicht alle diese Informationen werden beim Aufruf mit *--help* ausgegeben. Die übrigen werden im *Help-About...*-Dialog angezeigt.

Besonderheiten bei der Übersetzung in andere Landessprachen

Einige der Parameter für *KAboutData* werden vor dem Anzeigen in die eingestellte Landessprache übersetzt. Wie wir bereits in Kapitel 2.3.3, *Landessprache: Deutsch*, gesehen haben, übernimmt die Funktion *i18n* diese Aufgabe. Diese Funktion können wir allerdings im Konstruktor von *KAboutData* nicht einsetzen, denn zu diesem Zeitpunkt gibt es noch kein *KApplication*-Objekt, das heißt die Übersetzungstabellen wurden noch nicht geladen. *KAboutData* hilft uns weiter, da es die Texte erst unmittelbar vor dem Anzeigen mit *i18n* übersetzt. (Es werden genau

die Texte übersetzt, bei denen dieses in der Liste der Parameter oben angegeben ist. Die anderen Texte bleiben unverändert.) Zu dem Zeitpunkt sind die Übersetzungstabellen schon geladen.

Wir haben aber immer noch ein Problem: Die Texte können nur übersetzt werden, wenn sie auch in der Übersetzungstabelle enthalten sind. Diese entsteht aber, indem mit dem Tool *xgettext* im Quellcode alle Strings gesucht werden, die hinter dem Wort *i18n* stehen. Unsere Texte im Konstruktor tun dies aber leider nicht.

Ein Ausweg ist hier das Makro *I18N_NOOP* (definiert in *klocale.h*). Dieses Makro macht nichts, es wird einfach durch den Text ersetzt, der in Klammern dahinter angegeben ist. *xgettext* findet aber auch Texte hinter diesem Makro und trägt sie in die Übersetzungstabelle ein. Um also für die Übersetzung perfekt vorbereitet zu sein, sollte unser Programm also folgendermaßen aussehen:

```
#include <kaboutdata.h>
#include <kapp.h>
#include <kcmdlineargs.h>
#include <klocale.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    KAboutData about
        ("hello-kde", I18N_NOOP ("HelloWorld-KDE"), "1.0",
         I18N_NOOP ("The Hello-World program for KDE"),
         KAboutData::License_GPL,
         "(c) 2000 Addison-Wesley-Germany",
         "http://www.addison-wesley.de",
         I18N_NOOP ("Here is more text\n"
                    "even with more lines"),
         "Burkhard.Lehner@gmx.de");
    about.addAuthor ("Burkhard Lehner",
                    I18N_NOOP ("Source and Testing"),
                    "Burkhard.Lehner@gmx.de",
                    "http://www.burkhardlehner.de/");
    about.addCredit ("My mother",
                    I18N_NOOP ("cooking coffee"),
                    "mom@gmx.de",
                    "http://www.mother.de/");
    KCmdLineArgs::init (argc, argv, &about);
    KApplication app;

    QLabel *l = new QLabel
        (i18n ("<H1>Hello, World!</H1>"), 0);
    l->show();
}
```

```
    app.setMainWidget (1);  
    return app.exec();  
}
```

Die Beschriftung des *QLabel*-Objekts kann wiederum ganz normal mit *i18n* erfolgen, denn zum Zeitpunkt der Ausführung ist das *KApplication*-Objekt bereits erzeugt, und damit sind die Übersetzungstabellen geladen.

Weitere Informationen zur Übersetzung finden Sie in Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*.

Analyse eigener Kommandozeilenoptionen

Der Klassenname *KCmdLineArgs* ist die Abkürzung für *Command Line Arguments*. Diese Klasse, die wir bisher dazu benutzt haben, Informationen zu unserem Programm abzulegen, dient also vor allem dazu, die Kommandozeilenparameter zu analysieren. Sie ist dabei flexibel ausgelegt, so dass wir ihr mitteilen können, dass wir für unser Programm eigene Optionen vorsehen wollen. Dazu müssen wir der Klasse mitteilen, welche Optionen wir erwarten und was für ein Format sie haben sollen. Außerdem müssen wir eine kurze Beschreibung zu jeder Option aufführen, damit der Anwender die Optionen auch versteht, wenn er das Programm mit *--help* aufruft.

Die Informationen für unsere Optionen legen wir in einem statischen Array ab, dessen Elemente vom Struktur-Typ *KCmdLineOptions* sind. Diese Struktur enthält drei Strings vom Typ *char**: einen Namen, eine Beschreibung und einen Defaultwert.

Erweitern wir unser Programm um drei mögliche Optionen: Mit der Option *-b* oder *--beep* erzeugen wir einen Lautsprecherpieps direkt nach der Initialisierung, und mit der Option *--message <text>* kann man einen anderen Text angeben, der angezeigt werden soll.

Unser Programmtext sieht nun folgendermaßen aus (der Quelltext für *KAboutData* wurde der Übersichtlichkeit halber weggelassen):

```
#include <kaboutdata.h>  
#include <kapp.h>  
#include <kcmlineargs.h>  
#include <klocale.h>  
#include <qlabel.h>  
  
static KCmdLineOptions options[] =  
{ {"b", 0, 0},  
  {"beep", I18N_NOOP ("Beep on startup"), 0},  
  {"message <text>", I18N_NOOP ("Display this message"),  
    "<H1>Hello, World!</H1>"},  
  {0, 0, 0}};
```

```

    int main (int argc, char **argv)
    {
        KAboutData about (...); // wie oben
        KCmdLineArgs::init (argc, argv, &about);
        KCmdLineArgs::addCmdLineOptions (options);
        KApplication app;

        KCmdLineArgs *args = KCmdLineArgs::parsedArgs();
        if (args->isSet ("b"))
            KApplication::beep();

        QLabel *l = new QLabel (args->getOption ("message"), 0);
        l->show();
        app.setMainWidget (l);
        return app.exec();
    }

```

Kompilieren Sie dieses Programm, und starten Sie es mit folgender Zeile:

```
% hello-kde --beep --message "Hallöchen, Universum"
```

Probieren Sie statt *--beep* auch einmal *-beep*, *-b* oder *--b* aus. Alle Varianten sollten einen Piepston erzeugen. Wenn Sie die Option *--message* weglassen, wird wieder der Standardtext ausgegeben.

Die statische Array-Variable *options* speichert die Informationen über die Optionen, die uns interessieren. Der zweite Eintrag in jeder Zeile ist die Beschreibung für eine Option. Diese sollte auch in die passende Landessprache übersetzt werden. Da *i18n* zu dem Zeitpunkt, zu dem die Variable angelegt wird, noch nicht funktioniert (*KApplication* ist noch nicht erzeugt), müssen wir auch hier das Hilfsmakro *I18N_NOOP* benutzen.

Die Liste muss mit einer Zeile mit drei Null-Zeigern abgeschlossen werden. (Vergessen Sie diese Zeile nicht, da es sonst zu einem Absturz kommt.)

Wir haben in unserer Liste die Option *-b* als abkürzendes Synonym für die Option *--beep* eingefügt, indem wir als zweiten Wert einen Nullzeiger benutzt haben. Dadurch wird diese Option automatisch identisch mit der folgenden. Beachten Sie auch, dass es unter Unix üblich ist, einbuchstabige Optionen mit einem einzelnen Minus-Zeichen beginnen zu lassen, Optionen aus ganzen Wörtern dagegen mit zwei Minus-Zeichen. Das gibt *KCmdLineArgs* beim Aufruf des Programms mit *--help* auch so aus. Dennoch versteht das Programm auch einbuchstabige Optionen mit zwei Minus-Zeichen sowie mehrbuchstabige mit einem.

Dieses Optionen-Objekt wird mit der statischen Methode *KCmdLineArgs::addCmdLineOptions* übergeben, und zwar noch *vor* dem Erzeugen des *KApplication*-Objekts. Im weiteren Programmverlauf kann man sich dann mit der

Methode *KCmdLineArgs::parsedArgs* ein Objekt der Klasse *KCmdLineArgs* verschaffen, in dem die gefundenen Optionen enthalten sind.

Es gibt drei grundlegende Typen von Optionen, die analysiert werden können:

- einfache Optionen, die der Anwender an- und ausschalten kann (hier sind es *-b* und *--beep*) – Sie werden durch einen einfachen Namen angegeben.
- Optionen mit einem nachfolgenden Parameter (hier *--message*) – Sie werden durch einen Namen angegeben, gefolgt von einem Leerschritt und einer Bezeichnung des erwarteten Parameters in spitzen Klammern.
- eine Liste einfacher Argumente ohne Schlüsselwort und ohne Minus-Zeichen am Anfang (wird hier nicht benutzt, siehe unten) – Sie wird durch ein Plus-Zeichen, gefolgt von einer Bezeichnung für die Art der Argumente spezifiziert.

Einfache Optionen kann man mit der Methode *isSet* abfragen. Der boolsche Rückgabewert zeigt an, ob die Option vorhanden ist (*true*) oder nicht (*false*). Eine solche Option ist per Default zunächst nicht gesetzt. Um eine Option zu erhalten, die per Default gesetzt ist, lassen Sie den Namen einfach mit einem *no* beginnen. Wenn Sie beispielsweise im Programm oben die Bezeichnung *beep* durch *nobeep* ersetzen, so piept das Programm, wenn keine Parameter angegeben sind (oder *--beep* benutzt wird), und piept nicht, wenn *--nobeep* angegeben wird. Beachten Sie also, dass bei der Definition einer Option *xyz* automatisch auch *noxyz* definiert ist, und umgekehrt bei der Definition von *noxyz* auch *xyz* definiert ist. Berücksichtigen Sie diesen Umstand, wenn Sie eine Bezeichnung wählen, die von sich aus mit *no* beginnt, z.B. *notorious*. Dadurch ist automatisch auch die Option *torious* möglich, auch wenn sie keinen Sinn macht. Mit *isSet* können Sie allerdings nur die Option abfragen, die Sie selbst definiert haben, nicht die gegenteilige.

Bei Optionen mit nachfolgendem Parameter benutzen Sie zur Abfrage die Methode *getOption*. Sie liefert den String zurück, der als nächster Parameter der Option folgt. Wenn er in der Kommandozeile in Anführungszeichen gesetzt wird, darf er sogar Leerschritte enthalten. Wird dieser Parameter nicht angegeben, wird stattdessen die dritte Angabe in der *options*-Variable als Default benutzt. Von solchen Parametern gibt es keine *no*-Variante.

Findet *KCmdLineArgs* Parameter, die auf keinen der Einträge in der Liste passen, wird das Programm mit einer entsprechenden Fehlermeldung beendet. Man kann aber auch alle noch übrig gebliebenen Parameter mit den Methoden *KCmdLineArgs::count* und *KCmdLineArgs::arg* erfragen. (Diese Parameter dürfen aber nicht mit einem Minus-Zeichen beginnen.) Dazu muss man allerdings eine Option in die *options*-Liste aufnehmen, deren Bezeichnung mit einem Plus-Zeichen beginnt. Man benutzt diese Argumente oft, um schon beim Programmstart

einen oder mehrere Dateinamen an ein Programm zu übergeben, das diese Datei(en) automatisch öffnet. Beispiele hierfür finden Sie in Kapitel 3.5.6, *Applikation für ein Dokument*.

Beschränkung auf eine einzige Instanz eines Programms

Manchmal kann es sinnvoll sein, dass ein Programm beim Starten zunächst einmal schaut, ob es nicht bereits läuft. Hier folgt eine kleine – sicherlich unvollständige – Liste von Gründen:

- Handelt es sich um ein sehr umfangreiches Programm, das viele Ressourcen belegt (Speicher, Rechenzeit), würde es die Performance unnötig belasten, das Programm mehrmals gleichzeitig gestartet zu haben. Vielleicht hat der Anwender das erste Fenster nur minimiert und vergessen, dass er es bereits geöffnet hatte.
- Soll das Programm als eine Art Server oder Daemon laufen, macht es auch meist keinen Sinn, dass es zwei laufende Instanzen gibt.
- Belegt das Programm exklusiv eine wichtige Ressource (z.B. die Soundkarte, eine Schnittstelle oder eine Datenbankdatei, die es für andere sperrt), so kann das zweite gestartete Programm ohnehin nicht vernünftig arbeiten.
- Auch kann es oftmals sehr praktisch sein, alle wichtigen Informationen, die das Programm betreffen, in einer einzigen Instanz zu sammeln. So kann es nicht zu Inkonsistenzen (Unstimmigkeiten) zwischen mehreren Instanzen kommen. Auch Probleme, die entstehen würden, wenn mehrere Instanzen des Programms gleichzeitig versuchen, eine Datei zu schreiben, kann man so von vornherein unterbinden.

KDE bietet eine einfache Möglichkeit, einen solchen Test durchzuführen und unter Umständen wichtige Informationen vom neu gestarteten Programm zum alten Programm zu übertragen. Kernpunkt ist dabei die Klasse *KUniqueApplication*, eine Unterklasse von *KApplication*. Dementsprechend werden wir in der *main*-Funktion nun statt eines *KApplication*-Objekts ein *KUniqueApplication*-Objekt erzeugen. Der Rest des Programms ändert sich in der Regel kaum. Technisches Hilfsmittel für *KUniqueApplication* ist der DCOP-Server, der die Kommunikation zwischen KDE-Programmen ermöglicht (siehe Kapitel 4.20, *Interprozesskommunikation mit DCOP*). Der Test funktioniert daher auch nur dann, wenn das Programm auf einem laufenden KDE-System gestartet wird. Ist der DCOP-Server nicht erreichbar, startet das Programm ganz normal.

Schauen wir uns nun einmal an, was wir an unserem bisherigen Programm verändern müssen, um diesen Test durchführen zu lassen. Hier folgt der Quelltext unseres modifizierten Programms, die Änderungen sind wieder fett gedruckt:

```

#include <kuniqueapp.h>
#include <kcmdlineargs.h>
#include <klocale.h>
#include <qlabel.h>

static KCmdLineOptions options[] =
    {{"b", 0, 0},
      {"beep", I18N_NOOP ("Beep on startup"), 0},
      {"message <text>", I18N_NOOP ("Display this message"),
        "<H1>Hello, World!</H1>"},
      {0, 0, 0}};

int main (int argc, char **argv)
{
    KCmdLineArgs::init (argc, argv, "hello-kde",
        "The Hello-World program for KDE", "1.0");
    KCmdLineArgs::addCmdLineOptions (options);
    KUniqueApplication::addCmdLineOptions();
    KUniqueApplication app;
    QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
    l->show();
    app.setMainWidget (l);
    return app.exec();
}

```

(Wir haben dieses Mal aus Platzgründen und zur besseren Übersichtlichkeit auf das *QAboutData*-Objekt verzichtet. Aber auch das würde natürlich problemlos funktionieren.) Wir haben also nur die Klasse *KApplication* durch *KUniqueApplication* ersetzt, und auch die entsprechende Header-Datei *kuniqueapp.h* eingebunden. Außerdem haben wir noch mit *KUniqueApplication::addCmdLineOptions()* eine Option zur Liste der Optionen hinzugefügt, doch dazu später. (Im Moment funktioniert das Programm auch ohne diese Zeile.)

Kompilieren Sie das Programm wie üblich, und starten Sie es. Bis dahin ist der einzige merkbare Unterschied, dass das Programm nun automatisch im Hintergrund läuft. Wenn Sie es beispielsweise aus einer Konsole heraus starten (ohne ein Kaufmanns-Und & anzuhängen), wird die Konsole trotzdem nicht blockiert.

Starten Sie das Programm nun ein weiteres Mal. Sie werden bemerken, dass sich kein weiteres Fenster öffnet. Stattdessen wird das alte Fenster aktiviert. Wenn Sie das alte Fenster vorher minimieren, erscheint es wieder. Sogar wenn es auf einem anderen virtuellen Desktop liegt, wechselt der Window-Manager automatisch auf diesen Desktop.

Es gibt nun auch eine neue Kommandozeilenoption namens *--nofork*, die bewirkt, dass das Programm wieder auf herkömmliche Art gestartet wird, also ohne Test, ob es bereits läuft. Damit hat der Anwender die Möglichkeit, den Test ganz gezielt abzuschalten.

Leider haben unsere eigenen Kommandozeilenoptionen `--beep` und `--message` nun keinerlei Wirkung mehr, aber dem werden wir nun abhelfen. Die Optionen werden automatisch auf das alte Programm übertragen, dort werden sie nur zur Zeit nicht ausgewertet. Zuständig dafür ist die virtuelle Methode `KUniqueApplication::newInstance`. Sie wird immer dann (im alten Programm) aufgerufen, wenn ein neues Programm seine Kommandozeilenoptionen gesendet hat. Wir können nun eine eigene Klasse von `KUniqueApplication` ableiten und diese Methode durch eine eigene Methode ersetzen. `KCmdLineArgs::parsedArgs` liefert uns in dieser Methode bereits die Kommandoliste des neuen Programms. `newInstance` wird übrigens auch im Konstruktor von `KUniqueApplication` aufgerufen; es reicht daher, wenn wir nur dort die Optionen untersuchen, und nicht mehr in `main`. Da wir eine eigene Klasse erzeugen, benötigen wir zwei weitere Dateien: die Header-Datei und die Code-Datei für diese Klasse.

Hier sehen Sie die Header-Datei namens *myuniqueapp.h*:

```
#ifndef _MYUNIQUEAPP_H_
#define _MYUNIQUEAPP_H_

#include <kuniqueapp.h>
class QLabel;

class MyUniqueApplication : public KUniqueApplication
{
    Q_OBJECT

public:
    MyUniqueApplication (bool allowStyles = true,
                        bool GUIenabled = true);
    int newInstance ();

private:
    QLabel *l;
};

#endif
```

Da wir später in `newInstance` auf das `QLabel`-Objekt zugreifen wollen, legen wir eine Attributvariable `l` an, die die Adresse speichert. Außerdem müssen wir den Konstruktor definieren (mit den gleichen Parametern wie `KUniqueApplication`, die wir einfach weitergeben werden) sowie die Methode `newInstance` überschreiben.

Der Code für die Methoden in *myuniqueapp.cpp* sieht nun so aus:

```
#include "myuniqueapp.h"
#include <kcmlineargs.h>
#include <qlabel.h>
```



```

MyUniqueApplication::MyUniqueApplication
    (bool allowStyles, bool GUIenabled)
: KUniqueApplication (allowStyles, GUIenabled)
{
    l = new QLabel (0);
    l->show();
    setMainWidget (l);
}

int MyUniqueApplication::newInstance ()
{
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();
    if (args->isSet ("beep"))
        KApplication::beep();
    l->setText (args->getOption ("message"));
}

```

Wir legen das *QLabel*-Objekt nun im Konstruktor unserer Klasse an, dieses Mal noch ohne Text-Inhalt, denn der kommt erst in *newInstance* hinein. In *newInstance* interpretieren wir nun die Kommando-Optionen und erzeugen entsprechend einen Piepton und ändern den Text im Fenster.

Das Hauptprogramm in *hello-kde.cpp* ist nun noch ein gutes Stück kürzer geworden, da der größte Teil der Initialisierung jetzt in *MyUniqueApplication* gemacht wird:

```

#include <kaboutdata.h>
#include "myuniqueapp.h"
#include <kcmdlineargs.h>
#include <klocale.h>

static KCmdLineOptions options[] =
    {{"b", 0, 0},
      {"nobeep", I18N_NOOP ("Beep on startup"), 0},
      {"message <text>", I18N_NOOP ("Display this message"),
        "<H1>Hello, World!</H1>"},
      {0, 0, 0}};

int main (int argc, char **argv)
{
    KCmdLineArgs::init (argc, argv, "hello-kde",
        "The Hello-World program for KDE", "1.0");
    KCmdLineArgs::addCmdLineOptions (options);
    MyUniqueApplication::addCmdLineOptions();
    MyUniqueApplication app;
    return app.exec();
}

```

Wenn Sie dieses Programm kompilieren wollen, dürfen Sie nicht vergessen, zunächst den *moc* auf die Header-Datei *myuniqueapp.h* anzuwenden. Genauere Informationen finden Sie in Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*.

Manchmal möchte man bei einem erneuten Aufruf des Programms aber auch ein neues Fenster öffnen. Besonders häufig benutzt man das in Programmen, die mehrere Dokumente anzeigen. Diese verwenden meist je ein Fenster pro Dokument (siehe Kapitel 3.5.7, *Applikationen für mehrere Dokumente*). Der Anwender merkt hierbei nicht, dass er gar keine zweite Instanz gestartet hat. Auch diese Möglichkeit können wir ganz leicht mit unserem Beispielprogramm demonstrieren. Dazu müssen Sie nur die Dateien *myuniqueapp.h* und *myuniqueapp.cpp* leicht ändern:

myuniqueapp.h:

```
#ifndef _MYUNIQUEAPP_H_
#define _MYUNIQUEAPP_H_

#include <kuniqueapp.h>

class MyUniqueApplication : public KUniqueApplication
{
    Q_OBJECT

public:
    MyUniqueApplication (bool allowStyles = true,
                        bool GUIenabled = true);
    int newInstance ();
};

#endif
```

myuniqueapp.cpp:

```
#include "myuniqueapp.h"
#include <kcmdlineargs.h>
#include <qlabel.h>

MyUniqueApplication::MyUniqueApplication
    (bool allowStyles, bool GUIenabled)
: KUniqueApplication (allowStyles, GUIenabled)
{
    connect (this, SIGNAL (lastWindowClosed()),
            this, SLOT (quit()));
}

int MyUniqueApplication::newInstance ()
```

```

{
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();
    if (args->isSet ("beep"))
        KApplication::beep();
    QLabel *l =
        new QLabel (args->getOption ("message"), 0);
    l->show();
}

```

Dieses Mal erzeugen wir das *QLabel*-Objekt in *newInstance*, und zwar bei jedem Aufruf. Da wir nun kein zentrales Hauptfenster haben, das mit *setMainWidget* eingetragen wird, müssen wir auf andere Art dafür sorgen, dass das Programm korrekt beendet wird. Dazu verbinden wir das Signal *lastWindowClosed* mit dem Slot *quit*. (Das brauchen Sie übrigens nicht, wenn Sie als Fensterklasse die Klasse *KMainWindow* bzw. Ihre eigene abgeleitete Klasse benutzen, wie es in Kapitel 3.5.1, *Das Hauptfenster*, beschrieben wird. Diese Klasse sorgt selbst dafür.) Es sieht fast so aus, als würde nun doch wieder pro Aufruf ein Programm gestartet, aber ein Blick auf die Prozessliste zeigt eindeutig, dass es nur einen laufenden Prozess, *hello-kde*, gibt.

Zum Schluss noch ein kleiner Hinweis zur Effizienz: Im Konstruktor von *KUniqueApplication* wird eine ganze Reihe von Initialisierungen vorgenommen: Beispielsweise wird die Verbindung zum X-Server aufgebaut, Konfigurationsdateien und Übersetzungstabellen werden geladen. Falls das Programm bereits vorher lief, so werden anschließend die Daten an die alte Instanz übergeben, und das Programm beendet sich wieder – alle Initialisierungen waren also unnötig. Um hier ein wenig Rechnerleistung zu sparen, besitzt die Klasse *KUniqueApplication* die statische Methode *start*. Sie führt die Überprüfung auf eine andere bereits laufende Instanz durch, ohne ein *KApplication*-Objekt mit allen daraus resultierenden Initialisierungen anzulegen. Um diese Methode zu benutzen, ersetzen Sie einfach in Ihrem Programm die Zeile

```
KUniqueApplication app;
```

durch die Zeilen:

```

if (!KUniqueApplication::start ())
{
    fprintf (stderr, "Das Programm läuft schon!\n");
    exit (0);
}
KUniqueApplication app;

```

3.4 Hintergrund: Event-Verarbeitung

Ereignisse, die von außerhalb des Programms kommen und die eine Auswirkung auf die Applikation haben, werden *Events* genannt. Die meisten dieser Events kommen vom X-Server, zum Beispiel Maus-Events, Tastatur-Events, Änderungen der Größe und Position von Fenstern, Sichtbarwerden von Fensterbereichen usw. Bereits die Klasse *QObject* besitzt die virtuelle Methode *event*, um Events entgegenzunehmen. Allerdings kann *QObject* nur Timer-Events und Child-Events erhalten, die von Qt selbst erzeugt werden, also nicht vom X-Server kommen. *QWidget*, das ja von *QObject* abgeleitet ist, übernimmt die Event-Funktionalität und baut sie aus, da *QWidget* eine große Anzahl verschiedener Events – insbesondere vom X-Server – erhalten kann. Abbildung 3.13 verdeutlicht die Abarbeitung von Events in Qt.

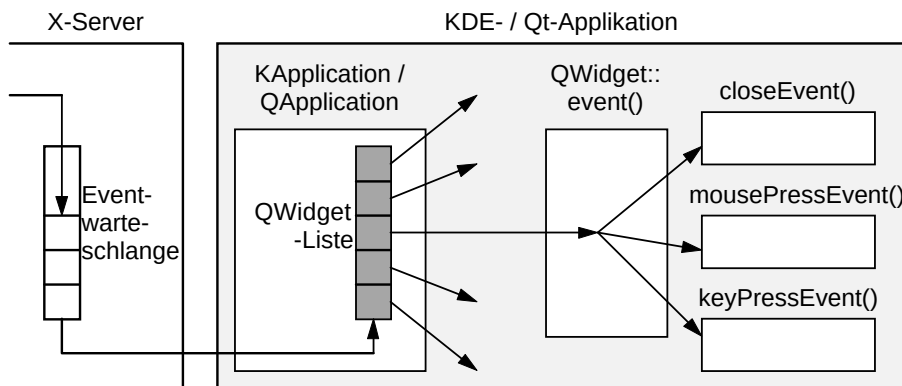


Abbildung 3-13 Die Event-Verteilung

Die Events werden im X-Server in einer Warteschlange – der so genannten *Event-Queue* – gespeichert. Dort werden sie in der Haupt-Event-Schleife des Objekts der Klasse *QApplication* (bzw. *KApplication*) abgefragt. Die Events werden an das zugehörige *QWidget*-Objekt (oder *QObject*) weitergeleitet, indem dessen Methode *event* aufgerufen wird. In dieser Methode wird die Art des Events festgestellt und abhängig von der Event-Art die passende Event-Methode aufgerufen. Dort findet dann die eigentliche Reaktion auf den Event statt.

Die meisten Events enthalten noch zusätzliche Informationen. Beim Drücken einer Maustaste werden beispielsweise auch die gedrückte Maustaste und die Mausposition zum Zeitpunkt des Klicks gespeichert. Diese Informationen werden in einem Objekt der Klasse *QEvent* abgelegt und der Methode *event* übergeben. Für die verschiedenen Event-Arten gibt es Unterklassen von *QEvent* (z.B. *QMouseEvent*, *QCloseEvent* usw.). Die Methode *event* führt eine Typumwandlung auf die entsprechende Unterklasse durch, bevor sie die spezielle Event-Routine aufruft. Das Objekt dieser Event-Klasse wird der Event-Routine als Parameter übergeben.

Alle GUI-Elemente müssen auf den einen oder anderen Event reagieren. Ein Button beispielsweise reagiert sowohl auf die Maus als auch auf die Tastatur, ebenso ein Eingabefeld. Aber auch ein Element wie *QLabel*, das nur einen Text anzeigt, der sich nicht ändert, reagiert zumindest auf den Event *paintEvent*, der signalisiert, dass ein Teil oder das ganze Widget neu gezeichnet werden muss. Bei der Entwicklung eines GUI-Elements muss man daher Events empfangen und geeignete Reaktionen ausführen. Es gibt dazu mehrere Möglichkeiten:

- Die gängigste Art, auf Events zu reagieren, ist es, die speziellen Event-Methoden wie *paintEvent* oder *mousePressEvent* in der neu definierten Klasse zu überschreiben. Dazu sind diese Methoden in *QWidget* als virtuelle Methoden definiert. Man braucht dann nur die Event-Methoden zu überschreiben, auf die das neue Widget speziell reagieren soll. Ein weiterer Vorteil ist, dass man die Event-Informationen bereits in einer Event-Klasse vom richtigen Typ erhält, da *event* bereits die Typumwandlung durchführt.
- Man kann auch die Methode *event* überschreiben, da auch sie virtuell ist. In diesem Fall hat man die volle Kontrolle über alle Events, muss aber selbst die verschiedenen Event-Arten unterscheiden. Insbesondere muss man alle wichtigen Events behandeln, also auch die, für die *QWidget* bereits eine Default-Implementierung vorgesehen hat. In seltenen Fällen ist es aber unumgänglich, *event* zu überschreiben, wenn man auf einen exotischen Event reagieren will, der von *event* nicht einer speziellen Event-Methode zugeordnet wird. (Eine Liste aller Event-Typen befindet sich in der Datei *qevent.h*.) So wird beispielsweise der Event-Typ *Hide*, der generiert wird, bevor ein Widget versteckt wird, von *event* nicht bearbeitet. Es empfiehlt sich, die Methode wie folgt zu überschreiben:

```
bool QWidget::event (QEvent *e)
{
    if (e->type() == QEvent::Hide)
    {
        // hier die spezielle Reaktion
        return true; // als Zeichen für "Event erkannt"
    }
    else
        // sonst einfach die alte Event-Routine nutzen
        return QWidget::event (e);
}
```

- Man kann Events über den so genannten Event-Filtermechanismus an ein anderes Objekt umleiten lassen. Das kann man nutzen, wenn man beispielsweise mehrere Klassen um die gleiche Funktionalität erweitern will, ohne von jeder Klasse eine neue Klasse abzuleiten. Dazu überladen Sie die Methode *eventFilter* in einer eigenen, von *QObject* abgeleiteten Klasse. Mit der Methode *QObject::setEventFilter* können Sie anschließend ein Objekt Ihrer selbst defi-

nierten Klasse als Filter einsetzen lassen. Dieses Objekt erhält nun alle Events, die an das andere Objekt gehen sollten. Dieses Konzept kann aber nur in wenigen Spezialfällen sinnvoll eingesetzt werden.

Widgets reagieren in der Regel auf ein Event mit dem Neuzeichnen des Bildschirminhalts oder von Teilen davon und/oder mit der Versendung von Signalen. Die Klasse *QPushButton* beispielsweise reagiert auf einen *mousePressEvent* damit, dass sie den Button eingedrückt zeichnet und das Signal *pressed()* aussendet. Ein anschließender *mouseReleaseEvent* bewirkt, dass der Button wieder nach vorn gezeichnet wird und das Signal *released()* ausgesendet wird. Befindet sich der Maus-Cursor bei diesem Event auch noch innerhalb der Schaltfläche, wird außerdem das Signal *clicked()* gesendet, und falls es sich um einen Toggle-Button handelt, wird auch noch das Signal *toggled(bool)* abgeschickt (wobei dann allerdings der Button nach dem Klicken weiterhin eingedrückt erscheint). Das Signal *clicked()*, das am häufigsten verwendet wird, wird also nur dann ausgesendet, wenn sowohl beim Drücken als auch beim Loslassen der Maustaste der Mauszeiger auf der Schaltfläche war.

Der Kontrollfluss innerhalb der meisten Widgets kann durch die Grafik in Abbildung 3.14 veranschaulicht werden.

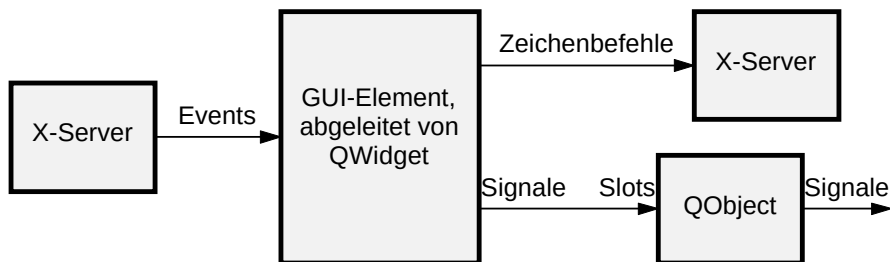


Abbildung 3-14 Kontrollfluss bei einem GUI-Objekt

Ein Event des X-Servers kann also eine ganze Kaskade von Zeichenbefehlen und Signalen auslösen. KDE- und Qt-Programme verbringen fast ihre ganze Zeit damit, in der Haupt-Event-Schleife auf einen neuen Event vom X-Server zu warten. (Das geschieht mit der Betriebssystemfunktion *select()*, die den Prozess schlafen legt, bis ein neuer Event vorliegt oder der nächste Timer-Event fällig ist.) Die ganze Arbeit innerhalb des Programms erfolgt als Reaktion auf einen Event. Erst nach der Rückkehr aus allen Slot- und Event-Methoden in die Haupt-Event-Schleife wird auf den nächsten Event gewartet.

Alle Events können auch von Hand erzeugt und an ein Objekt geschickt werden. Qt benutzt diesen Mechanismus, um auch andere Ereignisse, die nicht vom X-Server kommen, an Objekte zu verteilen (zum Beispiel Child-Events). Außerdem kann man auf diese Weise leicht ein Verhalten der Maus oder eine Tastatur-

eingabe simulieren. Einen »gefälschten« Mausklick mit der linken Maustaste an den Koordinaten (10,10) des Widgets *w* (Typ: *QWidget**) kann man zum Beispiel folgendermaßen verschicken:

```
QMouseEvent e (QEvent::MouseButtonPress,  
               QPoint (10, 10),  
               LeftButton, NoButton);  
QApplication::sendEvent (w, &e);
```

3.5 Das Hauptfenster

Nahezu alle Programme mit einer grafischen Benutzeroberfläche stellen auf dem Bildschirm ein mehr oder weniger großes *Hauptfenster* dar, dessen Aufbau immer ähnlich ist: Es enthält – in der Regel am oberen Rand – eine *Menüleiste* mit einer Hand voll Befehlskategorien (DATEI, BEARBEITEN, HILFE), hinter denen sich jeweils Befehlslisten in Form von Popup-Menüs befinden. Darunter befinden sich eine oder mehrere *Werkzeugleisten* mit Icons, die die wichtigsten Befehle aus der Menüleiste darstellen. Darunter liegt der *Anzeigebereich* – der Bereich, der die eigentlichen Daten darstellt, mit denen das Programm arbeitet. Dieser Bereich ist natürlich von der Funktion des Programms abhängig und sieht daher auch für jedes Programm anders aus. Am unteren Rand des Hauptfensters befindet sich meistens eine Statuszeile, in der der aktuelle Zustand, wichtige Meldungen oder kurze Hilfetexte angezeigt werden. Abbildung 3.15 zeigt das Hauptfenster von *Konqueror*. Unter der Menüleiste befinden sich in diesem Fall zwei Werkzeugleisten (eine mit Icons, eine mit einem Feld zur Eingabe einer URL). Darunter ist der Anzeigebereich, der sich hier noch einmal in einen linken Teil mit einem Verzeichnisbaum und einen rechten Teil aufspaltet, in dem eine Liste von Dateien angezeigt wird. Am unteren Rand des Hauptfensters ist die Statuszeile, die aktuelle Informationen über die ausgewählten Dateien anzeigt.

Abbildung 3.16 zeigt das Hauptfenster eines einfachen Texteditors, den wir im Laufe dieses Kapitels entwickeln wollen. Auch er hat eine Menü- und eine Werkzeugleiste. Auf eine Statuszeile haben wir verzichtet, um das Beispiel möglichst einfach zu halten.

In diesem Kapitel werden wir uns anschauen, wie ein solches Hauptfenster in einem KDE-Programm erstellt wird und wie man die Menüs mit Daten füllt. Die KDE-Bibliothek bietet hierfür eine Reihe von Klassen an, die im Einzelnen in den Kapiteln 3.5.1 bis 3.5.5 beschrieben werden.

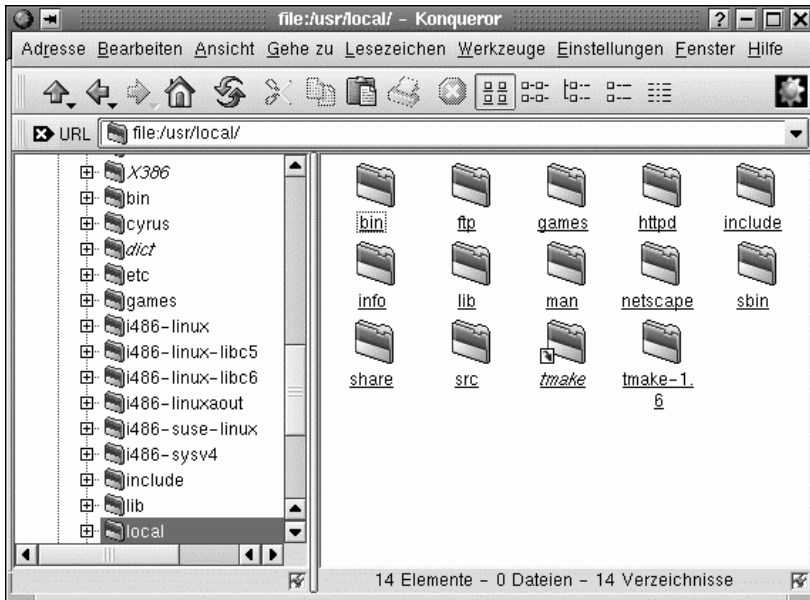


Abbildung 3-15 Das Hauptfenster von Konqueror

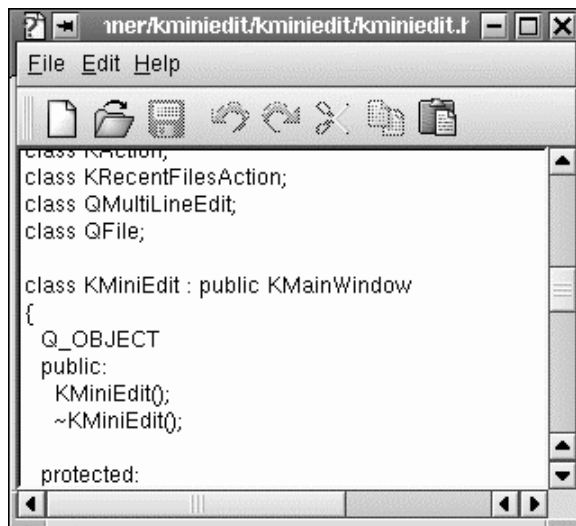


Abbildung 3-16 Das Hauptfenster eines einfachen Texteditors

Neben einfachen Hauptfenstern gibt es auch Applikationen, die mehrere Hauptfenster öffnen, beispielsweise um mehrere Dokumente gleichzeitig zu öffnen. Wie Sie ein solches Programm aufbauen, erfahren Sie in Kapitel 3.5.7, *Applikation für mehrere Dokumente*, im Abschnitt *KMiniEdit mit SDI*. Alternativ kann man auch alle Dokumente als kleine Unterfenster in einem einzigen Hauptfenster dar-

stellen. Diese Variante wird im darauffolgenden Abschnitt, *KMiniEdit mit MDI*, beschrieben. Zur besseren Programmstrukturierung hat es sich außerdem durchgesetzt, dass man möglichst die Verwaltung eines Dokuments und die Anzeige des Dokuments in verschiedenen Klassen realisiert. Wie Sie dieses Konzept in eigenen Programmen einsetzen, erfahren Sie in Kapitel 3.5.8, *Das Document-View-Konzept*.

In reinen Qt-Programmen muss man leider auf die Klassen der KDE-Bibliothek verzichten und stattdessen auf die Klassen der Qt-Bibliothek zurückgreifen. Welche Klassen das sind und welche Unterschiede es dabei gibt, erfahren Sie in Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*.

3.5.1 Ableiten einer eigenen Klasse von *KMainWindow*

Das Hauptfenster wird in KDE-Programmen durch die Klasse *KMainWindow* erzeugt. Da es sich dabei natürlich um ein Fenster handelt, ist *KMainWindow* eine Unterklasse von *QWidget*. Diese Klasse implementiert eine Reihe von Funktionalitäten:

- Sie erzeugt und verwaltet die Menüleiste (Objekt der Klasse *KMenuBar*).
- Sie erzeugt und verwaltet beliebig viele Werkzeugleisten (Objekte der Klasse *KToolBar*).
- Sie kann die Reihenfolge und Anordnung der Befehle in der Menüleiste und den Werkzeugleisten aus einer XML-Datei auslesen.
- Sie erzeugt und verwaltet die Statuszeile (Objekt der Klasse *KStatusBar*).
- Sie enthält virtuelle Methoden, in denen man Sicherheitsabfragen beim Schließen des Fensters unterbringen kann.
- Sie enthält Methoden zum Speichern und Laden der Fensterkonfiguration, wenn sich der Anwender ausloggt, während das Programm noch geöffnet ist. Nähere Informationen hierzu finden Sie in Kapitel 4.16, *Session-Management*.

Es ist im Allgemeinen üblich, beim Entwurf eines eigenen Programms eine eigene Unterklasse der Klasse *KMainWindow* zu schreiben. In unserem Minimalprogramm in Kapitel 2.3, *Das erste KDE-Programm*, war das noch nicht nötig, da wir nur ein extrem simples Hauptfenster hatten. Sobald aber viele Menübefehle benutzt werden und der Anzeigebereich komplexer wird, ist das Implementieren einer eigenen Klasse sehr zu empfehlen. In dieser eigenen Klasse füllt man im Konstruktor die Menüs mit Befehlen und erzeugt das Unterfenster für den Anzeigebereich. Außerdem kann man in der abgeleiteten Klasse Slots definieren, die mit den Menübefehlen verbunden werden und die die gewünschten Reaktionen ausführen.

Beispiel

Als einfaches Beispiel wollen wir unser Programm aus Kapitel 2.3, *Das erste KDE-Programm*, nun mit einer eigenen Unterklasse von *KMainWindow* implementieren. Zusätzlich erweitern wir unser Programm um eine Sicherheitsabfrage beim Aufruf von DATEI-BEENDEN, um zu demonstrieren, wie zusätzliche Slots definiert werden. Unsere Klasse nennen wir *MyMainWindow*, und wir implementieren sie in den Dateien *mymainwindow.h* und *mymainwindow.cpp*. (Wie Sie die Klasse nennen, ist natürlich Ihnen überlassen. Der Klassenname *MyMainWindow* ist natürlich nicht sehr aussagekräftig. Es hat sich hier eingebürgert, für diese Klasse den Namen der Applikation selbst zu benutzen. In unserem Fall wäre also der Klassenname *KHelloWorld* möglich und sinnvoll. Dementsprechend sollten die Dateien *khelloworld.h* und *khelloworld.cpp* heißen. Um Verwirrungen zu vermeiden, benutzen wir hier aber zunächst einmal den Namen *MyMainWindow*.) Auch unser Hauptprogramm in *main.cpp* muss natürlich angepasst werden.

Datei *main.cpp*:

```
#include <kapp.h>
#include <kcmdlineargs.h>
#include <klocale.h>

#include "mymainwindow.h"

int main(int argc, char **argv)
{
    QString aboutText ("KDE- und Qt-Programmierung\n"
                      "(c) 2000 Addison-Wesley-Germany");
    KCmdLineArgs::init (argc, argv, "khelloworld",
                      aboutText, "1.0");
    KApplication app;

    MyMainWindow *top = new MyMainWindow ();
    top->show();

    return app.exec();
}
```

Datei *mymainwindow.h*:

```
#ifndef MYMAINWINDOW_H
#define MYMAINWINDOW_H

#include <kmainwindow.h>

class QLabel;

class MyMainWindow : public KMainWindow
{
```

```

    Q_OBJECT
public:
    MyMainWindow();
    ~MyMainWindow();

private slots:
    void fileQuit();

private:
    QLabel *text;
};

#endif

```

Datei *mymainwindow.cpp*:

```

#include <kaction.h>
#include <kapp.h>
#include <klocale.h>
#include <kmenubar.h>
#include <kmessagebox.h>
#include <kstdaction.h>
#include <qlabel.h>
#include <qpopupmenu.h>

#include "mymainwindow.h"

MyMainWindow::MyMainWindow() : KMainWindow()
{
    text = new QLabel (i18n
        ("<H1>Hello, World!</H1>"), this);
    setCentralWidget (text);

    QPopupMenu *filePopup = new QPopupMenu (this);
    KAction *quitAction = KStdAction::quit (this,
        SLOT (fileQuit()), actionCollection());
    quitAction->plug (filePopup);

    menuBar()->insertItem (i18n ("%File"), filePopup);
    menuBar()->insertSeparator();
    menuBar()->insertItem (i18n ("%Help"), helpMenu());
}

MyMainWindow::~MyMainWindow()
{
}

void MyMainWindow::fileQuit()
{
}

```

```

    int really = KMessageBox::questionYesNo (this,
        i18n ("Do you really want to quit?"));
    if (really == KMessageBox::Yes)
        kapp->quit();
}

```

Kompilieren

Da wir eine eigene Unterklasse von *KMainWindow* – und damit auch eine Unterklasse von *QObject* – erzeugen, müssen wir zunächst den *moc* aufrufen (siehe Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*):

```
% moc mymainwindow.h -o mymainwindow.moc.cpp
```

Anschließend können wir die Code-Dateien einzeln kompilieren und dann linkern:

```

% g++ -c mymainwindow.cpp -I$QTDIR/include -I$KDEDIR/include
% g++ -c mymainwindow.moc.cpp -I$QTDIR/include -I$KDEDIR/include
% g++ -c main.cpp -I$QTDIR/include -I$KDEDIR/include

```

Als letzten Schritt linkern wir alle erzeugten Objektdateien zu einer ausführbaren Datei zusammen:

```
% g++ *.o -o khelloworld -lkdeui -lkdecore -lqt
```

Hier müssen Sie eventuell wieder die Optionen *-L\$KDEDIR/lib* und *-L\$QTDIR/lib* anfügen, damit die KDE- und Qt-Bibliotheken auch wirklich gefunden werden.

Um den Aufwand beim Kompilieren zu verringern, empfiehlt es sich, ein Makefile zu benutzen. Am einfachsten verwenden Sie zur Erstellung des Makefiles ein Tool wie beispielsweise *tmake* (siehe Kapitel 5.1, *tmake*) oder benutzen direkt eine integrierte Entwicklungsumgebung wie *KDevelop* (siehe Kapitel 5.5, *KDevelop*). Der Zeitaufwand, den Sie für die Einarbeitung in diese Tools benötigen, wird schnell durch die Zeit wettgemacht, die Sie beim Kompilieren sparen.

Wenn Sie das Programm fertiggestellt haben, können Sie es starten. Sie sollten ein Fenster erhalten, das ähnlich wie in Abbildung 3.17 aussieht.

Analyse des Beispiels

In unserem Hauptprogramm in *main.cpp* erzeugen wir wie gehabt das *KApplication*-Objekt. Anschließend erzeugen wir dynamisch auf dem Heap ein Fenster-Objekt unserer selbst geschriebenen Hauptfensterklasse *MyMainWindow*. Der Konstruktor benötigt keine Parameter, denn das Fenster wird immer ein vaterloses Toplevel-Widget. Anschließend zeigen wir das Fenster mit der Methode *show*. (Der Aufruf von *show* könnte auch im Konstruktor von *MyMainWindow* erfolgen, es hat sich allerdings eingebürgert, dem Hauptprogramm hier die Kontrolle zu überlassen.) Nachdem nun das Hauptfenster angezeigt wird, kann die Haupt-Event-Schleife mit der Methode *exec* gestartet werden.



Abbildung 3-17 Die Ausgabe unserer eigenen Hauptfensterklasse

Die Datei *mymainwindow.h* enthält die Deklaration der neuen Klasse *MyMainWindow*. (Wie allgemein üblich, rahmen wir die Header-Dateien in ein beginnendes `#ifndef xxx #define xxx` und ein abschließendes `#endif` ein. *xxx* ist dabei ein String, der aus dem Dateinamen abgeleitet ist, um eindeutig zu sein. Auf diese Weise verhindert man, dass Header-Dateien mehrfach durchlaufen werden.) Unsere Klasse ist von *KMainWindow* abgeleitet, erbt also automatisch alle Methoden – und somit auch die Funktionalitäten. Da *KMainWindow* eine Unterklasse von *QObject* ist, müssen wir das Makro `Q_OBJECT` als erste Zeile einfügen. Nun können wir eigene Slots deklarieren, wie beispielsweise den Slot *fileQuit*. Er ist hier als *private slot* deklariert, da wir ihn nur innerhalb dieser Klasse ansprechen. Dieser Slot soll aufgerufen werden, wenn der Anwender den Menübefehl FILE-QUIT aufruft. Er fragt dann den Anwender noch einmal, ob das Programm tatsächlich beendet werden soll. (Sofern beim Beenden eines Programms keine wichtigen ungesicherten Einstellungen oder Daten verloren gehen, sollte man diese Sicherheitsabfrage nicht stellen. Man kann davon ausgehen, dass der Anwender weiß, was er tut, wenn er das Programm beenden will. Die Sicherheitsabfrage ist in diesem Fall eher lästig und wenig hilfreich. In unserem Beispiel wird diese Sicherheitsabfrage nur eingeführt, um die Nutzung eigener Slots zu demonstrieren. Man kann die Sicherheitsabfrage übrigens auch umgehen, indem man einfach das Fenster schließt. Wie man das verhindert, wird in Kapitel 3.5.6, *Applikation für ein Dokument*, besprochen.) Weiterhin enthält die Klasse einen Zeiger auf ein *QLabel*-Objekt namens *text*. Dieses Objekt stellt den Begrüßungstext im Hauptfenster dar. Das Objekt wird im Konstruktor von *MyMainWindow* angelegt und hier abgelegt, so dass wir auch in anderen Methoden noch auf das Objekt zugreifen könnten (was im derzeitigen Zustand des Programms aber noch nicht passiert).

In der Datei *mymainwindow.cpp* ist nun der Code für die neue Klasse enthalten. Der Konstruktor erzeugt zunächst das *QLabel*-Objekt mit dem Text »Hello, World!« (bzw. dem Text in der eingestellten Landessprache). Das *QLabel*-Objekt muss ein Unter-Widget des Hauptfenster-Objekts sein, denn es wird innerhalb

dieses Fensters dargestellt. Damit es automatisch den verbleibenden Platz im Hauptfenster einnimmt, wird *setCentralWidget* aufgerufen. Von nun an übernimmt das Hauptfenster die Kontrolle über die Größe und Position des *QLabel*-Objekts. Anschließend wird das Menü erzeugt, ganz analog wie in unserem ursprünglichen Minimalprogramm. Es entfallen lediglich einige Zugriffe auf die Variable *top*, da wir uns hier ja innerhalb der Hauptfensterklasse selbst befinden. Außerdem verbinden wir die Aktion *quit* nicht direkt mit dem Slot *quit* von *KApplication*. Stattdessen verbinden wir sie mit dem eigenen Slot *fileQuit*. (Durch die Übergabe von *this* als erstem und *SLOT(fileQuit())* als zweiten Parameter wird automatisch ein *connect* durchgeführt zwischen dem *KAction*-Objekt und dem angegebenen Slot des Hauptfensters.) Diese selbst definierte Slot-Methode *fileQuit* lässt zunächst einmal einen Dialog auf dem Bildschirm erscheinen, den der Anwender beantworten muss. (An dieser Stelle wollen wir nicht näher auf die Klasse *KMessageBox* eingehen. Nähere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.) Je nach Ergebnis dieser Frage wird die Methode *quit* des *KApplication*-Objekts aufgerufen oder nicht.

3.5.2 Der Anzeigebereich

Der größte Bereich des Hauptfensters wird vom Anzeigebereich belegt, in dem die Applikation in der Regel die Daten darstellt, mit denen sie arbeitet. Was hier angezeigt wird, hängt natürlich stark von der Applikation und ihrer Aufgabe ab. Ein Texteditor wird hier das Textfenster anzeigen, ein Grafikeditor die erstellte Grafik, ein CD-Player wird an dieser Stelle vielleicht die Anzeigeeinheit mit Titelnummer und verstrichener Zeit anzeigen und ein Schachprogramm das Brett mit den Spielfiguren sowie eine Liste der gespielten Züge.

Der Anzeigebereich wird in jedem Fall von einem einzelnen Objekt der Klasse *QWidget* oder einer abgeleiteten Klasse ausgefüllt, dem so genannten Anzeigefenster. Der *parent*-Eintrag des Widgets muss in jedem Fall das Hauptfenster sein. Mit der Methode *KMainWindow::setCentralWidget* aktiviert man schließlich das Anzeigefenster. Von dem Moment an wird die Größe des Anzeigefensters automatisch immer so angepasst, dass das Hauptfenster vollständig von Menüleiste, Werkzeugleiste, Statuszeile und Anzeigefenster ausgefüllt ist. Da immer nur ein Widget pro Hauptfenster Anzeigefenster sein kann, hat immer nur der letzte Aufruf von *setCentralWidget* eine Wirkung.

Als Klasse für ein Anzeigefenster kommen alle von *QWidget* abgeleiteten Klassen in Frage. In der Regel wird einer der folgenden drei Fälle auftreten:

Fertiges Widget der Qt- oder KDE-Bibliothek

Im einfachsten Fall enthält ein bereits fertig definiertes Widget der Qt- oder KDE-Bibliothek die benötigte Funktionalität zum Anzeigen und Manipulieren der Daten. In unserem Beispiel hatten wir *QLabel* zum Anzeigen eines festen Texts

benutzt. Für einen einfachen Texteditor kommt beispielsweise *QMultiLineEdit* in Frage (siehe auch Kapitel 3.5.6, *Applikation für ein Dokument*, sowie Kapitel 3.7.6, *Eingabeelemente*). Für die Anzeige einfacher Tabellendaten – auch hierarchisch geordnet – können Sie beispielsweise *QListView* benutzen (siehe auch Kapitel 3.7.5, *Auswahlelemente*).

Unterteilter Anzeigebereich

Soll der Anzeigebereich unterteilt werden, so benutzen Sie eine der in Kapitel 3.7.7, *Verwaltungselemente*, beschriebenen Klassen und fügen in dieses Objekt die gewünschten Unter-Widgets ein. Wollen Sie beispielsweise den Anzeigebereich fest unterteilen und im oberen Bereich ein *QLabel*-Objekt und im unteren ein *QMultiLineEdit*-Objekt verwenden, so benutzen Sie die Klasse *QVBox*. (Alternativ können Sie auch die Klasse *QWidget* verwenden und ein Layout hinzufügen. Genauere Informationen finden Sie in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*.) Der entsprechende Code im Konstruktor Ihrer Hauptfensterklasse sieht dann so aus:

```
MyMainWindow::MyMainWindow() : KMainWindow()  
{  
    QVBox *box = new QVBox (this);  
    new QLabel (i18n("<H1>Hello, World!</H1>"), box);  
    new QMultiLineEdit (box);  
    setCentralWidget (box);  
    ...  
}
```



Abbildung 3-18 Der Anzeigebereich wurde in einen oberen und einen unteren Bereich unterteilt.

Wollen Sie die Größenverhältnisse nachträglich noch ändern, so verwenden Sie *QSplitter*. Dadurch erhalten Sie eine Trennlinie zwischen den Unter-Widgets, die der Anwender mit der Maus verschieben kann:

```

MyMainWindow::MyMainWindow() : KMainWindow()
{
    QSplitter *box = new QSplitter (Qt::Vertical, this);
    new QLabel (i18n("<H1>Hello, World!</H1>"), box);
    new QMultiLineEdit (box);
    setCentralWidget (box);
    ...
}

```

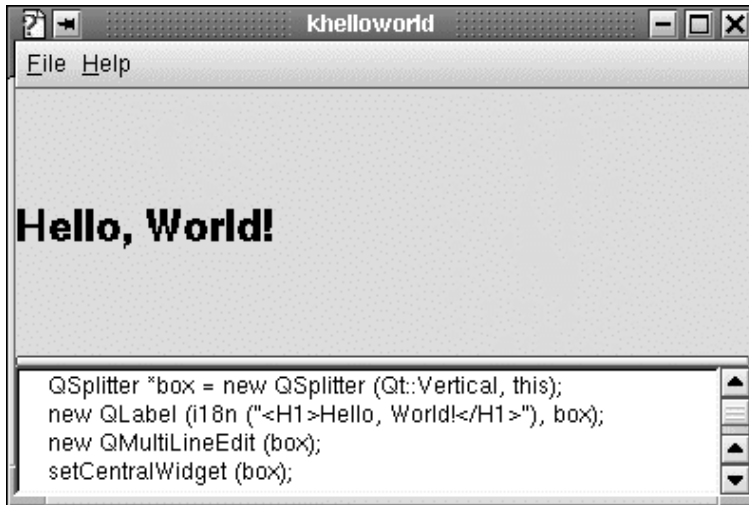


Abbildung 3-19 Mit QSplitter unterteilter Bereich

Eigene Widget-Klasse

Reicht die Funktionalität der vorhandenen Widget-Klassen nicht aus (das ist bei nahezu allen aufwendigeren Darstellungsarten der Fall), erstellen Sie eine eigene Widget-Klasse. Eine genaue Anleitung finden Sie in Kapitel 4.4, *Entwurf eigener Widget-Klassen*. Es ist allgemein üblich, als Klassennamen den Applikationsnamen zu benutzen und das Wort *View* anzuhängen.

Um wirklich international zu sein, könnte unser Hello-World-Programm statt eines Texts ein lachendes Gesicht zeigen. Das wird von jedem verstanden. Der Code dazu kann etwa folgendermaßen aussehen. Auf die Klasse *KHelloWorldView* wollen wir hier nicht weiter eingehen. Die Zeichenbefehle werden in Kapitel 4.2, *Zeichnen von Grafikprimitiven*, genau beschrieben. (Alternativ könnte man auch ein *QLabel*-Objekt benutzen und darin eine Grafikdatei darstellen lassen. Dieses *QLabel*-Objekt hätte aber eine feste Größe.)

```

class KHelloWorldView : public QWidget
{
public:
    KHelloWorldView (QWidget *parent)

```



```

        : QWidget (parent) {}
protected:
    void paintEvent (QPaintEvent *)
    {
        QPainter p (this);
        p.setPen (QPen (black, 5));
        p.drawEllipse (width () / 3 - 5,
                        height () / 3 - 5, 10, 10);
        p.drawEllipse (2 * width () / 3 - 5,
                        height () / 3 - 5, 10, 10);
        p.drawArc (0, 0, width(), height() - 20,
                   200 * 16, 140 * 16);
    }
};

MyMainWindow::MyMainWindow() : KMainWindow()
{
    KHelloWorldView *view = new KHelloWorldView (this);
    setCentralWidget (view);
    ...
}

```

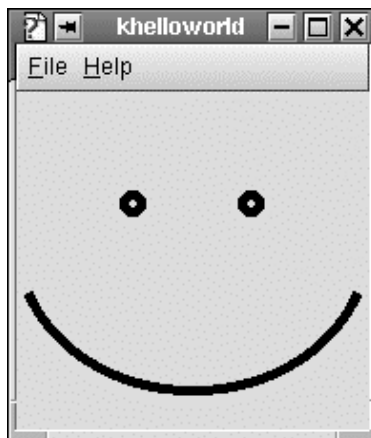


Abbildung 3-20 KHelloWorld mit eigener Klasse als Anzeigefenster

Wenn Sie Menübefehle vorsehen, die ausschließlich auf den Daten in der selbst definierten Anzeigeklasse arbeiten, so sollten Sie dort auch die entsprechenden Slots definieren. Die Befehle werden dann direkt mit den Slots der Anzeigeklasse verbunden und nicht mehr mit Slots der Hauptfensterklasse. So vereinfachen Sie die Hauptfensterklasse. Oftmals sind es vor allem die Befehle des Menüpunkts EDIT (deutsch BEARBEITEN), die nur auf die Anzeigeklasse wirken, zum Beispiel UNDO (RÜCKGÄNGIG), CUT (AUSSCHNEIDEN), COPY (KOPIEREN), PASTE (EINFÜGEN), FIND (SUCHEN), SELECT ALL (ALLES MARKIEREN) usw.

3.5.3 Definition von Aktionen für Menü- und Werkzeugleisten

Für die einzelnen Befehle einer Menü- oder Werkzeugleiste erzeugt man in der Regel Objekte der Klasse *KAction*. Ein solches Objekt speichert dabei eine Reihe von Eigenschaften für den Befehl:

- Befehlsbezeichnung – Dies ist der Text, der in einem Popup-Menü angezeigt wird, zum Beispiel NEW, SAVE AS... oder QUIT.
- Icon – In einer Werkzeugleiste wird diese Aktion durch dieses Icon dargestellt. In Popup-Menüs erscheint es links neben dem Befehltext.
- Tastatur-Kurzbefehl (Accelerator) – Über diese Tastenkombination kann der Befehl auch ausgeführt werden, ohne dass der Menüpunkt angewählt werden muss.
- Objekt und Slot-Methode – Dieser Slot wird aufgerufen, wenn die Aktion vom Anwender aufgerufen wird. Er führt den Befehl aus.

Einfache Befehlsaktionen

Als einfaches Beispiel wollen wir unser *KHelloWorld*-Programm um einen Befehl CLEAR (oder auf deutsch LÖSCHEN) erweitern. Dieser Befehl löscht den Begrüßungstext. Außerdem erzeugen wir eine Werkzeugleiste, die in unserem Fall zwei Icons enthält, einen für unseren neuen Befehl CLEAR und einen für den Befehl QUIT. Abbildung 3.21 zeigt unser neues Programm mit einer Werkzeugleiste. In diesem Fall ist die Werkzeugleiste so konfiguriert, dass sie auch den Text der Befehle anzeigt. Der Anwender kann diese Einstellung selbst vornehmen, indem er mit der rechten Maustaste auf die Werkzeugleiste klickt und unter TEXTPOSITION den Eintrag TEXT UNTER SYMBOLEN auswählt. Das Popup-Menü zum Befehl EDIT sehen Sie in Abbildung 3.22.



Abbildung 3-21 Jetzt gibt es eine Werkzeugleiste mit zwei Befehlen.



Abbildung 3-22 Popup-Menü zum Eintrag EDIT, mit neuem Befehl CLEAR

Die Aktion zum Befehl CLEAR muss nun erzeugt werden. Das geschieht am besten im Konstruktor der selbst definierten Hauptfensterklasse. Wir erzeugen dazu mit *new* ein neues *KAction*-Objekt, dem wir im Konstruktor nacheinander die Werte für Bezeichnung, Dateiname der Icon-Datei, Tastatur-Kurzbefehl, Objekt, Slot-Methode, Vater-Objekt und Name übergeben. Unsere Aktion QUIT wird auf die alte Weise erzeugt, mit Hilfe der Klasse *KStdAction*. Diese Klasse wird weiter unten im Abschnitt *KDE-Standard-Aktionen* genauer beschrieben. Anschließend können die Aktionen mit der Methode *plug* in die Popup-Menüs und die Werkzeugleisten eingefügt werden. Der Code sieht dann folgendermaßen aus:

```
MyMainWindow::MyMainWindow() : KMainWindow()
{
    text = new QLabel (i18n
        ("<H1>Hello, World!</H1>"), this);
    setCentralWidget (text);

    QPopupMenu *filePopup = new QPopupMenu (this);
    KAction *quitAction = KStdAction::quit (this,
        SLOT (fileQuit()), actionCollection());
    KAction *clearAction =
        new KAction (i18n ("%Clear"),    // Bezeichnung
                    "remove",           // Icon
                    Qt::CTRL + Qt::Key_X, // Tastatur
                    text, SLOT (clear()), // Objekt/Slot
                    actionCollection(),   // Vater
                    "file_clear");       // Name

    clearAction->plug (filePopup);
    clearAction->plug (toolBar());
    quitAction->plug (filePopup);
    quitAction->plug (toolBar());

    menuBar()->insertItem (i18n ("%File"), filePopup);
    menuBar()->insertSeparator();
    menuBar()->insertItem (i18n ("%Help"), helpMenu());
}
```

Die Adressen der *KAction*-Objekte werden in lokalen Variablen abgelegt, denn sie werden in diesem Fall nachher nicht mehr benötigt.

Hier folgt noch einmal eine kurze Beschreibung der Parameter für den Konstruktor eines *KAction*-Objekts:

1. *Die Bezeichnung der Aktion* – Dieser Text wird im Popup-Menü benutzt sowie für den Fall, dass bei den Buttons der Werkzeugleiste die Bezeichnungen eingefügt werden. In der Regel schließen Sie den String in die Funktion *i18n* ein, damit er in die eingestellte Landessprache übersetzt wird. Sie können einen Buchstaben unterstrichen darstellen, indem Sie das Kaufmanns-Und »&« voranstellen. Bei geöffnetem Popup-Menü kann dieser Buchstabe auch einfach auf der Tastatur gedrückt werden, um den Befehl auszuführen. Achten Sie darauf, dass die Befehle innerhalb eines Popup-Menüs nicht den gleichen Buchstaben benutzen. (Das gilt auch für die Übersetzung der Bezeichnungen in eine andere Landessprache.) Wollen Sie ein Kaufmanns-Und in der Bezeichnung haben (z.B. DRAG&DROP), so müssen Sie && schreiben (also hier *Drag&&Drop*). Bei Befehlen, die zunächst ein neues Dialogfenster öffnen, hängt man an die Bezeichnung drei Punkte an. Der Anwender erkennt dadurch sofort, dass der Befehl nicht direkt ausgeführt wird, sondern zunächst weitere Einstellungen nötig sind. Beispiele hierfür sind DATEI-ÖFFNEN... (zunächst öffnet sich der Datei-Dialog zur Auswahl der Datei), DATEI-SPEICHERN UNTER... (hier öffnet sich der Datei-Dialog zur Angabe des Dateinamens), DATEI-DRUCKEN... (zunächst müssen noch Druckereinstellungen vorgenommen werden) oder HILFE-ÜBER KDE... (ein neues Fenster mit der Information öffnet sich).
2. *Der Dateiname der Icon-Datei* – Die Datei mit dem passenden Dateinamen wird automatisch aus den KDE-Verzeichnissen gesucht. Sie dürfen die Dateiendung hier nicht angeben. Die Standard-Icons befinden sich in der Regel im Verzeichnis *\$KDEDIR/share/icons*. Dort befinden sich weitere Unterverzeichnisse, die die Art des Icons festlegen (*locolor* enthält Icons mit maximal 40 Farben aus der Standard-KDE-Palette, siehe Anhang C; *hicolor* Icons in TrueColor; weiterhin werden verschiedene Icongrößen unterschieden und ob es sich um Icons für Programme oder für Aktionen handelt). Wollen Sie eigene Icons entwerfen, so legen Sie diese in einer *png*- oder *xpm*-Datei ab und speichern sie im entsprechenden Verzeichnis. Wenn Sie der Aktion kein Icon zuordnen wollen (nur für die Menüleiste sinnvoll), so lassen Sie diesen Parameter einfach weg.
3. *Tastatur-Kurzbefehl* – Diese Angabe legt fest, mit welcher Tastenkombination diese Aktion von (fast) jeder Stelle des Programms aus aufgerufen werden kann. Diese Angabe ist eine der Konstanten, die in der Klasse *Qt* definiert sind. Benutzen Sie am besten nur Kombinationen mit der Taste Strg, da Kombinationen mit Alt bereits für den Aufruf der Menüleiste benutzt werden, und normale Tastendrücke in Eingabefeldern benutzt werden. Achten Sie auch darauf, dass Sie keine Kombinationen nutzen, die auch anderweitig verwendet werden. Benutzen Sie den Wert 0, um keinen Tastaturkurzbefehl zu definieren.

4. *Objekt* – An dieser Stelle geben Sie die Adresse des Objekts an, das informiert werden soll, wenn die Aktion aufgerufen wird. Sie können hier auch einen Null-Zeiger angeben. In diesem Fall wird zunächst kein Objekt mit der Aktion verbunden. Sie können das Signal `KAction::activated()` auch selbst mit einem Slot-Objekt und einem Slot verbinden. So können Sie beispielsweise mehrere Slots von einer Aktion aktivieren lassen.
5. *Slot* – Hier legen Sie den Slot fest, der vom angegebenen Objekt aufgerufen werden soll. Dieser Slot muss parameterlos sein, da auch das Signal von `KAction`, mit dem er verbunden wird, parameterlos ist (`KAction::activated()`). Achten Sie darauf, den Slot-Methodennamen in das Makro `SLOT()` einzufügen, und vergessen Sie auch nicht das leere Klammernpaar hinter dem Methodennamen.
6. *Vater-Objekt* – `KAction` ist von `QObject` abgeleitet, und kann daher auch ein Vater-Objekt besitzen, mit dem es zusammen gelöscht wird. Hier könnten Sie das Hauptfenster angeben (in diesem Fall also *this*). In der Regel sollten Sie aber das Ergebnis der Methode `KMainWindow::actionCollection()` verwenden. Das dort zurückgelieferte Objekt sammelt alle definierten Aktionen als Kind-Objekte. Das ist spätestens dann notwendig, wenn Sie die Aktionen automatisch in die Menü- und Werkzeugleisten eintragen lassen wollen (siehe Kapitel 3.5.4, *Aufbau der Menü- und Werkzeugleisten per XML-Datei*).
7. *Name* – Auch der Name des Objekts kann wie bei jeder von `QObject` abgeleiteten Klasse angegeben werden. Sie können ihn wie so oft auch weglassen, sollten ihn aber setzen, wenn Sie die Aktionen automatisch in die Menü- und Werkzeugleiste eintragen lassen wollen (siehe Kapitel 3.5.4, *Aufbau der Menü- und Werkzeugleiste per XML-Datei*). Wählen Sie einen beliebigen Namen für die Aktion, nur eindeutig sollte er sein. Eingebürgert haben sich Namen im Format *file_save_as* oder *bookmark_add*. Da dieser Text nie angezeigt wird, brauchen Sie die Texte nicht zu übersetzen. Englisch hat sich für diese Namen eingebürgert, ist aber nicht verpflichtend.

Oftmals sind Menübefehle nicht zu jedem Zeitpunkt sinnvoll. Der Befehl `FILE-SAVE` macht zum Beispiel nur Sinn, wenn die Daten auch tatsächlich geändert wurden. `EDIT-PASTE` macht keinen Sinn, wenn die Zwischenablage leer ist. Befehle, die zur Zeit nicht sinnvoll sind, sollten auch nicht aufgerufen werden können. Sie sollten aber im Menü verbleiben, damit der Anwender sieht, dass es diesen Befehl prinzipiell gibt (nur eben im Moment nicht). Dazu wird der Befehl in den Popup-Menüs ausgegraut, und in der Werkzeugleiste wird das Icon zum Befehl grau und kontrastarm dargestellt. Sie erreichen diesen Zustand, indem Sie zum `KAction`-Objekt die Methode `setEnabled(false)` aufrufen. Mit `setEnabled(true)` wird der Befehl wieder benutzbar.

Wir erweitern unser *KHelloWorld*-Beispiel noch einmal: Dieses Mal soll der Befehl CLEAR (LÖSCHEN) nach dem ersten Aufruf nicht mehr benutzbar sein. Eine Wirkung hätte er ohnehin nicht mehr. Daraus ergeben sich zwei Änderungen in unserem Programm: Das *KAction*-Objekt für diesen Befehl muss in einer Attribut-Variablen unserer Hauptfensterklasse abgelegt werden, denn zum Deaktivieren müssen wir darauf zugreifen können. Außerdem brauchen wir jetzt einen zusätzlichen Slot *fileClear*, denn der Aufruf des Befehls muss nun nicht nur das *QLabel*-Objekt löschen, sondern auch den Befehl deaktivieren. Der Code des Programms sieht nun so aus (die geänderten Stellen sind fett gedruckt, die Datei *main.cpp* ist unverändert):

Datei *mymainwindow.h*:

```
#ifndef MYMAINWINDOW_H
#define MYMAINWINDOW_H

#include <kmainwindow.h>

class KAction;
class QLabel;

class MyMainWindow : public KMainWindow
{
    Q_OBJECT
public:
    MyMainWindow();
    ~MyMainWindow();

private slots:
    void fileQuit();
    void fileClear();

private:
    QLabel *text;
    KAction *clearAction;
};

#endif
```

Datei *mymainwindow.cpp*:

```
#include <kaction.h>
#include <kapp.h>
#include <klocale.h>
#include <kmenubar.h>
#include <kmessagebox.h>
#include <kstdaction.h>
#include <qlabel.h>
```

```
#include <qpopupmenu.h>

#include "mymainwindow.h"

MyMainWindow::MyMainWindow() : KMainWindow()
{
    text = new QLabel (i18n
        ("<H1>Hello, World!</H1>"), this);
    setCentralWidget (text);

    QPopupMenu *filePopup = new QPopupMenu (this);
    KAction *quitAction = KStdAction::quit (this,
        SLOT (fileQuit()), actionCollection());
    clearAction =
        new KAction (i18n ("%Clear"),
            "remove",
            Qt::CTRL + Qt::Key_X,
            this, SLOT (fileClear()),
            actionCollection(),
            "file_clear");

    clearAction->plug (filePopup);
    clearAction->plug (toolBar());
    quitAction->plug (filePopup);
    quitAction->plug (toolBar());

    menuBar()->insertItem (i18n ("%File"), filePopup);
    menuBar()->insertSeparator();
    menuBar()->insertItem (i18n ("%Help"), helpMenu());
}

MyMainWindow::~MyMainWindow()
{
}

void MyMainWindow::fileQuit()
{
    int really = KMessageBox::questionYesNo (this,
        i18n ("Do you really want to quit?"));
    if (really == KMessageBox::Yes)
        kapp->quit();
}

void MyMainWindow::fileClear()
{
    text->clear();
    clearAction->setEnabled (false);
}
```

Beachten Sie, dass wir das *clearAction*-Objekt nun nicht mehr in einer lokalen Variable speichern, sondern in der Attributvariable. Vergessen Sie also nicht, die Zeile im Konstruktor zu ändern, die die lokale Variable angelegt hat!

Abbildung 3.23 zeigt unser Programm, in dem die Aktion *CLEAR* deaktiviert wurde. Das Icon in der Werkzeugleiste kann nun nicht mehr angeklickt werden, und auch der Befehl im Menü ist abgeschaltet.



Abbildung 3-23 *CLEAR* ist deaktiviert worden.

Von *KAction* abgeleitete Klassen

Neben diesen einfachen Befehlsaktionen gibt es noch einige von *KAction* abgeleitete Klassen, die komplexere Aktionen ausführen können. Tabelle 3.4 zeigt eine Liste der wichtigsten Klassen und ihrer Fähigkeiten.

Klasse	abgeleitet von	Fähigkeit
<i>KToggleAction</i>	<i>KAction</i>	ein-/ausschaltbare Option
<i>KRadioAction</i>	<i>KToggleAction</i>	auswählbare Option aus mehreren Optionen
<i>KActionMenu</i>	<i>KAction</i>	Untermenü mit weiteren Aktionen
<i>KActionSeparator</i>	<i>KAction</i>	Trennlinie bzw. Zwischenraum (z.B. in <i>KActionMenu</i>)
<i>KSelectAction</i>	<i>KAction</i>	Liste von Einträgen, von denen einer aktiviert sein kann
<i>KListAction</i>	<i>KSelectAction</i>	Liste von Befehlen, die aufgerufen werden können
<i>KRecentFilesAction</i>	<i>KListAction</i>	Liste der zuletzt geöffneten Dateien
<i>KFontAction</i>	<i>KSelectAction</i>	Liste der verfügbaren Schriftarten
<i>KFontSizeAction</i>	<i>KSelectAction</i>	Liste der sinnvollen Schriftgrößen

Tabelle 3-4 Die wichtigsten von *KAction* abgeleiteten Klassen

Die Klasse *KToggleAction* stellt eine Option zur Verfügung, die an- und ausgeschaltet werden kann. In der Menüleiste erscheint vor eingeschalteten Optionen ein Häkchen. In der Werkzeugleiste wird eine eingeschaltete Option als einge-

drückter Button angezeigt. Die Klasse bietet ein Signal *toggled (bool)*, das nach jeder Änderung den neuen Zustand liefert. Mit der Methode *KToggleAction::isChecked* können Sie jederzeit den aktuellen Zustand erfragen. Viele Programme bieten dem Anwender beispielsweise die Möglichkeit, die Werkzeugleiste und/oder die Statuszeile auszublenden, um mehr Platz für den Anzeigebereich zu bekommen. Dazu gibt es in der Menüleiste unter *SETTINGS* die Befehle *SHOW TOOLBAR* und *SHOW STATUSBAR*. Jedes Wählen eines der Befehle blendet die Werkzeugleiste bzw. Statuszeile ein oder aus. (Beachten Sie, dass *KStdAction* bereits Methoden besitzt, um diese Aktionsobjekte für die Befehle *Show Toolbar* und *Show Statusbar* zu erzeugen.) Der entsprechende Code dazu kann beispielsweise so aussehen:

```
MyMainWindow::MyMainWindow () : KMainWindow()
{
    ...
    // Aktion erzeugen
    KToggleAction *toolbarAction =
        new KToggleAction ("Show &Toolbar", // Bezeichnung
                           0,              // Tastatur
                           actionCollection(),
                           "settings_show_toolbar");

    // Anfangszustand korrekt setzen
    toolbarAction->setChecked (true);

    // Mit eigenem Slot verbinden
    connect (toolbarAction, SIGNAL (toggled (bool)),
            this, SLOT (toggleToolBar (bool)));

    // in das Menü mit aufnehmen
    QPopupMenu *settingsMenu = new QPopupMenu ();
    toolbarAction->plug (settingsMenu);
    menuBar()->insertItem ("&Settings", settingsMenu);
    ...
}

void MyMainWindow::toggleToolBar (bool doShow)
{
    if (doShow)
        toolbar()->show();
    else
        toolbar()->hide();
}
```

Da wir das Signal von Hand verbinden, brauchen wir im Konstruktor kein Objekt und keinen Slot angeben. Ebenso lassen wir die Angabe für das Icon weg. Dadurch ist das Aktionsobjekt nur in der Menüleiste einsetzbar, aber auch nur dort ist es ja sinnvoll.

Eine andere typische Anwendung in Textverarbeitungsprogrammen ist die Auswahlmöglichkeit, ob nichtdruckende Steuerzeichen wie Leerzeichen, Tabulatoren oder Absatzmarken angezeigt werden sollen oder nicht.

Beachten Sie auch, dass Sie ein *KToggleAction*-Objekt gleichzeitig in der Werkzeugleiste und in der Menüleiste aufführen können. Wenn Sie es an einer Stelle verändern, wird die Änderung auch an der anderen Stelle sofort dargestellt.

Wollen Sie mehrere *KToggleAction*-Objekte so miteinander verbinden, dass immer nur höchstens eine der Optionen ausgewählt ist, so setzen Sie bei allen Objekten mit der Methode *setExclusiveGroup* den gleichen String ein. Wird eine Option eingeschaltet, werden automatisch alle anderen Optionen mit dem gleichen String ausgeschaltet. Weitaus häufiger wird diese Fähigkeit aber mit *KRadioAction*-Objekten benutzt.

Die Klasse *KRadioAction* ist von *KToggleAction* abgeleitet und unterscheidet sich von dieser Klasse nur dadurch, dass Sie eine eingeschaltete Option durch Auswählen oder Anklicken nicht wieder ausschalten können. Die einzige Möglichkeit dazu besteht darin, eine andere Option einzuschalten, die mit *setExclusiveGroup* mit dieser Option verbunden ist und diese deswegen ausschaltet. Ein typisches Beispiel in einem Textverarbeitungsprogramm ist die Auswahl der Textausrichtung: linksbündig, rechtsbündig, zentriert oder im Blocksatz. Das Anlegen solcher Aktionen kann im Listing beispielsweise so aussehen:

```
KRadioAction *leftJustifyAction =
    new KRadioAction ("Justify &Left",
                      "leftjust", 0,
                      this, SLOT (justifyLeft()),
                      actionCollection(),
                      "justify_left");
leftJustifyAction->setExclusiveGroup ("justify");
KRadioAction *rightJustifyAction =
    new KRadioAction ("Justify &Right",
                      "left_right", 0,
                      this, SLOT (justifyRight()),
                      actionCollection(),
                      "justify_right");
rightJustifyAction->setExclusiveGroup ("justify");
... // Analog für zentriert

// in die Werkzeugleiste einfügen
leftJustifyAction->plug (toolBar());
rightJustifyAction->plug (toolBar());
centerJustifyAction->plug (toolBar());

// linksbündig als Standard setzen
leftJustifyAction->setChecked(true);
```

Achten Sie darauf, dass bereits beim Start eine der Optionen ausgewählt ist (hier *leftJustifyAction*). Benutzen Sie dazu die Methode *KToggleAction::setChecked(true)*.

Abbildung 3.24 zeigt die Werkzeugleiste mit den Aktionen, wobei zur Zeit die Einstellung »linksbündig« aktiviert ist.



Abbildung 3-24 Drei Aktionen vom Typ *KRadioAction*

Die Klasse *KActionMenu* erzeugt ein Untermenü, das bei der Auswahl der Aktion aufklappt. Auf diese Weise können komplexe Menüstrukturen besser organisiert werden. Fügen Sie nacheinander die Aktionen für das Untermenü mit der Methode *KActionMenu::insert* ein. Dabei sind alle Aktionen möglich, also einfache *KAction*-Objekte genauso wie alle anderen hier beschriebenen Klassen. Auch weitere *KActionMenu*-Objekte sind möglich, so dass die hierarchische Struktur noch tiefer wird. Diese Aktion wird in der Regel in der Menüzelle verwendet, kann aber auch in die Werkzeugleiste eingefügt werden.

Mit der Klasse *KActionSeparator* können Sie einen Trennstrich bzw. einen Zwischenraum zwischen Aktionen in einem Menü (z.B. *KActionMenu*) einfügen, um so die einzelnen Aktionen in Gruppen voneinander abzugrenzen. Sie führt keine Aktionen aus und kann auch nicht vom Anwender aktiviert werden.

Die Klasse *KSelectAction* enthält eine Liste von Optionen (Strings), von denen der Anwender eine auswählt. Diese Auswahl wird über die Signale *activated(int)* und *activated(const QString &)* gemeldet. Das erste Signal liefert den Index des Eintrags zurück, das zweite den Text. In der Menüzelle erscheint diese Aktion als Untermenü. Der ausgewählte Eintrag wird mit einem Häkchen markiert. In der Werkzeugleiste erscheint die Aktion als Auswahlbox. Der Anwender kann auf die Box klicken, woraufhin ein Untermenü erscheint, in dem die Einträge angezeigt werden. Die oben beschriebene Auswahl aus verschiedenen Varianten der Textausrichtung kann auch mit einem *KSelectAction*-Objekt erzeugt werden (was aber unüblich ist). Der Code dazu sieht folgendermaßen aus:

```
KSelectAction *justifyAction =
    new KSelectAction("&Justify",
                    "justify",
                    0,
                    actionCollection(),
                    "justify");

QStringList l;
l << "Left" << "Right" << "Center" << "Block";
justifyAction->setItems(l);
```

(Nähere Informationen zur Klasse *QStringList* finden Sie in Kapitel 4.7.2, *Container-Klassen*.) Abbildung 3.25 zeigt diese Variante der Auswahl in der Werkzeugleiste. Andere Einsatzgebiete von *KSelectAction* sind beispielsweise die Einstellung des Zoom-Faktors (sofern Sie nur feste Werte für den Zoom-Faktor vorgeben wollen) oder die Auswahl einer Formatvorlage.



Abbildung 3-25 *KSelectAction* in der Werkzeugleiste

Mit der Methode *setEditable (true)* hat der Anwender in der Werkzeugleiste auch die Möglichkeit, den Eintrag selbst über die Tastatur einzugeben. In der Menüleiste gibt es diese Möglichkeit nicht. Ein Einsatzgebiet ist hier beispielsweise auch die Einstellung des Zoom-Faktors. Der Anwender kann einen Zoom-Faktor aus der Liste wählen, oder er kann einen eigenen Zahlenwert eingeben.

Die Texte, die Sie einfügen, können das Kaufmanns-Und »&« enthalten. In der Menüleiste wird im Untermenü dann das &-Zeichen nicht angezeigt, sondern der folgende Buchstabe unterstrichen dargestellt, wie in Popup-Menüs üblich. In der Werkzeugleiste werden dagegen die &-Zeichen mit ausgegeben. Achten Sie also darauf, dass Sie das *KSelectAction*-Objekt entweder in der Menüleiste oder der Werkzeugleiste verwenden, möglichst aber nicht in beiden.

Die Klasse *KListAction* ist von *KSelectAction* abgeleitet und ist dieser sehr ähnlich. Der einzige Unterschied besteht darin, dass in *KListAction* der ausgewählte Eintrag bei der Verwendung in der Menüleiste nicht mit einem Häkchen versehen wird. *KListAction* setzen Sie daher ein, wenn Sie eine Reihe von Befehlen in einem Untermenü einfügen wollen, *KSelectAction* dagegen, wenn Sie einen der Einträge auswählen wollen. *KListAction* dient damit als Ersatz für ein *KActionMenu*-Objekt. Gegenüber diesem hat es den Vorteil, dass es einfacher mit Befehlen zu füllen ist und nur ein einziger Slot zur Bearbeitung aller Befehle benötigt wird. (Das Signal *activated (const QString &)* liefert ja den gewählten Befehl zurück.) *KListAction* wird daher vor allem dort eingesetzt, wo alle Befehle sehr ähnlich sind. Ihr Nachteil ist, dass keine komplexeren *KAction*-Objekte eingesetzt werden können. Obwohl *KListAction*-Objekte auch in der Werkzeugleiste eingesetzt werden können, ist das unüblich.

Neben diesen allgemein einsetzbaren Unterklassen von *KAction* stellt KDE auch noch drei sehr spezielle Unterklassen zur Verfügung:

KFontAction dient zur Auswahl einer Schriftart. Es handelt sich hierbei um eine Unterklasse von *KSelectAction*, die im Konstruktor automatisch alle installierten Schriftarten in die Liste der Einträge schreibt und *setEditable (true)* aufruft, so dass der Anwender hier auch einen eigenen Font-Namen eintragen kann. Sie wird

üblicherweise in der Werkzeugleiste benutzt. In der Menüleiste ist sie eher unpraktisch, da durch die unter Umständen sehr langen Untermenüs das Auswählen schwer ist.

KFontSizeAction ist ebenfalls eine Unterklasse von *KSelectAction* und dient zur Auswahl der Schriftgröße. Hierbei sind die üblichen Standardwerte bereits per Konstruktor eingetragen. *setEditable (true)* wird ebenfalls automatisch aufgerufen, so dass der Anwender eigene Werte eintragen kann, wenn keiner der vorgegebenen Werte passt. Die Werte sind dabei eingeschränkt auf ganze Zahlen zwischen 0 und 128. Bei jeder Änderung wird das Signal *fontSizeChanged (int)* aufgerufen. Auch *KFontSizeAction* wird meist nur in der Werkzeugleiste verwendet.

Die Anwendung dieser beiden Klassen in eigenen Programmen ist sehr einfach. Abbildung 3.26 zeigt die Klassen in Aktion. Der Code sieht beispielsweise so aus:

```
KFontAction *fontAction = new KFontAction
    ("&Font", 0, actionCollection (), "choose_font");
connect (fontAction, SIGNAL (activated (const QString &)),
        this, SLOT (changeFont (const QString &)));
fontAction->plug (toolBar());
KFontSizeAction *sizeAction = new KFontSizeAction
    ("Font &Size", 0, actionCollection (), "choose_size");
connect (sizeAction, SIGNAL (fontSizeChanged (int)),
        this, SLOT (changeSize (int)));
sizeAction->plug (toolBar());
```

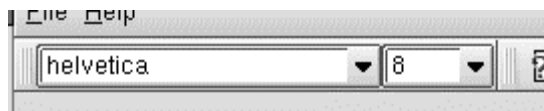


Abbildung 3-26 *KFontAction* und *KFontSizeAction*

Die Klasse *KRecentFilesAction* ist von *KListAction* abgeleitet. Sie stellt die Liste der letzten zehn geöffneten Dokumente in einem Popup-Menü dar und bietet die Möglichkeit, einen dieser Einträge auszuwählen, in der Regel, um das zugehörige Dokument wieder zu öffnen. Dieses wird in der Methode *urlSelected (const KURL&)* gemeldet. (Mit diesem Signal wird auch der Slot verbunden, den Sie im Konstruktor der Klasse angeben können.) Meist wird diese Klasse als Aktion zum Menüpunkt FILE-RECENTLY OPENED FILES... (auf deutsch DATEI-ZULETZT GEÖFFNETE DATEIEN) verwendet. Sie wird auch als Zeile für die Eingabe einer URL in der Werkzeugleiste eines Browsers verwendet. Die Klasse bietet auch die Möglichkeit, die aktuelle Liste der Einträge in einer KDE-Konfigurationsdatei abzuspeichern, um sie beim nächsten Start des Programms wieder einzulesen. Eine Anwendung dieser Klasse finden Sie in Kapitel 3.5.6, *Applikation für ein Dokument*. Weitere Informationen zu Konfigurationsdateien finden Sie in Kapitel 4.10, *Konfigurationsdateien*.

KDE-Standardaktionen

Die Menüstruktur fast aller KDE-Applikationen ist ähnlich aufgebaut, und auch die Befehle sind meist sehr ähnlich. Das wird auch ganz bewusst so gemacht, damit die Einarbeitungszeit in neue Programme möglichst gering ist. Auch wenn ein Anwender ein neues Programm zum ersten Mal startet, so kann er doch sicher sein, dass er unter DATEI-BEENDEN den Befehl zum Schließen findet und dass die Befehle für die Zwischenablage in der Kategorie BEARBEITEN zu finden sind. Auch die bekannten Icons in der Werkzeugleiste sind immer gleich: Eine Diskette symbolisiert das Speichern einer Datei und eine Schere das Ausschneiden von markierten Objekten. Damit diese in vielen Programmen vorkommenden Aktionen nicht in jedem Programm neu erzeugt werden müssen, stellt KDE mit der Klasse *KStdAction* eine einfache Möglichkeit zur Verfügung, diese Aktionen mit minimalem Programmieraufwand zu nutzen. Wir haben diese Klasse bereits in unserem Minimalprogramm in Kapitel 2.3, *Das erste KDE-Programm*, benutzt, um die Aktion für den Eintrag FILE-QUIT (DATEI-BEENDEN) zu generieren. Hier folgt eine Liste der Vorteile, die sich ergeben, wenn man so weit wie möglich die Standardaktionen nutzt:

- Das Listing wird einfacher, da weder die Bezeichnung noch der Icon-Dateiname noch der Tastatur-Kurzbefehl angegeben werden muss. Diese Angaben sind bereits vorgegeben.
- Zu allen Standardaktionen enthält das KDE-System bereits fertige Icons für die Popup-Menüs und die Werkzeugleiste, so dass hier keine eigenen Icons entworfen werden müssen.
- Ein Vertippen beim Eingeben der Bezeichnung ist nicht mehr möglich. Die Aktionen haben damit in allen Programmen die gleiche Bezeichnung. Der Anwender findet sich so noch schneller zurecht.
- Das KDE-System enthält bereits die Übersetzungen der Bezeichnungen in eine Vielzahl von Sprachen. Diese Texte müssen also nicht mehr übersetzt werden.
- Verwendet man das XML-System zur automatischen Generierung von Menü- und Werkzeugleisten (siehe Kapitel 3.5.4, *Aufbau der Menü- und Werkzeugleiste per XML-Datei*), so werden die Standardaktionen automatisch platziert. Es brauchen also nur noch die ganz spezifischen Aktionen angegeben werden.

Die Klasse *KStdAction* besitzt eine große Anzahl an statischen Methoden – je eine für eine Standardaktion –, die ein *KAction*-Objekt erzeugen und zurückliefern. Als Parameter müssen dabei nur das Objekt und der Slot angegeben werden, die beim Aktivieren der Aktion benachrichtigt werden sollen, sowie das Vater-Objekt. In unserem Einführungsprogramm sah der entsprechende Code für die Erzeugung der QUIT-Aktion so aus:

```
KAction *quitAction = KStdAction::quit  
    (this, SLOT (fileQuit()), actionCollection());
```

Alle anderen Angaben – die Bezeichnung, das Icon, der Tastatur-Kurzbefehl und der Name des Objekts – werden automatisch gesetzt.

Tabelle 3.5 enthält eine Liste aller zur Zeit definierten Standardaktionen, die *KStdAction* bereitstellt. Angegeben sind jeweils der Methodenname der statischen Methode von *KStdAction*, der englischsprachige Menübefehl und der deutschsprachige Menübefehl. Die mit *) gekennzeichneten Aktionen weisen Besonderheiten auf, die weiter unten erläutert werden. Einige der Standardaktionen werden in Kapitel 3.5.6, *Applikation für ein Dokument*, und Kapitel 3.5.7, *Applikation für mehrere Dokumente*, angewendet und besprochen.

Methodenname	engl. Menüname	dt. Menüname
<i>openNew</i>	FILE- <u>N</u> EW	DATEI- <u>N</u> EU
<i>open</i>	FILE- <u>O</u> PEN...	DATEI- <u>Ö</u> FFNEN...
<i>openRecent</i> *)	FILE-OPEN <u>R</u> ECENT	DATEI-ZU <u>L</u> ETZT GEÖFFNETE DATEIEN
<i>save</i>	FILE- <u>S</u> AVE	DATEI- <u>S</u> PEICHERN
<i>saveAs</i>	FILE-SAVE <u>A</u> S...	DATEI-SPEICHERN <u>U</u> NTER...
<i>revert</i> *)	FILE- <u>R</u> EVERT	DATEI-ZU <u>L</u> ETZT GESPEICHERTE FASSUNG
<i>print</i>	FILE- <u>P</u> RI ^N T	DATEI- <u>D</u> RUCKEN
<i>printPreview</i>	FILE-PRINT <u>P</u> REVIEW	DATEI-DRUCK <u>V</u> ORSCHAU
<i>close</i> *)	FILE- <u>C</u> LOSE	DATEI-SCH <u>L</u> IESSEN
<i>mail</i>	FILE- <u>M</u> AIL...	DATEI- <u>V</u> ERSENDEN
<i>quit</i>	FILE- <u>Q</u> UIT	DATEI- <u>B</u> EENDEN
<i>undo</i>	EDIT- <u>U</u> ND <u>O</u>	BEARBEITEN- <u>R</u> ÜCKGÄNGIG
<i>redo</i>	EDIT- <u>R</u> E <u>D</u> O	BEARBEITEN-WIEDER <u>H</u> ERSTELLEN
<i>cut</i>	EDIT- <u>C</u> UT	BEARBEITEN-AUSSCHNEIDEN
<i>copy</i>	EDIT- <u>C</u> OPY	BEARBEITEN- <u>K</u> OPIEREN
<i>paste</i>	EDIT- <u>P</u> ASTE	BEARBEITEN-EIN <u>E</u> ÜGEN
<i>selectAll</i>	EDIT-SELECT <u>A</u> LL	BEARBEITEN- <u>A</u> LLS AUSWÄHLEN
<i>find</i>	EDIT- <u>F</u> IND...	BEARBEITEN- <u>S</u> UCHEN...
<i>findNext</i>	EDIT-FIND <u>N</u> EXT	BEARBEITEN- <u>W</u> EITERSUCHEN
<i>findPrev</i>	EDIT-FIND <u>P</u> REVIOUS	BEARBEITEN- <u>E</u> RÜHERE SUCHEN
<i>replace</i>	EDIT- <u>R</u> EPLACE...	BEARBEITEN- <u>E</u> RSETZEN
<i>actualSize</i>	VIEW- <u>A</u> CTUAL SIZE	ANSICHT- <u>T</u> ATSÄCHLICHE GRÖSSE
<i>fitToPage</i>	VIEW- <u>F</u> IT TO PAGE	ANSICHT-AUF SEITE EIN <u>P</u> ASSEN
<i>fitToWidth</i>	VIEW-FIT TO PAGE <u>W</u> IDTH	ANSICHT-AUF SEITEN <u>B</u> REITE EINPASSEN
<i>fitToHeight</i>	VIEW-FIT TO PAGE <u>H</u> HEIGHT	ANSICHT-AUF SEITEN <u>H</u> ÖHE EINPASSEN

Tabelle 3-5 Standardaktionen der Klasse *KStdAction*

Methode	engl. Menüname	dt. Menüname
<i>zoomIn</i>	VIEW-ZOOM <u>I</u> N	ANSICHT-VERGRÖßERN
<i>zoomOut</i>	VIEW-ZOOM <u>O</u> UT	ANSICHT-VERKLEINERN
<i>zoom</i>	VIEW-Z <u>U</u> OM...	ANSICHT-Z <u>U</u> OM...
<i>redisplay</i>	VIEW-R <u>E</u> DISPLAY	ANSICHT-ANZEIGE <u>A</u> UFFRISCHEN
<i>up</i>	GO- <u>U</u> P	GEHE ZU-NACH <u>O</u> PEN
<i>back</i>	GO-B <u>A</u> CK	GEHE ZU-ZURÜCK
<i>forward</i>	GO-F <u>O</u> RWARD	GEHE ZU-NACH <u>V</u> ORNE
<i>home</i>	GO-H <u>O</u> ME	GEHE ZU-D <u>A</u> TEIANFANG
<i>prior</i>	GO-P <u>R</u> VIOUS PAGE	GEHE ZU-V <u>O</u> RIGE SEITE
<i>next</i>	GO-N <u>E</u> XT PAGE	GEHE ZU-N <u>A</u> CHSTE SEITE
<i>goTo *)</i>	GO-G <u>O</u> TO...	GEHE ZU-G <u>E</u> HE ZU...
<i>gotoPage *)</i>	GO-G <u>O</u> TO PAGE...	GEHE ZU-GEHE ZU <u>S</u> EITE...
<i>gotoLine *)</i>	GO-G <u>O</u> TO LINE...	GEHE ZU-GEHE ZU <u>Z</u> EILE...
<i>firstPage</i>	GO-F <u>I</u> RST PAGE	GEHE ZU-E <u>R</u> STE SEITE
<i>lastPage</i>	GO-L <u>A</u> ST PAGE	GEHE ZU-L <u>E</u> TZTE SEITE
<i>addBookmark</i>	BOOKMARKS-A <u>D</u> D BOOKMARK	LESEZEICHEN-LESEZEICHEN HINZUFÜGEN
<i>editBookmarks</i>	BOOKMARKS-E <u>D</u> IT BOOKMARKS...	LESEZEICHEN-LESEZEICHEN B <u>E</u> ARBEITEN
<i>spelling</i>	TOOLS-S <u>P</u> ELLING...	WERKZEUGE-R <u>E</u> CHTSCHREIBUNG
<i>showMenubar *)</i>	SETTINGS-SHOW M <u>E</u> NUBAR	EINSTELLUNGEN-M <u>E</u> NÜLEISTE ANZEIGEN
<i>showToolBar *)</i>	SETTINGS-SHOW T <u>O</u> OLBAR	EINSTELLUNGEN-WERKZ <u>E</u> UGLEISTE ANZEIGEN
<i>showStatusbar *)</i>	SETTINGS-SHOW S <u>T</u> ATUSBAR	EINSTELLUNGEN-S <u>T</u> ATUSZEILE ANZEIGEN
<i>saveOptions *)</i>	SETTINGS-S <u>A</u> VE OPTIONS	EINSTELLUNGEN-O <u>P</u> TIONEN SPEICHERN
<i>keyBindings</i>	SETTINGS-CONFIGURE <u>K</u> EY BIN-DINGS...	EINSTELLUNGEN-I <u>A</u> STENZUORDNUNGEN VORNEHMEN...
<i>preferences</i>	SETTINGS-P <u>R</u> EFERENCES...	EINSTELLUNGEN-P <u>E</u> RSÖNLICHE EINSTELLUNGEN...
<i>configureToolbars</i>	SETTINGS-CONFIGURE T <u>O</u> OLB <u>A</u> RS...	EINSTELLUNGEN-WERKZ <u>E</u> UGLEISTE EINRICHTEN...
<i>help *)</i>	HELP-H <u>E</u> LPH	HILFE-H <u>I</u> LFE
<i>helpContents</i>	HELP-C <u>O</u> NENTS	HILFE-I <u>N</u> HALT
<i>whatsThis</i>	HELP-WHAT'S T <u>H</u> IS?	HILFE-WAS IST <u>D</u> AS?
<i>reportBug</i>	HELP-R <u>E</u> PORT BUG	HILFE-B <u>E</u> RICHTEN SIE PROBLEME ODER WÜNSCHE...
<i>aboutApp</i>	HELP-A <u>B</u> OUT <i>appname</i> ...	HILFE-Ü <u>B</u> ER <i>appname</i> ...
<i>aboutKDE</i>	HELP-ABOUT <u>K</u> DE...	HILFE-Ü <u>B</u> ER <u>K</u> DE...

Tabelle 3-5 Standardaktionen der Klasse KStdAction (Forts.)

Der Rückgabewert der Methode *openRecent* ist ein *KRecentFileAction*-Objekt, das bereits im vorangegangenen Abschnitt, *Von KAction abgeleitete Klassen*, kurz beschrieben wurde. Dieses Aktionsobjekt kann eine Liste von bisher benutzten Dateinamen speichern. Eine Anwendung dieser Klasse finden Sie in Kapitel 3.5.6, *Applikation für ein Dokument*.

Die Aktion, die *revert* zurückliefert, wird eingesetzt, wenn die aktuellen Änderungen an einem Dokument verworfen werden sollen und stattdessen der Zustand der Datei wiederhergestellt werden soll. Diese Aktion ist aber nicht sehr üblich, da sie nur selten benötigt wird. Der Anwender kann alternativ auch das Dokument schließen (ohne es zu speichern) und es erneut öffnen.

Die von *close* erzeugte Aktion ist nur dann nötig, wenn ein Programm mehrere Dokumente gleichzeitig verwalten kann. Kann es nur ein Dokument auf einmal verwalten, ist die Funktionalität mit *quit* weit gehend identisch, *close* ist also unnötig.

Welche Aktionen üblicherweise im GO-Menü benutzt werden, hängt von der Art der Applikation ab. Zeigt es Dokumente an, die seitenweise aufgebaut sind (z.B. PostScript- oder PDF-Dateien), so sind die Aktionen *prior*, *next*, *gotoPage*, *firstPage* und *lastPage* sinnvoll. Ist es dagegen eher ein Browser, der Querverweise verfolgt, so benutzen Sie die Aktionen *up*, *back*, *forward*, *home* und *goTo*. Für ein Programm, das Textdateien darstellt, verwenden Sie die Aktion *gotoLine*.

Die Rückgabewerte von *showMenubar*, *showToolbar* und *showStatusbar* sind Zeiger auf *KToggleAction*-Objekte. Diese Klasse wurde bereits im vorherigen Abschnitt, *Von KAction abgeleitete Klassen*, beschrieben. Dort ist auch die Anwendung dieser Aktion gezeigt. Die Aktionen werden in der Regel nur in die Menüleiste aufgenommen, nicht in die Werkzeugleiste. Beachten Sie natürlich auch, dass die Aktion *showMenubar* nur dann benutzt werden sollte, wenn es für den Anwender auch eine Möglichkeit gibt, die Menüleiste wieder einzublenden. Möglich ist das beispielsweise durch einen Eintrag dieser Aktion in das Kontextmenü auf der rechten Maustaste. Besonders anwenderfreundlich ist es, wenn beim Ausblenden der Menüleiste der Anwender in einem kleinen Dialogfenster darauf hingewiesen wird, wie er sie wieder einblenden kann. Um den Anwender im Weiteren aber nicht mehr mit diesem Dialogfenster zu langweilen, kann man eine zusätzliche Option mit dem Text »Diesen Hinweis in Zukunft nicht mehr anzeigen« in das Dialogfenster mit aufnehmen und den Zustand in den Konfigurationsdateien abspeichern. Für die meisten Applikationen ist es aber ohnehin nicht sinnvoll, das Ausblenden der Menüleiste überhaupt anzubieten.

Falls möglich sollten alle Einstellungen, die verändert werden, automatisch in den Konfigurationsdateien gesichert werden. Die Aktion *saveOptions* ist daher meist unnötig. Sie kommt daher nur zum Einsatz, wenn ein automatisches Speichern der Optionen nicht möglich oder nicht sinnvoll ist.

Die Aktion, die *help* zurückliefert, kann genutzt werden, wenn man nur ein minimales Hilfesystem in das Programm integrieren möchte. Für »normale« Applikationen ist es dagegen besser, das vollständige Hilfesystem zu nutzen, das alle fünf folgenden Aktionen (*helpContents*, *whatsThis*, *reportBug*, *aboutApp* und *aboutKDE*) benutzt. Dieses Hilfesystem wird ohnehin automatisch in die Menüs integriert, wenn Sie die Methode *KMainWindow::helpMenu* benutzen oder das Menü per XML-Datei erstellen lassen (siehe Kapitel 3.5.4, *Aufbau der Menü- und Werkzeugleiste per XML-Datei*).

Weitere Möglichkeiten von Menü- und Werkzeugleisten

Bisher haben wir die Menü- und die Werkzeugleiste nur am Rande benutzt, um Aktionen einzufügen. Sie besitzen aber noch mehr Möglichkeiten, die wir hier ein wenig näher beleuchten wollen. Für die meisten Programme sind diese aber nicht nötig.

Die Menüleiste ist in der Klasse *KMenuBar* implementiert. Diese übernimmt die Darstellung auf dem Bildschirm und die Reaktion auf Mausklicks oder die speziellen Tastenkombinationen. Jedes Hauptfenster besitzt genau eine Menüleiste. Die Adresse des *KMenuBar*-Objekts kann erfragt werden mit der Methode *KMainWindow::menuBar()*. Beim ersten Aufruf dieser Methode wird das Objekt erzeugt, bei allen weiteren Aufrufen wird nur die Adresse des Objekts zurückgeliefert.

Im Gegensatz zur Menüleiste kann ein Hauptfenster aber beliebig viele Werkzeugleisten enthalten. So kann man die Befehle gruppieren und auf verschiedene Werkzeugleisten verteilen. Auch die Verwaltung der Werkzeugleisten (Objekte der Klasse *KToolBar*) übernimmt die Klasse *KMainWindow* für uns. Benötigt man nur eine Werkzeugleiste, so kann man einfach den Aufruf *toolBar()* verwenden. Bei mehreren Werkzeugleisten vergibt man Bezeichnungen für die Leisten, die man als Parameter an *toolBar* übergibt. Existiert bereits eine Werkzeugleiste mit dieser Bezeichnung, so wird die Adresse zurückgegeben, sonst wird eine Leiste erzeugt. Der Code zum Anlegen von zwei Werkzeugleisten und zum Einfügen von Aktionen kann beispielsweise so aussehen:

```
KAction *aktion1 = ...;
KAction *aktion2 = ...;
KAction *aktion3 = ...;
KAction *aktion4 = ...;

// aktion1 und aktion2 in eine Werkzeugleiste
aktion1->plug (toolBar ("firstBar"));
aktion2->plug (toolBar ("firstBar"));
// aktion3 und aktion4 in die andere Werkzeugleiste
aktion3->plug (toolBar ("secondBar"));
aktion4->plug (toolBar ("secondBar"));
```

Bisher hatten wir die Befehle über Aktionsobjekte in die Menü- und Werkzeugleiste eingefügt. Sie können aber auch Befehle direkt einfügen. Dazu bietet Ihnen die Klasse *KMenuBar* die Methode *insertItem*, die sie von der Vaterklasse *QMenuBar* geerbt hat, die diese wiederum von der Vaterklasse *QMenuData* hat. In der Regel legen Sie selbst Objekte der Klasse *QPopupMenu* an, die Sie ebenso mittels *insertItem* mit Befehlen füllen, und fügen dann diese Popup-Menüs in die Menüleiste ein.

KToolBar besitzt zum Einfügen die Methoden *insertButton* (einfacher Icon-Knopf), *insertLined* (Eingabezeile, z. B. für Schriftgröße), *insertCombo* (Auswahlbox, auch editierbar) und *insertWidget* (für ein beliebiges *QWidget*-Objekt). Mit der Methode *insertAnimatedWidget* kann man das bei Browsern übliche Icon (ganz rechts in der Werkzeugleiste) einfügen, das animiert ist, solange ein Dokument geladen wird. Dazu müssen Sie nur eine Liste der Dateinamen der Icons angeben. Es wird automatisch ein Objekt der Klasse *KAnimWidget* erzeugt. Weitere Informationen finden Sie in der Klassenreferenz zur Klasse *KToolBar*.

Alle von Hand in eine Menü- oder Werkzeugleiste oder in ein Popup-Menü eingefügten Befehle können mit einer Identifikationsnummer versehen werden. Diese gibt man als Parameter bei den *insert*-Methoden an. Lässt man diesen Parameter weg (Defaultwert ist -1), so wird automatisch eine eindeutige Identifikationsnummer erzeugt. Diese Nummer ist in jedem Fall der Rückgabewert der Methode. Über diese Nummer kann man später auf einzelne Elemente zurückgreifen, zum Beispiel um Einträge zu ändern oder zu löschen. Ein anderes Einsatzgebiet für die Identifikationsnummer ist die Nutzung der Signale der Klasse *QPopupMenu*, *KMenuBar* und *KToolBar*. *QPopupMenu* und *KMenuBar* besitzen das Signal *activated* (*int*), *KToolBar* besitzt das Signal *clicked* (*int*). Der übergebene *int*-Parameter gibt die Identifikationsnummer des Eintrags an, der aktiviert wurde. Nutzt man dieses Signal, so kann man alle Befehle eines Menüs oder einer Werkzeugleiste mit einem einzigen Slot verarbeiten. Das macht allerdings in der Regel nur Sinn, wenn alle Befehle auch tatsächlich mit einer einzigen Routine abgearbeitet werden können. Ein großes *case*-Statement ist hier nicht sinnvoll, da es das Programm unübersichtlicher macht. Die gleichen Möglichkeiten bieten Ihnen übrigens auch die Aktionsklassen *KSelectAction* und *KListAction*, die bereits im Abschnitt *Von KAction abgeleitete Klassen* besprochen wurden.

3.5.4 Aufbau der Menü- und Werkzeugleiste per XML-Datei

Bisher haben wir die Menü- und Werkzeugleisten in zwei Schritten aufgebaut: Im ersten Schritt haben wir *KAction*-Objekte erzeugt, und in einem zweiten Schritt haben wir diese Aktionen mit der Methode *plug* in *QPopupMenu*-Objekte und in die Werkzeugleiste eingefügt. Die Klasse *KMainWindow* bietet eine noch komfortablere und flexiblere Art, bei der der zweite Schritt nahezu vollständig entfällt. Es werden nach wie vor zunächst die *KAction*-Objekte erzeugt. Wie diese aber in

die Menüs einzufügen sind, wird durch eine Datei im XML-Format festgelegt, die mit dem Befehl *KMainWindow::createGUI* eingelesen und ausgewertet wird. Diese Vorgehensweise hat eine Reihe von Vorteilen:

- Das Programm wird übersichtlicher, und auch die Menüstruktur ist in der XML-Datei sehr übersichtlich dargestellt.
- Es ist einfach möglich, die Anordnung der Befehle in der Menüleiste und den Werkzeugleisten zu ändern, ohne dass das Programm neu kompiliert werden muss. Insbesondere die Werkzeugleisten lassen sich sehr einfach vom Programm aus konfigurieren (mit Hilfe der Klasse *KEditToolBar*); das Ergebnis wird automatisch in der XML-Datei abgespeichert. So kann jeder Anwender ohne jegliche Kenntnisse von C++ oder XML seine eigenen Einstellungen vornehmen.
- Für die Standardaktionen aus *KStdAction* muss keine Position mehr festgelegt werden. Sie werden automatisch an ihre üblichen Positionen in den Menüs gesetzt.

Für unser einfaches Beispiel des *Hello-World*-Programms können wir den Konstruktor unserer selbst definierten Hauptfensterklasse ein gutes Stück vereinfachen:

```
MyMainWindow::MyMainWindow() : KMainWindow()
{
    text = new QLabel (i18n
        ("<H1>Hello, World!</H1>"), this);
    setCentralWidget (text);

    KAction *quitAction = KStdAction::quit (this,
        SLOT (fileQuit()), actionCollection());
    clearAction =
        new KAction (i18n ("%Clear"),
            "remove",
            Qt::CTRL + Qt::Key_X,
            this, SLOT (fileClear()),
            actionCollection(),
            "file_clear");

    createGUI();
}
```

Wie Sie im Listing sehen, werden die Aktionen nur noch erzeugt, aber nicht mehr von Hand in die Menüs eingefügt. Alle Zeilen zur Erzeugung von Popup-Menüs und zum Einfügen von Einträgen in die Menü- oder Werkzeugleiste wurden entfernt. Das geschieht automatisch durch den Aufruf der Methode *createGUI()*. Diese lädt die Datei mit dem Namen *khelloworldui.rc* aus dem Verzeichnis *\$KDEDIR/share/apps/khelloworld/*. Diese Datei legt nun fest, wie die Aktionen in

den Menüs verteilt werden. Solange wir diese Datei noch nicht angelegt haben, können aber nur die Aktionen von *KStdAction* eingefügt werden (in unserem Fall also die QUIT-Aktion).

Erstellen wir nun also die XML-Datei, die festlegt, an welchen Stellen die CLEAR-Aktion eingefügt werden soll. XML ist ein Dateiformat in ASCII-Text, das mit dem HTML-Format verwandt ist. Es setzt sich in der letzten Zeit immer mehr durch und wird vor allem für Konfigurationsdaten genutzt. Dieses Dateiformat kann relativ einfach vom Rechner gelesen und geschrieben werden, aber dennoch (zumindest theoretisch) mit einem normalen Texteditor auch von Hand geändert werden. Der Inhalt der Datei besteht aus so genannten Tags – das sind Schlüsselbegriffe, die in spitzen Klammern, also < und >, eingeschlossen sind. Tags können entweder allein für sich stehen (dann werden sie mit einem Slash » / « abgeschlossen), oder sie stehen paarweise für Anfang und Ende eines Bereichs. (Das Ende-Tag beginnt dazu mit einem Slash » / »). Auf diese Weise lassen sich komplexe Hierarchien aufbauen. Wichtig ist es, bei XML-Dateien darauf zu achten, dass die formalen Regeln zum Aufbau eingehalten werden. Im Gegensatz zu HTML muss laut Spezifikation der Computer beim Einlesen alle formalen Kriterien peinlich genau prüfen. Das bedeutet unter anderem, dass durch Anfangs- und End-Tags eingeschlossene Bereiche sich nicht überlappen dürfen: Einer der Bereiche muss vollständig im anderen enthalten sein, oder beide müssen völlig getrennt voneinander sein. Ebenso darf zu keinem Anfangs-Tag das zugehörige Ende-Tag fehlen. Eine genaue Beschreibung des XML-Formats würde den Rahmen dieses Buchs sprengen. Umfassende Informationen finden Sie beispielsweise im Buch *Das XML-Handbuch*, von Charles F. Goldfarb und Paul Prescod, Addison-Wesley Verlag, ISBN 3-8273-1712-6, oder im Internet unter <http://www.w3.org/XML/>.

Hier sehen Sie die XML-Datei für unser Beispiel:

```
<!DOCTYPE kpartgui>
<kpartgui name="khelloworld">
  <MenuBar>
    <Menu name="file"><text>&amp;File</text>
      <Action name="file_clear"/>
    </Menu>
  </MenuBar>

  <ToolBar>
    <Action name="file_clear"/>
  </ToolBar>
</kpartgui>
```

Geben Sie diese Datei mit einem beliebigen Texteditor ein, und speichern Sie sie im Verzeichnis *\$KDEDIR/share/apps/khelloworld/khelloworldui.rc* ab. Starten Sie das Programm erneut; nun müsste auch der Menüeintrag für *Clear* wieder erscheinen, und zwar sowohl in der Menüleiste als auch in der Werkzeugleiste.

Den Dateinamen der XML-Datei bilden Sie aus dem Namen Ihres Programms (den Sie zum Beispiel am Anfang der *main*-Funktion in *KCmdLineArgs::init* festgelegt haben), an den Sie die Endung »*ui.rc*« hängen. Wollen Sie einen anderen Dateinamen wählen, so können Sie diesen auch als Parameter bei *createGUI* explizit angeben.

Für aufwendigere Programme mit mehr Befehlen fügen Sie einfach mehrere Zeilen mit dem Tag `<Action name = "..."/>` hintereinander ein. In genau dieser Reihenfolge werden die Befehle dann auch in das Menü oder die Werkzeugleiste aufgenommen. Der angegebene Name muss dabei dem Namen des *KAction*-Objekts entsprechen, den Sie beim Anlegen (als siebten Parameter) vergeben haben. Wie Sie sehen, ist der Parameter an dieser Stelle wichtig. Achten Sie also darauf, dass Sie diesen Namen in allen selbst definierten Aktionen setzen und dass er eindeutig ist. Achten Sie auch darauf, dass Sie als Vater-Objekt unbedingt *actionCollection()* benutzen. *createGUI* verwendet nur Aktionen, die Kindobjekte dieser Gruppe sind.

Wollen Sie mehrere Werkzeugleisten erzeugen, so benutzen Sie mehrere Bereiche des Tags *ToolBar*, und fügen Sie in die Tags die Angabe *name="bezeichnung"* ein. *bezeichnung* kann dabei ein beliebiger Ausdruck sein, der die entsprechende Werkzeugleiste charakterisiert. Für zwei Werkzeugleisten (in der ersten kommt zweimal die Aktion *CLEAR* vor, in der anderen dreimal – nicht besonders sinnvoll, aber nur ein einfaches Beispiel) kann die XML-Datei zum Beispiel so aussehen:

```
<!DOCTYPE kpartgui>
<kpartgui name="khelloworld">
  <MenuBar>
    <Menu name="file"><text>&File</text>
      <Action name="file_clear"/>
    </Menu>
  </MenuBar>

  <ToolBar name="TheFirstBar">
    <Action name="file_clear"/>
    <Action name="file_clear"/>
  </ToolBar>
  <ToolBar name="AnotherBar">
    <Action name="file_clear"/>
    <Action name="file_clear"/>
    <Action name="file_clear"/>
  </ToolBar>
</kpartgui>
```

Es müsste übrigens noch eine Frage offen sein: Warum ist die *CLEAR*-Aktion in der Werkzeugleiste aufgetaucht, die *QUIT*-Aktion dagegen nicht? Aktionen von *KStdAction* sollten doch eigentlich automatisch aufgenommen werden. In der Menüleiste taucht die Aktion ja auch auf. Der Grund dafür ist, dass die *QUIT*-

Aktion normalerweise eben nicht in die Werkzeugleiste aufgenommen wird. Diese Aktion wird nicht sehr oft benötigt (eben genau einmal pro Programmstart), daher ist es nicht nötig, sie besonders schnell erreichen zu können. Andere Standardaktionen wie FILE-OPEN... oder EDIT-PASTE erscheinen automatisch in der Werkzeugleiste. Wollen Sie die QUIT-Aktion auch unbedingt aufnehmen, so können Sie sie auch in der XML-Datei von Hand einfügen:

```
<!DOCTYPE kpartgui>
<kpartgui name="khelloworld">
<MenuBar>
  <Menu name="file"><text>&amp;File</text>
    <Action name="file_clear"/>
  </Menu>
</MenuBar>

<ToolBar>
  <Action name="file_clear"/>
  <Action name="file_quit"/>
</ToolBar>
</kpartgui>
```

Wollen Sie Befehlsgruppen in der Menüleiste oder der Werkzeugleiste voneinander optisch abgrenzen, so fügen Sie einfach das Tag `<Separator/>` an die entsprechende Stelle in der XML-Datei ein. In der Menüleiste werden im Popup-Menü die Einträge dort durch eine Linie voneinander getrennt, in der Werkzeugleiste durch einen kleinen Zwischenraum.

```
<!DOCTYPE kpartgui>
<kpartgui name="khelloworld">
<MenuBar>
  <Menu name="file"><text>&amp;File</text>
    <Action name="file_clear"/>
    <Action name="file_clear"/>
    <Separator />
    <Action name="file_clear"/>
  </Menu>
</MenuBar>

<ToolBar>
  <Action name="file_clear"/>
  <Action name="file_clear"/>
  <Separator />
  <Action name="file_clear"/>
</ToolBar>
</kpartgui>
```

Um dem Anwender die Möglichkeit zu geben, sich die Werkzeugleiste nach eigenem Geschmack einzurichten, bietet die KDE-Bibliothek die Klasse *KEditToolBar*

an. Zusammen mit der Aktion *KStdAction::configureToolBar* ist die individuelle Gestaltung dieser Leiste ein Kinderspiel. Der Code in der selbst abgeleiteten Hauptfensterklasse kann dabei folgendermaßen aussehen:

Datei *mymainwindow.h*:

```
...
class MyMainWindow : public KMainWindow
{
    Q_OBJECT
public:
    MyMainWindow();
    ~MyMainWindow();

private slots:
    void fileQuit();
    void configToolBar();

private:
    ...
}
```

Datei *mymainwindow.cpp*:

```
MyMainWindow::MyMainWindow() : KMainWindow()
{
    text = new QLabel (i18n
        ("<H1>Hello, World!</H1>"), this);
    setCentralWidget (text);

    KAction *quitAction = KStdAction::quit (this,
        SLOT (fileQuit()), actionCollection());
    clearAction =
        new KAction (i18n ("&Clear"),
            "remove",
            Qt::CTRL + Qt::Key_X,
            this, SLOT (fileClear()),
            actionCollection(),
            "file_clear");
    KStdAction::configureToolbars
        (this, SLOT (configToolBar()), actionCollection());
    createGUI();
}

void MyMainWindow::configToolBar ()
{
    KEditToolBar dialog (actionCollection());
    if (dialog.exec())
        createGUI();
}
```


Der Dialog, der sich beim Auswählen der Aktion öffnet, ist in Abbildung 3.27 zu sehen.

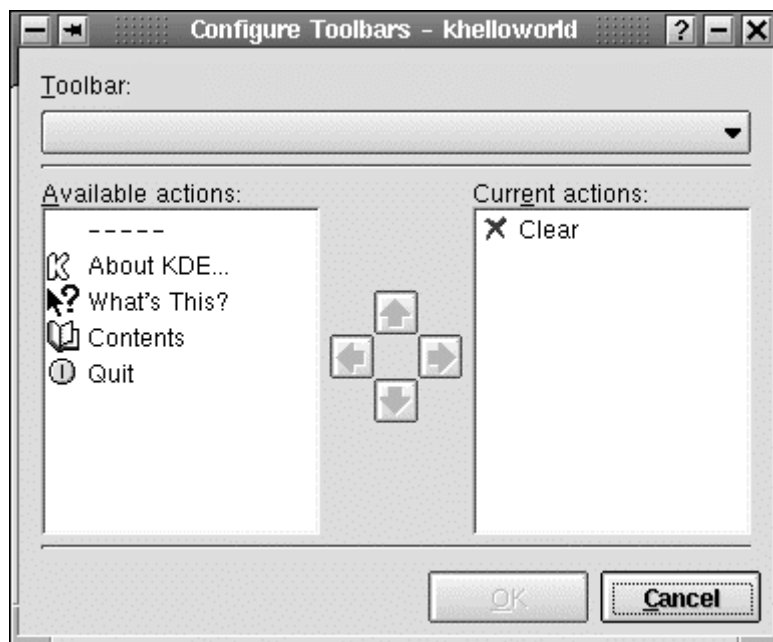


Abbildung 3-27 Dialogfenster zur Einstellung der Werkzeugleisten

Beim Beenden des Programms werden die neuen Daten über den Aufbau der Werkzeugleiste automatisch in der Datei *khelloworldui.rc* abgespeichert, allerdings dieses Mal nicht im Verzeichnis *\$KDEDIR/share/apps/khelloworld/* (dieses Verzeichnis hat für einen gewöhnlichen User keine Schreibrechte), sondern im Verzeichnis *\$HOME/.kde/share/apps/khelloworld/*. Die Datei an dieser Stelle hat beim nächsten Programmstart Vorrang.

Weitere Informationen zu XML-Dateien zum Aufbau von Menü- und Werkzeugleisten finden Sie im Internet auf der KDE-Developer-Homepage unter <http://developer.kde.org/documentation/tutorials/xmlui/preface.html>.

3.5.5 Die Statuszeile

In der Statuszeile stellt die Applikation die wichtigsten Einstellungen sowie Informationen über den aktuellen Zustand dar, in den meisten Fällen in Form eines Textes oder einer Abkürzung. Aber auch Icons, Fortschrittsbalken und andere Anzeigeelemente sind möglich. Grundsätzlich unterscheidet man zwei Arten von Statusmeldungen:

- Dauernd angezeigte Meldungen, zum Beispiel die aktuelle Cursorposition oder den Vergrößerungsfaktor, ob das aktuelle Dokument verändert wurde oder ob es schreibgeschützt ist.
- Längere Textmeldungen, die oft nur kurzzeitig angezeigt werden. Das sind beispielsweise Meldungen über längere Aktionen, die gerade durchgeführt werden (z.B. das Laden und Speichern von Dateien) oder Fehlermeldungen. In diese Kategorie fallen auch Hilfetexte, die in der Statuszeile erscheinen, wenn man mit dem Mauszeiger über Icons in der Werkzeugleiste fährt.

Wie die Menüleiste und die Werkzeugleiste auch, verwaltet *KMainWindow* das Objekt der Statuszeile. Sie erhalten es mit der Methode *KMainWindow::statusBar()*; der Rückgabewert ist ein Zeiger auf das *KStatusBar*-Objekt.

Textmeldungen über den Zustand

Um eine längere Textmeldung auszugeben, benutzen Sie die Methode *message*. Dieser übergeben Sie den Text als *QString*-Objekt, der in der Statuszeile angezeigt werden soll. Dort verbleibt er so lange, bis ein anderer Text mit *message* angezeigt wird oder bis die Methode *clear* aufgerufen wird. Sie löscht die Textmeldung. Sie können stattdessen auch eine Meldung nur für eine bestimmte Zeit anzeigen lassen, indem Sie als zweiten Parameter in *message* die Zahl der Millisekunden angeben. Nach dieser Zeit wird die Textmeldung automatisch gelöscht. Beachten Sie, dass immer nur eine längere Textmeldung angezeigt werden kann.

Ein typisches Anwendungsbeispiel ist die Anzeige einer Meldung, während das Programm mit etwas anderem beschäftigt ist. Der folgende Code könnte eine solche Meldung anzeigen, wenn ein Dokument gespeichert werden soll. Das Codestück befindet sich in einer Methode einer selbst definierten Hauptfensterklasse (abgeleitet von *KMainWindow*) und wird mit dem Namen der Datei als Parameter aufgerufen:

```
bool MyMainWindow::fileSave (QString filename)
{
    QString text = i18n ("Saving document %1...");
    statusBar()->message (text.arg (filename));
    ...
    // Speichern der Daten in der Datei
    // Variable success gibt an, ob erfolgreich
    ...
    if (success)
        text = i18n ("Saved %1 successfully");
    else
        text = i18n ("Saving of %1 failed!");
    statusBar()->message (text.arg (filename), 2000);
    return success;
}
```

Beim Beginn des Speichervorgangs wird die Meldung gesetzt, dass das Programm zur Zeit das Dokument speichert. (An dieser Stelle sollte möglichst auch der Mauscursor als Sanduhr gesetzt werden, siehe Kapitel 4.13, *Blockierungsfreie Programme*.) Nachdem der Speichervorgang abgeschlossen ist, wird für weitere zwei Sekunden angezeigt, ob das Ergebnis erfolgreich war oder nicht.

Beachten Sie, dass bei einem Aufruf von *message* mit einer Zeitangabe nicht garantiert ist, dass die Meldung für die angegebene Zeit sichtbar bleibt. Wird in der Zwischenzeit die Methode *message* oder *clear* aufgerufen, so wird die Textmeldung vorzeitig ersetzt bzw. gelöscht. Das bedeutet ganz generell, dass Sie in die Statuszeile keine entscheidend wichtigen Meldungen hineinsetzen sollten, die nicht übersehen werden dürfen. Sie haben nämlich keine Kontrolle darüber, ob die Meldung lange genug sichtbar ist. Die Texte in der Statuszeile sind eher für Meldungen gedacht, die dem Anwender erklären, was gerade passiert, während er auf eine Reaktion des Programms wartet.

Wenn Sie die Statuszeile nur für längere Textmeldungen benutzen, müssen Sie im Konstruktor der Hauptfensterklasse einmal die Methode *statusBar()* aufrufen, so dass die Statuszeile angelegt und angezeigt wird. Sonst würde die Statuszeile erst bei der ersten angezeigten Textmeldung eingeblendet, was den Anwender irritieren könnte. Oftmals benutzt man im Konstruktor auch eine Zeile wie die folgende:

```
MyMainWindow::MyMainWindow ()
    : KMainWindow ()
{
    ...
    // andere Initialisierungen
    ...
    statusBar()->message (i18n ("Ready."));
}
```

Dauernd angezeigte Meldungen

Die Statuszeile kann mehrere dauernd angezeigte Meldungen aufnehmen, die jeweils ein Feld belegen. Meist benutzt man einen kurzen Text oder einen Zahlenwert für die Anzeige. Ein solches Feld können Sie mit der Methode *KStatusBar::insertItem* einfügen. Als Parameter übergeben Sie vier Werte:

- Den Text für die Meldung (als *QString*-Objekt)
- Eine eindeutige Identifikationsnummer des Eintrags, mit deren Hilfe der Eintrag nachher wieder gelöscht oder verändert werden kann
- Einen Stretch-Faktor, der angibt, ob das Feld mit dem minimalen Platz auskommen soll (0, Defaultwert) oder ob es den restlichen zur Verfügung stehenden Platz belegen soll (Werte größer als 0)

- Einen *bool*-Wert für die Angabe, ob das Feld permanent sichtbar sein soll (*true*) oder ob es von temporären Textmeldungen übermalt werden darf (*false*, Defaultwert)

Der folgende Code erzeugt drei Felder in der Statuszeile, die für die Anzeige der Zeilennummer, der Spaltennummer und des Vergrößerungsfaktors vorbereitet werden. Als Identifikationsnummern benutzen wir der Einfachheit halber die Zahlen 1, 2 und 3.

```
statusBar()->insertItem ("Line xxx", 1);
statusBar()->insertItem ("Column xxx", 2);
statusBar()->insertItem ("Zoom: 100%", 3);
```

Der folgende Code setzt nun beispielsweise die neue Cursorposition in die Statuszeile:

```
QString text;
text = QString (i18n ("Line %1")).arg (line);
statusBar()->changeItem (text, 1);
text = QString (i18n ("Column %1")).arg (col);
statusBar()->changeItem (text, 2);
```

Welchen Text Sie initial in die Felder schreiben, ist nicht von Bedeutung, wenn Sie unmittelbar danach die Felder mit den korrekten Werten füllen lassen. Die Breite der Felder passt sich automatisch dem benötigten Platz an.

Sie können darauf reagieren, wenn diese Felder mit der Maus angeklickt werden, indem Sie einen eigenen Slot mit den Signalen *KStatusBar::pressed (int)* oder *KStatusBar::released (int)* verbinden. Der übergebene Parameter ist die Identifikationsnummer des Feldes, auf dem der Mauscursor stand.

Wollen Sie eine aufwendigere Statusleiste erzeugen, so können Sie mit der Methode *KStatusBar::addWidget* (geerbt von *QStatusBar*) auch ein eigenes Widget in die Statuszeile einfügen. Typische Widget-Klassen sind hierbei zum Beispiel *QLabel*, *QProgressBar*, *QPushButton*, *QComboBox* oder Ähnliches. Achten Sie darauf, dass das eingefügte Widget nur eine kleine Höhe besitzt (üblicherweise nicht mehr als 30 Pixel), damit die Statuszeile nicht zu viel Platz belegt. Das eingefügte Widget muss die Statuszeile als Vater-Widget haben. Die Methode *addWidget* besitzt neben dem ersten Parameter, der das einzufügende Widget angibt, noch zwei Parameter wie oben: den Stretch-Faktor und die Angabe, ob das Feld von Textmeldungen überdeckt werden darf.

Sie können die gleichen Felder wie oben auch mit folgendem Code erzeugen, indem Sie *QLabel*-Objekte in die Statuszeile einfügen:

```
lineLabel = new QLabel ("Line xxx", statusBar());
statusBar()->addWidget (lineLabel);
colLabel = new QLabel ("Column xxx", statusBar());
```

```
statusBar()->addWidget (colLabel);  
zoomLabel = new QLabel ("Zoom: 100%", statusBar());  
statusBar()->addWidget (zoomLabel);
```

Der Code zum Eintragen neuer Werte sieht in diesem Fall so aus:

```
QString text;  
text = QString (i18n ("Line %1")).arg (line);  
lineLabel->setText (text);  
text = QString (i18n ("Column %1")).arg (col);  
colLabel->setText (text);
```

Um ein Widget wieder aus der Statuszeile zu entfernen, benutzen Sie *KStatusBar::removeWidget*. Um es kurzzeitig auszublenden, verstecken Sie es mit *hide()*, um es später mit *show()* wieder anzuzeigen. (In der aktuellen Version von Qt bleibt jedoch ein kleiner, rechteckiger Rand stehen.)

3.5.6 Applikation für ein Dokument

Oftmals arbeiten Programme auf einem Dokument eines bestimmten Dateityps. Dieses Dokument kann üblicherweise geöffnet, angezeigt, geändert und wieder abgespeichert werden. In diesem Kapitel wollen wir uns die Struktur eines solchen Programms anschauen, ohne dabei natürlich zu sehr ins Detail zu gehen. Als Beispielprogramm entwerfen wir einen minimalen Texteditor. Als Anzeigebereich benutzen wir dazu die Klasse *QMultiLineEdit*.

Für unser einfaches Beispiel benutzen wir ausschließlich Aktionen aus *KStdAction*, und zwar die folgenden:

- NEW – löscht das aktuelle Dokument und lässt das Editorfenster leer
- OPEN... – löscht das aktuelle Dokument und fragt nach dem Namen einer Datei, die eingelesen werden soll
- OPEN RECENT – zeigt die Liste der letzten zehn geöffneten Dateien und ermöglicht das Laden einer der letzten Dateien
- SAVE – speichert das aktuelle Dokument unter seinem Dateinamen ab und fragt nach einem Dateinamen, falls noch keiner vorhanden ist
- SAVE AS... – fragt nach einem Dateinamen und speichert das Dokument unter diesem Namen ab
- QUIT – beendet das Programm
- UNDO – macht die letzte Änderung am Text rückgängig
- REDO – nimmt den letzten UNDO-Befehl zurück
- CUT – schneidet den markierten Text aus und legt ihn in die Zwischenablage

- COPY – kopiert den markierten Text in die Zwischenablage
- PASTE – fügt den aktuellen Inhalt der Zwischenablage an der Cursorposition ein
- SELECT ALL – markiert den gesamten Text

Die letzten sechs Aktionen werden direkt mit Slots des *QMultiLineEdit*-Objekts verbunden. *QMultiLineEdit* erledigt die gesamte Ausführung für uns. Wir müssen nur noch dafür sorgen, dass die Aktionen möglichst nur dann aktiviert werden können, wenn sie auch sinnvoll sind, d.h. wenn sie eine Auswirkung haben.

Um die Abarbeitung der ersten sechs Aktionen müssen wir uns selbst kümmern. Dazu definieren wir sechs Slots, die von diesen Aktionen aktiviert werden: *fileNew*, *fileOpen*, *fileOpenRecent*, *fileSave*, *fileSaveAs* und *fileQuit*.

Das Hauptprogramm in *main.cpp*

Werfen wir zunächst einen Blick auf das Hauptprogramm in *main.cpp*. Es ist nach der Struktur aufgebaut, die in Kapitel 3.3.2, *Grundstruktur einer KDE-Applikation*, beschrieben ist.

```
#include <kapp.h>
#include <kcmdlineargs.h>
#include <kaboutdata.h>
#include <klocale.h>

#include "kminiedit.h"

static const char *description =
    I18N_NOOP("KMiniEdit - A simple example editor");

static KCmdLineOptions options[] =
{
    { "+[arg1]", I18N_NOOP("Load that file or URL"), 0 },
    { 0, 0, 0 }
};

int main(int argc, char *argv[])
{
    KAboutData aboutData
        ("kminiedit", I18N_NOOP("KMiniEdit"),
         "0.1", description, KAboutData::License_GPL,
         "(c) 2000, Burkhard Lehner");
    aboutData.addAuthor ("Burkhard Lehner", 0,
                        "Burkhard.Lehner@gmx.de");
    KCmdLineArgs::init (argc, argv, &aboutData);
    KCmdLineArgs::addCmdLineOptions (options);
```

```

KCmdLineArgs *args = KCmdLineArgs::parsedArgs();
if (args->count() > 1)
    KCmdLineArgs::usage ("Only one file allowed!");

KApplication a;
KMiniEdit *kminiedit = new KMiniEdit();
if (args->count() > 0)
    kminiedit->loadFile (args->url(0));
kminiedit->show();

return a.exec();
}

```

Die einzige Besonderheit ist, dass ein Dateiname, der als Kommandozeilenparameter übergeben wurde, nach dem Erzeugen des Hauptfensterobjekts automatisch in das Hauptfenster geladen wird. Dazu benutzen wir die Methode *loadFile* der Hauptfensterklasse *KMiniEdit*, die wir später noch genauer erläutern werden.

Die Deklaration der Hauptfensterklasse in *kminiedit.h*

Schauen wir uns nun die Datei *kminiedit.h* an, die Datei, in der unsere Hauptfensterklasse *KMiniEdit* deklariert wird.

```

#ifndef KMINIEDIT_H
#define KMINIEDIT_H

#include <kurl.h>
#include <kmainwindow.h>

class KAction;
class KRecentFilesAction;
class QMultiLineEdit;
class QFile;

class KMiniEdit : public KMainWindow
{
    Q_OBJECT
public:
    KMiniEdit();
    ~KMiniEdit();

protected:
    bool queryClose();

private slots:
    void fileNew();
    void fileOpen();
    void fileOpenRecent(const KURL&);
    void fileSave();

```

```

    void fileSaveAs();
    void fileQuit();

    void checkClipboard();
    void checkEdited();

public:
    bool loadFile (KURL newurl);

private:
    void saveToLocalFile (QFile *);
    bool saveFile (KURL newurl);

    void resetEdited();

    KAction *saveAction, *pasteAction;
    KRecentFilesAction *recentAction;

    QMultiLineEdit *edit;
    KURL url;
};

#endif

```

Betrachten wir nun genauer, was in der Klasse deklariert wird und wozu es benutzt wird:

- Wir binden die Header-Dateien *kurl.h* und *kmainwindow.h* ein, da wir die Klassen *KURL* und *KMainWindow* in dieser Header-Datei als Objekte benutzen: *KMainWindow* als Oberklasse unserer neuen Klasse *KMiniEdit*, und *KURL* als Attributvariable. Die anderen Klassen – *KAction*, *KRecentFileAction*, *QMultiLineEdit* und *QFile* – benutzen wir nur als Zeiger oder Referenzen. Daher reicht es aus, diese Klassen als existierend anzugeben.
- Konstruktor und Destruktor sind ganz normal. Der Konstruktor wird das Anzeigefenster (*QMultiLineEdit*) anlegen und die Menüs aufbauen.
- Die Methode *queryClose* ist von *KMainWindow* geerbt. Sie ist eine virtuelle Methode und wird in unserer Klasse überschrieben. Da sie in *KMainWindow* als *protected* deklariert ist, deklarieren wir sie auch in unserer Klasse so. (Wir brauchen nicht explizit anzugeben, dass diese Methode virtuell sein soll. Das ist sie automatisch, da sie es bereits in *KMainWindow* war.) *KMainWindow* ruft diese Methode immer dann auf, wenn das Fenster geschlossen werden soll. In dieser Methode werden wir testen, ob sich noch ungespeicherte Daten im Editor befinden. Ist das der Fall, so fragen wir den Anwender, ob er sie speichern möchte oder nicht oder ob er abbrechen will. Im letzten Fall (oder wenn das Abspeichern nicht funktioniert) liefern wir als Rückgabewert *false*, woraufhin

KMainWindow das Fenster nicht schließt. Zum einen wird diese Methode immer dann aufgerufen, wenn der Anwender das Hauptfenster (mit dem X-Button in der Titelleiste) schließen will. Zum anderen benutzen wir diese Methode aber auch selbst an allen Stellen, an denen wir das Dokument schließen wollen. Somit bekommen wir eine lückenlose Sicherheitsabfrage.

- Die Slots *fileNew*, *fileOpen*, *fileSave*, *fileSaveAs* und *fileQuit* werden unmittelbar von den entsprechenden Aktionen aufgerufen. Wir müssen sie mit dem entsprechenden Code füllen, so dass sie auch die passenden Tätigkeiten ausführen. Der Slot *fileOpenRecent* wird nicht vom *activated*-Signal der Aktion aufgerufen, sondern wir verbinden ihn von Hand mit dem Signal *urlSelected* dieser Aktion. Dieses Signal liefert auch gleich die ausgewählte Datei als Parameter (vom Typ *KURL*) mit, so dass auch unser Slot diesen Parameter benötigt.
- Der Slot *checkClipboard* wird immer dann aufgerufen, wenn die Zwischenablage (*QClipboard*) meldet, dass sich die gespeicherten Daten geändert haben. In diesem Fall testen wir, ob sich ein Text in der Zwischenablage befindet, und aktivieren dann (und nur dann) die Aktion PASTE. Nähere Informationen zur Zwischenablage finden Sie in Kapitel 4.15.1, *Die Zwischenablage – QClipboard*.
- Der Slot *checkEdited* wird immer dann aufgerufen, wenn sich der Text im Anzeigebereich geändert hat. Er testet, ob der Text im *QMultiLineEdit*-Objekt ungespeicherte Daten enthält. Ist das der Fall, so aktiviert er die Aktion SAVE (nur in diesem Fall ist Speichern sinnvoll). Außerdem passt er den Text in der Titelzeile an, indem er das Wort *[modified]* dort einfügt, sowie den Dateinamen der aktuellen Datei.
- Die Methoden *loadFile* und *saveFile* laden und speichern eine Textdatei. (In unserem Programm können sie diese sogar von einem HTTP- oder FTP-Server laden.) Sie bekommen die URL (also eine Pfadangabe und einen Dateinamen, eventuell auch eine Protokollangabe und einen Rechner) als Parameter übergeben. Sie laden bzw. speichern die angegebene Datei und melden auftretende Fehler. Der Rückgabewert gibt an, ob die Operation erfolgreich war (*true*) oder nicht (*false*). Diese Methoden führen keinen Test durch, ob sich noch ungesicherte Änderungen im Editor befinden. Sie führen die eigentliche Operation aus, während die Slots *fileOpen*, *fileOpenRecent*, *fileSave* und *fileSaveAs* auf diese Methoden zurückgreifen. Nähere Informationen zu URLs und zum Laden und Speichern von Dateien über ein Netzwerk finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*. Informationen zum Einlesen und Schreiben von Textdateien finden Sie in Kapitel 4.18, *Dateizugriffe*.
- Die Methode *saveToLocalFile* ist eine Hilfsmethode, die den Inhalt des Editors in einer Hilfsdatei zwischenspeichert (das ist für Uploads nötig). Sie wird in *saveFile* benutzt.

- Die Methode *resetEdited* setzt den Zustand des Texts im Editor auf »keine ungespeicherten Änderungen« zurück. Wir benutzen diese Methode, wenn wir eine neue Datei geöffnet oder die aktuelle Datei gespeichert haben.
- Die Zeiger *saveAction*, *pasteAction* und *recentAction* speichern die Adressen der entsprechenden Aktionen. Auf diese Aktionen müssen wir im Verlauf des Programms zugreifen. *saveAction* wird deaktiviert, wenn keine ungespeicherten Änderungen im Editor vorhanden sind, *pasteAction* wird deaktiviert, wenn die Zwischenablage keinen Text enthält. *recentAction* enthält die Liste der zuletzt geöffneten Dateien; wir müssen diese Liste aktualisieren und in die Konfigurationsdatei schreiben. Alle anderen Aktionen werden im Konstruktor erzeugt, initialisiert und verbunden. Später müssen wir nicht mehr darauf zugreifen.
- Der Zeiger *edit* speichert die Adresse des *QMultiLineEdit*-Objekts, das unser Anzeigefenster ausfüllt. Auf dieses Objekt müssen wir sehr oft zurückgreifen, um den Inhalt auszulesen oder zu schreiben oder um festzustellen, ob es Änderungen gab.
- Die Variable *url* speichert den Dateinamen des aktuellen Dokuments. Haben wir noch keine Datei geöffnet (direkt nach dem Programmstart oder nach NEW), enthält diese Variable eine leere URL. Eine nähere Beschreibung zur Klasse *KURL* finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Zugriff mit KIO*, im Abschnitt *Festlegen einer URL*.

Der Code für die Methoden in *kminiedit.cpp*

Kommen wir nun zur längsten Datei, *kminiedit.cpp*. Sie enthält den Code für alle deklarierten Methoden, insbesondere den Konstruktor der Klasse *KMiniEdit*. Wir wollen uns den Inhalt der Datei schrittweise anschauen. Den gesamten Quellcode finden Sie auch auf der CD, die diesem Buch beiliegt.

Zunächst einmal müssen wir für alle Klassen, die wir benutzen wollen, die entsprechenden Header-Dateien einbinden. *kurl.h* und *kmainwindow.h* werden bereits in *kminiedit.h* eingebunden.

```
#include <kaction.h>           // Klasse KAction
#include <kapp.h>               // Zeiger kapp
#include <kfiledialog.h>       // Klasse KFileDialog
#include <kio/netaccess.h>      // Klasse KIO::NetAccess
#include <klocale.h>            // Funktion i18n
#include <kstdaction.h>         // Klasse KStdAction
#include <ktempfile.h>          // Klasse KTempFile
#include <kmessagebox.h>        // Klasse KMessageBox
#include <qclipboard.h>         // Klasse QClipboard
#include <qmultilineedit.h>     // Klasse QMultiLineEdit
#include <qfile.h>              // Klasse QFile
#include <qtextstream.h>        // Klasse QTextStream

#include "kminiedit.h"
```

Nun folgt der Konstruktor unserer Hauptfensterklasse:

```
KMiniEdit::KMiniEdit() : KMainWindow()  
{
```

Zuerst legt er das *QMultiLineEdit*-Widget an und setzt es in den Anzeigebereich:

```
    edit = new QMultiLineEdit (this);  
    setCentralWidget (edit);
```

Er erzeugt anschließend die Aktionen für das FILE-Menü. Alle Aktionen kommen dabei aus der Klasse *KStdAction*. Die meisten Aktionen werden direkt mit den zugehörigen Slots der Hauptfensterklasse verbunden, so dass sie gar nicht zwischengespeichert werden müssen.

```
    KStdAction::openNew (this, SLOT (fileNew()),  
                        actionCollection());  
    KStdAction::open (this, SLOT (fileOpen()),  
                     actionCollection());
```

Für die Aktion *Open Recent* ist das anders: Wir benutzen das normale Signal *activated* nicht, geben also als Objekt und als Slot einen Nullzeiger an. Dafür speichern wir das Objekt aber in unserer Attributvariablen *recentAction* ab. Wir holen aus der Konfigurationsdatei unseres Programms die Liste der zuletzt geöffneten Dateien, die dort beim letzten Aufruf des Programms gespeichert wurden. Nähere Informationen zu Konfigurationsdateien finden Sie in Kapitel 4.10, *Konfigurationsdateien*. Außerdem verbinden wir das Signal *urlSelected* mit unserem eigenen Slot:

```
    recentAction = KStdAction::openRecent  
        (0, 0, actionCollection());  
    recentAction->loadEntries (KGlobal::config());  
    connect (recentAction,  
            SIGNAL (urlSelected (const KURL &)),  
            this, SLOT (fileOpenRecent (const KURL &)));
```

Auch die Aktion *SAVE* hat eine Besonderheit: Sie soll nicht immer aktiv sein, sondern nur, wenn der Editor ungespeicherte Änderungen enthält. Wir müssen uns daher das *KAction*-Objekt merken (in der Attributvariable *saveAction*), um später die Aktion ein- und ausschalten zu können. Die Überprüfung soll immer dann stattfinden, wenn sich im Editor etwas getan hat. Also verbinden wir das Signal *textChanged* des Editor-Fensters mit unserem Slot *checkEdited*, der den Test durchführt. Damit der Zustand auch jetzt beim Initialisieren schon korrekt gesetzt wird, rufen wir *checkEdited* auch einmal von Hand auf:

```

saveAction = KStdAction::save (this, SLOT (fileSave()),
                                actionCollection());
checkEdited();
connect (edit, SIGNAL (textChanged()),
        this, SLOT (checkEdited()));

```

Für die Aktionen **SAVE AS** und **QUIT** gibt es wiederum keine Besonderheit:

```

KStdAction::saveAs (this, SLOT (fileSaveAs()),
                    actionCollection());
KStdAction::quit (this, SLOT (fileQuit()),
                  actionCollection());

```

Nun erzeugen wir die Aktionen für das **EDIT**-Menü. Alle Aktionen werden mit Slots des *QMultiLineEdit*-Objekts in *edit* verbunden. Die meisten Aktionen sind aber auch hier nicht immer sinnvoll, so dass sie im Verlauf des Programms deaktiviert werden. Dafür liefert *QMultiLineEdit* bereits passende Signale, die wir direkt mit den Slots *setEnabled* der *KAction*-Objekte verbinden. Außerdem setzen wir den Anfangszustand dieser Aktionen korrekt. Wir benötigen die Aktionen später nicht mehr, da sie komplett mit ihren Signalen und Slots gesteuert werden. Daher reicht es, die erzeugten *KAction*-Objekte in einer lokalen Zeigervariablen zwischenzuspeichern. Wir benutzen in diesem Fall sogar nur eine einzige Variable *a*, die wir immer wieder neu belegen:

```

KAction *a;
a = KStdAction::undo (edit, SLOT (undo()),
                      actionCollection());
a->setEnabled (false);
connect (edit, SIGNAL (undoAvailable(bool)),
        a, SLOT (setEnabled(bool)));

a = KStdAction::redo (edit, SLOT (redo()),
                     actionCollection());
a->setEnabled (false);
connect (edit, SIGNAL (redoAvailable(bool)),
        a, SLOT (setEnabled(bool)));

a = KStdAction::cut (edit, SLOT (cut()),
                    actionCollection());
a->setEnabled (false);
connect (edit, SIGNAL (copyAvailable(bool)),
        a, SLOT (setEnabled(bool)));

a = KStdAction::copy (edit, SLOT (copy()),
                     actionCollection());
a->setEnabled (false);
connect (edit, SIGNAL (copyAvailable(bool)),
        a, SLOT (setEnabled(bool)));

```

Die Aktion PASTE hat wieder eine Besonderheit: Nur wenn die Zwischenablage Text enthält, sollte sie aktiv sein. Diesen Test machen wir in der Slot-Methode *checkClipboard*. Wir verbinden diese Slot-Methode mit dem Signal *dataChanged* der Zwischenablage und rufen sie außerdem einmal direkt auf, um den Anfangszustand korrekt zu setzen. Weitere Informationen zur Zwischenablage finden Sie in Kapitel 4.15.1, *Die Zwischenablage – QClipboard*.

```
pasteAction = KStdAction::paste (edit, SLOT (paste()),
                                actionCollection());

checkClipboard ();
connect (kapp->clipboard(), SIGNAL (dataChanged()),
        this, SLOT (checkClipboard()));
```

Die Aktion SELECT ALL ist nun wieder ganz normal:

```
KStdAction::selectAll (edit, SLOT (selectAll()),
                      actionCollection());
```

Schließlich rufen wir *createGUI* auf, um Menü- und Werkzeugleisten anlegen zu lassen:

```
createGUI();
}
```

Der Destruktor ist arbeitslos, da alle anderen Objekte automatisch gelöscht werden:

```
KMiniEdit::~KMiniEdit()
{
}
```

Die Methode *queryClose* wird von *KMainWindow* geerbt und hier überschrieben. Sie wird aufgerufen, wenn das Fenster geschlossen werden soll. Wir prüfen in ihr ab, ob es noch ungespeicherte Änderungen gibt, und fragen in diesem Fall den Anwender, was damit zu tun ist. Als Rückgabewert liefern wir *true* zurück, wenn das Fenster geschlossen werden kann (keine Änderungen, Änderungen korrekt gespeichert oder der Anwender will sie verwerfen). Wollen wir das Fenster dagegen offen halten, so liefern wir *false* zurück. Wir rufen diese Methode auch selbst an allen Stellen auf, an denen wir das Dokument schließen wollen (z.B. in *fileNew*, *fileOpen* und *fileQuit*). Weitere Informationen zur Klasse *KMessageBox* finden Sie in Kapitel 3.7.8, Dialoge. Die Funktion *i18n* wird in Kapitel 4.9.1, *KDE-Übersetzungen – Die Funktion i18n*, die Klasse *QString* in Kapitel 4.7.4, *Die String-Klassen – QString und QCString*, genauer beschrieben.

```
bool KMiniEdit::queryClose ()
{
    if (!edit->edited())
```

```

    return true;
    QString text
        = i18n("<H2>Unsaved Changes</H2>"
               "Save the changes in document %1 before "
               "closing it?");
    int result = KMessageBox::warningYesNoCancel
        (this, text.arg(url.prettyURL()));
    if (result == KMessageBox::Yes)
        return saveFile(url);
    return (result == KMessageBox::No);
}

```

Der selbst definierte Slot *fileNew* wird vom Menübefehl **NEW** aufgerufen. Wir prüfen zunächst, ob alle Änderungen gespeichert wurden. Dann setzen wir den Dateinamen der »geöffneten« Datei auf ein neues, leeres *KURL*-Objekt, löschen den Inhalt des Editor-Fensters und rufen *resetEdited* auf, um anzuzeigen, dass es bisher keine Änderungen an unserem neuen Dokument gab.

```

void KMiniEdit::fileNew()
{
    if (queryClose())
    {
        url = KURL ();
        edit->clear();
        resetEdited();
    }
}

```

Der Slot *FILEOPEN* wird von **OPEN** aufgerufen. Wir prüfen wiederum zunächst, ob alle Änderungen gespeichert wurden. Dann öffnen wir ein Dialogfenster zur Eingabe eines Dateinamens (statische Methode *getOpenURL* von *KFileDialog*). Bei Abbruch des Dialogs wird eine leere URL zurückgeliefert. Ist sie nicht leer, so laden wir die angegebene Datei mit der Methode *loadFile* in unser Editorfenster:

```

void KMiniEdit::fileOpen()
{
    if (queryClose())
    {
        KURL newurl = KFileDialog::getOpenURL ();
        if (!newurl.isEmpty())
            loadFile (newurl);
    }
}

```

Der Slot *fileOpenRecent* wird aufgerufen, wenn der Anwender aus der Liste der zuletzt geöffneten Dateien (im Menüpunkt **OPEN RECENT**) eine ausgewählt hat. Als Parameter wird die URL übergeben. Nach dem üblichen Test, ob alle Änderungen gespeichert wurden, laden wir die angegebene Datei mit *loadFile*:

```
void KMiniEdit::fileOpenRecent(const KURL &newurl)
{
    if (queryClose())
        loadFile (newurl);
}
```

Der Slot *fileSave* speichert das Dokument unter dem vorhandenen Dateinamen ab, der in der Attributvariablen *url* gespeichert ist. Dieser Dateiname ist entweder der Dateiname, mit dem das Dokument geöffnet wurde oder unter dem es beim letzten Aufruf von SAVE AS... gespeichert wurde. Für neue Dokumente ist dieser Dateiname leer. Dieser Sonderfall wird aber von *saveFile* abgefangen, das dann selbst nach einem Dateinamen fragt:

```
void KMiniEdit::fileSave()
{
    saveFile (url);
}
```

Der Slot *fileSaveAs* speichert das Dokument unter einem neuen Dateinamen ab. Wir rufen dazu einfach *saveFile* mit einer leeren URL auf. *saveFile* fragt dann selbstständig nach einem neuen Dateinamen:

```
void KMiniEdit::fileSaveAs()
{
    saveFile (KURL());
}
```

Der Slot *fileQuit* beendet das Programm, indem er die Methode *quit* unseres *KApplication*-Objekts aufruft, natürlich nicht, ohne vorher die Sicherheitsabfrage durchzuführen. (Das Makro *kapp* ist in der Header-Datei *kapp.h* definiert. Es liefert einen Zeiger auf das *KApplication*-Objekt zurück, von dem es ja nur eines gibt.)

```
void KMiniEdit::fileQuit()
{
    if (queryClose())
        kapp->quit();
}
```

Die Methode *loadFile* übernimmt für uns das Laden einer Datei in das Editor-Fenster. Der Dateiname wird dazu als Parameter übergeben. Der Rückgabewert gibt an, ob die Datei korrekt geladen werden konnte (*true*):

```
bool KMiniEdit::loadFile (KURL newurl)
{
```

Wir benutzen zum Laden die *KIO*-Klassen von KDE, die es uns ermöglichen, sogar Dateien von einem FTP- oder HTTP-Server zu laden. Genauere Informatio-

nen über diese Klassen finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*. Wir wollen hier nicht näher auf den Code eingehen, sondern sie einfach als gegeben hinnehmen. Wir kopieren die Datei auf die lokale Festplatte – allerdings nur für den Fall, dass es sich nicht ohnehin um eine lokale Datei handelt.

```

if (newurl.isMalformed())
{
    QString text =
        i18n("<B>The URL %1 is not correct!</B>");
    KMessageBox::sorry (this,
                        text.arg (newurl.prettyURL()));
    return false;
}
QString filename;
if (newurl.isLocalFile())
    filename = newurl.path();
else
{
    if (!KIO::NetAccess::download (newurl, filename))
    {
        QString text =
            i18n("<B>Error downloading %1!</B>");
        KMessageBox::sorry (this,
                            text.arg (newurl.prettyURL()));
        return false;
    }
}

```

Nachdem sich nun die Datei auf unserer lokalen Festplatte befindet, öffnen wir sie mit der Klasse *QFile* und lesen sie in einem Stück ein. Mit der Methode *QMultiLineEdit::setText* tragen wir den Inhalt in das Editorfenster ein. Nähere Informationen zu den Klassen zur Dateiverarbeitung finden Sie in Kapitel 4.18, *Dateizugriffe*. An dieser Stelle müssten eigentlich noch weitere Tests stattfinden, ob die Datei auch tatsächlich geöffnet und geladen werden konnte. Wir haben hier darauf verzichtet, um das Beispiel einfach zu halten.

```

QFile file (filename);
file.open (IO_ReadOnly);
QTextStream stream (&file);
stream.setEncoding (QTextStream::UnicodeUTF8);
edit->setText (stream.read());
file.close();

```

Die folgende Zeile löscht die lokale Kopie (allerdings nur, wenn sich die Datei nicht ohnehin auf unserer lokalen Festplatte befunden hat):

```

KIO::NetAccess::removeTempFile (filename);

```


Das Laden ist nun abgeschlossen. Wir speichern den neuen Dateinamen in unserer Attributvariablen *url*:

```
url = newurl;
```

Nun aktualisieren wir noch die Liste der zuletzt geöffneten Dateien, indem wir den neuen Dateinamen zum *KRecentFilesAction*-Objekt hinzufügen. Wir speichern den neuen Inhalt der Liste direkt in der Konfigurationsdatei ab, so dass er auch beim nächsten Programmstart wieder zur Verfügung steht. (Nähere Informationen zu den Konfigurationsdateien finden Sie in Kapitel 4.10, *Konfigurationsdateien*.)

```
recentAction->addURL (url);  
recentAction->saveEntries (KGlobal::config());
```

Wir setzen noch den Editor zurück (keine Änderungen bisher) und geben *true* zurück als Zeichen, dass das Laden erfolgreich war:

```
resetEdited();  
return true;  
}
```

Die Hilfsmethode *saveToLocalFile* speichert den Inhalt unseres Editors in einer Datei ab, die als Zeiger auf ein *QFile*-Objekt übergeben wird. Informationen zum Zugriff auf Dateien finden Sie in Kapitel 4.18, *Dateizugriffe*. (Auch an dieser Stelle sollte man genauer prüfen, ob beim Speichern keine Fehler auftraten. Wir haben diesen Test der Einfachheit halber weggelassen.)

```
void KMiniEdit::saveToLocalFile (QFile *file)  
{  
    QTextStream stream (file);  
    stream.setEncoding (QTextStream::UnicodeUTF8);  
    stream << edit->text();  
    file->close();  
}
```

Die Methode *saveFile* speichert die Datei unter dem übergebenen Dateinamen ab. Dieser Dateiname kann auch zum Beispiel auf einen FTP- oder HTTP-Server verweisen. Der Rückgabewert ist wieder *true*, falls die Datei erfolgreich gespeichert werden konnte.

```
bool KMiniEdit::saveFile (KURL newurl)  
{
```

Ist der übergebene Dateiname leer (neues Dokument oder Aufruf aus *fileSaveAs* heraus), so wird ein Dialog geöffnet, in dem der Anwender den Dateinamen eingeben kann:

```
if (newurl.isEmpty())
    newurl = KFileDialog::getSaveURL ();
```

Ist der Dateiname immer noch leer, so bedeutet das, dass der Anwender den Dialog zur Auswahl abgebrochen hat. Wir beenden einfach die Methode:

```
if (newurl.isEmpty())
    return false;
```

Im anderen Fall – es liegt also eine URL vor – testen wir, ob diese sinnvoll gebildet ist:

```
if (newurl.isMalformed())
{
    QString text =
        i18n ("<B>The URL %1 is not correct!</B>");
    KMessageBox::sorry (this,
                        text.arg (newurl.prettyURL()));
    return false;
}
```

Verweist der Dateiname auf ein Verzeichnis auf der eigenen Festplatte, benutzen wir einfach die Methode *saveToLocalFile*. Wir legen dazu ein *QFile*-Objekt zu diesem Dateinamen an und übergeben es:

```
if (newurl.isLocalFile())
{
    QFile file (newurl.path());
    file.open (IO_WriteOnly);
    saveToLocalFile (&file);
}
```

Im anderen Fall öffnen wir zunächst eine temporäre Hilfsdatei, in der wir die Daten abspeichern. Diese wird dann auf den FTP- oder HTTP-Server kopiert. Wir gehen hier nicht näher auf den Code ein, er wird ausführlich in Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*, beschrieben.

```
else
{
    KTempFile tempfile;
    saveToLocalFile (tempfile.file());
    if (!KIO::NetAccess::upload (tempfile.name(), newurl))
    {
        QString text = i18n ("<B>Error uploading %1!</B>");
        KMessageBox::sorry (this,
                            text.arg (newurl.prettyURL()));
        tempfile.unlink();
        return false;
    }
    tempfile.unlink();
}
```

Nachdem die Datei nun erfolgreich gespeichert wurde, legen wir den Dateinamen in der Attributvariablen *url* ab. Danach fügen wir ihn in die Liste der zuletzt geöffneten Dateien ein. (Ist er dort bereits vorhanden, wird der alte Eintrag gelöscht und der neue vorn angehängt. Der Dateiname rutscht so an die erste Stelle der Liste.) Anschließend vermerken wir im Editorfenster, dass keine Änderungen vorliegen, und geben *true* als Zeichen für ein erfolgreiches Speichern zurück.

```
url = newurl;
recentAction->addURL (url);
recentAction->saveEntries (KGlobal::config());
resetEdited();
return true;
}
```

Der Slot *checkClipboard* wird immer aufgerufen, wenn sich der Inhalt der Zwischenablage geändert hat. Wir prüfen in diesem Slot, ob die Zwischenablage einen Text enthält (oder einen Datentyp, der sich in einen Text umwandeln lassen kann). Ist das der Fall, aktivieren wir die Aktion PASTE. Sonst deaktivieren wir sie. Nähere Informationen zur Zwischenablage finden Sie in Kapitel 4.15.1, *Die Zwischenablage – QClipboard*.

```
void KMiniEdit::checkClipboard()
{
    pasteAction->setEnabled (kapp->clipboard()->
                           text() != QString::null);
}
```

Die Slot-Methode *checkEdited* testet, ob der Editor ungespeicherte Änderungen enthält. Ist das der Fall, wird die Aktion SAVE aktiviert. Außerdem setzen wir die Titelzeile des Programms entsprechend. Dazu benutzen wir die Methode *KMainWindow::setCaption*. Als ersten Parameter übergeben wir die gerade bearbeitete Datei (oder »New Document«, falls sie noch keinen Dateinamen hat). Als zweiten Parameter übergeben wir die Information, ob es noch ungespeicherte Änderungen gab. Die Methode *setCaption* erzeugt daraus automatisch eine Titelzeile, die all diese Informationen enthält:

```
void KMiniEdit::checkEdited()
{
    bool modified = edit->edited();
    saveAction->setEnabled (modified);
    if (url.isEmpty())
        setCaption (i18n ("New Document"), modified);
    else
        setCaption (url.prettyURL(), modified);
}
```

Die Methode *resetEdited* ist eine kleine Hilfsmethode. Sie setzt zunächst im *QMultiLineEdit*-Widget das Flag für Modifikationen zurück und ruft anschließend *checkEdited* auf, um die Anzeige entsprechend zu aktualisieren.

```
void KMiniEdit::resetEdited()
{
    edit->setEdited (false);
    checkEdited();
}
```

Kompilieren und Starten des Beispiels

Um das Programm zu kompilieren, führen Sie folgende Schritte aus:

```
% moc kminiedit.h -o kminiedit.moc.cpp
% g++ -c kminiedit.cpp -I$QTDIR/include -I$KDEDIR/include
% g++ -c kminiedit.moc.cpp -I$QTDIR/include -I$KDEDIR/include
% g++ -c main.cpp -I$QTDIR/include -I$KDEDIR/include
% g++ -o kminiedit *.o -lkio -lkfile -lkdeui -lkdecore -lqt
```

Vergessen Sie also insbesondere nicht, beim Linken die Bibliotheken *libkio* (für den Netzwerkzugriff auf Dateien) und *libkfile* (für den Dateidialog) einzubinden. Anschließend können Sie das Programm zum ersten Mal aufrufen und testen.

Außerdem sollten Sie noch die XML-Datei zur Anordnung der Aktionen in Menü- und Werkzeugleiste anlegen. Da wir nur Aktionen aus *KStdAction* benutzt haben, kann diese Datei minimal sein:

```
<!DOCTYPE kpartgui>
<kpartgui name="kminiedit">
</kpartgui>
```

Speichern Sie diese Datei unter *\$KDEDIR/share/apps/kminiedit/kminieditui.rc* ab.

Wenn Sie nun das Programm starten, sollte sich ein Fenster öffnen, das dem in Abbildung 3.28 ähnelt.

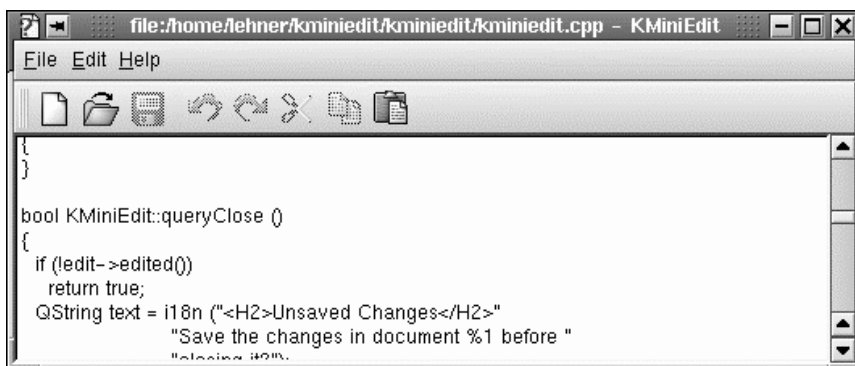


Abbildung 3-28 KMiniEdit in Aktion

Experimentieren Sie ein bisschen mit dem Editor, und vergleichen Sie sein Verhalten mit dem anderer Editoren, die Sie kennen.

Wenn Sie eine Textdatei öffnen, die deutsche Umlaute (oder andere europäische Sonderzeichen) enthält, so werden diese anscheinend nicht korrekt angezeigt. Wenn Sie andererseits Umlaute im Editor eingeben, so werden sie korrekt dargestellt, die gespeicherte Datei enthält an diesen Stellen aber scheinbar Unsinn. Dennoch ist nach dem Laden in den Editor alles wieder korrekt. Woran liegt das? Ist das ein Fehler unseres Editors?

Nein, das ist es nicht, unser Editor ist seiner Zeit nur schon etwas voraus! Er lädt und speichert seine Textdateien im *Unicode*-Format ab (genauer: im *UTF-8*-Format). Dieses Format speichert alle ASCII-Zeichen im Bereich von 0 bis 127 wie üblich ab. Alle anderen Sonderzeichen (und dazu zählen auch exotische Schriften wie Chinesisch und Japanisch) werden als Kombination von Zeichen des Bereichs 128 bis 255 dargestellt.

Das UTF-8-Format setzt sich langsam immer weiter durch. Es hat den Vorteil, dass es problemlos auch exotische Zeichen speichert. (Kennen Sie nicht auch das Problem, dass Sie eine E-Mail von einem Bekannten aus Russland nicht lesen können, weil dummerweise der falsche Zeichensatz eingestellt ist?) Auch KDE benutzt für seine Konfigurations- und Übersetzungsdateien zunehmend Unicode UTF-8. Öffnen Sie zum Beispiel einmal die Datei *\$KDEDIR/share/applnk/Editors/kwrite.desktop* – zum einen mit einem »normalen« Editor, zum anderen mit unserem selbst entwickelten Editor. Der normale Editor zeigt bei vielen Übersetzungen nur wilden Zeichensalat an, unser Editor dagegen zeigt die richtigen Sonderzeichen an. (Wenn bei vielen Übersetzungen trotzdem nur Fragezeichen dargestellt werden, so liegt das daran, dass Ihr X-Server keinen Font installiert hat, der diese exotischen Zeichen enthält.)

Noch sind allerdings die meisten Textdokumente, die man im europäischen Raum verwendet, nicht im UTF-8-Format, sondern im Latin-1-Format abgespeichert. Wenn Sie unseren Editor für solche Dateien verwenden wollen, müssen Sie im Programm in den Methoden *loadFile* und *saveToLocalFile* jeweils die Zeile

```
stream.setEncoding (QTextStream::UnicodeUTF8);
```

entfernen. Nun benutzt unser Editor zum Laden und Speichern von Dateien das Format Latin-1 (die Default-Einstellung). Besser wäre es natürlich, wenn der Anwender innerhalb des Programms wählen könnte, welches Format er benutzen möchte.

Weitere Informationen zu Unicode finden Sie in Kapitel 4.8, *Der Unicode-Standard*. Die Klasse *QTextStream* wird in Kapitel 4.18.4, *Stream-basierte Ein-/Ausgabe*, genauer beschrieben.

Anpassen des Beispiels an eigene Projekte

Wenn Sie dieses Beispiel für Ihr eigenes Programm anpassen wollen, so müssen Sie wahrscheinlich folgende Änderungen vornehmen:

- Im Konstruktor der Hauptfensterklasse müssen Sie als Anzeigebereich die Widget-Klasse wählen, die Ihren gewünschten Dokumenttyp darstellen und eventuell editieren kann. Wählen Sie möglichst eine Klasse, die die passenden Slots für die Aktionen aus dem EDIT-Menü bereits enthält (insbesondere UNDO und REDO). Alle anderen müssen Sie in Ihrer Hauptfensterklasse deklarieren und ausführen.
- Die Methoden zum Laden und Speichern des Dokuments (hier insbesondere Teile von *loadFile* und *saveToLocalFile*) müssen an Ihren Dokumenttyp angepasst werden.
- Falls die Widget-Klasse des Anzeigebereichs keine Möglichkeit bietet, Änderungen abzufragen, müssen Sie eine eigene *bool*-Variable *modified* zur Hauptfensterklasse hinzunehmen. Diese müssen Sie möglichst immer aktuell halten, und dementsprechend die Titelzeile und die Aktion SAVE anpassen (wie in *checkEdited*).
- In *checkClipboard* müssen Sie testen, ob der Inhalt der Zwischenablage von einem geeigneten Format ist, um in Ihr Dokument eingefügt werden zu können. Nähere Informationen zur Zwischenablage und zu Mime-Typen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag & Drop*.
- In der Regel werden Sie noch viele weitere Aktionen definieren und implementieren, sowohl solche aus *KStdAction* als auch selbst definierte.

3.5.7 Eine Applikation für mehrere Dokumente

Oftmals ist es sinnvoll, wenn man in einem Programm gleich mehrere Dokumente geöffnet haben kann. Mit unserem bisherigen Ansatz musste der Anwender dafür das Programm mehrfach starten. Das belegt aber unnötige Ressourcen und ist für den Anwender unpraktisch, insbesondere da viele Fenster geöffnet sind, die sich leicht gegenseitig verdecken.

Alle großen Office-Pakete erlauben es dem Anwender daher, mehrere Dokumente gleichzeitig innerhalb eines Programms geöffnet zu haben. Dazu ergeben sich in den Aktionen folgende Änderungen:

- OPEN schließt das aktuelle Dokument nun nicht mehr, sondern öffnet nur ein weiteres.
- NEW öffnet ebenfalls ein neues leeres Dokument, ohne das alte Dokument zu schließen.

- CLOSE kommt als neue Aktion hinzu. Sie schließt das aktuelle Dokument; natürlich nicht ohne vorher abzufragen, ob alle geänderten Daten gesichert sind.
- QUIT schließt jetzt alle noch geöffneten Dokumente (mit Sicherheitsabfrage), bevor es das Programm wirklich beendet.

Es gibt nun grundsätzlich zwei Möglichkeiten, die Unterfenster auf dem Bildschirm darzustellen:

Im *Single Document Interface (SDI)* erhält jedes Dokument ein eigenes Hauptfenster, mit Titelzeile, Menü- und Werkzeugleiste, Statuszeile und Anzeigebereich. Die Verwaltung der Fenster übernimmt der Window-Manager. Der Anwender kann kaum unterscheiden, ob es sich um ein einziges Programm handelt oder ob das Programm mehrfach gestartet wurde. Hier empfiehlt es sich sogar, den Menüpunkt FILE-QUIT ganz wegzulassen, da der Anwender überrascht sein könnte, dass beim Beenden in einem Fenster alle anderen Fenster ebenfalls geschlossen werden.

Im *Multi Document Interface (MDI)* hat das Programm nur ein einziges Hauptfenster, also auch nur eine Titelzeile, eine Menü- und Werkzeugleiste und eine Statuszeile. Im Anzeigebereich werden die einzelnen Dokumente wie kleine Fenster im Fenster dargestellt. Das Programm selbst simuliert dabei eine Art Window-Manager: Nur eines der Dokumente ist gerade aktiv. Die Dokumente lassen sich innerhalb des Anzeigebereichs verschieben und in der Größe ändern. Sie lassen sich auch maximieren und minimieren.

Abbildung 3.29 zeigt unseren Editor mit dem Single Document Interface, Abbildung 3.30 mit dem Multi Document Interface, jeweils mit zwei geöffneten Dateien.

MDI hat den Vorteil, dass es weit weniger Platz verbraucht. (Es gibt nur ein Hauptfenster und nur einmal die Menü- und Werkzeugleisten sowie die Statuszeile.) Auch tummeln sich nicht so viele unabhängige Fenster auf dem Bildschirm, so dass die Fensterleiste übersichtlich bleibt. Sein Nachteil ist jedoch, dass es oftmals nicht so intuitiv in der Bedienung ist. Für den Anwender ist es manchmal nicht sofort ersichtlich, auf welches Dokument sich die Befehle aus den Menüs auswirken, und hat er ein Dokument maximiert, sind die anderen Dokumente unsichtbar, so dass er leicht vergisst, dass er noch andere Dokumente geöffnet hat. Auch in vielen Microsoft-Programmen geht der Trend inzwischen wieder weg von MDI, zurück zu SDI.

MDI ist aber nicht in jedem Fall schlecht. Als Faustregel kann man sich vielleicht merken, dass SDI die bessere Wahl ist, wenn es sich um voneinander unabhängige Dokumente handelt. MDI ist dagegen gut geeignet, wenn es Querverbindungen zwischen den Dokumenten gibt. Ein anderes Entscheidungskriterium ist die

durchschnittliche Größe und Anzahl der Dokumentenfenster: Hat man oft viele, aber relativ kleine Dokumentenfenster, so ist MDI durch die Platzersparnis besser; bei wenigen und eher großen Dokumentenfenstern ist der Vorteil nicht mehr so gravierend, so dass SDI vorzuziehen ist.

KMiniEdit mit SDI

Als Beispiel wollen wir unserem simplen Texteditor *KMiniEdit* die Fähigkeit verleihen, mehrere Dokumente gleichzeitig geöffnet zu haben. Zunächst wollen wir dazu das SDI verwirklichen. Abbildung 3.29 zeigt das Programm mit mehreren geöffneten Dokumenten. Es ist auf den ersten Blick nicht zu erkennen, ob das Programm nicht doch mehrfach gestartet wurde.

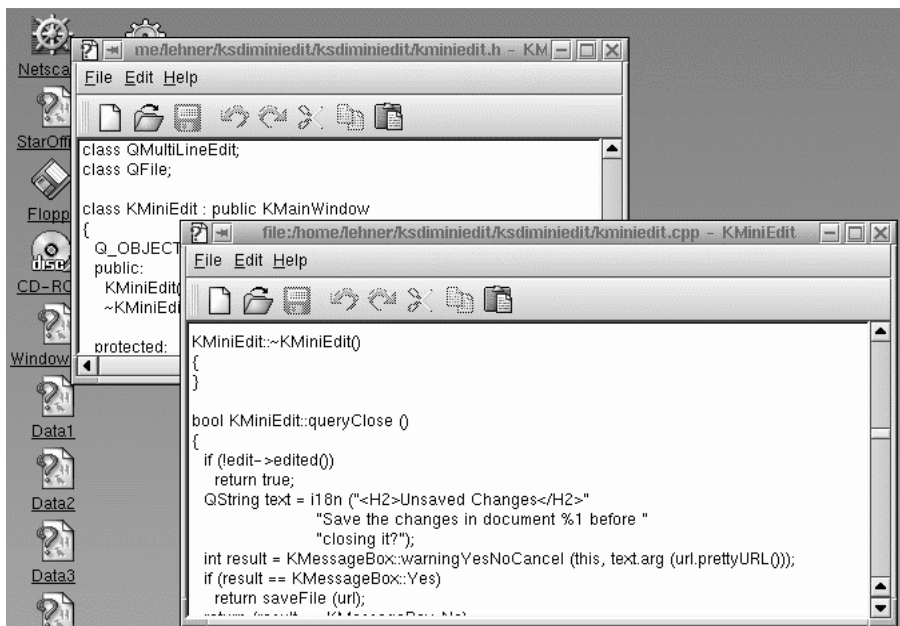


Abbildung 3-29 Zwei Dokumente in KMiniEdit mit Single Document Interface

Wir wollen an dieser Stelle nicht den ganzen Quellcode des Beispielprogramms abdrucken, da die Änderungen zum vorherigen Beispiel nur klein sind. Nur die wichtigsten Änderungen sind hier abgedruckt. Der gesamte Quelltext ist auf der CD enthalten, die dem Buch beiliegt.

An der Klassenstruktur ändert sich fast nichts, da wir nun einfach mehrere Hauptfenster-Objekte erzeugen und anzeigen lassen.

Im Hauptprogramm in *main.cpp* bleibt auch nahezu alles beim Alten. Der einzige Unterschied ist, dass wir nun auch mehrere Dateinamen auf der Kommandozeile zulassen und für jeden Dateinamen automatisch ein Hauptfenster öffnen. In die-

ses laden wir die Datei mit der Methode *KMiniEdit::loadFile*. Wenn das fehlschlug, löschen wir das Fenster gleich wieder. Sind anschließend keine Fenster geöffnet (entweder gab es keine Dateinamen auf der Kommandozeile oder das Laden der Dateien schlug fehl), erzeugen wir ein einziges, leeres Hauptfenster. Das Attribut *KMainWindow::memberList* enthält eine Liste der zur Zeit existierenden Hauptfenster. Wir benutzen dieses Attribut, um festzustellen, ob Fenster geöffnet sind.

```
int main(int argc, char *argv[])
{
    ... // Unverändert
    KCmdLineArgs *args = KCmdLineArgs::parsedArgs();

    KApplication a;

    if (args->count() > 0)
    {
        for (int i = 0; i < args->count(); i++)
        {
            KMiniEdit *window = new KMiniEdit();
            if (window->loadFile (args->url (i)))
                window->show();
            else
                delete window;
        }
    }

    if (!KMainWindow::memberList ||
        KMainWindow::memberList->isEmpty())
    {
        KMiniEdit *kminiedit = new KMiniEdit();
        kminiedit->show();
    }

    return a.exec();
}
```

Den Befehl FILE-QUIT nehmen wir nun aus dem Menü heraus. Das Programm wird automatisch beendet, sobald das letzte Fenster geschlossen wird; dafür sorgt die Klasse *KMainWindow* von ganz allein, ohne dass wir uns darum kümmern müssten. Stattdessen fügen wir den Befehl FILE-CLOSE ein, der das eine Fenster schließt. Diese Aktion wird direkt mit dem Slot *close()* des Hauptfensters verbunden; ein eigener Slot ist daher nicht nötig. (Man kann darüber streiten, ob man den Befehl nicht weiterhin QUIT nennen sollte, da der Anwender jedes Hauptfenster als eigenes Programm wahrnimmt.) Die anderen Befehle bleiben erhalten. Ein paar Befehle haben nun aber eine leicht unterschiedliche Wirkung: FILE-NEW öffnet jetzt ein neues Hauptfenster, indem es ein *KMiniEdit*-Objekt erzeugt. FILE-OPEN... öffnet ebenfalls ein neues Hauptfenster, aber nur, wenn das aktuelle

Hauptfenster nicht ein neues, noch leeres Dokument ist. In diesem Fall wird das aktuelle Hauptfenster benutzt. (Die übliche Vorgehensweise des Anwenders ist es, das Programm zunächst zu starten und anschließend ein Dokument zu öffnen. Er hätte somit zwei Fenster, das erste, leere vom Programmstart und das zweite der geöffneten Datei.) Ebenso arbeitet FILE-OPEN RECENT. Die entsprechenden Slot-Methoden sind nun einfacher geworden, die Abfrage nach ungesicherten Änderungen entfällt überall. (Diese wird nur noch von CLOSE aufgerufen, und zwar automatisch von *KMainWindow*.)

```
void KMiniEdit::fileNew()
{
    KMiniEdit *window = new KMiniEdit ();
    window->show();
}

void KMiniEdit::fileOpen()
{
    KURL newurl = KFileDialog::getOpenURL ();
    if (!newurl.isEmpty())
    {
        if (url.isEmpty() && !edit->edited())
            loadFile (newurl);
        else
        {
            KMiniEdit *window = new KMiniEdit ();
            window->show();
            window->loadFile (newurl);
        }
    }
}

void KMiniEdit::fileOpenRecent(const KURL &newurl)
{
    if (url.isEmpty() && !edit->edited())
        loadFile (newurl);
    else
    {
        KMiniEdit *window = new KMiniEdit ();
        window->show();
        window->loadFile (newurl);
    }
}
```

Eine Besonderheit betrifft die Liste der zuletzt geöffneten Dateien: Sie sollte bei allen Hauptfenstern identisch sein. (Das ist insbesondere deswegen wichtig, weil diese Liste auch in der Konfigurationsdatei abgelegt wird, und diese existiert nur einmal für das gesamte Programm. Daher muss die Liste eindeutig sein.)

Eine Lösung wäre es, das Aktionsobjekt vom Typ *KRecentFileAction* nur einmal anzulegen und in allen Hauptfenstern gemeinsam zu benutzen. Man legt in diesem Fall das Objekt praktischerweise in einer statischen Klassenvariablen ab. Der entsprechende Code sähe beispielsweise folgendermaßen aus (beachten Sie aber, dass wir diese Lösung nicht gewählt haben):

- in der Datei *kminiedit.h*:

```
class KMiniEdit : public QMultiLineEdit
{
    ...
    static KRecentFilesAction *recentAction;
};
```

- in der Datei *kminiedit.cpp*:

```
KRecentFilesAction *KMiniEdit::recentAction = 0;

KMiniEdit::KMiniEdit (...)
{
    ...
    if (recentAction == 0)
    {
        recentAction = KStdAction::openRecent (0, 0, 0);
        connect (recentAction, SIGNAL (urlSelected
                                      (const KURL &)),
                this, SLOT (fileOpenRecent (const KURL&)));
    }
    ...
    recentAction->plug (filePopup);
    ...
}
```

Das *recentAction*-Objekt würde auf diese Weise in mehrere Menüleisten eingebunden werden. Es ergeben sich aber zwei Probleme:

- Die Aktion ist nun nur mit dem ersten Hauptfenster verbunden. Wird dieses geschlossen, so geht das Signal *urlSelected* ins Leere, da es mit keinem Slot mehr verbunden ist. Man bräuchte daher ein zusätzliches Objekt, das so lange geöffnet ist, wie das Programm besteht (beispielsweise eine eigene, von *KApplication* abgeleitete Klasse).
- Als Vater-Objekt für *recentAction* kann nur ein Objekt eingetragen sein. Benutzt man daher eine XML-Datei, um die Menü- und Werkzeugleisten festzulegen, wird es nur dort aufgenommen, wo es auch in die *actionCollection*-Gruppe aufgenommen wurde.

Diese beiden Probleme sind nicht so leicht zu lösen. Daher wählen wir einen anderen Weg: Jedes Hauptfenster bekommt sein eigenes *recentAction*-Objekt. (Der Code zum Erzeugen bleibt also der gleiche wie bei unserem Programm mit nur

einem Hauptfenster.) Wir müssen nur darauf achten, dass eine Änderung der Liste in einem Hauptfenster auch gleich an alle anderen Hauptfenster berichtet wird. Hier kann uns die statische Klassenvariable *KMainWindow::memberList* helfen: Sie enthält immer eine Liste der aktuellen Hauptfenster-Objekte (alle Instanzen der von *KMainWindow* abgeleiteten Klassen). Diese Liste durchlaufen wir und aktualisieren in jedem Objekt das *recentAction*-Objekt. Wir benutzen dazu eine neue Methode *addURLtoRecent*, die diese Aufgabe erledigt. Der Code dazu sieht folgendermaßen aus:

```
void KMiniEdit::addURLtoRecent()
{
    recentAction->addURL (url);
    recentAction->saveEntries (KGlobal::config());
    if (memberList)
    {
        for (uint i = 0; i < memberList->count(); i++)
        {
            KMiniEdit *w = (KMiniEdit *) (memberList->at (i));
            w->recentAction->loadEntries (KGlobal::config());
        }
    }
}
```

Diese Methode ist nicht sehr effizient programmiert: Auch das eigene Hauptfenster, von dem die Änderung ausging, wird erneut geladen. Auch wäre es effizienter, die Einträge der Objekte direkt zu setzen, anstatt den Umweg über die Konfigurationsdatei zu gehen. Da diese Methode aber nur selten aufgerufen wird, braucht sie nicht besonders effizient zu sein.

KMiniEdit mit MDI

Im *Multi Document Interface* (MDI) stellt das Programm immer nur ein Hauptfenster dar. Innerhalb des Anzeigebereichs werden die einzelnen Dokumente in eigenen Fenstern (ohne Menü- oder Werkzeugleiste) dargestellt. Eines der Dokumente ist aktiv. Auf dieses Dokument beziehen sich die Befehle aus Menü- und Werkzeugleiste. Abbildung 3.30 zeigt unseren Editor mit Multi Document Interface.

Im Vergleich zu unserem Beispiel für ein Dokument ergibt sich eine ganze Reihe von Unterschieden. Zunächst ändert sich die Klassenstruktur. Unsere Hauptfensterklasse ist ähnlich wie vorher. Der Anzeigebereich ist nun aber nicht mehr ein *QMultiLineEdit*-Objekt, sondern ein *QWorkspace*-Objekt. Diese Klasse hat eine sehr einfache Schnittstelle, ist aber dennoch sehr mächtig. Alle Unter-Widgets, die in dieses *QWorkspace*-Objekt eingefügt werden, werden automatisch als eigene kleine Fenster dargestellt und verwaltet. Wir könnten theoretisch Widgets der Klasse *QMultiLineEdit* direkt verwenden, um sie in das *QWorkspace*-Objekt

einzufügen. Dabei tritt jedoch das Problem auf, dass wir nur noch ein Hauptfensterobjekt haben, aber mehrere Dokumente, deren Dateinamen gespeichert werden müssen.

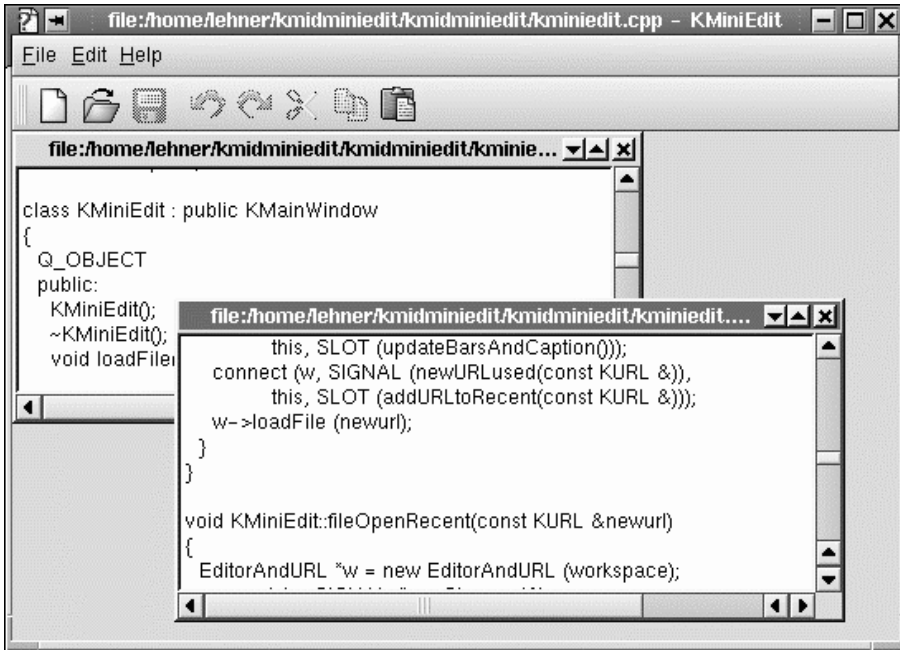


Abbildung 3-30 Zwei Dokumente in KMiniEdit mit Multi Document Interface

Es empfiehlt sich daher (und auch aus anderen Gründen), eine eigene Widget-Klasse zu implementieren, die als Dokumentenfenster benutzt werden. In unserem Fall leiten wir unsere Klasse namens *EditorAndURL* von der Klasse *QMultiLineEdit* ab. So ist für die Darstellung des Texts bereits gesorgt. Die Klasse *EditorAndURL* enthält aber unter anderem auch den Dateinamen in einem Attribut namens *url*.

Die Methoden zum Speichern und Laden von Dateien verschieben wir nun auch in unsere neue Klasse, denn dort passen sie am besten hin.

Unsere Klasse erhält noch weitere Attribute. Sie beschreiben den Zustand des Editorfensters, der sich auf die Aktivierbarkeit von Aktionen auswirken kann. *QMultiLineEdit* liefert Änderungen dieser Zustände nur per Signal (*copyAvailable*, *redoAvailable*, *undoAvailable*), bietet aber keine direkte Möglichkeit, die aktuellen Werte zu erfragen. Wenn der Anwender aber ein anderes Dokument aktiviert, müssen die Aktionen in Menü- und Werkzeugleiste anhand des Zustands dieses Dokuments an- oder ausgeschaltet werden. Unsere eigene Klasse fängt daher die Signale ab und merkt sich stattdessen diese Zustände in den drei Attributen

hasUndo, *hasRedo* und *hasCopy*. Diese Attribute (ebenso wie das *url*-Attribut) sind der Einfachheit halber als *public* deklariert, so dass wir sie sehr einfach abfragen können.

Die Header-Datei unserer Klasse *EditorAndURL* sieht nun so aus:

```
#ifndef EDITORANDURL_H
#define EDITORANDURL_H

#include <kurl.h>
#include <qwidget.h>
#include <qmultilineedit.h>

class QFile;

class EditorAndURL : public QMultiLineEdit {
    Q_OBJECT
public:
    EditorAndURL (QWidget *parent, const char *name = 0);
    ~EditorAndURL () {}

    KURL url;
    bool hasUndo;
    bool hasRedo;
    bool hasCopy;

signals:
    void barsChanged();
    void newURLused (const KURL &);

protected:
    void closeEvent (QCloseEvent *ev);

private slots:
    void updateUndo (bool b);
    void updateRedo (bool b);
    void updateCopy (bool b);

public:
    bool loadFile (KURL newurl);
    bool saveFile (KURL newurl);

private:
    void saveToLocalFile (QFile *);
};

#endif
```

Sie enthält insbesondere zwei Signale: *barsChanged* und *newURLused*. Das Signal *barsChanged* wird ausgesendet, wenn sich der Zustand des Editors geändert hat (sei es, dass nun REDO oder UNDO vorhanden sind oder nicht, dass COPY möglich ist oder dass das Dokument geändert wurde). Es wird dazu genutzt, im Hauptfenster die entsprechenden Aktionen zu setzen und die Statuszeile zu aktualisieren. Das Signal *newURLused* wird ausgesendet, wenn ein neuer Dateiname gelesen oder geschrieben wurde. Das Signal wird vom Hauptfenster aufgefangen, das daraufhin die *Recent*-Liste entsprechend aktualisiert.

Die Quellcode-Datei zur Klasse *EditorAndURL* enthält nur eine besondere Methode, auf die wir hier näher eingehen. Die anderen Methoden sind weitgehend selbsterklärend, deshalb verzichten wir auf den Abdruck. Der gesamte Quellcode befindet sich auf der CD, die dem Buch beiliegt.

Ein Editor-Fenster, das im *QWorkspace*-Objekt liegt, kann durch Anklicken des X-Buttons oder die Auswahl von FILE-CLOSE geschlossen werden. Dabei wird ein Close-Event an dieses Widget geschickt. Nun muss abgefragt werden, ob im Editor noch ungespeicherte Veränderungen enthalten sind. Aber wo können wir diese Abfrage durchführen? Die Methode *queryClose* gibt es nur in der Klasse *KMainWindow*. Wir müssen daher selbst den Close-Event abfangen. Dazu überschreiben wir die virtuelle Methode *closeEvent* (*QCloseEvent **). In dieser Methode führen wir den Test und die Abfrage an den Benutzer aus, genau wie wir es vorher in der Methode *queryClose* im Hauptfenster getan haben. Wollen wir das Fenster wirklich schließen, so setzen wir das übergebene Event-Objekt auf *accept*; wollen wir den Close-Event dagegen abschmettern (wenn der Anwender CANCEL gewählt hat oder das Abspeichern fehlschlug), so setzen wir es auf *ignore*. Der Code dieser Methode sieht also folgendermaßen aus:

```
void EditorAndURL::closeEvent (QCloseEvent *ev)
{
    ev->accept();
    if (edited())
    {
        QString text =
            i18n("<H2>Unsaved Changes</H2>"
                "Save the changes in document %1 before "
                "closing it?");
        int result = KMessageBox::warningYesNoCancel
            (this, text.arg(url.prettyURL()));
        if (result == KMessageBox::Yes)
            if (!saveFile(url))
                ev->ignore();
        if (result == KMessageBox::Cancel)
            ev->ignore();
    }
}
```

Betrachten wir nun die Hauptfensterklasse. Auch hier gibt es eine ganze Reihe von Änderungen. Werfen wir zunächst einen Blick auf die Header-Datei der Klasse *KMiniEdit*:

```
#ifndef KMINIEDIT_H
#define KMINIEDIT_H

#include <qmultilinedit.h>
#include <kurl.h>
#include <kmainwindow.h>

#include "editorandurl.h"

class KAction;
class KRecentFilesAction;
class KSelectAction;
class QFile;
class QWorkspace;

class KMiniEdit : public KMainWindow
{
    Q_OBJECT
public:
    KMiniEdit();
    ~KMiniEdit();
    void loadFile(const KURL &url);

protected:
    bool queryClose();

private slots:
    void fileNew();
    void fileOpen();
    void fileOpenRecent(const KURL&);
    void fileSave();
    void fileSaveAs();
    void fileClose()
        {if (active()) active()->close();}
    void fileQuit();

    void editUndo()
        {if (active()) active()->undo();}
    void editRedo()
        {if (active()) active()->redo();}
    void editCut()
        {if (active()) active()->cut();}
    void editCopy()
        {if (active()) active()->copy();}
```



```

void editPaste()
    {if (active()) active()->paste();}
void editSelectAll()
    {if (active()) active()->selectAll();}

void checkClipboard();
void updateBarsAndCaption();
void activateWindow (int);
void addURLtoRecent (const KURL&);

private:
    EditorAndURL *active ();

    KAction *saveAction, *closeAction,
            *undoAction, *redoAction,
            *cutAction, *copyAction, *pasteAction;
    KRecentFilesAction *recentAction;
    KSelectAction *windowSelectAction;

    QWorkspace *workspace;
};

#endif

```

Es fällt zunächst auf, dass viel mehr Slots deklariert sind. Jeder Menübefehl hat nun einen eigenen Slot bekommen. Vorher war das nicht nötig, denn viele Aktionen konnten direkt mit dem entsprechenden Slot der Klasse *QMultiLineEdit* verbunden werden. Nun haben wir aber mehrere Editor-Objekte gleichzeitig geöffnet. Nur eines von ihnen soll aber aufgerufen werden, nämlich das Objekt des aktiven Dokuments. Da es unnötiger Aufwand ist, beim Wechsel des aktiven Dokuments alle Signal-Slot-Verbindungen aufzuheben und neu aufzubauen, bilden wir hier eigene Slots für diese Aktionen. Die Slots rufen dann die Methode des aktiven Elements auf. Dazu benutzen sie die selbst geschriebene Hilfsmethode *active*, die einen Zeiger auf das gerade aktive Dokument liefert. (Sie benutzt dazu die Methode *QWorkspace::activeWindow*, die das Widget zurückliefert. Sie führt einen Type-Cast mit diesem Zeiger auf den Klassentyp *EditorAndURL* durch, damit auf die Methode – insbesondere die von *QMultiLineEdit* – zugegriffen werden kann. In unserem Fall ist uns ja bekannt, dass wir nur Widgets der Klasse *EditorAndURL* im *QWorkspace* abgelegt haben.) Wichtig ist es noch, darauf zu achten, dass *active* auch einen Null-Zeiger zurückliefern kann, wenn kein Dokument aktiv ist. (Das passiert genau dann, wenn alle Dokumente geschlossen wurden.)

Werfen wir noch einen Blick auf den Quellcode der wichtigsten Methoden in *KMiniEdit*. Zunächst sehen Sie hier den Code für den Konstruktor:

```

KMiniEdit::KMiniEdit() : KMainWindow()
{
    workspace = new QWorkspace (this);
    setCentralWidget (workspace);
    connect (workspace,
            SIGNAL (windowActivated (QWidget *)),
            this, SLOT (updateBarsAndCaption()));

    // Erzeugen der Aktionen für das File-Menü
    ...
    closeAction = KStdAction::close
        (this, SLOT (fileClose()), actionCollection());
    ...

    // Erzeugen der Aktionen für das Edit-Menü
    undoAction = KStdAction::undo
        (this, SLOT (editUndo()), actionCollection());
    redoAction = KStdAction::redo
        (this, SLOT (editRedo()), actionCollection());
    cutAction = KStdAction::cut
        (this, SLOT (editCut()), actionCollection());
    copyAction = KStdAction::copy
        (this, SLOT (editCopy()), actionCollection());
    KStdAction::selectAll
        (this, SLOT (editSelectAll()), actionCollection());

    pasteAction = KStdAction::paste
        (this, SLOT (editPaste()), actionCollection());
    connect (kapp->clipboard(), SIGNAL (dataChanged()),
            this, SLOT (checkClipboard()));
    checkClipboard ();

    // Erzeugen der Aktionen für das Window-Menü
    new KAction ("&Tile", "tile",
        0, workspace, SLOT (tile()),
        actionCollection(), "window_tile");
    new KAction ("&Cascade", "cascade",
        0, workspace, SLOT (cascade()),
        actionCollection(), "window_cascade");
    windowSelectAction =
        new KSelectAction ("&Windows", 0, 0, 0, 0,
            actionCollection(),
            "window_windowlist");
    connect (windowSelectAction, SIGNAL (activated (int)),
        this, SLOT (activateWindow (int)));

    createGUI();
    updateBarsAndCaption();
}

```

Im Konstruktor werden die Aktionen aus dem EDIT-Menü nun mit Slots des Hauptfensters verbunden und nicht mehr mit Slots eines *QMultiLineEdit*-Objekts. Außerdem werden drei weitere Aktionen angelegt, dieses Mal nicht aus *KStdAction*, sondern selbst definierte. Die Aktion *TILE* verteilt die Dokumente so im Anzeigebereich, dass der gesamte Platz möglichst gleichmäßig aufgeteilt wird. Die Aktion *CASCADE* dagegen macht alle Dokumentenfenster etwa gleich groß und ordnet sie so an, dass sie sich überlappen. Beide Aktionen sind mit den gleichnamigen Slots des *QWorkspace*-Objekts verbunden. Eine weitere Aktion, *WINDOWS*, enthält immer die aktuelle Liste aller Dokumente. Diese Aktion kann benutzt werden, um über die Menüleiste zwischen den Dokumenten zu wechseln. Nur so kann gewährleistet werden, dass das Programm auch vollständig über die Tastatur bedient werden kann. Der Klassentyp dieser Aktion ist *KSelectAction*. Wird ein Eintrag ausgewählt, so wird das Signal *KSelectAction::activated(int)* gesendet. Dieses Signal fangen wir im Slot *KMiniEdit::activateWindow(int)* auf und aktivieren dort das entsprechende Fenster:

```
void KMiniEdit::activateWindow (int i)
{
    QWidget *w = workspace->windowList().at (i);
    if (w)
        w->setFocus();
}
```

Aktualisiert wird die Liste der Fenster in der Slot-Methode *updateBarsAndCaption*:

```
void KMiniEdit::updateBarsAndCaption()
{
    QWidgetList l = workspace->windowList();
    QStringList urls;
    for (uint i = 0; i < l.count(); i++)
    {
        QString urlname;
        EditorAndURL *w = (EditorAndURL *) (l.at(i));
        if (w->url.isEmpty())
            urlname = i18n ("New Document");
        else
            urlname = w->url.prettyURL();
        if (w->edited())
            urlname += " *";
        urls << urlname;
    }
    windowSelectAction->setItems (urls);
    windowSelectAction->setCurrentItem
        (l.findRef (active()));

    if (active())
```

```

{
    bool modified = active()->edited();
    if (active()->url.isEmpty())
        setCaption (i18n ("New Document"), modified);
    else
        setCaption (active()->url.prettyURL(), modified);
    saveAction->setEnabled (modified);
    closeAction->setEnabled (true);

    undoAction->setEnabled (active()->hasUndo);
    redoAction->setEnabled (active()->hasRedo);
    cutAction->setEnabled (active()->hasCopy);
    copyAction->setEnabled (active()->hasCopy);
}
else
{
    setCaption (i18n ("No Document"), false);
    saveAction->setEnabled (false);
    closeAction->setEnabled (false);

    undoAction->setEnabled (false);
    redoAction->setEnabled (false);
    cutAction->setEnabled (false);
    copyAction->setEnabled (false);
}
}

```

Wir greifen dabei mit der Methode *QWorkspace::windowList* auf die Liste der Dokumentenfenster zurück. Zu jedem Dokument legen wir den Dateinamen in der Aktion ab und schreiben einen Stern (*) dahinter, wenn das Dokument noch ungesicherte Änderungen enthält. Das zur Zeit aktive Dokument wird in der Aktion auf den Zustand »ausgewählt« gesetzt, so dass es mit einem Häkchen versehen wird.

In der Methode *updateBarsAndCaption* wird außerdem der Text der Titelzeile des Hauptfensters aktualisiert, und die Aktionen werden entsprechend des aktiven Dokuments an- oder abgeschaltet. Dieser Slot wird immer aufgerufen, wenn eines der Dokumente eine Änderung des Zustands meldet.

Achtung: Aufgrund eines Fehlers in der Klasse *QWorkspace* wird zur Zeit das Signal *QWorkspace::windowActivated* nicht ausgesendet, wenn das letzte Dokument geschlossen wird. Daher wird auch *updateBarsAndCaption* nicht aufgerufen und somit die Menü- und Werkzeugleiste nicht aktualisiert. Auch die Window-Liste wird nicht auf den neuesten (leeren) Stand gebracht. Diesen Bug zu umgehen ist sehr aufwendig, weshalb wir es hier nicht machen. Wahrscheinlich wird er in einer der nächsten Qt-Versionen behoben sein.

Die Methoden zum Öffnen einer Datei oder eines neuen Dokuments erzeugen einfach ein neues Objekt der Klasse *EditorAndURL*. Es reicht aus, als Vater-Widget das *QWorkspace*-Objekt anzugeben. Außerdem muss die Methode *show* aufgerufen werden (das geschieht im Konstruktor von *EditorAndURL*), und die Signale des Objekts müssen entsprechend verbunden werden. Der Code sieht dann folgendermaßen aus:

```
void KMiniEdit::fileNew()
{
    EditorAndURL *w = new EditorAndURL (workspace);
    connect (w, SIGNAL (barsChanged()),
            this, SLOT (updateBarsAndCaption()));
    connect (w, SIGNAL (newURLused(const KURL &)),
            this, SLOT (addURLtoRecent(const KURL &)));
}

void KMiniEdit::fileOpen()
{
    KURL newurl = KFileDialog::getOpenURL ();
    if (!newurl.isEmpty())
    {
        EditorAndURL *w = new EditorAndURL (workspace);
        connect (w, SIGNAL (barsChanged()),
                this, SLOT (updateBarsAndCaption()));
        connect (w, SIGNAL (newURLused(const KURL &)),
                this, SLOT (addURLtoRecent(const KURL &)));
        w->loadFile (newurl);
    }
}

void KMiniEdit::fileOpenRecent(const KURL &newurl)
{
    EditorAndURL *w = new EditorAndURL (workspace);
    connect (w, SIGNAL (barsChanged()),
            this, SLOT (updateBarsAndCaption()));
    connect (w, SIGNAL (newURLused(const KURL &)),
            this, SLOT (addURLtoRecent(const KURL &)));
    w->loadFile (newurl);
}
```

Schließlich ist die Methode *queryClose* nun anders realisiert. Sie wird aufgerufen, wenn das Hauptfenster geschlossen werden soll. Sie arbeitet dabei die Liste der Dokumente ab, die im *QWorkspace*-Objekt enthalten sind. Sie versucht, jedes Dokument durch Aufruf von *close* zu schließen. Weigert sich eines der Dokumente (weil es nicht gesicherte Änderungen gab und der Anwender CANCEL gewählt hat oder das Speichern fehlschlug), so bricht *queryClose* ab. Das Hauptfenster bleibt in diesem Fall offen.

```

bool KMiniEdit::queryClose ()
{
    while (!workspace->windowList().isEmpty())
    {
        QWidget *w = workspace->windowList().first();
        w->close(true);
        if (workspace->windowList().containsRef (w))
            return false;
    }
    return true;
}

```

Wie Sie sehen, ist eine Implementierung eines MDI-Konzepts komplizierter und potenziell fehleranfälliger. Auch wenn dieses Konzept moderner aussieht, sollten Sie es wirklich nur dann benutzen, wenn es echte Vorteile bringt – also nur dann, wenn Ihr Programm viele und eher kleine Dokumentenfenster anzeigen soll oder wenn es einen Zusammenhang zwischen den Dokumenten gibt, so dass es besser ist, diese Dokumente übersichtlich nebeneinander zu sehen.

Wenn Sie dieses Beispielprogramm für ein eigenes Projekt anpassen wollen, so müssen Sie vor allem die Klasse *EditorAndURL* entsprechend implementieren. Achten Sie dabei am besten schon darauf, dass die für die Aktionen wichtigen Zustände (*hasUndo*, *hasRedo* und *hasCopy* in unserem Fall) nicht nur über Signale mitgeteilt werden, sondern jederzeit auch direkt abgefragt werden können (wichtig beim Wechsel des aktiven Dokuments). Außerdem müssen Sie alle Stellen in *KMiniEdit* ändern, an denen direkt auf die Attribute von *EditorAndURL* zugegriffen wird, also insbesondere in *updateBarsAndCaption*.

Auch *KDevelop* bietet Ihnen bereits fertige Programmgerüste für Programme mit MDI-Konzept. Verwenden Sie möglichst dieses Gerüst, da Ihnen dann viele potenzielle Fehlerquellen direkt erspart bleiben (siehe Kapitel 5.5, *KDevelop*).

3.5.8 Das Document-View-Konzept

Bei unseren bisherigen Programmen war das Dokument direkt in der Widget-Klasse abgelegt, die zum Anzeigen auf dem Bildschirm benutzt wurde. In unserem Fall war das die Klasse *QMultiLineEdit*, die gleichzeitig den Text des Dokuments speichert und ihn auf dem Bildschirm anzeigt.

Moderne Programme – insbesondere wenn sie eine bestimmte Komplexität erreichen – sind im Gegensatz dazu oft nach dem *Document-View-Konzept* entworfen worden. In diesem Fall gibt es eine Klasse, die die Daten des Dokuments enthält (*Document*), und eine andere Klasse, die ein Dokument auf dem Bildschirm anzeigt (*View*). Einem Document-Objekt sind dazu in der Regel ein oder mehrere View-Objekte zugeordnet. Dieses Konzept bietet eine Reihe von Vorteilen:

- Verschiedene View-Objekte können verschiedene Ausschnitte des gleichen Document-Objekts anzeigen. So kann ein View-Objekt beispielsweise den Anfang einer langen Datei anzeigen, und direkt daneben zeigt ein anderes View-Objekt das Ende an. Änderungen in einem View-Objekt verändern das Document-Objekt, was wiederum ein Neuzeichnen des anderen View-Objekts bewirkt. Beide View-Objekte zeigen so immer den aktuellen Zustand an.
- Die View-Objekte können nicht nur verschiedene Ausschnitte, sondern auch verschiedene Darstellungen des Dokuments zeigen. Eines zeigt zum Beispiel die Gesamtübersicht, während ein anderes einen vergrößerten Ausschnitt zeigt. Oder eines zeigt eine Zahlenreihe, während ein anderes das dazugehörige Diagramm darstellt. Änderungen am Dokument haben immer Auswirkungen auf alle View-Objekte.
- Zwischen der Document-Klasse und der View-Klasse ist eine eindeutige Arbeitsteilung definiert: Die Document-Klasse enthält den Code zum Laden und Speichern der Daten. Die einzelnen Klassen sind dadurch kleiner und übersichtlicher.
- Die Speicherung des Dokuments und die Anzeige auf dem Bildschirm können nun unabhängig voneinander realisiert werden. Ein Portieren auf eine andere grafische Plattform ist dadurch leichter, da die Document-Klasse weit gehend übernommen werden kann. Nur die View-Klasse muss angepasst oder neu geschrieben werden. Umgekehrt ist auch das Umsteigen auf ein anderes Dokumentenformat einfacher: In diesem Fall kann die View-Klasse weit gehend erhalten bleiben, nur die Realisierung der Document-Klasse muss geändert werden.

Abbildung 3.31 verdeutlicht den Zusammenhang.

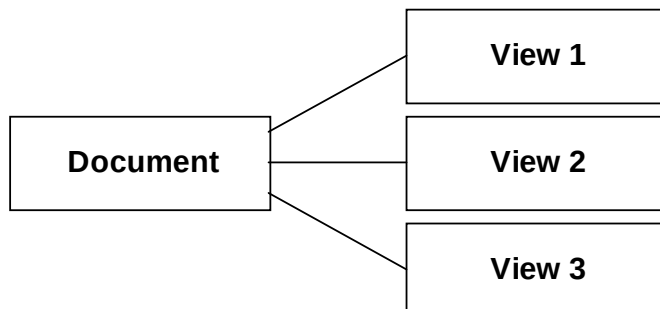


Abbildung 3-31 Zuordnung mehrerer View-Objekte zu einem Document-Objekt

In der Praxis sind die Klassen in KDE- oder Qt-Programmen etwa folgendermaßen aufgebaut:

- Die *Document-Klasse* ist meist von *QObject* abgeleitet. Dadurch kann sie Signale und Slots definieren. *QWidget* ist als Vaterklasse nicht notwendig, da die Document-Klasse nichts darstellt. Die Document-Klasse verwaltet in der Regel eine Liste der angeschlossenen View-Objekte. (Diese Liste ist aber nicht unbedingt nötig.) Die View-Objekte melden sich dazu bei der Document-Klasse an und wieder ab. Wird das letzte View-Objekt vom Document-Objekt getrennt, kann in der Regel das Document-Objekt freigegeben werden (natürlich nicht ohne die geänderten Daten – eventuell nach Rückfrage – zu sichern). Die Document-Klasse besitzt in der Regel ein parameterloses Signal namens *dataChanged* (oder so ähnlich), mit dem sie den angeschlossenen View-Objekten mitteilt, dass sich das Dokument geändert hat und daher alle View-Objekte ihre Anzeige erneuern müssen. Der Klassenname lautet üblicherweise *MyAppDoc*, wobei *MyApp* der Name des Programms ist.
- Die *View-Klasse* ist in der Regel von *QWidget* oder einer Unterklasse abgeleitet. Sie speichert in einem Attribut einen Zeiger auf das Document-Objekt, dem sie zugeordnet ist. Dieser Zeiger wird dem View-Objekt meist im Konstruktor übergeben und bleibt während der Lebensdauer des View-Objekts konstant. Verwechseln Sie aber nicht diesen Zeiger auf das Document-Objekt mit dem Vater-Widget der View-Klasse. Dieses Vater-Widget ist in der Regel eine Hauptfensterklasse (bei einem Single Document Interface) oder eine Klasse wie *QWorkspace* (bei einem Multi Document Interface). Das View-Objekt verbindet sich mit dem Signal *dataChanged* des Document-Objekts, um von Änderungen unterrichtet zu werden. Will ein View-Objekt eine Änderung am Dokument vornehmen (aufgrund einer Aktion des Anwenders), so meldet es diesen Änderungswunsch über eine Methode der Document-Klasse. In einem Texteditor würde das View-Objekt bei einem Tastendruck dem Document-Objekt beispielsweise mitteilen, dass ein einzelner Buchstabe an einer bestimmten Stelle im Dokument eingefügt werden muss. Die eigene Darstellung ändert das View-Objekt aber (noch) nicht, es wird ja unmittelbar darauf vom Document-Objekt danach aufmerksam gemacht, dass sich das Dokument geändert hat. Erst dann holt es die neuen Daten aus dem Dokument und zeigt sie an.

Abbildung 3.32 verdeutlicht die Reihenfolge der Aufrufe, die bei einer Änderung des Dokuments entstehen: Die Änderung wird (in der Regel) von einem View-Objekt ausgelöst, woraufhin das Document-Objekt das Signal *dataChanged* aussendet. Daraufhin zeichnen alle View-Objekte ihren Inhalt neu und holen dazu die aktuellsten Daten vom Document-Objekt.

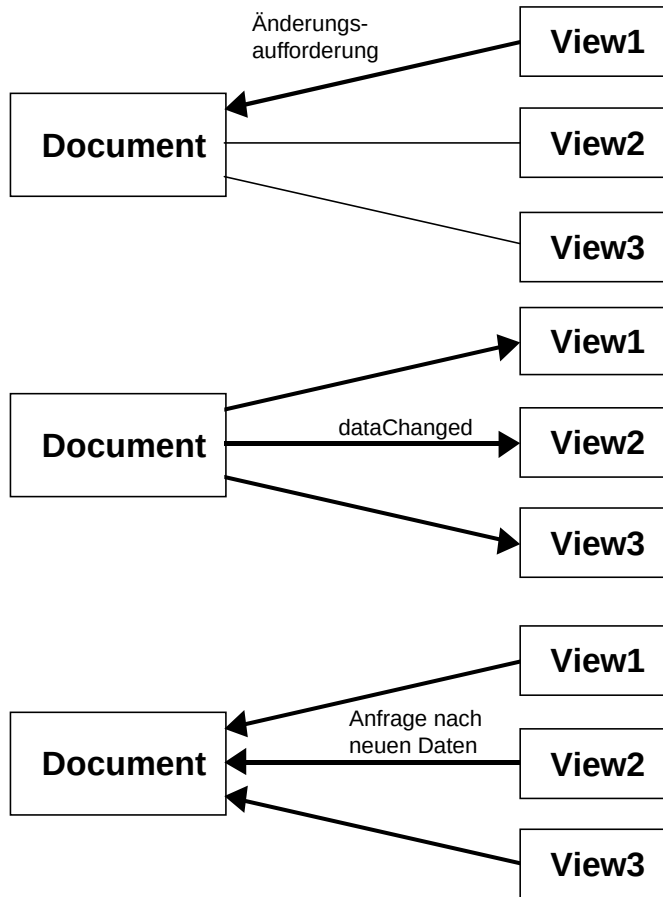


Abbildung 3-32 Aufrufe, die von einer Änderung ausgelöst werden

Viele der bereits existierenden Unterklassen von *QWidget* sind nicht oder nur schlecht als Basisklasse für eine View-Klasse geeignet, da sie meist selbst die gesamten angezeigten Daten speichern. So ist *QMultiLineEdit* als View-Klasse in einem Texteditor völlig ungeeignet, da bei jeder Änderung am Dokument (also jedem getippten Buchstaben) der gesamte Text des Dokuments in das *QMultiLineEdit*-Objekt kopiert werden müsste, was insbesondere bei großen Texten sehr ineffizient ist. Stattdessen schreibt man diese Klasse oft selbst. In der Methode *paintEvent*, die für das Zeichnen zuständig ist, holt sich das View-Objekt dann die zu zeichnenden Daten (und nur diese) direkt vom Document-Objekt. Nähere Informationen hierzu finden Sie in Kapitel 4.4, *Entwurf eigener Widget-Klassen*.

Programme, die auf dem Document-View-Konzept beruhen, stellen in der Regel mehr als ein Fenster dar. Sie sind also meist nach dem Grundgerüst für eine SDI-

oder eine MDI-Applikation aufgebaut. Das SDI-Konzept hat hierbei den Nachteil, dass manchmal nicht so leicht klar wird, dass zwei View-Fenster dem gleichen Dokument zugeordnet sind. Kann das Programm aber auch noch mehrere Dokumente gleichzeitig öffnen, so ist das MDI-Konzept oft verwirrender, da nicht klar ist, welche View-Fenster zu welchen Document-Objekten gehören. (Bedenken Sie, dass der Anwender die Document-Objekte in der Regel nie zu Gesicht bekommt. Er sieht nur die dazugehörenden View-Fenster.) Hier ist oft eine Mischung aus SDI und MDI am günstigsten: Jedes Dokument erhält ein eigenes Hauptfenster, die View-Fenster zu einem Dokument werden dagegen alle innerhalb dieses Hauptfensters dargestellt. Hierbei ist *QWorkspace* nicht immer die beste Wahl, da die Platzierung der Fenster oft mühsam ist. Besser ist beispielsweise ein *QSplitter*-Objekt, das die View-Fenster neben- oder untereinander anordnet und keinen Platz verschwendet. (Nähere Informationen zu *QSplitter* finden Sie in Kapitel 3.7.7, *Verwaltungselemente*).

Wie neue View-Objekte zu einem Dokument erzeugt werden, kann je nach Applikation sehr unterschiedlich sein. Eine Möglichkeit ist, im Menü einen weiteren Menüpunkt FILE-NEW VIEW anzubieten, der vom aktuellen Dokument (also dem Dokument, das das aktuelle View-Fenster anzeigt) ein weiteres View-Objekt erzeugt. Es kann auch sinnvoll sein, nur eine feste Anzahl von View-Fenstern vorzusehen (zum Beispiel wenn sie das Dokument auf verschiedene Arten darstellen). Diese View-Objekte lassen sich dann zum Beispiel über eine *KToggleAction*-Option ein- und ausschalten. Diese Aktionen sind üblicherweise im Menüpunkt VIEW untergebracht.

Es gibt keine speziellen Klassen in der KDE- oder Qt-Bibliothek, die die Entwicklung einer Applikation nach dem Document-View-Prinzip unterstützen würden. Sowohl *KDevelop* als auch *KAppTemplate* erzeugen aber auf Wunsch das Grundgerüst einer Applikation nach diesem Prinzip, so dass Sie sich an diese Vorgaben halten können. (Siehe Kapitel 5.3, *kapptemplate* und *kappgen*, und Kapitel 5.5, *KDevelop*.)

3.5.9 Das Hauptfenster für reine Qt-Programme

Alle Programme, die wir bisher in diesem Kapitel 3.5, *Das Hauptfenster*, kennen gelernt haben, benutzen zum Teil sehr intensiv Klassen aus der KDE-Bibliothek. Wenn Sie auf diese Bibliothek verzichten wollen (oder müssen), so müssen Sie auf ähnliche Klassen aus der Qt-Bibliothek zurückgreifen. (Oftmals sind die Klassen der KDE-Bibliothek nur abgeleitete Klassen der Qt-Bibliothek, die um einige Methoden erweitert wurden.) Tabelle 3.6 zeigt die Alternativklassen, die Sie benutzen können:

KDE-Klasse	Alternativklasse der Qt-Bibliothek
<i>KMainWindow</i>	<i>QMainWindow</i>
<i>KAction</i>	<i>QAction</i>
<i>KMenuBar</i>	<i>QMenuBar</i>
<i>KToolBar</i>	<i>QToolBar</i>
<i>KStatusBar</i>	<i>QStatusBar</i>
<i>KStdAction</i>	Erzeugen der Aktionen von Hand

Tabelle 3-6 Klassen für das Hauptfenster in reinen Qt-Applikationen

Die Umsetzung eines KDE-Programms geht meist nicht ohne zusätzliche Änderungen vonstatten. Deshalb folgen hier einige Bemerkungen zu den oben genannten Klassen:

- Die Klasse *QMainWindow* ist in vielen Punkten ganz ähnlich zu benutzen wie *KMainWindow*. Leiten Sie also Ihre eigene Hauptfensterklasse von *QMainWindow* ab. Die Methoden *menuBar* und *statusBar* liefern Ihnen wie gehabt die Menüleiste bzw. die Statuszeile. Bei der Werkzeugleiste müssen Sie allerdings anders vorgehen: Erstellen Sie zunächst selbst ein Objekt der Klasse *QToolBar*, füllen Sie es mit den gewünschten Buttons bzw. Aktionen, und fügen Sie anschließend das Objekt mit *QMainWindow::addToolBar* in das Hauptfenster ein. Weiterhin fehlt in *QMainWindow* die virtuelle Methode *queryClose*, in der wir die Sicherheitsabfrage implementiert hatten. Überschreiben Sie stattdessen die virtuelle Methode *closeEvent*, so wie es auch in der Klasse *EditorAndURL* im Abschnitt *KMiniEdit mit MDI* in Kapitel 3.5.7, *Applikation für mehrere Dokumente*, gemacht wurde.
- Die Klasse *QAction* wird zwar ähnlich wie *KAction* verwendet, ist aber etwas anders zu benutzen. *QAction* vereint dabei die Klassen *KAction* und *KToggleAction*. Benutzen Sie die Methode *QAction::setToggleAction*, um eine ein- und ausschaltbare Aktion zu erzeugen. Für die anderen von *KAction* abgeleiteten Klassen gibt es keine Entsprechung in der Qt-Bibliothek. Deren Funktionalität müssen Sie selbst programmieren. Oftmals reicht es aber auch, wenn Sie andere Widget-Klassen in der Werkzeugleiste *QToolBar* benutzen (z.B. *QCombo* oder *QLineEdit*).
- In reinen Qt-Programmen werden Icons nicht in Standardverzeichnissen gesucht. Stattdessen wird an vielen Stellen die Klasse *QIconSet* benutzt (z.B. im Konstruktor von *QAction*). Diese Klasse generiert aus einem *QPixmap*-Objekt einen ganzen Satz von Icons in verschiedenen Größen. (Nähere Informationen zur Klasse *QPixmap* finden Sie in Kapitel 4.3.2, *QPixmap*). Um beispielsweise eine Datei namens *open.png* aus dem aktuellen Verzeichnis zu benutzen, können Sie folgende Zeile schreiben:

```
QIconSet openIcon (QPixmap ("open.png"));
```

- Qt besitzt keine Klasse wie *KStdAction*, die die häufig benutzten Aktionen bereits definiert hat. Alle Aktionen müssen Sie daher selbst definieren. Sie können sich dabei an den Menübezeichnungen orientieren, die *KStdAction* benutzt, da sie allgemein üblich sind.
- Es gibt keine Möglichkeit, die Anordnung der Menüpunkte aus einer XML-Datei zu laden, wie es die Methode *KMainWindow::createGUI* macht. Sie müssen also den umständlicheren und unflexibleren Weg wählen, dass Sie die Aktionen von Hand in die Menü- und Werkzeugleisten eintragen. Die Methode von *QAction* lautet dabei *QAction::addTo* (im Gegensatz zu *KAction::plug*).
- Das Hilfemenü wird in reinen Qt-Applikationen nicht automatisch generiert. Sie müssen es also von Hand erzeugen. Auch das Anzeigen von Hilfedateien müssen Sie selbst implementieren. Hier ist oftmals die Klasse *QTextBrowser* nützlich, die Dateien mit Querverweisen im RichText-Format (einer Untermenge von HTML) anzeigen kann.
- Die Klasse *QStatusBar* besitzt keine Methoden wie *insertItem*. Um einzelne Textfelder in die Statuszeile einzufügen, können Sie stattdessen *QLabel*-Objekte benutzen.

3.6 Anordnung von GUI-Elementen in einem Fenster

Wie wir bereits kurz in Kapitel 3.2 erwähnt haben, werden die meisten Dialoge und Fenster in Qt und KDE durch ein Fenster der Klasse *QWidget* (oder einer abgeleiteten Klasse) erzeugt. Dieses Widget enthält Unter-Widgets, die verschiedene GUI-Elemente darstellen. Diese Unter-Widgets werden im Fenster positioniert, so dass alle Unter-Widgets vollständig sichtbar sind und die Anordnung einen möglichst guten Überblick über die Funktionalität bietet. Eine geschickte Anordnung der Unter-Widgets – ein so genanntes Layout – ist oftmals nicht leicht als Quelltext zu erstellen.

In Kapitel 3.6.1, *Anordnung auf feste Koordinaten*, und Kapitel 3.6.2, *Anordnung der Widgets im resize-Event*, werden zwei aufwendige und unzureichende Konzepte vorgestellt – der Vollständigkeit halber. Sie können diese Kapitel auch überspringen, um direkt in Kapitel 3.6.3 mit den Klassen *QHBoxLayout*, *QVBoxLayout* und *QGridLayout* ein einfaches Konzept kennen zu lernen. Ein mächtigeres Konzept wird in Kapitel 3.6.4, *Die Layout-Klassen QBoxLayout und QGridLayout*, vorgestellt. Allerdings ist es sehr viel schwieriger zu handhaben.

Spätestens seit dem Programm *Qt Designer* von Trolltech steht dem Qt- und KDE-Entwickler aber auch ein sehr mächtiges Werkzeug zur Verfügung, mit dem sich Dialoge mit gutem Layout mit ein paar Mausklicks zusammenstellen lassen. Damit kann die Entwicklungszeit zum Teil gravierend verkürzt werden. Insbesondere um die Layout-Konzepte kennen zu lernen, sollten Sie mit *Qt Designer* experimentieren. Weitere Informationen finden Sie in Kapitel 5.4, *Qt Designer*.

3.6.1 Anordnung auf feste Koordinaten

Wir wiederholen hier noch einmal das Beispiel aus Kapitel 3.2: Wir erzeugen ein Fenster, das Meldungen in einem Rechteck anzeigt und das zwei Buttons hat, die unterhalb des Rechtecks angeordnet werden. Der Code dafür sieht folgendermaßen aus:

```
#include <qapplication.h>
#include <qwidget.h>
#include <qpushbutton.h>
#include <qmultilineedit.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    // Toplevel-Widget anlegen, Größe setzen
    QWidget *messageWindow = new QWidget ();
    app.setMainWidget (messageWindow);
    messageWindow->resizeSize (220, 150);

    // Drei Unter-Widgets anlegen, Größe setzen
    QMultiLineEdit *messages =
        new QMultiLineEdit (messageWindow);
    messages->setGeometry (10, 10, 200, 100);
    QPushButton *clear =
        new QPushButton ("Clear", messageWindow);
    clear->setGeometry (10, 120, 95, 20);
    QPushButton *hide =
        new QPushButton ("Hide", messageWindow);
    hide->setGeometry (115, 120, 95, 20);

    // noch ein paar Eigenschaften festlegen
    messageWindow->setCaption ("einfacher Dialog");
    messages->setReadOnly (true);
    // ersten Eintrag in das Fenster einfügen
    messages->append ("Initialisierung abgeschlossen\n");

    // Toplevel-Widget anzeigen
    messageWindow->show ();

    return app.exec ();
}
```

Wir wollen dieses Programm noch einmal Schritt für Schritt durchgehen. Unser Toplevel-Widget ist von der Klasse *QWidget*, hat also keinen besonderen Inhalt. Es wird unser Hauptfenster werden. Dieses Widget erzeugen wir mit *new* auf dem

Heap. Angezeigt wird es noch nicht, da Toplevel-Widgets standardmäßig den Modus »versteckt« besitzen. Erst nach dem Aufruf von *show* in der letzten Zeile wird das Fenster mit seinen drei Unter-Widgets auf den Bildschirm gezeichnet.

Mit *resize* legen wir die neue Größe des Hauptfensters fest. Sie wird hier auf eine Breite von 220 Pixel und eine Höhe von 150 Pixel eingestellt.

Anschließend erzeugen wir nacheinander die drei Unter-Widgets *messages*, *clear* und *hide*, und zwar ebenfalls auf dem Heap. Sie sind Objekte der Klassen *QMultiLineEdit* bzw. *QPushButton*. Bei *QMultiLineEdit* ist der einzige Parameter beim Konstruktoraufruf das Vater-Widget. Dieses ist hier unser Hauptfenster, so dass *messages* ein Unter-Widget von *messageWindow* wird. Bei *QPushButton* kann man zusätzlich im ersten Parameter des Konstruktors die Beschriftung der Schaltfläche festlegen. Der zweite Parameter bestimmt hier das Vater-Widget, wiederum *messageWindow*. Allen drei Unter-Widgets wird anschließend mit *setGeometry* gleichzeitig eine Position (die ersten beiden Koordinaten) und eine Größe (die letzten beiden Koordinaten) zugewiesen. Die Parameter müssen hier von Hand berechnet werden, um zu verhindern, dass sich die Unter-Widgets überlappen oder aus dem Toplevel-Widget herausragen (wodurch sie nicht vollständig angezeigt würden). Auch soll der Platz zwischen den Widgets gleich sein, damit das Dialogfenster ansprechend wirkt.

Anschließend wird noch die Titelzeile des Toplevel-Widgets auf den Text »einfacher Dialog« gesetzt, und *messages* wird auf den Modus *ReadOnly* gesetzt, wodurch es nur Text anzeigen kann, der aber nicht editiert werden kann. Danach wird eine erste Textzeile (zu Demonstrationszwecken) in *messages* eingetragen.

Das Ergebnis, das auf dem Bildschirm erscheint, sehen Sie in Abbildung 3.33. Die Aufteilung der Widgets in Vater- und Unter-Widgets wird noch einmal in Abbildung 3.34 verdeutlicht. Beachten Sie auch hier, dass beim Löschen des Vater-Widgets auch die Unter-Widgets gelöscht werden. Um die Unter-Widgets brauchen wir uns also nicht mehr zu kümmern.



Abbildung 3-33 Dialogfenster mit drei Unter-Widgets

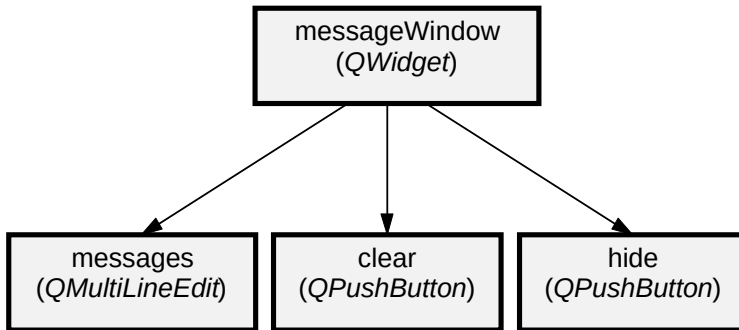


Abbildung 3-34 Die Widget-Hierarchie

Ein kleines Problem unseres Dialogs können wir schnell beheben: Wird das Toplevel-Widget vom Benutzer in der Größe verändert (indem er am Rahmen zieht), so sind die Unter-Widgets zum Teil nicht mehr sichtbar, oder es entsteht sehr viel Freiraum an einer Seite. Das unterbinden wir einfach, indem wir das Toplevel-Widget auf eine feste Größe einstellen. Dazu ersetzen wir in unserem Programm die Zeile

```
messageWindow->resize (220, 150);
```

durch:

```
messageWindow->setFixedSize (220, 150);
```

Unser Fenster hat aber immer noch viele unschöne Eigenschaften:

- Dialogfenster sollten in der Größe veränderbar sein, damit der Benutzer das Fenster seinen Bedürfnissen anpassen kann. Beim Ändern der Größe sollten sich die Unter-Widgets automatisch anpassen, so dass der neue Raum gut ausgefüllt wird und alle Widgets sichtbar bleiben.
- Das Fenster muss (sofern die Größe veränderbar ist) eine Minimalgröße haben. Sonst könnten zum Beispiel die Buttons so klein werden, dass die Beschriftung nicht mehr in sie hineinpasst. Diese Minimalgröße hängt aber zum Beispiel auch vom Button-Text und der verwendeten Schriftart ab.
- Alle Koordinaten müssen von Hand berechnet werden. Das ist eine zeitaufwendige und fehleranfällige Arbeit, die den Programmierer von den eigentlichen Designproblemen abhält. Eine Änderung der Anordnung zieht dabei oft eine völlige Neuberechnung der Koordinaten nach sich.

Die ersten beiden Probleme werden wir im nächsten Abschnitt lösen, indem wir im Toplevel-Widget den *resize*-Event abfangen und dort die Unter-Widgets neu positionieren. Aber auch diese Methode ist zu umständlich, so dass wir in den beiden darauf folgenden Abschnitten das Konzept der Layout-Klasse behandeln, das alle drei Probleme auf einfache Weise löst.

3.6.2 Anordnung der Widgets im *resize*-Event

Immer wenn das Toplevel-Widget durch den Benutzer oder durch den Befehl *resize* in der Größe geändert wird, erhält das Widget einen Event des Typs *resize*. Wenn wir nun die Event-Methode überschreiben, können wir in ihr die Unter-Widgets neu anordnen lassen. So passen sich die Fensterelemente immer der Fenstergröße an. Vorhandener Platz wird genutzt, und trotzdem bleiben alle Unter-Widgets sichtbar.

Auch dieser Ansatz ist noch nicht optimal und kann eigentlich nur bei einfachen Dialogen eingesetzt werden. Besser ist es, das Konzept der Layout-Klassen zu benutzen, wie es in den beiden folgenden Abschnitten 3.6.3 und 3.6.4 besprochen wird. Sie können den Rest dieses Abschnitts also überspringen, wenn Sie nur mit Layout-Klassen arbeiten wollen.

Die Koordinaten der Unter-Widgets berechnen wir nach folgenden Vorgaben:

- Die Button-Höhe bleibt konstant bei 20 Pixel. Zusätzlicher Freiraum in der Höhe wird vom *QMultiLineEdit*-Objekt genutzt.
- Das *QMultiLineEdit*-Objekt nutzt die Breite des Fensters.
- Die beiden Buttons teilen sich die Breite des Fensters gleichmäßig.
- Der Abstand zwischen den Widgets und der Abstand zwischen Widget und Rand beträgt immer 10 Pixel.

Daraus ergeben sich folgende Größen für die Unter-Widgets, wobei h und w die Höhe und Breite des Toplevel-Widgets bezeichnen, während $h1$ und $w1$, $h2$ und $w2$ bzw. $h3$ und $w3$ die Höhe und Breite des Anzeigefensters, des Clear-Buttons bzw. des Hide-Buttons angeben:

$$h1 = h - 10 - 10 - 20 - 10 \quad (\text{Gesamthöhe minus drei Zwischenräume und Button-Höhe})$$

$$w1 = w - 10 - 10 \quad (\text{Gesamtbreite minus zwei Zwischenräume})$$

$$h2 = h3 = 20 \quad (\text{Buttons haben eine konstante Höhe})$$

$$w2 = w3 = (w - 30) / 2 \quad (\text{Breite minus drei Zwischenräume, auf zwei Buttons verteilt})$$

Da wir die Methode *resizeEvent* überschreiben wollen, müssen wir eine neue Klasse, *MessageWindow*, von *QWidget* ableiten. Um aus der Klasse heraus Zugriff auf die Unter-Widgets zu haben, legen wir sie als Objektvariablen in der neuen Klasse an.


```
class MessageWindow : public QWidget {
    Q_OBJECT

public:
    MessageWindow ();
    ~MessageWindow () {}

protected:
    void resizeEvent (QResizeEvent *);

private:
    QMultiLineEdit *messages;
    QPushButton *clear, *hide;
};

MessageWindow::MessageWindow ()
    : QWidget ()
{
    setCaption ("Dialog mit resizeEvent");

    messages = new QMultiLineEdit (this);
    clear = new QPushButton ("Clear", this);
    hide = new QPushButton ("Hide", this);

    messages->setReadOnly (true);
    setMinimumSize (140, 110);
    resize (220, 150);
}

void MessageWindow::resizeEvent (QResizeEvent *)
{
    int buttonWidth = (width () - 30) / 2;
    messages->setGeometry (10, 10,
                           width () - 20, height () - 50);
    clear->setGeometry (10, height () - 30,
                       buttonWidth, 20);
    hide->setGeometry (width () - 10 - buttonWidth,
                      height () - 30, buttonWidth, 20);
}
```

Im Konstruktor von *MessageWindow* werden nun die drei Unter-Widgets angelegt; der *parent*-Eintrag lautet also *this*. Weiterhin wird die Minimalgröße des Toplevel-Widgets auf 140 x 110 Pixel gesetzt und die initiale Größe auf 220 x 150 Pixel. Die Position und Größe der Unter-Widgets muss noch nicht gesetzt werden, denn vor dem ersten Anzeigen mit *show* wird automatisch ein Event vom Typ *Resize* an das Widget geschickt, so dass die Unter-Widgets in der Methode *resizeEvent* platziert werden.

Die minimale Größe des Hauptfensters muss gesetzt werden, um zu verhindern, dass der Benutzer das Fenster beliebig verkleinern kann. Bei einem zu kleinen Fenster sind die Beschriftungen der Buttons nicht mehr lesbar, im Extremfall kann die Höhe oder Breite der Unter-Widgets sogar negativ werden.

Abbildung 3.35 zeigt unser Fenster in zwei verschiedenen Größen. Im ersten Bild ist das Fenster auf Minimalgröße verkleinert, im zweiten ist es vergrößert.



Abbildung 3-35 Das Fenster in zwei verschiedenen Größen

Die hier beschriebene Methode des Layouts wirft immer noch ein paar Probleme auf:

- Die Berechnungsvorschriften für die Platzierung der Unter-Widgets können bei komplexen Dialogen schnell unübersichtlich und kompliziert werden.
- In jedem Fall muss für das Dialogfenster eine eigene Klasse abgeleitet werden, um die Methode *resizeEvent* überschreiben zu können. Für einfache Dialoge ist das oft ein zu großer Aufwand.
- Die minimale Fenstergröße ist oft nur durch Ausprobieren zu ermitteln. Da sie außerdem auch von der verwendeten Schriftart und oft auch von der gewählten Landessprache abhängt, sind konstante Werte hier ohnehin nicht die optimale Wahl.

Die Abbildungen 3.36 und 3.37 verdeutlichen die Probleme. Abbildung 3.36 zeigt einen Button, der auf eine feste Breite gesetzt worden ist, um das englische Wort »Reset« anzeigen zu können. Da der Benutzer jedoch als Landessprache Deutsch gewählt hat, steht auf dem Button stattdessen »Zurücksetzen«. Dieser Text ist aber zu lang für den Button, daher wird er nicht vollständig angezeigt. Abbildung 3.37 zeigt das Problem, das entsteht, wenn die minimale Fenstergröße

zu klein gewählt wird. Beim Verkleinern des Fensters ergeben sich Positionen für die Unter-Widgets, die sich zum Teil überlappen und es damit fast unmöglich machen, die Buttons noch zu lesen oder zu bedienen.

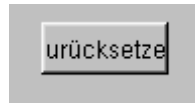


Abbildung 3-36 Falsche Button-Größe durch gewählte Landessprache

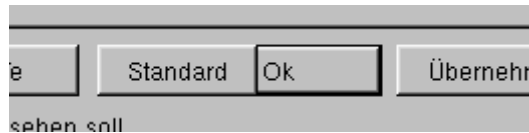


Abbildung 3-37 Falsche minimale Fenstergröße führt zu überlappenden Widgets

Ein ganz anderer Ansatz, der alle beschriebenen Probleme löst, ist die Benutzung der Layout-Klassen von Qt. Ihr Einsatz wird in den beiden folgenden Abschnitten beschrieben.

3.6.3 Einfache Layouts mit *QHBoxLayout*, *QVBoxLayout* und *QGridLayout*

Die Qt-Bibliothek stellt drei Widget-Klassen zur Verfügung, die sich automatisch um die Platzierung ihrer Unter-Widgets kümmern: *QHBoxLayout*, *QVBoxLayout* und *QGridLayout*. Diese Klassen sind von *QFrame* abgeleitet, wodurch das Widget ganz einfach mit der Methode *setFrameStyle* mit einem Rahmen versehen werden kann, der die Unter-Widgets umschließt (siehe Kapitel 3.7.1, *Statische Elemente*).

QHBoxLayout ordnet alle eingefügten Unter-Widgets nebeneinander von links nach rechts an, *QVBoxLayout* untereinander von oben nach unten. Zwischen den Unter-Widgets wird standardmäßig kein Platz gelassen, so dass die Unter-Widgets direkt aneinander stoßen und auch direkt am Rand des Widgets liegen. Die Konstruktor dieser beiden Klassen benutzen nur die normalen Parameter *parent* und *name*, die an den Konstruktor von *QFrame* weitergegeben werden.

Ein erstes, einfaches Beispiel zeigt, wie drei Buttons nebeneinander angeordnet werden:

```
QHBoxLayout *topWidget = new QHBoxLayout(); // no parent
QPushButton *b1 = new QPushButton ("Ok", topWidget);
QPushButton *b2 = new QPushButton ("Defaults", topWidget);
QPushButton *b3 = new QPushButton ("Cancel", topWidget);
topWidget->show ();
```

Das Ergebnis sehen Sie in Abbildung 3.38. Wenn Sie statt *QHBoxLayout* die Klasse *QVBoxLayout* benutzen, werden die Buttons untereinander angeordnet. Das Ergebnis zeigt Abbildung 3.39.

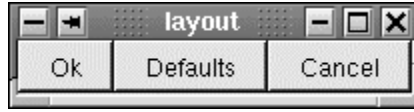


Abbildung 3-38 Drei Buttons im *QHBoxLayout*-Widget



Abbildung 3-39 Drei Buttons im *QVBoxLayout*-Widget

Unser Standardbeispiel aus den beiden letzten Abschnitten kann auch mit *QHBoxLayout* und *QVBoxLayout* realisiert werden. Um die relative Position der Widgets zueinander festzulegen, unterteilt man das Dialogfenster zunächst rekursiv in Bereiche mit übereinander bzw. nebeneinander angeordneten Widgets. Für unser einfaches Beispiel zeigt Abbildung 3.40 diese Unterteilung.

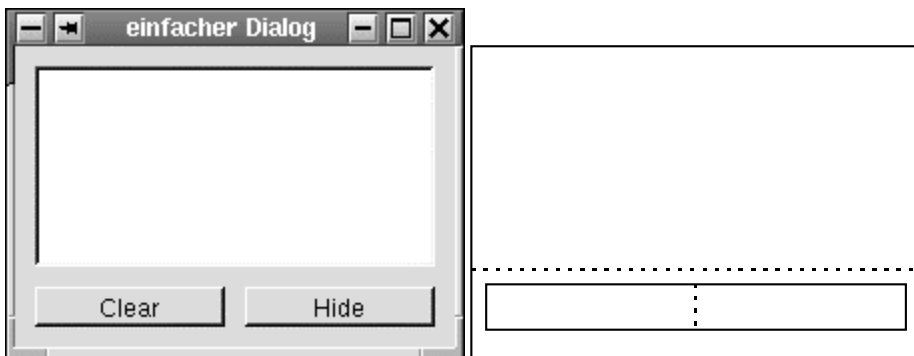


Abbildung 3-40 Unterteilung des Fensters

Unser Dialogfenster besteht also aus einem horizontal unterteilten Bereich mit zwei Teilen: Im oberen Teil befindet sich der Meldungsbereich, im unteren Teil liegen die Buttons. Der untere Teil ist wiederum vertikal in zwei Teile unterteilt, die jeweils einen Button enthalten.

Als Toplevel-Widget benutzen wir also ein *QVBox*-Objekt, in das wir zunächst das Meldungsfenster und anschließend ein *QHBox*-Objekt einfügen. In dieses Objekt kommen dann die beiden Buttons. Der entstehende Widget-Baum ist aufgebaut wie in Abbildung 3.41.

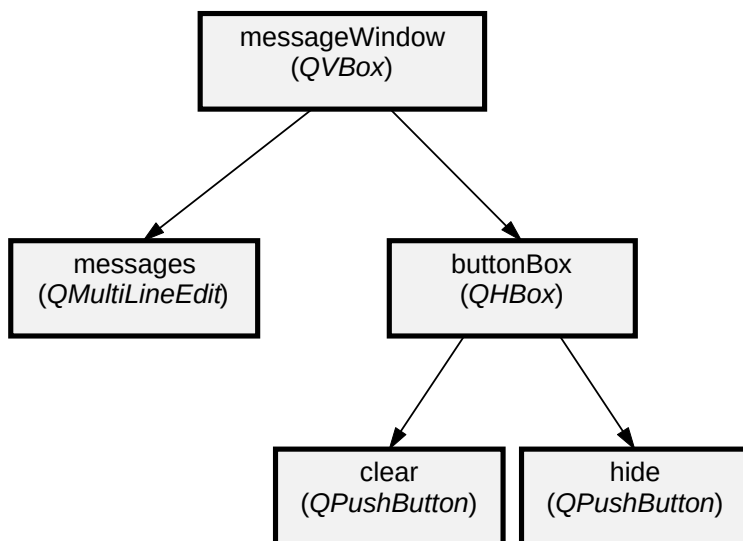


Abbildung 3-41 Widget-Baum mit *QVBox* und *QHBox*

Das Listing sieht so aus:

```
#include <qapplication.h>
#include <qhbox.h>
#include <qmultilineedit.h>
#include <qpushbutton.h>
#include <qvbox.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);

    QVBox *messageWindow = new QVBox ();

    QMultiLineEdit *messages =
        new QMultiLineEdit (messageWindow);
    QHBox *buttonBox = new QHBox (messageWindow);
    QPushButton *clear = new QPushButton
        ("Clear", buttonBox);
    QPushButton *hide = new QPushButton
        ("Hide", buttonBox);
```

```
messageWindow->show();  
app.setMainWidget (messageWindow);  
return app.exec();  
}
```

Das Ergebnis auf dem Bildschirm sehen Sie in Abbildung 3.42.

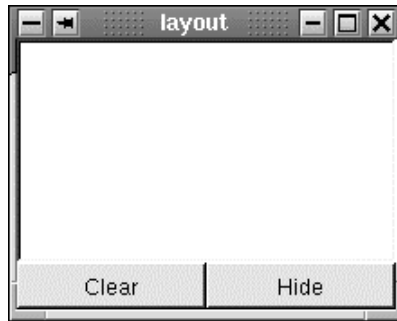


Abbildung 3-42 Das Beispielfenster mit QVBoxLayout und QHBoxLayout

QHBoxLayout und *QVBoxLayout* bieten auch die Möglichkeit, den Platz zwischen den Widgets festzulegen (mit *setSpacing*) sowie den Platz zwischen dem Rand und den Widgets (mit *setMargin*, einer von *QFrame* geerbten Methode). Wenn Sie unmittelbar vor *messageWindow->show* folgende drei Zeilen einfügen, erhalten Sie ein Fenster, das dem aus Abbildung 3.43 ähnelt.

```
messageWindow->setSpacing (10);  
messageWindow->setMargin (10);  
buttonBox->setSpacing (10);
```

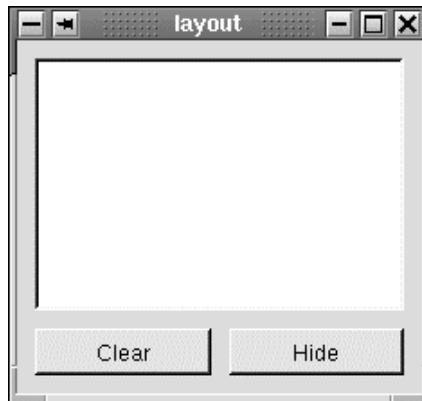


Abbildung 3-43 Zusätzliche Abstände mit *setSpacing* und *setMargin*

Die Klassen *QHBoxLayout* und *QVBoxLayout* sorgen nicht nur für die Anordnung der Unter-Widgets, sie bestimmen auch die minimale und maximale Gesamtgröße. So ist gewährleistet, dass die Widgets immer in vernünftigen Größen dargestellt werden. Dazu benutzen sie die Werte, die die Unter-Widgets in den Methoden *sizeHint* und *sizePolicy* zurückliefern. Somit ist z.B. bei den Buttons gewährleistet, dass die Beschriftung immer vollständig sichtbar ist, unabhängig vom verwendeten Zeichensatz und der Beschriftung. Diese Methoden sind für alle vordefinierten Widgets passend überladen, so dass sich in fast allen Fällen eine ergonomisch günstige und ästhetisch ansprechende Aufteilung ergibt. Wie dieses Verfahren genau funktioniert, erfahren Sie in Kapitel 3.6.5, *Platzbedarfsberechnung für Widgets*. In der Regel reicht es aber aus, einfach alle Widgets in die entsprechenden *QHBoxLayout*- oder *QVBoxLayout*-Fenster zu stecken. Die Platzverteilung, die dabei automatisch entsteht, ist immer sinnvoll, meist sogar optimal.

Wenn die Unter-Widgets in Form eines Gitters oder einer Tabelle angeordnet werden sollen, helfen die Klassen *QHBoxLayout* und *QVBoxLayout* meist nicht weiter. Mit *QGridLayout* kann man eine solche Anordnung der Widgets erreichen. Alle eingefügten Unter-Widgets werden in Spalten und Zeilen organisiert, wobei alle Elemente einer Spalte gleich breit und alle Elemente einer Zeile gleich hoch sind (sofern die Angabe von *sizePolicy* des jeweiligen Widgets eine Streckung in diese Richtung zulässt). Die einzelnen Spalten können dabei jedoch durchaus unterschiedliche Breiten, die Zeilen unterschiedliche Höhen haben. Die Höhe einer Zeile ist immer größer oder gleich der größten Mindesthöhe ihrer Elemente, und die Breite einer Spalte ist immer größer oder gleich der größten Mindestbreite.

Standardmäßig werden die Unter-Widgets zeilenweise von oben nach unten in ein *QGridLayout*-Objekt eingefügt, und innerhalb der Zeilen von links nach rechts (so wie man schreibt). Im Konstruktor des *QGridLayout*-Objekts muss dazu im ersten Parameter die Anzahl der Spalten festgelegt werden. Sobald die erste Zeile diese Anzahl von Elementen besitzt, wird eine neue Zeile begonnen. Die Anzahl der Zeilen ergibt sich so aus der Anzahl der eingefügten Widgets.

Will man die Elemente spaltenweise einfügen (von links nach rechts, innerhalb der Spalten von oben nach unten), so kann man einen alternativen Konstruktor benutzen, bei dem man hinter dem ersten Parameter einen weiteren Parameter vom Typ *QGridLayout::Direction* benutzt, dem man den Wert *QGridLayout::Vertical* übergibt. Der erste Parameter im Konstruktor gibt nun die Anzahl der Zeilen an.

Ein einfaches Beispiel soll die Benutzung zeigen. Das Ergebnis sehen Sie in Abbildung 3.44. Wenn Sie stattdessen den auskommentierten Konstruktor benutzen, der die Elemente spaltenweise anordnet, ergibt sich das Fenster aus Abbildung 3.45. Wie Sie sehen, werden übereinander liegende Elemente auf die gleiche Breite gestreckt, nebeneinander liegende auf die gleiche Höhe.

```
QGrid *window = new QGrid (3); // Toplevel-Widget

// Benutzen Sie diesen Konstruktor, um die Elemente
// spaltenweise einzufügen:
// QGrid *window = new QGrid (3, QGrid::Vertical);

QPushButton *b1 = new QPushButton ("Pause", window);
QPushButton *b2 = new QPushButton ("Stop", window);
QPushButton *b3 = new QPushButton ("Slow", window);
QPushButton *b4 = new QPushButton ("Rewind", window);
QPushButton *b5 = new QPushButton ("Play", window);
QPushButton *b6 = new QPushButton ("Fast Forward", window);
```



Abbildung 3-44 QGrid mit zeilenweise angeordneten Elementen

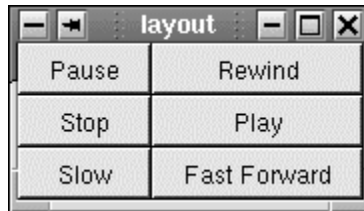


Abbildung 3-45 QGrid mit spaltenweise angeordneten Elementen

Genauso wie bei *QHBoxLayout* und *QVBoxLayout* kann man auch mit *QGrid* die Befehle *setSpacing* und *setMargin* benutzen, um zusätzlichen Zwischenraum zwischen die Widgets bzw. zwischen Widgets und Rand einzufügen.

3.6.4 Die Layout-Klassen *QBoxLayout* und *QGridLayout*

Im letzten Abschnitt haben wir die Widget-Klassen *QHBoxLayout*, *QVBoxLayout* und *QGrid* kennen gelernt. Ihre Anwendung war sehr einfach, aber ihre Flexibilität ist leider begrenzt. Im Gegensatz dazu ist die Anwendung der Layout-Klassen *QBoxLayout* und *QGridLayout* etwas komplizierter. Ihre Möglichkeiten sind dafür aber auch vielfältiger.

Die Klassen *QBoxLayout* und *QGridLayout* sind nicht von *QWidget* abgeleitet, noch nicht einmal von *QObject*. Da sie sich also den Overhead des Signal-Slot-Mechanismus sparen und auch kein eigenes Widget benötigen, sind sie effizien-

ter und Ressourcen schonender als *QHBox*, *QVBox* und *QGrid*. In der Tat benutzen *QHBox*, *QVBox* und *QGrid* intern ein Objekt der Klasse *QGridLayout*, um Elemente anzuordnen.

Die Aufgabengebiete entsprechen denen der Klassen *QHBox*, *QVBox* und *QGrid*: *QBoxLayout* ordnet Unter-Widgets eines Widgets nebeneinander oder übereinander an, *QGridLayout* ordnet sie in einer Tabellenform an. Da es sich hier aber nicht um eigene Widgets handelt, werden die Layout-Klassen parallel zu bestehenden Widgets benutzt.

Beginnen wir zunächst wieder mit unserem Standardbeispiel – dem Fenster mit einem *QMultiLineEdit*-Objekt und zwei Buttons. Das Fenster ist in unserem Fall eine Instanz der Klasse *QWidget* und enthält drei Unter-Widgets. Für die Widget-Hierarchie gilt also wieder Abbildung 3.34. Daneben erzeugen wir aber aus *QBoxLayout*-Elementen einen zweiten Hierarchiebaum, der in Abbildung 3.46 dargestellt ist. Beachten Sie, dass diese Hierarchie *nicht* aufgrund der *QObject*-Hierarchie entsteht, sondern eine interne Realisierung in den Layout-Klassen ist!

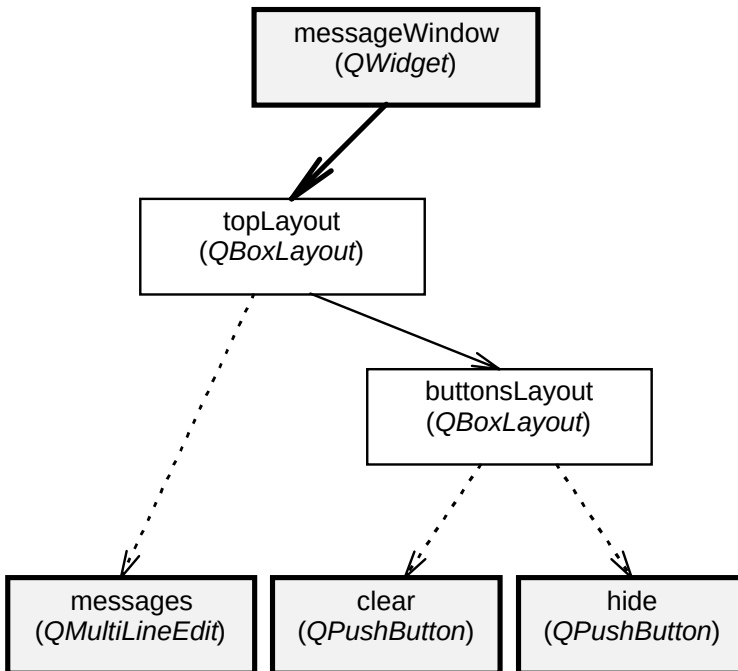


Abbildung 3-46 Die Layout-Hierarchie

In Abbildung 3.46 ist die Baumstruktur mit drei verschiedenen Pfeilarten dargestellt. Die Pfeilarten haben folgende Bedeutung:

- *dicker Pfeil (von Widget zu Layout-Objekt)* – Das oberste Layout-Element (hier *topLayout*) übernimmt die Aufgabe, auf *resize*-Events vom zugeordneten Widget (hier *messageWindow*) zu reagieren und die Unter-Widgets neu anzuordnen. Umgekehrt berechnet es bei seiner Aktivierung die Minimal- und Maximalgröße, die die Unter-Widgets benötigen, und legt diese Werte für das zugeordnete Widget mit den Methoden *setMinimumSize* und *setMaximumSize* fest. So ist gewährleistet, dass alle Unter-Widgets immer korrekt dargestellt werden. Die Zuordnung des Layout-Objekts zum Widget geschieht, indem Sie im Konstruktor von *topLayout* das Widget *messageWindow*.
- *dünner Pfeil (von Layout-Objekt zu Layout-Objekt)* – Ein Layout-Objekt kann andere Layout-Objekte zugewiesen bekommen, die Teilaufgaben übernehmen. Hier übernimmt *buttonsLayout* die Anordnung der Buttons *clear* und *hide*, während *topLayout* nun nur noch die beiden Elemente *messages* und *buttonsLayout* verwaltet. *buttonsLayout* darf keinem Widget zugeordnet sein (daher kein dicker Pfeil auf *buttonsLayout*). Im Konstruktor von *buttonsLayout* lässt man dazu den Widget-Parameter weg. (Er ist nicht 0, sondern wird weggelassen, wodurch ein anderer Konstruktor benutzt wird.) Die Zuordnung von *buttonsLayout* zu *topLayout* geschieht durch den Aufruf der Methode *topLayout->addLayout(buttonsLayout)*.
- *gestrichelter Pfeil (von Layout-Objekt zu Widget)* – Ein Layout-Objekt übernimmt die Anordnung von Unter-Widgets. Diese Widgets werden dem Layout-Objekt durch den Aufruf der Methode *addWidget* zugeordnet.

Damit der Baum korrekt aufgebaut ist, gibt es ein paar Anforderungen, die unbedingt erfüllt sein müssen:

- Ein Layout-Hierarchiebaum bezieht sich immer nur auf die Beziehung zwischen einem Widget und *allen direkten* Unter-Widgets. *Alle* Unter-Widgets müssen im Baum auftauchen, aber deren Unter-Widgets dürfen *nicht* auftauchen. (Wenn eines der Unter-Widgets selbst Unter-Widgets hat, kann man für diese Beziehung einen eigenen Layout-Hierarchiebaum erstellen.)
- Das Vater-Widget steht ganz oben im Baum, die Unter-Widgets ganz unten.
- Genau ein Layout-Objekt ist dem Vater-Widget zugeordnet (also genau ein dicker Pfeil). Dieses Layout-Objekt ist das höchste in der Hierarchie.
- Jedes Unter-Widget ist genau einem Layout-Objekt zugeordnet (ein gestrichelter Pfeil pro Unter-Widget).
- Jedes Layout-Objekt – außer dem obersten – ist genau einem anderen Layout-Objekt zugeordnet (dünner Pfeil).

Hier sehen Sie den Code, der diesen Layout-Baum erzeugt:

```
// Widgets erzeugen (ein Toplevel-Widget und
// drei Unter-Widgets)
QWidget *messageWindow = new QWidget ();
QMultiLineEdit *messages =
    new QMultiLineEdit (messageWindow);
QPushButton *clear =
    new QPushButton ("Clear", messageWindow);
QPushButton *hide =
    new QPushButton ("Hide", messageWindow);

// Layout-Objekte erzeugen
// topLayout wird messageWindow zugeordnet
QBoxLayout *topLayout =
    new QBoxLayout (QBoxLayout::TopToBottom,
                    messageWindow, 10);
// buttonsLayout wird keinem Widget zugeordnet
QBoxLayout *buttonsLayout =
    new QBoxLayout (QBoxLayout::LeftToRight);

// Hierarchiebaum aufbauen
topLayout->addWidget (messages);
topLayout->addLayout (buttonsLayout);

buttonsLayout->addWidget (clear);
buttonsLayout->addWidget (hide);
```

Die hier benutzte Reihenfolge beim Aufbau der Layout-Struktur ist wohl die übersichtlichste: Zuerst werden alle Widgets erzeugt. Danach werden alle Layout-Objekte erzeugt (eines ist dem Vater-Objekt zugeordnet, die anderen sind noch nicht zugeordnet), und schließlich wird der Layout-Baum von oben nach unten aufgebaut. Beim Aufbau des Baums muss darauf geachtet werden, dass bei *QBoxLayout* die Reihenfolge beim Einfügen entscheidend ist (in der Reihenfolge, wie es die Angabe im Konstruktor vorschreibt). Ebenso darf man erst dann Objekte in ein Layout-Objekt einfügen, wenn es selbst zugeordnet worden ist. Es muss also entweder das oberste Layout-Objekt sein oder mit *addLayout* in ein anderes Objekt eingefügt worden sein.

Da die Layout-Objekte nicht von *QObject* abgeleitet sind, stehen sie nicht im Hierarchiebaum der Widgets. Es stellt sich also die Frage, wann und wie die Layout-Objekte gelöscht werden, wann also ihr Speicher freigegeben wird. Ein Layout-Objekt, das einem Widget zugeordnet ist, wird im Destruktor dieses Widgets automatisch gelöscht. Weiterhin löscht ein Layout-Objekt in seinem Destruktor alle untergeordneten Layout-Objekte, nicht jedoch die Widgets. Insgesamt ergibt sich also, dass beim Löschen des ganzen Widget-Baums auch alle Layout-Objekte freigegeben werden. Beim Freigeben des obersten Layout-Objekts werden alle Layout-Objekte freigegeben, die Widgets bleiben jedoch unverändert erhalten.

Die Layout-Struktur kann in Grenzen dynamisch verändert werden: Es können jederzeit weitere Widgets oder Unter-Layouts in die vorhandene Struktur eingebaut werden. Daraus ergibt sich sofort eine Neuberechnung des Größenbedarfs und der Aufteilung des Platzes an die Widgets. Auch das Löschen eines Widgets (mit *delete*) funktioniert in der Regel reibungslos: Das Widget wird aus dem Layout-Objekt entfernt, dem es zugeordnet ist, und dieses Objekt berechnet seinen Platzbedarf neu. Dennoch ist Vorsicht geboten: Ein zu konfuse Umgang mit den Layout-Objekten kann unter Umständen auch zu Abstürzen des Programms führen. In der Regel legen Sie ohnehin zuerst alle Widgets und danach alle Layout-Objekte an, und anschließend wird an dieser Struktur nichts mehr geändert.

Wie die genaue Platzberechnung und Verteilung funktioniert, wird ausführlicher in Kapitel 3.6.5, *Platzbedarfsberechnung für Widgets*, beschrieben.

Die Klasse *QBoxLayout*

Wir kommen nun zu den vielfältigen, aber nicht immer ganz übersichtlichen Kontrollmöglichkeiten, die man beim Einsatz der Klasse *QBoxLayout* hat, wodurch diese mächtiger als *QHBox* und *QVBox* ist.

QBoxLayout kann die eingefügten Elemente in der Reihenfolge *LeftToRight*, *RightToLeft*, *TopToBottom* (oder abgekürzt *Down*) und *BottomToTop* (oder abgekürzt *Up*) anordnen. Den entsprechenden Wert setzt man für den Parameter *Direction* im Konstruktor der Klasse ein. Alternativ kann man die abgeleiteten Klassen *QHBoxLayout* oder *QVBoxLayout* benutzen. Der Unterschied besteht nur darin, dass der Richtungsparameter im Konstruktor fehlt und die Richtung *LeftToRight* (bei *QHBoxLayout*) bzw. *TopToBottom* (bei *QVBoxLayout*) benutzt wird.

Mit zwei weiteren Parametern im Konstruktor kann man zusätzlich Platz einfügen. Der Parameter *border* gibt an, wie viel Platz (in Pixel) zwischen dem Rand des Vater-Widgets und den Unter-Widgets gelassen werden soll. Der Parameter *space* gibt an, wie viel Platz (in Pixel) zwischen den einzelnen Elementen des Layouts (sowohl Widgets als auch Unter-Layouts) gelassen werden soll. Beide Werte sind natürlich vom persönlichen Geschmack und vom konkreten Fenster abhängig. Gängig ist für beide Parameter ein Wert von zehn Pixel. Den Parameter *border* hat übrigens nur der Konstruktor für das oberste Layout-Objekt, da sich die Unter-Layouts »im Inneren« des Widgets befinden.

Zusätzlicher Platz zwischen den Elementen des Layouts lässt sich mit der Methode *QBoxLayout::addSpacing(int space)* schaffen. Damit können Sie zum Beispiel eine lange Liste gleichartiger GUI-Elemente in sinnvolle Gruppen gliedern.

Mit den Methoden *QBoxLayout::addWidget* und *QBoxLayout::addLayout* kann ein Widget oder ein Unter-Layout an das Ende der Liste angehängt werden. Mit *QBoxLayout::insertWidget* und *QBoxLayout::insertLayout* kann man sie aber auch an eine beliebige Stelle zwischen andere Elemente einfügen. Dazu gibt man im ersten Parameter die Position an, an der das Widget oder Unter-Layout eingefügt werden soll.

Jedem Widget und jedem Layout, das mit *addWidget* bzw. *addLayout* zu einem Layout-Objekt hinzugefügt wird, kann als zweiter Parameter ein Streckungsfaktor (*stretch factor*) zugeordnet werden. Dieser Parameter vom Typ *int* gibt an, wie noch verbleibender Platz auf die Elemente zu verteilen ist. Wird der Parameter weggelassen, entspricht er einem Faktor von 0.

Die Aufteilung des vorhandenen Platzes an die Elemente erfolgt in folgenden Schritten:

1. Zunächst bekommt jedes Widget (oder Layout) die Mindestbreite (für horizontale *QBoxLayouts*) bzw. die Mindesthöhe (für vertikale *QBoxLayouts*) zugewiesen, die es benötigt (siehe Kapitel 3.5.6, *Platzbedarfsberechnung für Widgets*). Außerdem werden die festen Freiräume (bestimmt durch *border*, *space* sowie *addSpacing*) festgelegt.
2. Sollte noch Platz verbleiben, der aufgeteilt werden muss, wird nachgeschaut, ob eines oder mehrere der Elemente vom *sizePolicy*-Typ *Expanding* ist bzw. sind (für die entsprechende Richtung). Diese Elemente zeigen nämlich durch diese Angabe an, dass sie sinnvollerweise so groß wie möglich dargestellt werden sollen. Gibt es solche Elemente, wird der Platz nur unter ihnen aufgeteilt, ansonsten unter allen Elementen.
3. Die Aufteilung geschieht nun so, dass der verbleibende Platz im Verhältnis der Streckungsfaktoren verteilt wird. Ein Element mit Streckungsfaktor 12 erhält also dreimal so viel zusätzlichen Platz wie ein Element mit Faktor 4. Elemente mit Faktor 0 bekommen in der Regel keinen zusätzlichen Platz; es sei denn, alle Elemente haben den Faktor 0. Dann wird der Platz gleichmäßig auf alle Elemente verteilt.

In den meisten Fällen ist es nicht nötig, einen Streckungsfaktor anzugeben. Auf gleiche Elemente wird der Platz dann gleichmäßig verteilt, Elemente vom Typ *Expanding* bekommen den Vorrang bei der Platzvergabe. Wollen Sie jedoch gezielt ein Element bevorzugen, so geben Sie ihm einen Streckungsfaktor größer als 0. Durch die Vergabe verschiedener Faktoren können Sie das Verhalten noch genauer beeinflussen.

Zusätzlich zu festem Abstand mit *addSpacing* können Sie auch einen flexiblen Freiraum mit der Methode *addStretch (int factor)* einfügen. Der so eingefügte Freiraum verhält sich wie ein leeres Widget mit dem Streckungsfaktor *factor*.

Anwendungen der verschiedenen Kontrollmöglichkeiten werden ausführlich in den Übungsaufgaben besprochen.

Die Klasse *QGridLayout*

Auch die Klasse *QGridLayout* kann in der Layout-Hierarchie benutzt werden. Dabei dürfen Elemente von *QGridLayout* und *QBoxLayout* beliebig gemischt und miteinander verbunden werden.

Genau wie *QGrid* platziert *QGridLayout* die eingefügten Elemente in einer Tabellenstruktur, so dass nebeneinander liegende Elemente die gleiche Höhe und übereinander liegende Elemente die gleiche Breite haben. Die Anzahl der Zeilen und Spalten der Tabelle muss dabei bereits im Konstruktor angegeben werden. Beim Einfügen der Widgets oder Unter-Layouts gibt man der Methode *addWidget* bzw. *addLayout* in zwei weiteren Parametern (Zeile und Spalte, mit Wertebereich von 0 bis Gesamtzeilen-/spaltenzahl – 1) die Position in der Tabelle an. Die Reihenfolge des Einfügens ist daher, anders als bei *QGrid* oder bei *QBoxLayout*, beliebig. Positionen in der Tabelle dürfen auch leer bleiben. Auch komplett leere Zeilen oder Spalten sind erlaubt. Vorsichtig müssen Sie allerdings bei Zellen sein, die von mehreren Elementen belegt werden. Diese verdecken sich gegenseitig, ohne dass eine Warnung ausgegeben wird.

Eine Besonderheit von *QGridLayout* gegenüber *QGrid* ist die Möglichkeit, dass sich ein Widget über mehrere Zeilen und/oder Spalten erstrecken kann, wenn Sie die Methode *addMultiCellWidget* (*QWidget *w, int fromRow, int toRow, int fromCol, int toCol*) benutzen. Auf diese Weise lassen sich viele Layouts erstellen, die so sonst nicht oder nur schwer möglich wären. Diese Methode gibt es allerdings nur für Widgets, nicht für Unter-Layouts.

Ebenso wie bei *QBoxLayout* kann man auch bei *QGridLayout* im Konstruktor die Parameter *border* und *space* angeben, mit denen man den Abstand zwischen dem Rand und den Elementen bzw. den Abstand zwischen den Elementen festlegen kann.

Auch Streckungsfaktoren können zugeordnet werden, allerdings nicht einzelnen Elementen des Layouts, sondern nur ganzen Zeilen oder Spalten. Die Methoden hierzu heißen *setColStretch(int column, int factor)* und *setRowStretch(int row, int factor)*.

Die Mindestbreite einer Spalte bzw. die Mindesthöhe einer Zeile können Sie mit den Methoden *addColSpacing(int column, int space)* bzw. *addRowSpacing(int row, int space)* bestimmen. Diese Methoden erweitern die Bedingungen, die durch die Elemente einer Zeile bzw. Spalte festgelegt sind. Sie können zum Beispiel zusätzlichen festen Freiraum zwischen zwei Spalten setzen, indem Sie eine Spalte ohne Elemente dazwischen einfügen und dieser Spalte mit *setColSpacing* eine feste Breite geben.

3.6.5 Platzbedarfsberechnung für Widgets

Das Layout-Konzept von Qt ist sehr ausgeklügelt und recht komplex. In der Regel braucht man sich um die Layouts nicht zu kümmern, da der Platzbedarf automatisch berechnet wird. In diesem Kapitel wollen wir ein wenig die Hintergründe beschreiben, nämlich auf welchen Methoden diese Berechnungen beruhen.

Anforderungen der Widgets

Die Grundlage aller Berechnungen sind die Widgets selbst, die mit einigen Methoden Auskunft geben, welchen Bedarf an Platz sie haben:

- Die Methode `QWidget::sizeHint` liefert ein Objekt vom Typ `QSize` zurück, das angibt, wie viel Platz das Widget am liebsten haben würde. Diese Methode ist in allen vordefinierten GUI-Element-Klassen so überschrieben, dass hier ausreichende und für die praktische Arbeit sinnvolle Werte zurückgeliefert werden. Das Ergebnis hängt in vielen Fällen auch vom gerade angezeigten Inhalt ab (z.B. die Beschriftung eines Buttons), aber auch vom gewählten Stil (Windows, Motif, ...), den eingestellten Schriftarten und anderen Einstellungen des Widgets.
- Die Methode `QWidget::sizePolicy` liefert ein Objekt der Klasse `QSizePolicy` zurück. Dieses Objekt enthält Flags – getrennt nach horizontal und vertikal –, die angeben, wie die Angabe aus `sizeHint` aufzufassen ist: ob als grobe Richtschnur oder als unbedingt einzuhaltender Wert. (Die Einzelheiten werden unten erläutert.) Bis einschließlich der Version Qt 2.1 musste diese Methode überschrieben werden, um für eine Klasse einen anderen Wert zurückgeben zu können. Seit Qt 2.2 wird dieser Wert in einer Attributvariablen abgelegt und kann mit `setSizePolicy` jederzeit geändert werden.
- Die Methode `QWidget::minimumSizeHint` liefert wiederum ein `QSize`-Objekt zurück, das angibt, wie viel Platz das Widget auf jeden Fall benötigt, damit ein sinnvolles Arbeiten überhaupt möglich ist.

Betrachten wir die Werte einmal genauer, um zu sehen, wie das von `QWidget::sizePolicy` zurückgegebene `QSizePolicy`-Objekt aufgebaut ist:

Es enthält zwei Zahlenwerte: eine Angabe für die Breite und eine für die Höhe des Widgets. Jede der Angaben ist vom Typ `SizeType`, das ist ein `int`-Typ, der durch eine Oder-Kombination von drei Flags entsteht. Das Flag `MayShrink` gibt an, dass das Widget unter Umständen auch kleiner werden darf als die Angabe von `sizeHint`, ohne dass die Funktionsfähigkeit oder Bedienbarkeit zu sehr leiden würde. Das Flag `MayGrow` gibt an, dass das Widget größer sein darf als die Angabe in `sizeHint`, ohne dass es zu einer störend großen oder unübersichtlichen Darstellung kommt. Das Flag `ExpMask` gibt (wenn es gesetzt ist) an, dass das Widget möglichst allen zur Verfügung stehenden Platz bekommt, da es umso effizienter genutzt werden kann, je größer es ist.

Statt dieser drei Flags werden sechs vordefinierte Konstanten benutzt, die eine Kombination der Flags darstellen. Tabelle 3.7 listet diese Konstanten mit ihrer Bedeutung auf. Beachten Sie, dass *QSizePolicy* unterschiedliche Konstanten für Höhe und Breite setzen kann.

Konstante	Flags	Beschreibung
<i>Fixed</i>	keine	<i>sizeHint</i> liefert den einzigen vernünftigen Wert, der unbedingt eingehalten werden muss
<i>Minimum</i>	<i>MayGrow</i>	darf nicht kleiner werden als <i>sizeHint</i> , kann aber größer sein, ohne dass es stört
<i>Maximum</i>	<i>MayShrink</i>	darf kleiner werden, aber auf keinen Fall größer als der Wert von <i>sizeHint</i>
<i>Preferred</i>	<i>MayGrow</i> <i>MayShrink</i>	<i>sizeHint</i> ist nur ein Vorschlag, darf größer oder kleiner sein
<i>MinimumExpanding</i>	<i>MayGrow</i> <i>ExpMask</i>	darf nicht kleiner sein als <i>sizeHint</i> , aber größer; je größer, desto besser
<i>Expanding</i>	<i>MayGrow</i> <i>MayShrink</i> <i>ExpMask</i>	darf größer und auch kleiner sein als <i>sizeHint</i> ; aber je größer, desto besser

Tabelle 3-7 Die Konstanten für *QSizePolicy*

Zwei Beispiele sollen dieses Konzept veranschaulichen:

- Ein Widget der Klasse *QPushButton* wird in Höhe und Breite durch den Text und den Zeichensatz festgelegt. Die Größe, die der Button hat, wenn die Schrift ihn gerade ausfüllt, wird durch *sizeHint* zurückgeliefert. Die Höhe darf diesen Wert nicht unterschreiten, sollte ihn aber auch nicht überschreiten, da der Button sonst unästhetisch wirkt. In der Breite darf das Widget diesen Wert ebenfalls nicht unterschreiten, wenn die Schrift sichtbar bleiben soll. Die Breite darf aber ruhig wachsen, ohne dass der Button dadurch wesentlich schlechter aussehen würde. Allerdings nutzt es dem Button auch nichts, breiter zu sein als nötig, weshalb hier die *sizePolicy* für die Breite nicht *ExpMask* gesetzt hat. *sizePolicy* liefert also für die Breite die Angabe *Minimum* zurück (darf größer werden, nutzt aber nichts) und für die Höhe *Fixed* (sollte genau die passende Höhe haben).
- *QMultiLineEdit*, die Klasse, die wir schon oft in Beispielen benutzt haben, stellt eine beliebige Menge Text dar. Die Größe, die dazu nötig ist, kann man nicht genau im Voraus bestimmen. *sizeHint* liefert daher einen Wert zurück, der »üblicherweise« benutzt werden sollte. (Im konkreten Fall ist das eine Breite von ca. 30 Zeichen und eine Höhe von sechs Zeilen des aktuellen Zeichensatzes.) Das Widget bleibt aber vollständig bedienbar, auch wenn es kleiner wird als der Wert von *sizeHint*. Durch die Rollbalken ist jeder Teil des Textes immer erreichbar. Ein *QMultiLineEdit*-Widget ist aber umso besser zu bedienen und

zeigt umso mehr an, je größer es ist. Daher fordert es Vorrang bei der Vergabe von freiem Platz. Der Rückgabewert von *sizePolicy* ist entsprechend für Breite und Höhe *Expanding* (also gibt *sizeHint* nur eine unverbindliche Größe an, nach dem Motto: je größer, desto besser).

Wenn Sie in einem konkreten Fall nicht mit dem Wert zufrieden sind, den *sizePolicy* für ein Widget liefert, können Sie ab Qt 2.2 diesen Wert für ein Widget mit *setSizePolicy* einfach ändern. Wenn Sie mit der Version 2.0 oder 2.1 arbeiten, ist das nicht möglich. Hier müssen Sie den Umweg über eine abgeleitete Klasse wählen, in der Sie die Methode *sizePolicy* überschreiben und den gewünschten Wert zurückgeben lassen.

Berechnungen in den Layouts

Die Layout-Objekte (*QBoxLayout*, *QGridLayout*) fragen die Daten der in ihnen enthaltenen Widgets ab und berechnen daraus den Platzbedarf für sich selbst. Dieser ist wiederum über die Methoden *sizeHint*, *minimumSize*, *maximumSize* und *expanding* zu erfragen. So werden die Daten von den Unter-Widgets entlang der Layout-Hierarchie nach oben durchgereicht und zusammengerechnet. Das oberste Layout-Objekt hat damit also die Gesamtgröße berechnet. Wenn es ein Toplevel-Widget kontrolliert, so setzt es mit *QWidget::setMinimumSize* und *QWidget::setMaximumSize* den minimalen und maximalen Platzbedarf. Damit ist dann automatisch gewährleistet, dass der Anwender das Fenster nur im erlaubten Rahmen vergrößern und verkleinern kann. Außerdem wird die Anfangsgröße des Fensters auf *sizeHint* gesetzt, um direkt mit einer möglichst optimalen Darstellung zu beginnen.

Durch folgende Ereignisse können nun Neuberechnungen nötig werden:

- Wenn das Toplevel-Widget in der Größe geändert wird (z.B. weil der Anwender mit der Maus den Rahmen zieht), verteilt das oberste Layout-Objekt den nun zur Verfügung stehenden Platz auf seine Elemente (Widgets und Unter-Layouts). Die Unter-Layouts verteilen den ihnen zugewiesenen Platz weiter an ihre Elemente und so weiter.
- Wenn sich der Größenbedarf eines Widgets ändert (z.B. wenn die Beschriftung eines Buttons geändert wird), meldet es dieses durch einen Aufruf der Methode *QWidget::updateGeometry*. (Es muss diese Meldung von sich aus machen, denn woher sollen die Layout-Objekte wissen, dass ein Aufruf von *sizeHint* oder *sizePolicy* nun plötzlich einen anderen Wert liefert?) Daraufhin wird eine komplette Neuberechnung des Gesamtplatzbedarfs ausgelöst (von unten nach oben in der Layout-Hierarchie). Reicht der zur Verfügung stehende Platz nicht aus, so wird das Widget vergrößert (durch Neusetzen von *setMinimumSize*). Verkleinert wird es allerdings in der Regel nicht. Anschließend wird wieder der vorhandene Platz aufgeteilt.

- Wenn Widgets zur Layout-Hierarchie hinzukommen (*addWidget*, *insertWidget* oder *show*) oder wegfallen (*hide* oder *delete*), ändert sich ebenfalls der Platzbedarf. Auch hierbei wird eine Neuberechnung des Bedarfs von unten nach oben durchgeführt, und anschließend wird der vorhandene Platz von oben nach unten verteilt.

3.6.6 Übungsaufgaben

Übung 3.7 – *QVBoxLayout*, *QHBoxLayout*

Wie erzeugen Sie mit Hilfe der Klasse *QVBoxLayout* und *QHBoxLayout* ein Fenster mit der Anordnung wie in Abbildung 3.47? Wie erzeugen Sie das Fenster aus Abbildung 3.48?



Abbildung 3-47 Fenster mit einem *QMultiLineEdit*-Objekt und drei Buttons

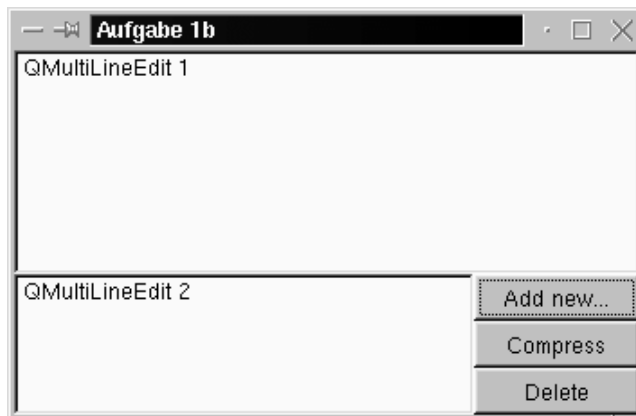


Abbildung 3-48 Fenster mit zwei *QMultiLineEdit*-Objekten und drei Buttons

Übung 3.8 – QGrid

Wie sieht der Code aus, der mit Hilfe der Klasse *QGrid* ein Fenster mit der Telefonastatur aus Abbildung 3.49 erzeugt? (Tipp: Mit *QGrid::skip()* können Sie ein Element des Gitters frei lassen.)



Abbildung 3-49 Telefonastatur

Übung 3.9 – QGrid

Wie sieht der Code aus, der ein Navigationsfenster wie in Abbildung 3.50 mit Hilfe von *QGrid* erzeugt?

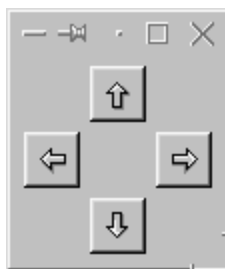


Abbildung 3-50 Navigationsfenster

Übung 3.10 – QGrid, QHBoxLayout, QVBoxLayout

Wie kann man *QGrid* als Ersatz für *QHBoxLayout* oder *QVBoxLayout* einsetzen, wenn man keinen Rahmen benötigt, aber dafür fünf Pixel Platz zwischen den Elementen frei lassen will?

Übung 3.11 – QHBoxLayout

Wie erreichen Sie die folgende Anordnung der Buttons in einem *QHBoxLayout*-Objekt?

a)



b)



c)



d)



e)



f)



g)



h)



i)



Übung 3.12 – *QHBoxLayout* und *QVBoxLayout*

Erstellen Sie mit *QHBoxLayout* und *QVBoxLayout* den Code, der das Fenster aus Abbildung 3.51 erzeugt.

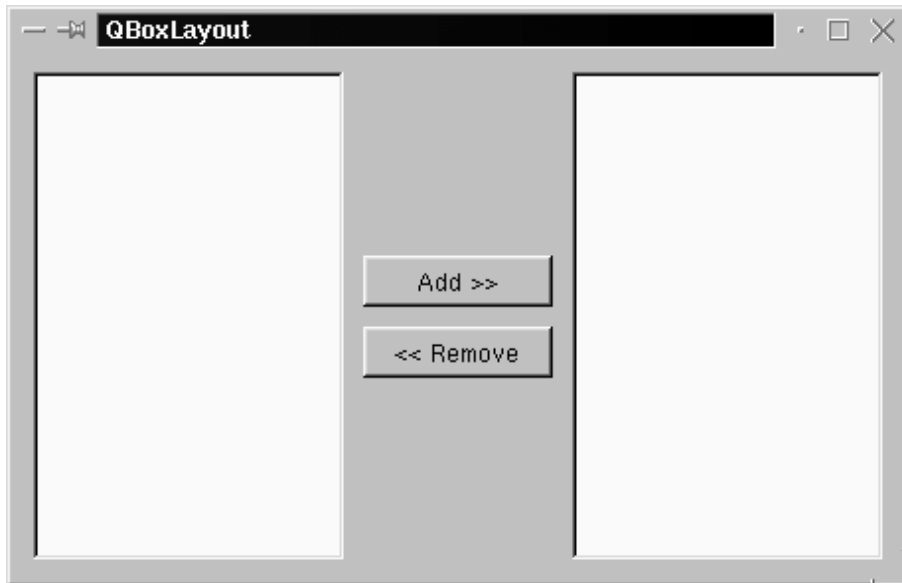


Abbildung 3-51 Dialogfenster mit *QBoxLayout*

Übung 3.13 – *QGridLayout*

Wie erreichen Sie die folgenden Anordnungen der Buttons in einem *QGridLayout*-Objekt?

a)

Button 1	Second Button
3	Number 4

b)

Button 1	Second Button
3	Number 4

c)

Button 1		Second Button
3		Number 4

d)

Button 1	Second Button
3	Number 4

Übung 3.14 – QGridLayout

Wie sieht der Code aus, der den Ziffernblock der Tastatur mit Hilfe von *QGridLayout* wie in Abbildung 3.52 darstellt?



Abbildung 3-52 Ziffernblock

Übung 3.15 – QVBoxLayout, QGridLayout

Die Anordnung der Buttons in Abbildung 3.53 lässt sich auf zwei Arten erreichen: durch zwei *QBoxLayout*-Objekte oder durch ein *QGridLayout*-Objekt. Wie sieht der Code dafür jeweils aus?

Button 1	Button 2
Button 3	

Abbildung 3-53 QVBoxLayout vs. QGridLayout

3.7 Überblick über die GUI-Elemente von Qt und KDE

Eine der wichtigsten Anforderungen an ein Programm mit grafischer Oberfläche ist die intuitive und einfache Bedienbarkeit der Bildschirmdialoge. Die Aufgabe des Programmierers ist es dabei kaum noch, die aufwendigen Bildschirmausgaben selbst zu programmieren. Er wird vielmehr zum Designer einer Oberfläche, die er zum Großteil aus fertigen und dem Benutzer schon aus vielen anderen Applikationen bekannten Elementen zusammensetzt. Nachdem er die benötigten Elemente ausgesucht hat, ordnet er diese so an, dass sie sinnvolle Blöcke ergeben und die wichtigen Informationen leicht zu erfassen sind (siehe auch Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*). Anschließend verbindet er die Elemente miteinander und mit dem darunter liegenden berechnenden Programm.

Qt enthält bereits eine große Anzahl dieser vorgefertigten Bedienelemente. Alle Elemente sind – direkt oder über mehrere Vererbungsstufen – von der Klasse *QWidget* abgeleitet. Ihre Komplexität reicht von einem einfachen Rahmen bis zu einem vollständigen Konfigurationsdialog für die Druckerausgabe. Auch KDE bietet eine Reihe von fertigen GUI-Elementen an, die in der Bibliothek *kdeui* zusammengefasst sind. Da KDE ja auf Qt aufbaut, sind die Aufgaben der KDE-Elemente meist spezieller, und ihre Anwendung ist zum Teil auf wenige Spezialfälle beschränkt.

An einigen Stellen bieten KDE und Qt sogar von der Funktionalität ähnliche oder identische Elemente an. Die Hauptursache für eine solche »erneute Erfindung des Rades« liegt darin, dass KDE in der Entwicklung dynamischer ist als Qt, so dass eine neue Idee dort schneller in eine neue Widget-Klasse umgesetzt werden kann und diese schneller in die offizielle *kdeui*-Bibliothek aufgenommen wird. Gute Ideen setzen sich aber meistens durch, so dass eine Klasse dieser Funktionalität – oft mehrmals überarbeitet, mit einer guten Schnittstelle versehen und gut dokumentiert – wenig später auch in der nächsten Version von Qt zu finden ist. Aus Kompatibilitätsgründen bleibt die Klasse in KDE jedoch weiter enthalten, so dass sie dann zweimal vorliegt.

Dem Programmierer stellt sich nun natürlich die Frage, welche Klasse er wählen soll, wenn die Funktionalität doch offensichtlich gleich ist. Diese Frage ist oft nicht leicht zu beantworten. Die Qt-Klassen sind meist besser durchdacht und sehen oft ästhetischer aus, während die KDE-Klassen oft etwas unbeholfen wirken und in der Schnittstelle häufig verwirrend sind. Auch werden Qt-Klassen besser gewartet, und die Dokumentation ist oft wesentlich besser – schließlich ist Qt ja ein Produkt, das verkauft werden soll und muss. Auf der anderen Seite haben manche KDE-Klassen ein besonderes Look&Feel, und die Benutzung der Qt-Klassen würde eine Abweichung vom Quasi-Standard sein, der sich unter bestehenden KDE-Programmen gebildet hat. Das ist zum Beispiel bei den Menü- und

Werkzeugleisten der Fall. Nur die KDE-Klassen sind in das KDE-spezifische Session-Management eingebunden und haben ein ganz charakteristisches Aussehen, durch das sich eine KDE-Applikation deutlich als eine solche zu erkennen gibt. In diesem Fall sollte man der KDE-Klasse den Vorzug geben.

Nahezu jedes der fertigen Bedienelemente kann in verschiedenen Stilarten auf dem Bildschirm dargestellt werden, wobei sich zwei Hauptlinien abzeichnen: das an Microsoft Windows angelehnte Aussehen der Elemente und die Motif-ähnliche Darstellung. Welche Stilart benutzt werden soll, wird für die ganze Applikation mit der Methode *QApplication::setStyle* festgelegt. In einer KDE-Umgebung kann der Benutzer diese Einstellung in einem KDE-Systemmenü vornehmen, so dass sich alle KDE-Programme ganz automatisch an seine Vorliebe anpassen. Dort werden ganze Themes – bestehend aus Stil, Bildschirmfarben, Hintergrundbildern, Icons und Systemklängen – zentral über das KDE-Kontrollzentrum gesteuert. Der Programmierer einer KDE-Applikation braucht sich also darüber keine Gedanken mehr zu machen.

Jedes GUI-Element kann mit *setEnabled(false)* deaktiviert werden. Es ist dann nicht mehr bedienbar und wird meist in Grau und mit geringerem Kontrast dargestellt. Das wird oft genutzt, wenn eine Einstellung zur Zeit nicht relevant ist. Bei einer guten Applikation können immer nur die GUI-Elemente bedient werden, die auch einen Sinn machen. Das gilt zum Beispiel auch für die Befehle in der Menüzeile: So ist der Befehl EINFÜGEN nur sinnvoll, wenn die Zwischenablage auch wirklich etwas enthält. Nur dann sollte er aktiviert werden können.

In den folgenden Abschnitten werden die vordefinierten GUI-Elemente in Gruppen eingeteilt und kurz mit ihren Aufgaben und Möglichkeiten vorgestellt. Obwohl diese Liste recht umfassend ist, sind nicht alle Klassen aufgenommen worden. Einige KDE-Klassen sind zu speziell, um in der alltäglichen Programmierpraxis von Nutzen zu sein. Nicht aufgeführt sind überdies alle Klassen, die von Programmierern auf der ganzen Welt programmiert wurden, aber nicht (oder noch nicht) den Weg in eine der beiden Bibliotheken geschafft haben. Sie sollten also regelmäßig im Internet Ausschau nach interessanten Klassen halten. Zentrale Anlaufpunkte sind dabei neben den Seiten <http://www.kde.org/> und <http://developer.kde.org/> zum Beispiel auch <http://www.ksource.org/>, <http://apps.kde.com/> oder <http://www.sourceforge.net/>.

3.7.1 Statische Elemente

Diese Gruppe von GUI-Elementen zeichnet sich dadurch aus, dass sie keine Aktion des Benutzers erforderlich machen. Sie stellen nur einen Text, ein Bild oder ein grafisches Element dar, der oder das zur Erläuterung, Beschriftung, Gliederung des Aufbaus oder zur Auflockerung dient. Sie tragen meist keine weiteren Informationen und haben immer das gleiche Aussehen.

Ein Beispiel, das wir schon einmal kurz in Kapitel 3.2.2, *Unter-Widgets*, kennen gelernt haben, ist die Klasse *QFrame*. Sie ist eine einfache Widget-Klasse, die einen Rahmen an ihrem Rand darstellt. Sie kann zum Beispiel benutzt werden, um GUI-Elemente, die inhaltlich zusammengehören, optisch zu gruppieren. Dazu fügt man diese GUI-Elemente als Unter-Widgets in ein *QFrame*-Objekt ein. Mit einem geeigneten Layout-Konzept gewährleistet man dann, dass die Elemente innerhalb des Rahmens so platziert werden, dass sie sich nicht überdecken.

Beachten Sie, dass die Layout-Klassen *QBoxLayout* und *QGridLayout* die Dicke des Rahmens bei der Platzverteilung nicht berücksichtigen. Wenn Sie also beispielsweise zehn Pixel Platz zwischen dem Inneren des Rahmens und den Unter-Widgets haben wollen (sowie fünf Pixel zwischen den einzelnen Unter-Widgets), sollten Sie folgende Technik benutzen:

```
QFrame *rahmen = new QFrame (topwidget);
rahmen->setFrameStyle (QFrame::Box | QFrame::Sunken);
rahmen->setLineWidth (1);
rahmen->setMidLineWidth (0);
QVBoxLayout *layout =
    new QVBoxLayout (rahmen, rahmen->frameWidth()+10, 5);
```

Beachten Sie, dass die Layout-Klassen *QHBox*, *QVBox* und *QGrid* von *QFrame* abgeleitete Klassen sind. Auch sie können also einfach mit der Methode *setFrameStyle* dazu gebracht werden, die in ihnen enthaltenen Widgets mit einem Rahmen zu versehen.

QFrame kann auch dazu benutzt werden, eine horizontale oder vertikale Linie zu zeichnen, indem Sie als Rahmenart *HLine* oder *VLine* eintragen (meist in der Schattierungsart *Sunken*, manchmal auch *Raised*, ungebräuchlich ist *Plain*). Damit können Sie zum Beispiel zwei unterschiedliche Bereiche eines Fensters grafisch voneinander trennen. Ein Beispiel dafür zeigt Abbildung 3.54, wo die Elemente von den Buttons im unteren Teil getrennt werden. Statt *QFrame* können Sie auch die KDE-Klasse *KSeparator* benutzen, die von *QFrame* abgeleitet ist und die solche Trennungslinien erzeugt. Einem Objekt von *KSeparator* müssen Sie nur eine Orientierung zuweisen – horizontal oder vertikal –, dann können Sie es zum Beispiel in eine Layout-Klasse einfügen. (Abbildung 3.54 benutzt in der Tat ein Objekt der Klasse *KSeparator*.)

Die Klasse *QGroupBox* ist von *QFrame* abgeleitet. Sie kann den Rahmen zusätzlich noch mit einem Titel versehen. So kann sich der Benutzer leichter einen Überblick über die gebotenen Optionen verschaffen. Eine typische Anwendung mit vier Unter-Widgets zeigt das linke Bild in Abbildung 3.55.



Abbildung 3-54 Trennlinien mit *QFrame* oder *KSeparator*

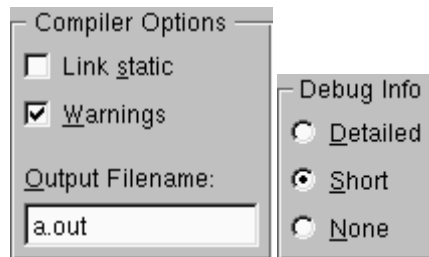


Abbildung 3-55 *QGroupBox* und *QButtonGroup* zeichnen einen Rahmen und eine Überschrift.

Neben *QGroupBox* gibt es noch die Klasse *QButtonGroup*, die sich vom Aussehen her nicht von *QGroupBox* unterscheidet. Sie kontrolliert allerdings alle eingefügten Elemente vom Typ *QRadioButton* und gewährleistet, dass höchstens einer der Buttons eingeschaltet ist (siehe auch Kapitel 3.7.3, *Buttons*). Das rechte Bild in Abbildung 3.55 zeigt eine Anwendung mit drei *QRadioButton*-Objekten.

Wenn Sie die Unter-Widgets einer *QGroupBox* oder *QButtonGroup* mit Layout-Klassen anordnen, sollten Sie unbedingt darauf achten, dass der Titel zusätzlichen Platz verbraucht, den die Layout-Klasse nicht berücksichtigt. Damit der Titel also lesbar bleibt, muss zunächst ein Leerraum eingefügt werden, zum Beispiel mit:

```
layout->addSpacing (box->fontMetrics().height());
```

Natürlich ist auch hier wie bei *QFrame* ein ausreichender Platz am Rand des Layouts freizulassen.

Die Klasse *QLabel* wird meist benutzt, um einen festen Text anzuzeigen. So beschriftet man zum Beispiel Eingabefelder. Der Schriftzug »Output Filename:« in Abbildung 3.55 links ist beispielsweise ein *QLabel*-Objekt. Der darzustellende Text kann im Konstruktor angegeben oder mit *setText* gesetzt (und auch nachträglich geändert) werden. Die Ausrichtung (linksbündig, rechtsbündig oder zentriert) kann festgelegt werden. Der Text kann mehrere Zeilen umfassen wie in Abbildung 3.56, indem Sie Zeilenvorschübe mit »\n« einfügen. Wenn Sie vor einen Buchstaben oder eine Ziffer das Zeichen »&« stellen, wird der Buchstabe unterstrichen und als Beschleuniger-Zeichen benutzt. Zusammen mit der **Alt**-Taste können Sie den Tastaturfokus auf ein Widget lenken, das Sie mit *setBuddy*

festlegen können. Beispielsweise springt der Cursor im Fenster in Abbildung 3.55 links automatisch in das Eingabefeld, wenn **[Alt]+[0]** gedrückt wird. Da *QLabel* von *QFrame* abgeleitet ist, kann es den Inhalt auch mit einem Rahmen versehen.

QLabel hat noch zwei Slot-Methoden, *setNum(int)* und *setNum(double)*, mit denen direkt eine Zahl dargestellt werden kann. Man spart sich so die Umwandlung einer Zahl in einen String.

QLabel kann aber viel mehr. Statt eines Textes kann es auch ein Bild (in Form eines *QPixmap*-Objekts), ein bewegtes Bild (in Form eines *QMovie*-Objekts, zum Beispiel eine bewegte GIF- oder PNG-Datei) oder einen formatierten Text im RichText-Format, einer Untermenge von HTML, darstellen (siehe Abbildung 3.57). Die Unterstützung weiterer Tags wird von Version zu Version ausgebaut. Eine Anwendung von RichText in einem *QLabel*-Objekt finden Sie in Kapitel 2.2.5, *Übungsaufgaben*.

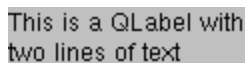


Abbildung 3-56 Ein zweizeiliger Text mit *QLabel*



Abbildung 3-57 RichText in einem *QLabel*-Objekt

Die Klasse *KDateTable* ist eine etwas exotischere, aber sehr nett anzuschauende Klasse. Sie stellt die Tage eines Monats in Form eines Kalenderblatts dar. Im Konstruktor kann man den darzustellenden Monat angeben. So ganz richtig ist die Zuordnung dieser Klasse zur Gruppe der statischen Elemente nicht, da man ein Datum markieren kann. Das wird zum Beispiel in der Klasse *KDatePicker* benutzt, die in Kapitel 3.7.5, *Auswahlelemente*, beschrieben wird.

Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Abbildung 3-58 KDateTable

3.7.2 Anzeigeelemente

Diese Gruppe von Elementen zeichnet sich dadurch aus, dass sie dem Benutzer einen Zustand oder eine Information anzeigen. Im Gegensatz zu den statischen Elementen verändern sie ihren Inhalt im Laufe des Programms.

Die Klasse *KLed* stellt zwei Zustände dar: »An« oder »Aus«. Im Zustand »An« ist sie grün, im Zustand »Aus« schwarz (siehe Abbildung 3.59). Diese Klasse kann eingesetzt werden, um einen schnellen Überblick über den Zustand eines Geräts oder einer Verbindung zu ermöglichen. Sie hat eine große Anzahl von Optionen, um die Darstellung dem eigenen Geschmack anzupassen. Da das Widget nicht interaktiv ist, lässt sich mit ihm der Zustand nicht ändern. Es dient nur zur Anzeige. Um eine »An«/»Aus«-Auswahl zu bieten, benutzen Sie die Klasse *QCheckBox*, die in Kapitel 3.7.3, *Buttons*, beschrieben wird.



Abbildung 3-59 KLed, hier im Zustand »An«

Bei einer längeren Aktion sollte ein Programm dem Benutzer melden, wie weit es ist. Dazu verwendet es typischerweise einen Fortschrittsbalken (*progress bar*). Diese Klasse gibt es sowohl in Qt (*QProgressBar*) als auch in KDE (*KProgress*). Die Darstellung ist unterschiedlich, bei beiden Klassen aber sehr flexibel einstellbar. Welche Klasse man vorzieht, bleibt dem eigenen Geschmack überlassen.



Abbildung 3-60 QProgressBar und KProgress

Die Klasse *QLCDNumber* bietet neben *QLabel* die Möglichkeit, Zahlen anzuzeigen. Die Zahlen werden dabei wie auf einer 7-Segment-Anzeige dargestellt, wodurch sie in jeder Größe gut lesbar sind. Gegenüber einem *QLabel*-Objekt hat *QLCDNumber* auch den Vorteil, dass sein Platzbedarf unabhängig von der dargestellten Zahl ist. Das Darstellungsformat kann binär, oktal, dezimal und hexadezimal sein und einen Dezimalpunkt enthalten.



Abbildung 3-61 *QLCDNumber*

3.7.3 Buttons

Buttons sind GUI-Elemente, die vom Benutzer angeklickt werden können. Dabei kann zwischen Klicken und Doppelklicken unterschieden werden, aber die Position des Klickens oder eine Bewegung der Maus auf dem Element hat keine Auswirkung. Die meisten Buttons haben zwei Zustände: »Gedrückt« und »Losgelassen«. Einige Buttons – so genannte Toggle-Buttons – unterscheiden darüber hinaus die Zustände »An« und »Aus«, wobei der Unterschied zwischen »Gedrückt« und »Losgelassen« dann meist nicht mehr wichtig ist.

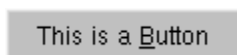


Abbildung 3-62 *QPushButton* mit Beschleuniger-Taste [Alt] + [B]

QPushButton ist die wohl am häufigsten verwendete Button-Klasse. Sie stellt auf dem Bildschirm ein Rechteck dar, das etwas nach vorn (auf den Benutzer zu) herausgezogen erscheint und in der Regel eine Beschriftung trägt. Klickt der Anwender auf das Objekt und lässt er die Maustaste wieder los, wird eine Aktion ausgeführt oder gestartet. Im Konstruktor von *QPushButton* kann man bereits einen Text angeben, der die Beschriftung des Buttons darstellt. Enthält dieser Text ein »&«-Zeichen, so wird der darauf folgende Buchstabe (oder die darauf folgende Ziffer) unterstrichen dargestellt und in Kombination mit der [Alt]-Taste als Beschleuniger-Zeichen benutzt. Wird also [Alt] mit diesem Buchstaben zusammen betätigt, so hat das die gleiche Wirkung wie das Anklicken mit der Maus. Die Schaltfläche wird auch kurz eingedrückt dargestellt, um anzuzeigen, dass sie aktiviert wurde.

Statt eines Textes kann ein Button der Klasse *QPushButton* auch ein Bild in Form eines *QPixmap*-Objekts als Beschriftung erhalten, indem man die Methode *setPixmap* benutzt.

Ein *QPushButton*-Objekt kann mit der Methode *setToggle(true)* zu einem Toggle-Button gemacht werden. Ein Klick (Drücken und Loslassen der Maus) führt dann

zu einem Umschalten des Zustandes zwischen »eingedrückt« (aktiv) oder »hervorstehend« (nicht aktiv). Davon sollte nur sehr sparsam Gebrauch gemacht werden, denn die allgemeine Funktionalität einer Schaltfläche dieser Art ist immer, direkt eine Aktion auszuführen. Toggle-Buttons haben sich zum einen in Werkzeugleisten etabliert, wenn eine von mehreren Auswahlmöglichkeiten aktiv sein soll. Toggle-Buttons können andererseits auch eingesetzt werden, wenn eine Aktion so lange läuft, wie der Button eingeschaltet ist (zum Beispiel eine PLAY/PAUSE-Taste für ein Wiedergabegerät). Wenn Sie dagegen zwischen zwei Zuständen einer Option wählen wollen, benutzen Sie besser ein *QCheckBox*-Objekt, das weiter unten in diesem Abschnitt besprochen wird.

Die Klasse *QToolButton* hat eine ähnliche Funktionalität wie die Klasse *QPushButton*. Im Unterschied zu dieser hebt sich der Button aber normalerweise nicht von der Umgebung ab, der Inhalt (in der Regel ein Icon) wird also einfach auf flachem Hintergrund dargestellt. Erst wenn man sich mit der Maus auf dem Button befindet, erscheint der Button in seiner typischen hervortretenden Darstellung und wird als Button erkennbar. Dieser Buttontyp wird in der Regel in Werkzeugleisten benutzt. Abbildung 3.63 zeigt eine Werkzeugleiste, in der sich die Maus auf dem Reload-Button befindet. Wenn mehrere kleine Buttons (insbesondere mit Icons statt mit Text als Beschriftung) nebeneinander angeordnet werden, ist es oft ästhetischer und übersichtlicher, wenn die Buttons keinen Rahmen haben, solange die Maus nicht auf sie zeigt. Sie benutzen diese Klasse nur selten direkt, sondern fügen Icons oder Aktionen in die Werkzeugleiste ein. Diese generiert dann das *QToolButton*-Objekt. (Siehe Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*, sowie Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*, für weitere Informationen über Werkzeugleisten.)



Abbildung 3-63 *QToolButton*

Zwei spezielle Buttons sind noch in der KDE-Bibliothek enthalten, nämlich die Buttons *KColorButton* und *KIconButton*, die beide von *QPushButton* abgeleitet sind. Sie ermöglichen die Auswahl einer Farbe bzw. eines Icons. Innerhalb des Buttons wird bei *KColorButton* ein Rechteck mit der aktuell eingestellten Farbe dargestellt, und beim Klick auf den Button öffnet sich ein Farbauswahl-Dialog (*KColorDialog*, siehe Kapitel 3.7.8, *Dialoge*), mit dem die Farbe neu festgelegt werden kann. *KIconButton* zeigt ein aktuelles Icon an, und durch einen Klick auf den Button öffnet sich ein Dialog (*KIconButton*, siehe Kapitel 3.7.8, *Dialoge*), mit dem man ein anderes Icon aus einem Verzeichnis des Dateisystems auswählen kann.



Abbildung 3-64 KColorButton und KIconButton

Es gibt noch zwei weitere Buttons von der Art des Toggle-Buttons (mit den Zuständen »An« und »Aus«) in der Qt-Bibliothek. Diese beiden haben allerdings ein ganz anderes Aussehen als *QPushButton*.

QCheckBox stellt einen kleinen, rechteckigen Kasten dar, in dem eine Markierung erscheint, wenn der Button im Zustand »An« ist. Rechts neben dem Kasten steht ein beschreibender Text. Um den Zustand zu wechseln, kann man auf den Kasten oder auf die Beschriftung klicken oder (wie bei *QPushButton*) eine Beschleunigtaste benutzen. Außer in Abbildung 3.65 ist ein Beispiel für *QCheckBox* auch in Abbildung 3.55 links zu sehen. Auch *QCheckBox* kann statt des beschreibenden Textes ein Bild in Form eines *QPixmap*-Objekts enthalten.



Abbildung 3-65 QCheckBox

QRadioButton stellt die Markierung, ob das Objekt im Zustand »An« oder »Aus« ist, in einem Kreis oder einer Raute (im Motif-Stil) dar, ansonsten ähneln das Aussehen und die Bedienung sehr *QCheckBox* (siehe Abbildung 3.66 sowie Abbildung 3.55 rechts). Der Unterschied tritt erst im Verhalten bei mehreren Objekten auf. *QCheckBox*-Objekte sind unabhängig voneinander. Jedes Objekt kann im Zustand »An« oder »Aus« sein. *QRadioButton* wird dagegen benutzt, wenn von mehreren Optionen nur eine einzige ausgewählt sein darf. Bei mehreren *QRadioButton*-Objekten werden automatisch alle anderen ausgeschaltet, sobald eines angeschaltet wird. Die Kontrolle über dieses Verhalten übernimmt die Klasse *QButtonGroup*, die schon im Abschnitt 3.7.1, *Statische Elemente* erwähnt wurde. Den anzeigenden Rahmen von *QButtonGroup* muss man nicht unbedingt benutzen. Man kann auch die *QRadioButton*-Objekte mit der Methode *insert* einfügen und das *QButtonGroup*-Objekt im Zustand »Versteckt« belassen.

Beachten Sie: Solange ein *QRadioButton*-Objekt nicht unter die Kontrolle eines *QButtonGroup*-Objekts gestellt wurde, hat es das gleiche Verhalten wie *QCheckBox*. Sie sollten es aber niemals wie dieses benutzen, denn der Anwender kennt diese Button-Art aus anderen Applikationen und wäre von diesem abweichenden Verhalten verwirrt. Als Faustregel kann man sagen, dass *QRadioButton* niemals allein auftritt, sondern immer in Gruppen zu mehreren Optionen, und dass dieser Button immer einem *QButtonGroup*-Objekt zugeordnet sein sollte.

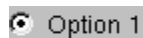


Abbildung 3-66 QRadioButton

3.7.4 Einstellungselemente

Mit den GUI-Elementen dieser Gruppe kann der Anwender einen Zahlenwert festlegen, der in einem bestimmten Bereich liegt. Typische Beispiele für solche Anwendungen sind die Einstellung der Rot-, Grün- und Blauanteile einer Farbe, die Festlegung eines Timeout-Intervalls oder das Verschieben eines Fensters mit einem Rollbalken.

Die Festlegung des Bereichs und der Schrittweite erfolgt bei vielen Elementen durch die Klasse *QRangeControl*, von der sie abgeleitet sind. (*QRangeControl* ist nicht von *QWidget* abgeleitet, daher sind die Klassen der Einstellungselemente mit Mehrfachvererbung von *QWidget* und *QRangeControl* abgeleitet.) Ein Objekt der Klasse *QRangeControl* verwaltet einen ganzzahligen *int*-Wert, für den eine obere und untere Grenze angegeben werden kann, sowie eine kleine Schrittweite (*lineStep*) und eine große Schrittweite (*pageStep*). Die Klasse hat Methoden, um den Wert um einen kleinen oder großen Schritt zu erhöhen oder zu verringern (immer unter Einhaltung der Grenzen) und um den Wert zu setzen und auszulesen. Bei jeder Änderung des Wertes wird die virtuelle Methode *valueChanged* aufgerufen, die in den abgeleiteten Klassen überschrieben ist und entsprechende Reaktionen hervorruft. (Beachten Sie, dass *RangeControl* kein Signal aussenden kann, da es nicht von *QObject* abgeleitet ist.)

Das einfachste Einstellungselement, das noch dazu den wenigsten Platz braucht, ist die Klasse *QSpinBox* (siehe Abbildung 3.67). Sie zeigt in einem Feld den Zahlenwert an, der mit den Pfeil-Buttons erhöht oder verringert werden kann. Der erwünschte Wert kann auch über die Tastatur direkt in das Feld eingegeben werden. In jedem Fall wird der Wertebereich getestet und gegebenenfalls korrigiert.

Das Element kann in den *zyklischen Modus* gesetzt werden, so dass nach dem größten möglichen Wert wieder der kleinste Wert angezeigt wird und entsprechend vor dem kleinsten wieder der größte. Vor und hinter die Zahl kann man einen String einfügen, der dargestellt wird. In Abbildung 3.67 wurde beispielsweise ein Prozentzeichen an die Zahl angehängt. Statt des kleinsten Wertes kann man auch einen speziellen Text anzeigen lassen. Hat man beispielsweise ein *QSpinBox*-Element benutzt, um die Dicke eines Rahmens in Pixel einzustellen, kann man statt des Werts 0 den Text »kein Rahmen« anzeigen lassen.

Durch Überladen der Klasse kann man mit dem Objekt relativ einfach auch andere Werte als ganze Zahlen benutzen, solange es eine eindeutige Zuordnung zwischen ganzen Zahlen und dem gewünschten Wertetyp gibt. Dazu muss man die Methoden *mapValueToText* und *mapTextToValue* überschreiben, die eine ganze Zahl in einen Text (zur Darstellung) bzw. einen Text in eine ganze Zahl (bei Tastatureingabe des Wertes) umwandeln. So könnte man zum Beispiel statt der Zahlen 1 bis 7 die Wochentage »Montag« bis »Sonntag« anzeigen lassen. (Hierfür eignet sich jedoch eher ein Auswahlelement wie *QComboBox*, siehe

Abschnitt 3.7.5, *Auswahlelemente*.) Man kann beispielsweise auch reelle Zahlen mit einer Nachkommastelle im Bereich von 0.0 bis 1.0 einstellen lassen, indem man den Wertebereich von 0 bis 10 wählt und dann in der Darstellung den Wert durch 10 teilt.

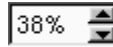


Abbildung 3-67 QSpinBox

Die anderen Einstellungselemente stellen den Wert im Gegensatz zu *QSpinBox* nicht als Text, sondern als Balken oder Position eines Zeigers dar, sind also analoge Regler. Am einfachsten lassen sie sich mit der Maus bedienen, indem der Zeiger oder Balken mit der Maus gefasst und verschoben wird. Aber auch über die Tastatur können die meisten dieser Elemente mit Hilfe der Pfeiltasten und der Tasten <PageUp> und <PageDown> bedient werden.

Die Klasse *QSlider* stellt einen Schieberegler auf einem Balken dar, der verschoben werden kann, um einen ganzzahligen Wert innerhalb bestimmter Grenzen auszuwählen. Wenn man den Wertebereich groß genug wählt, ist der Wert nahezu kontinuierlich einstellbar. Der Balken kann horizontal oder vertikal liegen. Man kann Markierungen in regelmäßigen Abständen anbringen. *QSlider* wird oft benutzt, wenn nicht ein exakter Wert benötigt wird. Typische Anwendungen sind Einstellungen von einem Helligkeitswert, einem Rotationswinkel oder der Lautstärke. *QSlider* kann nur ganzzahlige Werte zurückliefern. Durch folgenden Trick kann man jedoch auch reelle Zahlen erhalten: Wenn man beispielsweise einen reellen Wertebereich von 1.0 bis 2.0 braucht, so wählt man als Zahlenbereich 1000 bis 2000 und teilt den zurückgelieferten Wert vor der Anwendung durch 1000. Die Einstellung kann dann bis auf 0,001 genau vorgenommen werden, was völlig ausreichend ist, da ein noch genauerer Wert mit der Maus auf dem Balken gar nicht angezeigt werden kann.

Man sollte die Werte von *lineStep* und *pageStep* von *QSlider* unbedingt auf vernünftige Werte setzen, damit das Element auch mit der Tastatur gut zu bedienen ist. Als ungefähre Richtschnur gilt, dass *pageStep* etwa ein Zwanzigstel des Wertebereichs sein sollte. *lineStep* ist oftmals 1, um die Einstellung so genau wie möglich zu machen (falls nötig), sollte aber nicht wesentlich kleiner als ein Zwanzigstel von *pageStep* sein. Will man beispielsweise einen Helligkeitswert zwischen 0 und 255 wählen, kann man *lineStep* = 1 und *pageStep* = 16 wählen.



Abbildung 3-68 QSlider, hier horizontal und ohne Markierungen

Ein ähnliches Element mit allerdings anderen Aufgabengebieten ist der Rollbalken (*scrollbar*), der durch die Klasse *QScrollBar* erzeugt wird (siehe Abbildung 3.69). Auch diesen Balken kann man mit der Maus hin- und herziehen, um einen

Wert einzustellen. Zusätzlich kann der Balken noch über zwei Buttons mit Pfeilen bewegt werden (Schrittweite *lineStep*), oder indem man in die Bereiche zwischen dem Balken und den Pfeil-Buttons klickt (Schrittweite *pageStep*). *QScrollBar* wird in der Regel dazu verwendet, den angezeigten Ausschnitt eines anderen Fensters zu kontrollieren. Intern wird diese Klasse zum Beispiel in den Klassen *QMultiLineEdit*, *QListBox*, *QListView* und vielen anderen mehr benutzt. Sobald dort der anzuzeigende Text nicht mehr in das Fenster passt, wird am Rand des Fensters ein Rollbalken eingeblendet.



Abbildung 3-69 *QScrollBar*, waagerecht

Das *QScrollBar*-Element kann senkrecht oder waagerecht stehen. Die Breite des verschiebbaren Balkens (bzw. die Höhe bei senkrechter Platzierung) verändert sich, so dass das Verhältnis zwischen der Breite des Balkens und dem Schiebereich in etwa dem Verhältnis zwischen dem angezeigten Teil des Textes und der Gesamtlänge des Textes entspricht. Somit kann der Benutzer abschätzen, wie lang der Text ungefähr ist. Als Grundlage für diese Berechnung dienen der Wert von *pageStep* und die Größe des Wertebereichs.

Oft wird ein *QScrollBar*-Objekt auch anstelle eines *QSlider*s zum Einstellen eines Wertes benutzt. Der Vorteil ist, dass die Position durch die Pfeiltasten genauer festgelegt werden kann. Da *QScrollBar* aber eigentlich für das Verschieben eines Ausschnittes vorgesehen ist, sollte man es nicht »zweckentfremden«. *QSlider* sieht professioneller aus und kann mit Markierungen versehen werden.

Ein Einstellelement mit kreisförmiger Anordnung ist *QDial* (siehe Abbildung 3.70). Dieses Element stellt einen Zeiger dar, der mit der Maus positioniert werden kann. Genau wie *QSlider* liefert *QDial* ganzzahlige Werte in einem definierbaren Bereich zurück.

QDial ist zwar ein »grafisches Schmankerl«, jedoch in der Bedienung schwieriger. Es benötigt mehr Platz auf dem Bildschirm, und durch die schrägstehenden Kanten, die oftmals nicht ganz exakt platziert werden können, sieht es etwas grober und unprofessioneller aus. Man sollte daher *QSlider* der Vorzug geben.



Abbildung 3-70 *QDial*

Zwei spezielle GUI-Elemente der KDE-Bibliothek sind *KValueSelector* und *KGradientSelector* (siehe Abbildung 3.71 und 3.72). Wenn man sie verwendet, kann man wie bei *QSlider* einen Wert aus einem definierbaren Wertebereich mit

der Maus auswählen, allerdings wird im Hintergrund ein Farbverlauf dargestellt. In *KGradientSelector* kann man zusätzlich je eine Beschriftung am linken und rechten Rand einfügen lassen. Diese beiden GUI-Elemente werden im *KColorDialog* benutzt (ebenso wie das *KHSSelector*-Element, das im Anschluss beschrieben wird). Sie können aber auch in anderen Programmen genutzt werden, insbesondere wenn ein Farb- oder Helligkeitswert auszuwählen ist, da sie grafisch sehr ansprechend gestaltet sind.



Abbildung 3-71 KValueSelector



Abbildung 3-72 KGradientSelector

Die Klasse *KXYSelector* aus der KDE-Bibliothek erlaubt es dem Anwender, gleichzeitig zwei Werte festzulegen, indem er eine Position in einem Rechteck anklickt. Der eine Wert wird durch die x-Koordinate des angeklickten Punktes festgelegt, der andere durch die y-Koordinate. Eine Spezialanwendung dieses GUI-Elements ist mit der Klasse *KHSSelector* realisiert worden, in der der Anwender den *Hue*- (Farbton-) und den *Saturation*- (Sättigungs-)Wert einer Farbe in diesem Feld auswählen kann, wobei in der Fläche des Rechtecks der entsprechende zweidimensionale Farbverlauf eingezeichnet ist. Auch dieses Element wird im *KColorDialog* verwendet, kann jedoch vielleicht auch in anderen Programmen nützlich sein.

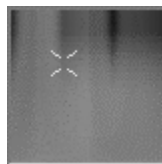


Abbildung 3-73 KHSSelector

3.7.5 Auswahlelemente

In dieser Gruppe der GUI-Elemente werden die Objekte zusammengefasst, bei denen der Anwender eine Auswahl aus mehreren Alternativen treffen kann. Typische Beispiele hier sind die Auswahl eines Wochentages, einer Landessprache oder einer vordefinierten Farbe.

Eine Möglichkeit haben wir bereits im Abschnitt 3.7.3, *Buttons*, kennen gelernt: Mehrere *QRadioButton*-Elemente ermöglichen es, eines der Elemente auszuwählen. Die anderen werden automatisch deaktiviert.

Ein anderes Element zur Auswahl ist *QListBox*. Die verschiedenen Optionen werden untereinander in einem Feld dargestellt. Reicht der Platz im Feld nicht aus, wird am rechten Rand automatisch ein Rollbalken eingefügt, mit dem man die Optionen nach oben und unten verschieben kann (siehe Abbildung 3.74). Durch Anklicken eines Objekts wird dieses aktiviert und hervorgehoben dargestellt. Normalerweise ist immer nur eine Option aktiviert. Mit der Methode *setMultiSelection* kann man jedoch den Modus ändern, so dass nun mehrere Elemente aktiviert werden können. Das Anklicken einer bereits aktivierten Option deaktiviert diese wieder. Neben Textelementen kann *QListBox* auch Bilder (als *QPixmap*-Objekte) enthalten. Durch das Überladen der Klasse *QListBoxItem* kann man sogar eigene Elemente definieren.



Abbildung 3-74 *QListBox*

Ein weiteres GUI-Element zur Auswahl einer Option, das weniger Platz benötigt und trotzdem eine übersichtliche Auswahl ermöglicht, wird von der Klasse *QComboBox* erzeugt. Die gerade ausgewählte Option wird in einem kleinen Fenster dargestellt, und erst durch das Klicken auf einen Button rechts neben dem Fenster klappt ein weiteres Fenster mit allen Optionen auf, aus denen mit der Maus die gewünschte ausgewählt werden kann. Danach schließt sich das untere Fenster wieder, so dass es nichts mehr verdeckt. Die Auswahl kann auch über die Tastatur mit den Pfeiltasten vorgenommen werden. Es gibt zwei verschiedene Erscheinungsformen von *QComboBox*: Die erste (ältere) realisiert das Auswahlfenster in Form eines Popup-Menüs, die zweite durch eine Art aufspringender *QListBox*. Die zweite Variante ist besser, da sie auch bei sehr vielen Optionen noch bedienbar bleibt, indem sie einen Rollbalken benutzt. Bei der ersten Variante kann es geschehen, dass ein Teil der Optionen unter der unteren Bildschirmkante verschwindet und so nicht mehr ausgewählt werden kann. Bei

QComboBox kann grundsätzlich nur eine Option aktiviert sein. Bei *QComboBox* kann eine Option ebenfalls durch einen Text oder eine QPixmap repräsentiert werden.

QComboBox kann in den Modus *ReadWrite* gesetzt werden, wobei dann der Benutzer nicht nur eine der vorhandenen Optionen auswählen kann, sondern die vorhandene auch ändern oder einen eigenen Wert eingeben kann. (Dann können die Optionen allerdings nur Textoptionen sein, keine Pixmaps). In diesem Fall enthält das Fenster eine Eingabezeile des Typs *QLineEdit* (siehe Abschnitt 3.7.6, *Eingabeelemente*). Dieser Modus ist sehr praktisch, wenn man dem Benutzer ein paar Standardwerte vorgeben, ihn aber nicht darauf beschränken will, zum Beispiel bei einer Schriftgröße oder einem Vergrößerungsfaktor. Er wird auch oft als eine Eingabezeile »mit Gedächtnis« eingesetzt, da man die selbst eingetragenen Werte automatisch in die Liste der Optionen einfügen lassen kann. So kann der Benutzer die alten Eintragungen nochmals auswählen.



Abbildung 3-75 *QComboBox* mit ausgeklapptem Menü

Welche der drei Varianten zur Auswahl einer Alternative man in seinen Dialogen benutzen sollte, hängt von den Umständen ab. Wenn es nur wenige (maximal acht), immer gleich bleibende Optionen sind, sollte man für jede Option ein *QRadioButton*-Element benutzen. Bei vielen Optionen braucht diese Lösung allerdings zu viel Platz und wird unübersichtlich. Sind die Optionen in Zahl und/oder Inhalt veränderlich, so ist diese Lösung auch ungeschickt, da sich bei jeder Änderung der Platzbedarf und damit das ganze Layout ändern würde. Daher kann man in solchen Fällen besser *QListBox* oder *QComboBox* benutzen. *QComboBox* benötigt weniger Platz, ist aber etwas unübersichtlicher, da der Anwender die Option, die er wünscht, erst im neuen Fenster suchen muss.

Sollen mehrere Optionen aktivierbar sein, ist *QListBox* die einzige Möglichkeit. Beachten Sie aber, dass *QListBox* auf jeden Fall die falsche Wahl ist, wenn die Optionen inhaltlich nicht zusammenhängen. Sollen unabhängige Optionen aktivierbar und deaktivierbar sein, sollten Sie auf jeden Fall ein *QCheckBox*-Objekt für jede Option benutzen.

Bei einer großen Anzahl von Optionen in *QListBox* und *QComboBox* kann es wichtig werden, die Optionen nach einer gut nachvollziehbaren Ordnung zu sortieren, damit der Anwender die gewünschte Option schneller findet. Bei Optionen wie z.B. Wochentagen bietet sich die natürliche Sortierung (Montag, Dienstag, Mittwoch, ...) an, ansonsten kann man sich oft mit einer alphabetischen Sortierung behelfen.

Eine spezielle Farbauswahlbox stellt die KDE-Bibliothek mit der Klasse *KColorCombo* zur Verfügung. Sie ist von *QComboBox* abgeleitet und zeigt im Fenster eine ausgewählte Farbe an. In den Optionen sind alle Standardfarben von KDE enthalten (siehe Anhang C, *Die KDE-Standardfarbpalette*) und können dort ausgewählt werden. Alternativ kann man auch den Eintrag »Custom« anklicken, wobei dann ein Farbdialog der Klasse *KColorDialog* erscheint, in dem man eine eigene Farbe einstellen kann.



Abbildung 3-76 *KColorCombo*

Die KDE-Bibliothek enthält weiterhin ein GUI-Element zur Auswahl einer Farbe aus einer Tabelle: *KColorCells* (siehe Abbildung 3.77). Durch Anklicken kann man eine Farbe auswählen. Dadurch, dass alle Farben gleichzeitig angezeigt werden, ist diese Klasse übersichtlicher als *QColorCombo*, braucht aber auch mehr Platz. Leider ist nur schwer zu erkennen, welche der Farben ausgewählt wurde.

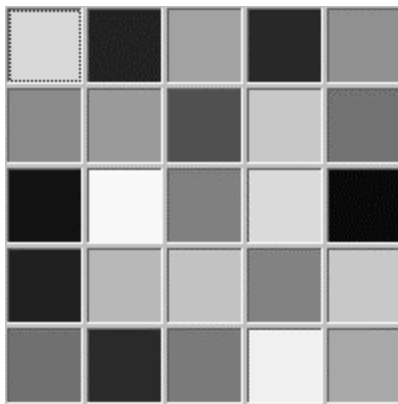


Abbildung 3-77 *KColorCells*

Eine Klasse, die ein Auswahlelement realisiert, wurde bereits im Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*, benutzt: *QPopupMenu* (siehe Abbildung 3.78). Hierbei handelt es sich um ein Fenster, das nur bei Bedarf geöffnet wird. Man kann es daher auch nicht als GUI-Element in einen Dialog einfügen.

Dieses Element bleibt meist auf die Spezialanwendungen Menüleiste und Kontext-Popup-Menü beschränkt. Auch von dieser Klasse gibt es eine KDE-Version, *KPopupMenu*, die von *QPopupMenu* abgeleitet ist und sich von dieser Klasse nur dadurch unterscheidet, dass man ihr einen Titel hinzufügen kann, der als erster (nicht aktivierbarer) Eintrag oben im Popup-Menü steht (siehe Abbildung 3.79).



Abbildung 3-78 QPopupMenu

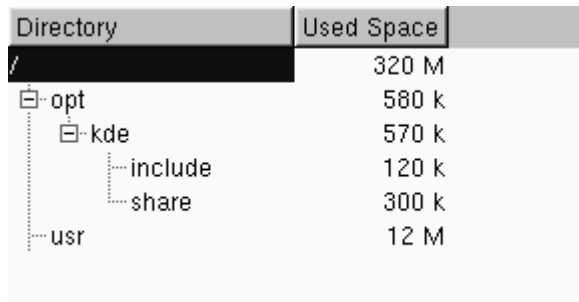


Abbildung 3-79 KPopupMenu

Eine extrem mächtige Klasse hat Qt mit *QListView* herausgebracht. Diese Universalklasse stellt Einträge in einer Tabellenform dar. Die einzelnen Spalten einer Tabelle sind in der Größe veränderbar, und die Reihenfolge der Spalten kann (durch Ziehen der Spaltenüberschrift mit der Maus) vertauscht werden. Durch Klicken auf die Spaltenüberschrift wird ein Signal aktiviert, mit dem man zum Beispiel die Sortierungsreihenfolge ändern kann. Die Einträge können in einer Baumstruktur angeordnet werden, die auch in der Darstellung durch Einrückung zum Ausdruck kommt (siehe Abbildung 3.80). Einzelne Teile des Baums kann man »kollabieren« lassen – also ausblenden –, um die Tabelle übersichtlicher zu machen. Auch einzelne Spalten können ausgeblendet werden.

Die Zahl der Spalten ist ebenso wie die Anzahl der Einträge nicht beschränkt. Passen nicht mehr alle Einträge in das Fenster, wird ein Rollbalken hinzugefügt, mit dem man die Einträge nach oben und unten verschieben kann. Auch wenn die Einträge zu breit für das Fenster sind, wird automatisch ein horizontaler Rollbalken ergänzt.

Wie bei *QListBox* kann ein Element markiert werden, wodurch alle anderen deaktiviert werden. Im Modus *MultiSelection* kann man dagegen mehrere Objekte markieren und die Markierung durch einen weiteren Mausklick wieder aufheben.



Directory	Used Space
/	320 M
└─ opt	580 k
└─ kde	570 k
└─ include	120 k
└─ share	300 k
└─ usr	12 M

Abbildung 3-80 *QListView*

Die Anwendungsgebiete von *QListView* sind vielfältig wie für kaum eine andere Klasse. Für jedes eingetragene Element kann man festlegen, ob es selektierbar sein soll oder nicht. Setzt man alle Elemente auf »nicht selektierbar«, so kann man *QListView* als reines Anzeigeelement benutzen. Wenn man die Möglichkeit ignoriert, mit Baumstrukturen zu arbeiten, kann man *QListView* für Tabellen jeglicher Art verwenden: Adresslisten, Druckerstatuslisten, Dateilisten, Datensätze, Zuordnungstabellen usw. Die Baumstruktur kann man einsetzen, um zum Beispiel einen Verzeichnisbaum anzuzeigen, um Hierarchien darzustellen oder um lange Listen in sinnvolle Gruppen zu unterteilen. Auch in diesem Fall können die Elemente natürlich mehrere Spalten belegen. Neben Text können die Felder der Tabelle auch Bilder (in Form von *QPixmap*-Objekten) enthalten.

Die Klasse *QIconView* (siehe Abbildung 3.81) zeigt in einem Fenster eine beliebige Anzahl von Icons, optional mit einer Textbeschriftung. Dieses Widget wird oft in Dateimanagern eingesetzt, in denen jedes Icon eine Datei repräsentiert, kann aber auch für andere Zwecke eingesetzt werden. *QIconView* bietet die Möglichkeit, die Icons mit der Maus innerhalb des Fensters zu verschieben oder auch per Drag&Drop in ein anderes Fenster zu schieben. Auch die Umbenennung der Beschriftung durch Anklicken ist möglich, so wie es im Explorer von Microsoft üblich ist.

Wiederum eine Spezialklasse ist *KDatePicker*, eine Klasse aus der KDE-Bibliothek, mit der der Anwender auf einem Kalenderblatt ein Datum auswählen kann. Er kann dabei das Jahr und den Monat mit Buttons aussuchen.



Abbildung 3-81 QIconView



Abbildung 3-82 KDatePicker

3.7.6 Eingabeelemente

Die Gruppe der Eingabeelemente zeichnet sich dadurch aus, dass per Tastatur ein Text oder eine Zahl eingetippt oder verändert werden kann.

Das einfachste Eingabeelement ist eine einzelne Zeile, die mit beliebigem Text gefüllt werden kann. Sie wird durch die Klasse *QLineEdit* realisiert (siehe Abbildung 3.83). Die Klasse besitzt Signale, die bei jeder Änderung des Textes oder beim Druck auf die Return-Taste Nachrichten versenden. Die Länge der Eingabe ist nicht begrenzt. Wenn der Text nicht in das Fenster passt, wird er am linken Rand aus dem Fenster hinausgeschoben, bleibt aber natürlich weiterhin gespeichert.

In einem *QLineEdit*-Objekt kann man Teile des Textes markieren oder (wie unter Unix-Systemen üblich) mit der mittleren Maustaste einfügen. Hier benutzt Qt allerdings den neueren Standard, der besagt, dass das Einfügen mit der mittleren Maustaste nicht an der aktuellen Einfügeposition, sondern an der aktuellen Mausposition geschieht.

Mit der Zusatzklasse *QValidator* kann man einem *QLineEdit*-Objekt ein Kontrollobjekt zuordnen, das überprüft, ob der eingegebene Text ein bestimmtes Format

erfüllt. Zwei bereits definierte Klassen sind zum Beispiel *QIntValidator* und *QDoubleValidator*, die nur die Eingabe einer ganzzahligen bzw. reellen Zahl in einem festlegbaren Bereich erlauben. In der KDE-Bibliothek gibt es die Klasse *KDateValidator*, die prüft, ob das eingegebene Datum gültig ist.

Burkhard Lehner

Abbildung 3-83 *QLineEdit*

Um mehrzeiligen Text einzugeben, gibt es unter Qt das GUI-Element *QMultiLineEdit* (siehe Abbildung 3.84). Auch hier ist die Menge des Textes, der eingegeben werden kann, nicht begrenzt. Falls die Fenstergröße nicht ausreicht, werden Rollbalken hinzugefügt, mit denen man den Inhalt des Fensters verschieben kann. *QMultiLineEdit* ist allerdings nicht optimiert, so dass bei zu großen Texten (über 10 kByte) die Bedienung spürbar langsamer wird.

Auch in diesem Objekt kann man Texte markieren und mit der mittleren Maustaste einfügen. Es kann Tabulatoren anzeigen und unterstützt einen automatischen Zeilenumbruch. Es kann markierte Texte per Drag&Drop innerhalb des Widgets, in andere Widgets oder aus anderen Widgets in das eigene verschieben. Es bietet einen Undo-Mechanismus, mit dem man die letzten Schritte rückgängig machen kann. In Kapitel 3.5.6, *Applikation für ein Dokument*, und 3.5.7, *Applikation für mehrere Dokumente*, wird das *QMultiLineEdit*-Widget eingesetzt, um einen einfachen, aber doch sehr mächtigen Texteditor zu implementieren. Es kann allerdings nicht einzelne Bereiche des Texts andersfarbig oder in einem anderen Zeichensatz darstellen. Für ein komplexes Textverarbeitungssystem ist die Klasse daher nicht geeignet. Es ist allerdings ein neues Widget namens *QTextEdit* in Vorbereitung, das wahrscheinlich in einer der nächsten Qt-Versionen verfügbar sein wird. Dieses Widget kann dann Text im RichText-Format nicht nur anzeigen, sondern auch editieren.

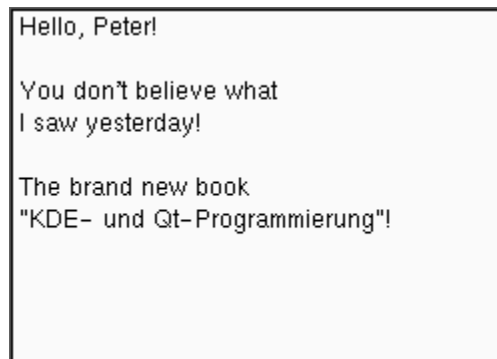


Abbildung 3-84 *QMultiLineEdit*

Wenn man ein *QMultiLineEdit*-Objekt in den Modus »ReadOnly« setzt, so stellt es keinen Cursor mehr dar und nimmt keine Eingaben von der Tastatur mehr entgegen. Der Programmierer kann aber weiterhin mit den Methoden *append* und *insertAt* Text hinzufügen oder ändern. Der Text kann weiterhin markiert werden, und die Rollbalken lassen sich bedienen. Damit eignet sich das Objekt auch ausgezeichnet als Anzeigeelement, zum Beispiel für einen längeren Text oder als Fenster für hereinkommende Meldungen oder Fehler.

3.7.7 Verwaltungselemente

Einige GUI-Elemente dienen dazu, andere Elemente zu kontrollieren oder anzuordnen. Drei der Klassen haben wir bereits in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster* kennen gelernt: Die Klassen *QVBox*, *QHBox* und *QGrid* ordnen ihre Unter-Widgets automatisch untereinander, nebeneinander oder in einer Tabelle an.

Ein anderes Element ist *QSplitter*. Es teilt das Fenster in zwei Teile auf, in denen die beiden ersten eingefügten Unter-Widgets dargestellt werden. Zwischen den beiden Teilfenstern wird eine Trennlinie dargestellt, die mit der Maus bewegt werden kann, so dass eines der Fenster vergrößert und das andere verkleinert wird. *QSplitter* kann dabei das Fenster in zwei (oder mehr) nebeneinander oder übereinander liegende Teilfenster trennen. Ein typisches Anwendungsbeispiel ist ein Editor in einer integrierten Umgebung, in der in einem zweiten Fenster am unteren Bildschirmrand Fehlermeldungen angezeigt werden. Je nach Situation kann der Anwender dann das Editor- oder das Fehlerfenster vergrößern. Auch der Filemanager *Konqueror* benutzt diese Technik, um im linken Fensterteil einen Verzeichnisbaum und im rechten die Liste der Dateien oder deren Inhalt darzustellen.

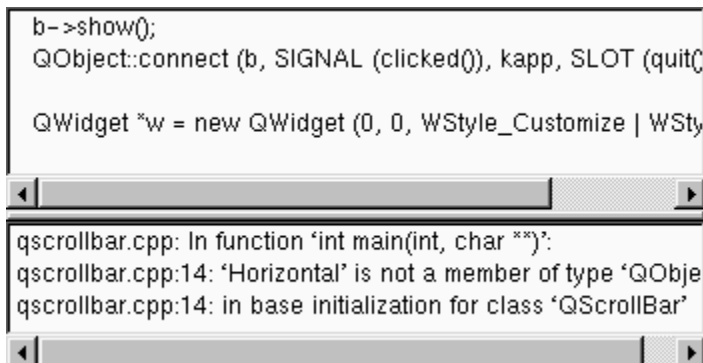


Abbildung 3-85 QSplitter

Ein GUI-Element, das mehrere Seiten alternativ in einem Fenster darstellt, ist die Klasse *QWidgetStack*. Sie stellt immer nur eines ihrer Unter-Widgets dar (die anderen sind im Modus »versteckt«, siehe Kapitel 3.2.3, *Die wichtigsten Widget-Eigenschaften*). Auf diese Weise wird zum Beispiel in einem *QTabWidget*-Element erreicht, dass durch Klicken auf einen der Tabs am oberen Bildschirmrand im darunter liegenden Fenster die entsprechende Seite (und nur diese) angezeigt wird. Man kann *QWidgetStack* aber zum Beispiel auch benutzen, um in einem separaten Fenster einen kontextabhängigen Optionendialog aufzubauen. Je nach ausgewählten Elementen wird das Fenster angezeigt, das die für dieses Objekt spezifischen Optionen festlegt.

Das *QTabWidget*-Element aus der Qt-Bibliothek erzeugt, wie oben bereits erwähnt, ein mehrseitiges Fenster. Am oberen Rand des Fensters werden Karteikartenreiter (*Tabs*) dargestellt, und ein Klick auf einen der Reiter bringt die entsprechende Seite nach vorn. Dieses Element kommt immer dann zum Einsatz, wenn die Zahl der einstellbaren Optionen zu groß ist, um sinnvoll in ein Fenster zu passen. Wichtig ist, dass bei der Aufteilung in mehrere Seiten die Gruppierung der Optionen einfach nachvollziehbar bleibt und dass die Beschriftung der Tabs kurz und prägnant ist. Wenn der Anwender eine bestimmte Option ändern will, muss er direkt die richtige Seite finden können, ohne alle Seiten durchsuchen zu müssen. Bei einer sinnvollen Aufteilung passiert es daher oft, dass einige Seiten voller sind als andere. Auf keinen Fall sollte man dann versuchen, Optionen von einer vollen Seite auf eine leerere zu verschieben oder zwei leerere Seiten zu einer zusammenzufassen, wenn sie inhaltlich nicht zusammengehören. Leerere Seiten sollten auch nicht durch unnötiges »Füllmaterial« oder zusätzlichen Freiraum mitten im Fenster aufgebläht werden. Lassen Sie alle Bedienelemente in ihrer natürlichen Größe am oberen Rand, und lassen Sie den überschüssigen Freiraum am unteren Rand stehen. *QTabWidget* wird zum Beispiel auch in der Klasse *QTabDialog* benutzt (siehe Kapitel 3.8.4, *Optionsdialoge*).

Die Zahl der Karteikartenreiter sollte nicht zu groß sein. Bei mehr als zwei Reihen dauert die Suche nach der richtigen Seite unter Umständen wieder zu lange. Da sich bei mehreren Reihen außerdem die Reihenfolge der oberen und unteren Reiter oft vertauscht, kann sich der Anwender ihre Position nicht gut merken. Eine alternative Möglichkeit, eine so große Anzahl von Unterfenstern mit einem *QListView*-Objekt zu verwalten, wird in Kapitel 3.8.4, *Optionsdialoge*, gezeigt.

3.7.8 Dialoge

Dialoge sind *Toplevel*-Widgets, die geöffnet werden, wenn eine komplexere Eingabe erfolgen soll. Der Anwender schließt sie meist mit einem OK- oder ABBRECHEN-Button, nachdem er alle Einstellungen vorgenommen hat. Bei modalen Dialogen (das ist die normale Anwendung) sind die anderen Fenster der Applikation so lange nicht bedienbar, bis das Dialogfenster wieder geschlossen wird.

Die Grundlage für fast alle Dialoge ist die Klasse *QDialog*. Sie ist von *QWidget* abgeleitet. Die GUI-Elemente, die im Dialog eingestellt werden sollen, werden einfach als Unter-Widgets in das *QDialog*-Objekt eingefügt. Im Gegensatz zu *QWidget* ist ein *QDialog*-Objekt in jedem Fall ein *Toplevel-Widget*. Zwar besitzt der Konstruktor einen Parameter *parent*, dieser Parameter wird hier jedoch eingesetzt, um zu kennzeichnen, in welchem Fenster das Dialogfenster zentriert werden soll. Es gibt zwei verschiedene Modi, in denen der Dialog ausgeführt werden kann. Im Modus »nicht-modal« (*modeless* oder *nonmodal*) wird der Dialog wie ein normales *QWidget*-Fenster ausgeführt. Der einzige Zusatz ist der *Default-Button-Mechanismus*, der etwas weiter unten beschrieben wird. Im Modus »modal« dagegen ist das Dialogfenster das einzige Fenster des Programms, das Maus- und Tastatur-Events zugeschickt bekommt. Alle anderen Fenster der Applikation können so lange nicht benutzt werden, bis das Dialogfenster wieder geschlossen wird. Andere Events vom X-Server (zum Beispiel *repaint*-Events) und Timer-Events werden weiterhin an die entsprechenden Objekte geleitet. Das *QDialog*-Objekt wird im modalen Modus mit der Methode *exec* gestartet, die eine eigene, lokale Event-Schleife startet, in der das Objekt alle hereinkommenden Events filtert. Erst nach dem Aufruf der Methode *done* kehrt das Programm aus dieser Methode zurück.

QDialog bietet einen Default-Button-Mechanismus an. Eines der *QPushButton*-Elemente im Fenster kann man mit der Methode *QPushButton::setDefault()* zum Standardbutton machen. Als Zeichen dafür wird es mit einem etwas breiteren Rahmen dargestellt. Wird nun innerhalb des Fensters die Return-Taste betätigt, so aktiviert das automatisch den Standardbutton. Weiterhin reagiert ein *QDialog*-Element auf die Eingabe der [Esc]-Taste und schließt das Fenster mit einem Rückgabewert, der einen Abbruch signalisiert.

Modale Bildschirmdialoge können eingesetzt werden, um zum Beispiel ein Fenster mit einer wichtigen Fehlermeldung zu öffnen, auf die der Benutzer zunächst reagieren muss, bevor er mit dem Programm weiterarbeiten kann. Sie werden außerdem oft benutzt, um eine komplexe Eigenschaft festzulegen oder auszuwählen, beispielsweise eine Datei oder eine Farbe. Wählt der Anwender zum Beispiel den Befehl **SPEICHERN** aus, so soll sich zunächst ein Fenster öffnen, in dem er den Dateinamen festlegen kann. Erst danach wird die Datei gespeichert, und die anderen Fenster können wieder bedient werden.

Eine andere Klasse, *QSemimodal*, erzeugt – genau wie *QDialog* – ein Fenster mit einem Dialog, der modal oder nicht-modal sein kann. Interessant ist aber auch hier vor allem der modale Modus. Er verhält sich ganz analog zu *QDialog*, arbeitet jedoch anders. Anstatt eine eigene lokale Event-Schleife zu starten, kehrt *QSemimodal* sofort vom Aufruf von *exec* zum Aufrufer zurück. Das Fenster bleibt jedoch geöffnet, und *QSemimodal* verwirft alle Maus- und Tastatur-Events, die für andere Fenster bestimmt sind, so dass diese nicht mehr bedient werden können.

So kann das Programm noch Berechnungen vornehmen, während das Dialogfenster geöffnet ist. Benutzt wird *QSemimodal* unter anderem im *QProgressDialog*, der etwas weiter unten beschrieben wird.

Es gibt bereits viele vordefinierte Dialogfenster in den Qt- und KDE-Bibliotheken, mit deren Hilfe man verschiedene Werte einstellen kann. Im Folgenden wollen wir die wichtigsten kurz erläutern.

Die Klasse *QMessageBox* stellt ein Fenster auf dem Bildschirm dar, das ein Icon, eine Meldung und ein bis drei Buttons mit frei wählbarer Beschriftung besitzt (siehe Abbildung 3.86). Diese Klasse kann sehr einfach dazu benutzt werden, um dem Anwender eine wichtige Meldung zu präsentieren oder ihm eine wichtige Frage zu stellen (»Wollen Sie die Festplatte wirklich formatieren?«). Am einfachsten geht das mit Hilfe einer der statischen Klassen-Methoden *about*, *information*, *warning* oder *critical*. Diese Methoden erzeugen automatisch ein *QMessageBox*-Objekt, füllen es mit den angegebenen Texten, stellen es dar, warten auf die Antwort, schließen das Fenster wieder, löschen das *QMessageBox*-Objekt und geben als Rückgabewert die Nummer des angeklickten Buttons zurück. Der Text kann im RichText-Format (einer Untermenge von HTML, siehe auch *QLabel* in Kapitel 3.7.1, *Statische Elemente*) angegeben werden.



Abbildung 3-86 *QMessageBox* mit drei Buttons

Auch die KDE-Bibliothek enthält eine Klasse mit dieser Funktionalität: *KMessageBox*. *KMessageBox* ist eine Klasse, die ausschließlich statische Methoden enthält, mit denen man sehr einfach typische Meldungsfenster öffnen lassen kann. Es stehen Methoden für Warnungen, Informationen, Ja-Nein-Abfragen und Ähnliches zur Verfügung. Auch *KMessageBox* kann den Informationstext in RichText darstellen, so dass er übersichtlich formatiert werden kann. Besonders interessant ist das Informationsfenster (*KMessageBox::information*, siehe Abbildung 3.87), das zusätzlich mit einer Check-Box die Möglichkeit gibt, dieses Fenster in Zukunft zu unterdrücken. Dieser Zustand wird automatisch in den KDE-Konfigurationsdateien abgelegt, ist also auch beim nächsten Programmstart vorhanden.



Abbildung 3-87 KMessageBox-Informationsfenster

Die Klasse *QProgressDialog* erzeugt ein Fenster mit einem Fortschrittsbalken (siehe Abbildung 3.88). Hiermit kann dem Anwender gezeigt werden, wie lange eine aufwendige Berechnung oder Ein-/Ausgabe-Operation noch dauern wird. Wie bereits oben erwähnt wurde, ist *QProgressDialog* nicht von *QDialog*, sondern von *QSemimodal* abgeleitet. Nach dem Aufruf von *exec* werden die anderen Fenster blockiert, das Programm arbeitet aber weiter. Während der Berechnung oder Ein-/Ausgabe sollte in regelmäßigen Abständen *QApplication::processEvents* aufgerufen werden, damit die Maus abgefragt wird. Anschließend kann getestet werden, ob der Anwender den Abbruch-Button gedrückt hat. Weiterhin muss dem *QProgressDialog*-Objekt natürlich regelmäßig mitgeteilt werden, wie weit die Arbeit jetzt vorangeschritten ist.

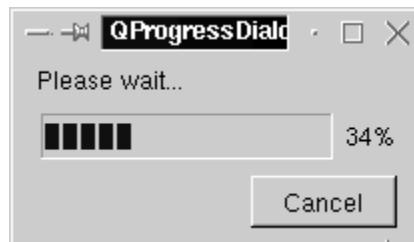


Abbildung 3-88 QProgressDialog

QProgressDialog öffnet das Fenster erst, wenn eine erste Abschätzung der benötigten Gesamtzeit eine Dauer von mehr als drei Sekunden ergibt. So wird verhindert, dass das Fenster nur für eine sehr kurze Zeit sichtbar wird, was den Anwender verunsichern könnte.

Ein wichtiger Dialog in fast jedem Programm ist das Fenster zur Festlegung eines Dateinamens. Auch hier gibt es zwei Klassen, die diese Aufgabe übernehmen: *QFileDialog* und *KFileDialog* (siehe Abbildung 3.89 bzw. 3.90). Beide öffnen ein Fenster und lassen den Benutzer das Verzeichnis und einen Dateinamen auswäh-

len. Dazu kann man bei beiden Dateifilter einstellen. Beide Dialoge können zum größten Teil mit der Maus bedient werden, sofern nur eine existierende Datei ausgewählt wird und nicht ein neuer Dateiname einzugeben ist. In *KFileDialog* kann man sogar Dateien auswählen, die auf einem HTTP- oder FTP-Server liegen. In Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*, wird genau beschrieben, wie man eine solche Datei lädt oder speichert. Beachten Sie, dass Sie die Bibliothek *kfile* zu Ihrem Programm hinzulinken müssen, wenn Sie *KFile* nutzen wollen.

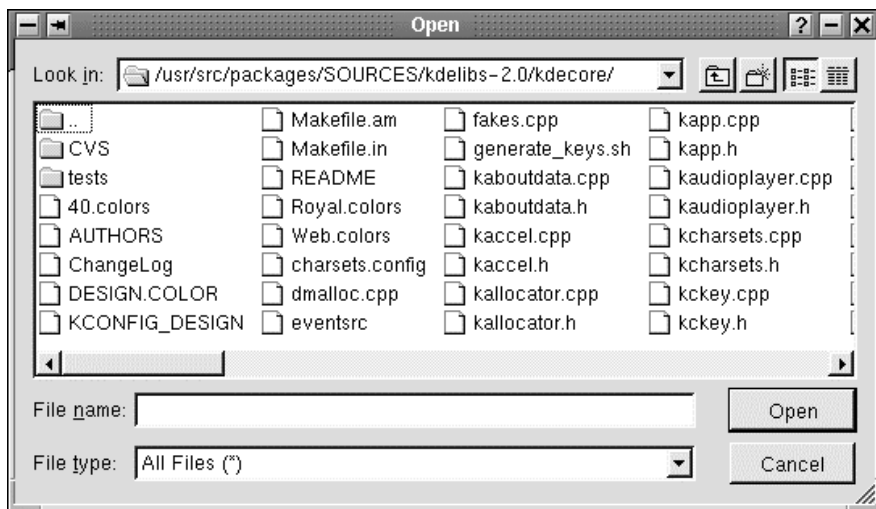


Abbildung 3-89 QFileDialog

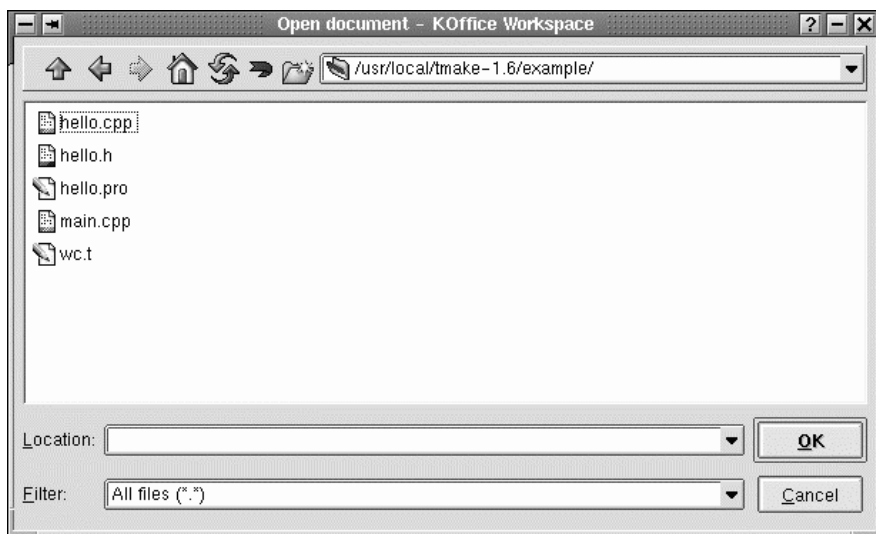


Abbildung 3-90 KFileDialog

Mit den Klassen *QColorDialog* und *KColorDialog* kann der Anwender eine Farbe wählen. Die Farbe kann in beiden Dialogen im HSV- oder im RGB-Farbmodell festgelegt werden (siehe auch Kapitel 4.1, *Farben unter Qt*). Welcher der beiden Klassen man den Vorzug geben will, ist wieder Geschmackssache. Abbildung 3.91 und Abbildung 3.92 zeigen die beiden Klassen in Aktion.

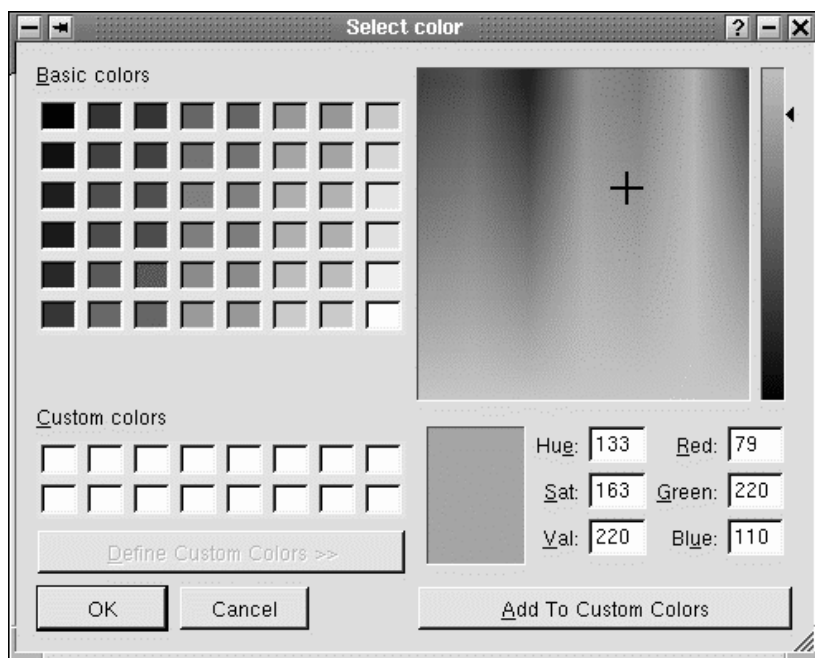


Abbildung 3-91 QColorDialog

Auch für die Auswahl eines Zeichensatzes stehen zwei Klassen zur Verfügung: *QFontDialog* und *KFontDialog*. Beide ermöglichen dem Anwender eine einfache Auswahl des Zeichensatzes und der Schriftparameter wie Schriftgröße, fette und/oder kursive Darstellung. Beide Dialoge zeigen die eingestellte Schrift als Beispieltext.

Die Klasse *QPrintDialog* zeigt in einem Fenster alle im System vorhandenen Drucker an, aus denen der Anwender auswählen kann (siehe Abbildung 3.95). Alternativ kann der Anwender den Ausdruck in eine Datei umleiten lassen. Weiterhin kann man einstellen, welche Seiten eines Dokuments gedruckt werden sollen, wie das Papierformat und wie die Orientierung ist. Als Endergebnis erzeugt das Dialogfenster ein Objekt der Klasse *QPrinter*, das direkt zum Zeichnen auf dem Drucker benutzt werden kann.

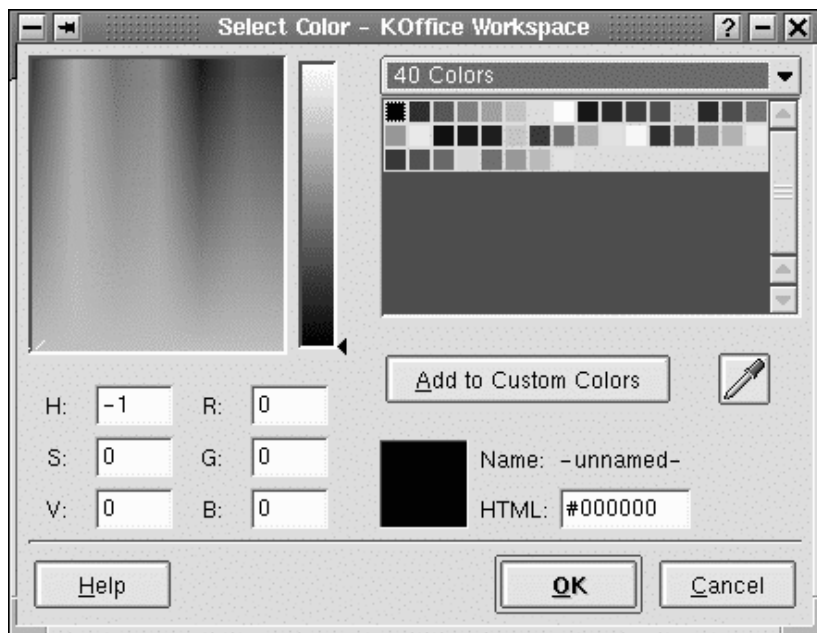


Abbildung 3-92 KColorDialog

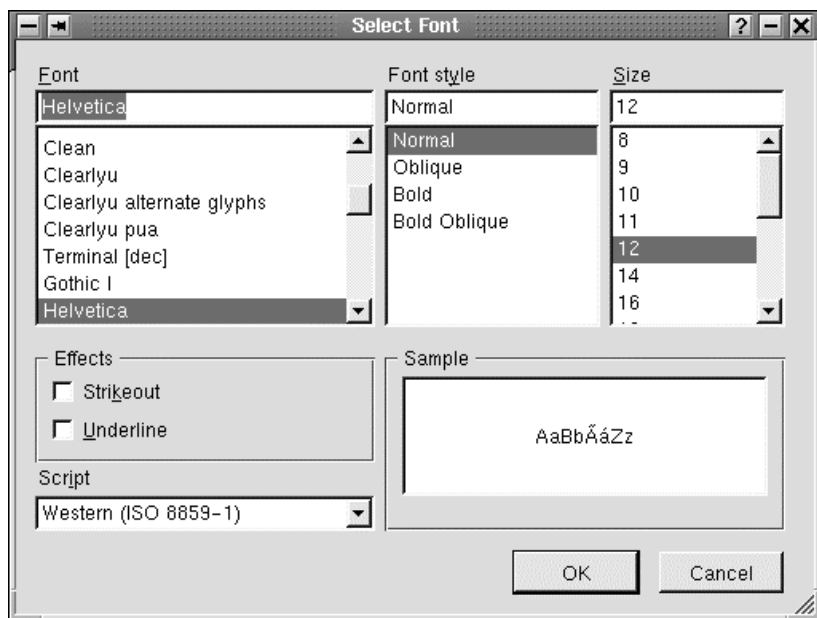


Abbildung 3-93 QFontDialog

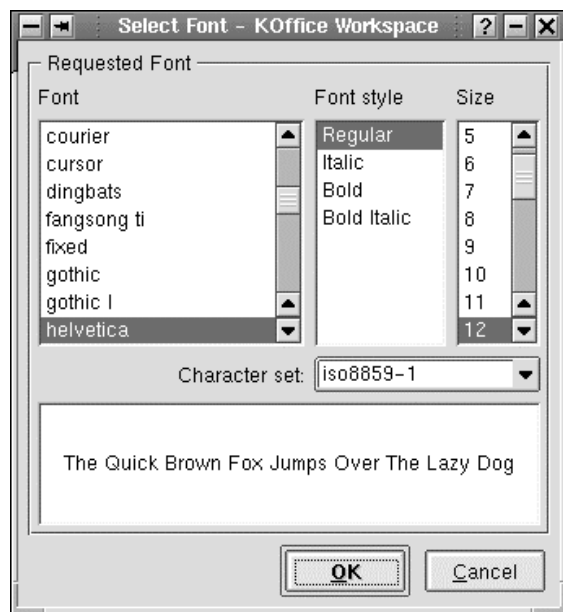


Abbildung 3-94 KFontDialog

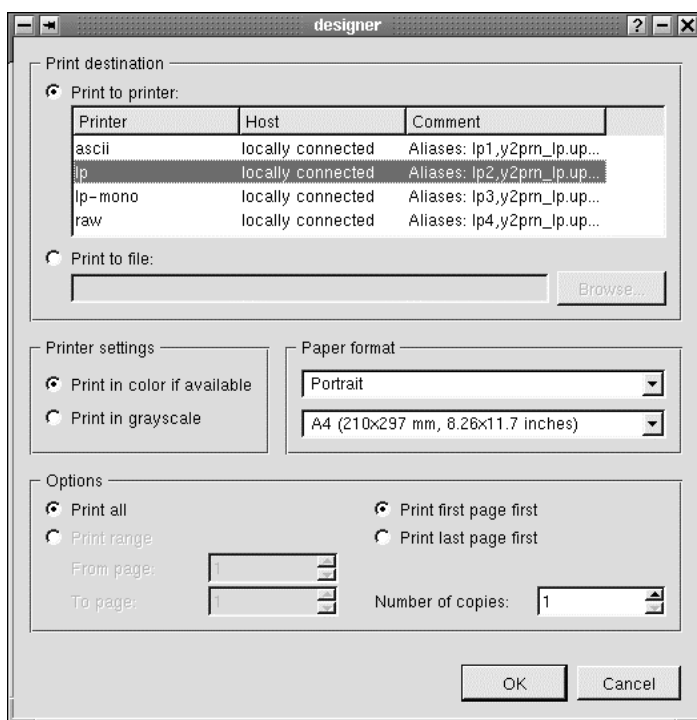


Abbildung 3-95 QPrintDialog

Eine weitere spezielle Klasse ist *KIconDialog*. Sie erzeugt ein Fenster, in dem der Anwender in verschiedenen Verzeichnissen nach Icons suchen kann, von denen er eines auswählen kann (siehe Abbildung 3.96).

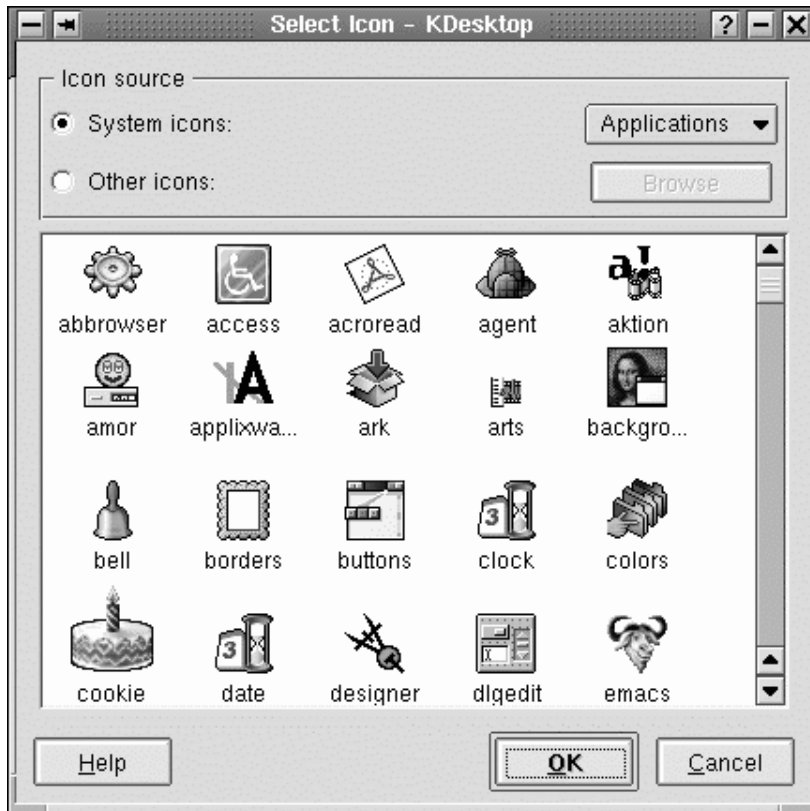


Abbildung 3-96 *KIconDialog*

3.8 Der Dialogentwurf

Der größte Teil der Interaktion zwischen Benutzer und Applikation findet in so genannten Dialogen statt. Das sind Fenster auf dem Bildschirm, in denen in Bedienelementen Optionen eingestellt, Auswahlen getroffen und Aktionen gestartet werden. Für die gängigsten Aufgaben gibt es in KDE und Qt bereits vordefinierte Dialoge. Eine Auflistung der fertigen Dialogklassen finden Sie in Kapitel 3.7.8, *Dialoge*.

In den meisten Fällen benötigt man aber sehr spezielle Dialoge, die auf das Problem zugeschnitten sind, das die Applikation lösen soll. Der Entwurf eines guten Dialogs ist jedoch mehr als nur die Zusammenstellung aller benötigten Elemente.

Dem Programmierer stellt sich hier die schwierige Aufgabe, den Dialog intuitiv und übersichtlich anzuordnen. Ein gutes Programm zeichnet sich dadurch aus, dass es ohne großen Lernaufwand schnell zu bedienen ist.

Der Grundaufbau eines Dialogs besteht meist aus einem *Toplevel-Widget*, das als umrahmendes Fenster dient und die Bedienelemente in Form von Unter-Widgets enthält. Für das *Toplevel-Widget* wird am häufigsten eine der folgenden Klassen benutzt:

- *QWidget* – Schon dieses einfache Element erlaubt die Anordnung der Unter-Widgets mit Layout-Klassen.
- *QFrame* – Diese Klasse hat die gleichen Möglichkeiten wie *QWidget*, es besteht hier jedoch außerdem noch die Möglichkeit, einen zusätzlichen Rahmen um den Widget-Rand zu zeichnen. Da aber das *Toplevel-Widget* ohnehin einen Rahmen vom Window-Manager zugewiesen bekommt, ist das nicht nötig, meist sogar störend. *QFrame* wird jedoch oft eingesetzt, um innerhalb eines anderen *Toplevel-Widgets* Bedienelemente zu gruppieren.
- *QDialog* – Die häufigste Anwendung dieser Klasse als *Toplevel-Widget* ist die des »modalen« Dialogs. Das heißt, solange das Dialogfenster geöffnet ist, können nur in diesem Fenster Einstellungen vorgenommen werden. Dazu startet Qt eine eigene Event-Schleife, in der nur die Events für dieses Fenster abgearbeitet werden. Events für andere Fenster werden ignoriert. Modale Dialoge werden häufig für Entscheidungen benutzt, die sofort getroffen werden müssen, oder wichtige Meldungen, die unbedingt beachtet werden müssen, z.B. Abfragen wie: »Die aktuelle Datei wurde noch nicht gespeichert! Soll die Datei gespeichert werden?« Die Klasse *QDialog* kann allerdings auch nicht-modal (*modeless* oder *nonmodal*) benutzt werden. In diesem Fall bleiben auch die anderen Fenster bedienbar. Zusätzlich bietet die Klasse *QDialog* die Möglichkeit, das Fenster immer über den anderen Fenstern der Applikation zu halten, so dass es nicht verdeckt wird. Außerdem ist in der Klasse der so genannte Default-Button-Mechanismus realisiert. Innerhalb eines *QDialog*-Objekts kann ein Button als Default-Button deklariert werden. Wird dann innerhalb des Fensters die Return-Taste betätigt, wird automatisch dieser Button aktiviert.

3.8.1 Ein erstes Beispiel für einen Dialog

Wir wollen zunächst noch einmal unser Beispiel aus den vorangegangenen Kapiteln benutzen. Diesmal konzentrieren wir uns jedoch auf die Anbindung des Fensters an den Rest des Programms: Auf dem Bildschirm wird ein Statusfenster erzeugt, in dem Meldungen (zum Beispiel über den Programmstatus oder Fehlermeldungen) angezeigt werden. Das Fenster soll neben dem Anzeigebereich für die Meldungen zwei Buttons enthalten, den einen, um die Meldungen zu

löschen, den anderen, um das Fenster zu verstecken. Bei jeder hereinkommenden Meldung soll das Fenster wieder dargestellt werden (falls es versteckt war) und die neue Meldung unten angehängt werden.

Zur Darstellung der Meldungen benutzen wir ein Widget der Klasse *QMultiLine Edit*, eine sehr mächtige Klasse zum Darstellen und Editieren mehrzeiliger Texte. Da wir nur den Text anzeigen lassen wollen – ohne die Möglichkeit, ihn zu editieren –, stellen wir das Widget auf *ReadOnly*. Die Funktionalität ist aber weiterhin sehr hoch: Die alten Meldungen werden nach oben aus dem Widget gescrollt, mit dem Rollbalken kann man sie aber wieder sichtbar machen. Der Text kann mit der Maus markiert und in die Zwischenablage kopiert werden. Die beiden Buttons sind Widgets der Klasse *QPushButton*.

Zur Erzeugung des Fensters können wir folgenden Programmcode benutzen (beachten Sie, dass er noch nicht optimal ist und wir noch einige Änderungen vornehmen wollen):

```
// Widgets erzeugen (ein Toplevel-Widget und
// drei Unter-Widgets)
QWidget *messageWindow = new QWidget ();
QMultiLineEdit *messages =
    new QMultiLineEdit (messageWindow);
QPushButton *clear =
    new QPushButton ("Clear", messageWindow);
QPushButton *hide =
    new QPushButton ("Hide", messageWindow);

// Layout erzeugen und starten
QVBoxLayout *topLayout =
    new QVBoxLayout (messageWindow);
QHBoxLayout *buttonsLayout = new QHBoxLayout ();
topLayout->addWidget (messages);
topLayout->addLayout (buttonsLayout);
buttonsLayout->addWidget (clear);
buttonsLayout->addWidget (hide);

// Signale der Buttons mit passenden Slots verbinden
QObject::connect (clear, SIGNAL (clicked ()),
                  messages, SLOT (clear ()));
QObject::connect (hide, SIGNAL (clicked ()),
                  messageWindow, SLOT (hide ()));
```

Nach der Erzeugung des Toplevel-Widgets *messageWindow* (ohne Vater-Widget) werden die drei Unter-Widgets *messages*, *clear* und *hide* mit *messageWindow* als Vater-Widget erzeugt. Der Titeltext des Dialogfensters wird mit *setCaption* gesetzt, und *messages* wird auf *ReadOnly* gesetzt, da die Meldungen nur angezeigt, aber

nicht editiert werden sollen. Die Position und Größe der Unter-Widgets wird von den Layout-Klassen *topLayout* und *buttonsLayout* kontrolliert (siehe Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*).

Die *clicked*-Signale der beiden Buttons werden mit passenden Slots verbunden. Praktischerweise gibt es bereits Slots, die genau die gewünschten Aktionen ausführen, so dass wir keine eigenen Slots definieren müssen.

Eine neue Meldung wird dann durch folgende zwei Zeilen in das Fenster eingefügt:

```
messages->append ("Initialisierung abgeschlossen\n");  
messageWindow->show ();
```

Das Ergebnis sieht dann etwa wie in Abbildung 3.97 aus.

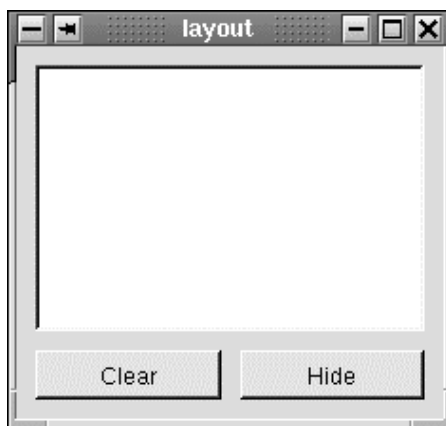


Abbildung 3-97 Dialogfenster zur Anzeige von Meldungen

Was kann dieses Programmstück?

- Sobald eine Meldung nach dem Schema oben eingefügt wird, wird das Fenster sichtbar.
- Auch bei weiteren Meldungen bleibt es sichtbar (weitere Aufrufe von *show* bewirken nichts).
- Wenn Sie CLEAR anklicken, werden die Meldungen gelöscht, das Fenster bleibt aber sichtbar. (Soll es automatisch nach dem Löschen versteckt werden, verbindet man einfach das *clicked*-Signal des *clear*-Buttons zusätzlich noch mit dem Slot *hide* von *messageWindow*.)
- Das Anklicken von HIDE macht das Fenster unsichtbar. Das geschieht auch, wenn das Fenster mit dem X-Button des Window-Managers geschlossen wird. Es bleibt jedoch erhalten und wird beim nächsten Aufruf von *show* wieder sichtbar.

- Am Ende des Programms reicht ein *delete messageWindow*, um alles wieder zu löschen. (Die Unter-Widgets werden automatisch gelöscht.)

Was sollte noch verbessert werden?

- Das ganze Objekt besteht aus vier Variablen, von denen zwei benutzt werden müssen, um Meldungen anzuzeigen. Die beiden anderen können schnell zu Namenskonflikten führen. Besser wäre es, nur eine Klasse zu haben, die im Konstruktor das Fenster anlegt und die mit einer einfachen Methode angesteuert werden kann.

Dieses Problem werden wir im nächsten Abschnitt beheben.

3.8.2 Eine neu definierte Dialogklasse

Um nur ein einzelnes Objekt für unser Meldungsfenster zu haben, definieren wir eine neue Klasse: *MessageWindow*. Diese Klasse wird von der Klasse des benutzten Toplevel-Widgets – also hier von *QWidget* – abgeleitet. Im Konstruktor werden die Unter-Widgets erzeugt, die richtige Platzierung wird vorgenommen, und alle Signal-Slot-Verbindungen werden hergestellt.

Es ist in den allermeisten Fällen sinnvoll, jedes Dialogfenster, das man selbst definiert, als eigene Klasse zu implementieren. Man erreicht dadurch eine Kapselung, die die Programmstruktur übersichtlicher macht. Man hat nur noch ein Objekt mit klar definierter Schnittstelle, und die Wiederverwertbarkeit in anderen Programmen ist dadurch stark vereinfacht. In vielen Fällen ist die Definition einer eigenen Klasse ohnehin unumgänglich, nämlich dann, wenn die Signale, die die Bedienelemente aussenden, nicht direkt mit Slots anderer Objekte verbunden werden können. In diesem Fall muss man eigene Slots definieren, die die entsprechenden Aktionen ausführen. Dazu muss man aber zwangsläufig eine eigene Klasse definieren.

Hier sehen Sie zunächst die Klassenschnittstelle für unser Dialogobjekt:

```
class MessageWindow : public QWidget {
    Q_OBJECT

public:
    // Konstruktor
    MessageWindow (const char *name = 0, WFlags f = 0);
    ~MessageWindow () {}

public slots:
    void insert (QString message);

private:
    QMultiLineEdit *messages;
};
```


Unsere Klasse ist von *QWidget* abgeleitet, das wiederum von *QObject* abgeleitet ist. Daher muss das Makro *Q_OBJECT* in unserer Klassendefinition stehen am Anfang. Auf diese Datei muss außerdem der Meta Object Compiler (*moc*) angewendet werden (siehe Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*).

Der Konstruktor unserer Klasse erhält zwei Parameter, die genau dem zweiten und dritten Parameter des Konstruktors der Klasse *QWidget* entsprechen. Da unser Dialogfenster als Toplevel-Widget benutzt werden soll, braucht es keinen *parent*-Parameter.

Der Destruktor braucht nichts zu tun, da alle GUI-Elemente Unter-Widgets vom Toplevel-Widget sind und damit automatisch gelöscht werden. Die Destruktoren der Basisklassen werden automatisch aufgerufen. Da einige Compiler Probleme haben, wenn der Destruktor nicht explizit angegeben wird, schreiben wir ihn in die Klassendefinition und tragen direkt ein, dass er nichts bewirkt.

Mit der Methode *insert* soll eine neue Meldung in das Fenster eingetragen und das Fenster dargestellt werden (wenn es versteckt war). Damit sie aufgerufen werden kann, muss sie natürlich in der Schutzkategorie *public* stehen. Wir definieren sie hier gleich als Slot, so dass sie auch direkt von einem Signal eines anderen Objekts aufgerufen werden kann, das einen Text in Form eines *QString*-Parameters aussendet. Trotzdem kann sie natürlich weiterhin wie eine normale Methode benutzt werden.

Das Fenster mit den Meldungen speichern wir in der Objektvariablen *messages* ab. Da sie nur noch innerhalb der Klasse benutzt wird, kann sie in der Schutzkategorie *private* stehen. Die beiden Buttons verwalten wir nicht als Objektvariablen, sondern wir benutzen dazu lokale Variablen im Konstruktor. Nachdem die Buttons nämlich erzeugt, in das Layout eingebunden und mit ihren Signalen verbunden sind, braucht man keinen direkten Zugriff mehr auf sie. Sie erledigen ihre Arbeit selbstständig (indem sie die verbundenen Slots aktivieren) und werden automatisch gelöscht, da sie Unter-Widgets vom Toplevel-Widget sind. Da wir *messages* noch benötigen, um die Texte hinzuzufügen, müssen wir es als Objektvariable definieren.

Als Nächstes müssen wir die Methoden der Klasse mit Code füllen:

```
MessageWindow::MessageWindow (const char *name, WFlags f) :
    QWidget (name, f)
{
    setCaption ("Dialog in eigener Klasse");

    messages = new QMultiLineEdit (this);
    messages->setReadOnly (true);
    QPushButton *clear = new QPushButton ("Clear", this);
    QPushButton *hide = new QPushButton ("Hide", this);
```

```

// Layout erzeugen und starten
QVBoxLayout *topLayout = new QVBoxLayout (this, 10);
QHBoxLayout *buttonsLayout = new QHBoxLayout ();
topLayout->addWidget (messages);
topLayout->addLayout (buttonsLayout);
buttonsLayout->addWidget (clear);
buttonsLayout->addWidget (hide);
topLayout->activate();

QObject::connect (clear, SIGNAL (clicked ()),
                  messages, SLOT (clear ()));
QObject::connect (hide, SIGNAL (clicked ()),
                  this, SLOT (hide ()));
}

void MessageWindow::insert (QString message)
{
    messages->append (message);
    show ();
}

```

Es empfiehlt sich, in den Konstruktor der selbst definierten Klasse alle Argumente der Basisklasse ebenfalls mit aufzunehmen und an den Konstruktor der Basis-Klasse weiterzugeben. In unserem Fall hat der Konstruktor unserer neuen Klasse *MessageWindow* die Parameter *name* und *f*, während *parent* entfällt, da das Dialogfenster immer als Toplevel-Widget eingesetzt wird. Entsprechend wird der Konstruktor der Basisklasse *QWidget* mit den Parametern *0* (kein Vater, also Toplevel-Widget), *name* und *f* aufgerufen.

Innerhalb des Konstruktors werden nun die drei Unter-Widgets mit dem Vater-Widget *this* angelegt. Wie bereits oben erwähnt, erzeugen wir die beiden Buttons in lokalen Zeigervariablen mit *new* auf dem Heap, da wir später nicht mehr auf sie zugreifen müssen.

Alle weiteren Zeilen im Konstruktor sind ganz analog zum vorherigen Beispiel geschrieben.

Die Methode *insert* übernimmt die beiden Befehle, die beim Einfügen einer Textzeile nötig sind: das Einfügen in das *messages*-Widget und den Aufruf von *show*.

Ein Objekt der neuen Klasse kann nun einfach mit dieser Zeile angelegt werden:

```
MessageWindow *messageWindow = new MessageWindow ();
```

Eine neue Meldung wird dann folgendermaßen eingefügt:

```
messageWindow->insert ("Initialisierung abgeschlossen\n");
```

Zum Schluss des Programms kann das Objekt mit *delete messageWindow* gelöscht werden.

Die Darstellung auf dem Bildschirm ist gleich geblieben, da sich nur die interne Struktur geändert hat. Das Fenster sieht nach wie vor aus wie in Abbildung 3.97.

3.8.3 Modale Dialoge

Manchmal ist es im Ablauf eines Programms nötig, dass der Anwender eine Entscheidung trifft oder eine Eingabe macht, bevor das Programm mit der Abarbeitung fortfahren kann. Dazu öffnet sich ein neues Fenster, das die Eingabe entgegennimmt und das mit einem Button geschlossen wird. Solange das Fenster noch offen ist, reagieren die anderen Fenster des Programms nicht mehr auf eine Eingabe.

In Qt wird dieses Verhalten von einem Objekt der Klasse *QDialog* erzeugt. *QDialog* ist von *QWidget* abgeleitet. Es kann im Modus *modal* oder *nicht-modal* benutzt werden. Im nicht-modalen Modus unterscheidet es sich kaum von *QWidget*, im modalen Modus dagegen übernimmt es genau die geforderten Aufgaben. Ein modales *QDialog*-Objekt ist immer ein *Toplevel-Widget*. Sobald es (mit der Methode *QDialog::exec*) aufgerufen wird, öffnet es das Fenster auf dem Bildschirm und blockiert die anderen Fenster des Programms. Dazu startet es eine eigene Event-Schleife, in der nur die für dieses Fenster relevanten Maus- und Tastatur-Events verarbeitet werden. Alle anderen Maus- und Tastatur-Events werden ignoriert und gelöscht. Ähnlich wie *quit* die Haupt-Event-Schleife von *QApplication* beendet, kann die lokale Event-Schleife mit der Methode *done* beendet werden. Ihr kann man auch noch einen Parameter in Form eines *int*-Wertes mitgeben, der als Resultat des Dialogs gespeichert wird und abgefragt werden kann. Nach dem Beenden der lokalen Event-Schleife wird das Fenster wieder versteckt, und die Kontrolle kehrt zum Aufrufer von *exec* zurück. Der Resultatwert des Dialogs ist der Rückgabewert der *exec*-Methode.

Standardmäßig werden für den Resultatwert eines Dialogs die Konstanten *QDialog::Accepted* und *QDialog::Rejected* benutzt. Die meisten Dialoge lassen sich regulär beenden (meist mit einem OK-Button am unteren Rand) oder abbrechen (mit einem ABBRECHEN-Button). Das erreicht man am einfachsten, indem man mittels *connect* den OK-Button mit dem Slot *accept* und den ABBRECHEN-Button mit dem Slot *reject* verbindet. Diese rufen ihrerseits *done(Accepted)* bzw. *done(Rejected)* auf, beenden damit den Dialog, schließen das Fenster und geben den Wert *Accepted* bzw. *Rejected* an den Aufrufer von *exec* zurück. Natürlich sind alle *int*-Werte als Rückgabewerte erlaubt, und ihre Interpretation ist beliebig.

Für einfache Meldungen, die nur vom Anwender bestätigt werden sollen, oder für einfache Ja/Nein-Fragen reicht es meist, die Klasse *QMessageBox* zu benutzen (siehe Kapitel 3.7.8, *Dialoge*), ohne eine eigene Klasse bilden zu müssen. Für kom-

plexere Dialoge, in denen eine Eingabe nötig ist oder spezielle Daten angezeigt werden sollen, leitet man am besten wieder eine eigene Klasse von *QDialog* ab, in der man im Konstruktor die benötigten GUI-Elemente erzeugt und platziert.

Als Beispiel für einen modalen Dialog wollen wir ein Fenster entwickeln, das von einer Adressverwaltung benutzt werden kann, um neue Adressen einzugeben. Es kann zum Beispiel immer dann aufgerufen werden, wenn der Anwender den Befehl NEUE ADRESSE HINZUFÜGEN... aktiviert.

Unser einfacher Beispieldialog soll nur drei Felder enthalten. In den ersten beiden kann man den Namen und die E-Mail-Adresse eingeben, im dritten kann man das Alter auswählen (siehe Abbildung 3.98). Eine vollständige Adressverwaltung hätte natürlich viel mehr Felder, es ist aber nicht schwierig, unsere hier entwickelte Klasse zu erweitern.



Abbildung 3-98 Ein einfacher Dialog zur Adresseingabe

Am unteren Rand hat unser Fenster zwei Buttons mit der Aufschrift ADD (also »hinzufügen«) und CANCEL (also »abbrechen, ohne hinzuzufügen«). Der CANCEL-Button ist für fast jeden Dialog wichtig, denn oftmals wird ein Dialog aus Versehen aufgerufen. Dann muss der Anwender eine einfache Möglichkeit haben, den Dialog zu schließen, ohne irgendeine Änderung zu bewirken.

Ermitteln wir zunächst, welche Klassen wir für unsere GUI-Elemente benutzen können. Zur Eingabe von Name und E-Mail-Adresse müssen sie beliebigen Text entgegennehmen können. Da in diesem Fall eine einzelne Zeile ausreicht, benutzen wir *QLineEdit*. Die Auswahl des Alters kann hier zum Beispiel durch ein Objekt der Klasse *QSpinBox* erfolgen, da der Wert nur ganzzahlig sein kann. Die beiden Buttons am unteren Rand sind vom Typ *QPushButton*.

Der Code, der die Klasse erzeugt, sieht wie folgt aus:

```
class AddressDialog : public QDialog
{
    Q_OBJECT
```

```
public:
    AddressDialog (QWidget *parent=0,
                  const char *name=0, WFlags f=0);
    ~AddressDialog () {}

    // Methoden zum Auslesen der eingegebenen Daten
    QString name() {return nameW->text();}
    QString email() {return emailW->text();}
    int age() {return ageW->value();}

public slots:
    // Löscht alle Einträge
    void clear();

private:
    // GUI-Elemente für die Daten
    QLineEdit *nameW, *emailW;
    QSpinBox *ageW;
};

AddressDialog::AddressDialog (QWidget *parent,
                              const char *name, WFlags f)
    : QDialog (parent, name, true, f)
{
    // Erzeuge die GUI-Elemente
    nameW = new QLineEdit (this);
    emailW = new QLineEdit (this);
    ageW = new QSpinBox (0, 130, 1, this);

    // Erzeuge und verbinde die Buttons
    QPushButton *add = new QPushButton ("Add", this);
    connect (add, SIGNAL (clicked()), SLOT (accept()));
    QPushButton *cancel = new QPushButton ("Cancel", this);
    connect (cancel, SIGNAL (clicked()), SLOT (reject()));

    // "Add" wird der Default-Button
    add->setDefault(true);

    // Layout (mit Beschriftungstexten) erzeugen
    QVBoxLayout *top = new QVBoxLayout (this, 10);
    QGridLayout *contents = new QGridLayout (3, 2);
    QHBoxLayout *buttons = new QHBoxLayout ();

    top->addLayout (contents);
    top->addWidget (new KSeparator (QFrame::HLine, this));
    top->addLayout (buttons);

    QLabel *l;
    l = new QLabel (nameW, "&Name : ", this);
```

```

l->setAlignment (AlignRight | AlignVCenter);
contents->addWidget (l, 0, 0);
contents->addWidget (nameW, 0, 1);
l = new QLabel (emailW, "e&Mail : ", this);
l->setAlignment (AlignRight | AlignVCenter);
contents->addWidget (l, 1, 0);
contents->addWidget (emailW, 1, 1);
l = new QLabel (ageW, "&Age : ", this);
l->setAlignment (AlignRight | AlignVCenter);
contents->addWidget (l, 2, 0);
contents->addWidget (ageW, 2, 1);

buttons->addStretch (1);
buttons->addWidget (add);
buttons->addWidget (cancel);

top->activate();
}

void AddressDialog::clear()
{
    nameW->clear();
    emailW->clear();
    ageW->setValue (20);
}

```

Der Konstruktor fällt in diesem Beispiel schon recht lang aus. Bei noch komplexeren Dialogen kann es übersichtlicher sein, den Aufbau des Fensters auf mehrere Methoden zu verteilen, die der Konstruktor dann nur noch aufruft.

Der ADD-Button wird mit dem *accept*-Slot und der CANCEL-Button mit dem *reject*-Slot verbunden. Außerdem ist der ADD-Button der Default-Button. Das bedeutet, dass jedes Mal, wenn innerhalb des Dialogfensters die -Taste gedrückt wird, automatisch der ADD-Button betätigt wird.

Seien Sie vorsichtig mit dem Mechanismus des Default-Buttons in *QDialog*-Objekten, da er den Anwender manchmal überraschen könnte. Wenn der Benutzer innerhalb eines der Eingabefelder beispielsweise die -Taste drückt, wird der Dialog beendet und die neue (wahrscheinlich noch unvollständige) Adresse eingetragen. Der Anwender hat aber vielleicht erwartet, dass der Fokus automatisch in das nächste Feld wechselt.

Noch unerwarteter ist wahrscheinlich die Reaktion, wenn sich der Fokus gerade auf einem anderen Button befindet (z.B. dem CANCEL-Button). Auch in diesem Fall wird bei  der ADD-Button ausgelöst. Um den CANCEL-Button auszulösen, hätte der Anwender auf die Leertaste drücken müssen.

Das Layout dieses Fensters besteht aus drei Bereichen: Oben befindet sich der Eingabebereich, der seinerseits aus einem Gitter aus drei Zeilen mit jeweils zwei Objekten besteht. In der Mitte sehen Sie eine waagerechte Linie, um den Eingabebereich grafisch von den Buttons zu trennen, und am unteren Rand die Buttons. Sie sind in einem *QHBoxLayout*-Objekt organisiert, das die Buttons so weit wie möglich nach rechts schiebt. Der Eingabebereich besteht aus einer rechtsbündigen Beschriftung im linken Teil und dem zugehörigen Eingabefeld rechts davon. Da die Beschriftungsobjekte nicht weiter gebraucht werden, erzeugen wir sie alle mit einer einzigen Variable *l* nacheinander auf dem Heap. Beachten Sie, dass es trotzdem drei verschiedene Widgets sind. *l* ist nur ein Zeiger, der auf das zuletzt erzeugte Widget zeigt.

Den Beschriftungselementen wurde als erster Parameter im Konstruktor ein so genannter *Buddy* zugeordnet. Das ist ein Widget, das den Fokus bekommen soll, wenn der mit & markierte Buchstabe (oder die so markierte Ziffer) in der Beschriftung zusammen mit der **Alt**-Taste gedrückt wird. Hier bewirkt ein Druck auf **Alt**+**N** zum Beispiel, dass der Fokus auf das Namensfeld im Eingabebereich springt. Dadurch kann das Fenster auch per Tastatur leichter gesteuert werden.

Wie wird dieser Dialog innerhalb eines Programms genutzt? Wenn wir zum Beispiel ein Objekt der Klasse *AddressBook* geschrieben haben, das in seinem Slot *addNewAddress* (z.B. aktiviert durch einen Menübefehl oder einen Button) den Dialog aufrufen will, kann das so geschehen:

```
void AddressBook::addNewAddress ()
{
    AddressDialog d();
    if (d.exec() == QDialog::Accepted)
    {
        // Adresse mit den Methoden
        // d.name, d.email und d.age auslesen
        // und in das Adressbuch eintragen
    }
}
```

Bei jedem Aufruf des Slots wird ein Dialog erzeugt und gestartet. Aber nur, wenn er mit dem ADD-Button beendet wurde, werden die eingetragenen Daten in das Adressbuch übernommen. Da *d* hier eine lokale Variable ist, wird *d* automatisch wieder gelöscht, nachdem der Slot abgearbeitet ist.

Anstatt jedes Mal ein neues Dialogobjekt zu erzeugen, kann man auch gleich beim Programmstart eines erzeugen und es dann immer wieder benutzen. Wenn die Klasse *AddressBook* zum Beispiel im Konstruktor ein *AddressDialog*-Fenster erzeugt und in der Objektvariablen *d* ablegt, so könnte der Slot auch so aussehen:

```

void AddressBook::addNewAddress ()
{
    d.clear();
    if (d.exec() == QDialog::Accepted)
    {
        // Adresse auslesen und in das Adressbuch eintragen
    }
}

```

Man spart hier den Aufwand, jedes Mal das Objekt mitsamt seinem Layout neu erzeugen zu lassen. Dadurch wird die Zeit bis zum Öffnen des Fensters kürzer. Man darf hier aber nicht vergessen, die Werte der letzten Eintragung zu löschen, bevor man den Dialog erneut startet (hier mit der Methode *clear*).

Für dieses spezielle Dialogfenster kann man sich ein anderes Verhalten vorstellen: Anstatt nach dem Klicken auf ADD das Fenster wieder zu schließen, kann es vorteilhaft sein, das Fenster geöffnet zu lassen, so dass der Anwender gleich eine weitere Adresse eingeben kann, ohne erneut den Befehl zum Öffnen des Fensters wählen zu müssen. Eine Realisierung könnte zum Beispiel so aussehen:

```

class AddressDialog : public QDialog
{
    Q_OBJECT
public:
    AddressDialog (QWidget *parent=0,
                  const char *name=0, WFlags f=0);
    ~AddressDialog () {}

    // Methoden zum Auslesen der eingegebenen Daten
    QString name() {return nameW->text();}
    QString email() {return emailW->text();}
    int age() {return ageW->value();}

public slots:
    // Löscht alle Einträge
    void clear();

signals:
    // Neue Adresse ist eingetragen worden
    void newAddressAdded();

private slots:
    // Wird von add->clicked aktiviert
    void addClicked();

private:
    // GUI-Elemente für die Daten
    QLineEdit *nameW, *emailW;

```



```
    QSpinBox *ageW;
};

AddressDialog::AddressDialog (QWidget *parent,
                              const char *name, WFlags f)
    : QDialog (parent, name, true, f)
{
    // Erzeuge die GUI-Elemente
    nameW = new QLineEdit (this);
    emailW = new QLineEdit (this);
    ageW = new QSpinBox (0, 130, 1, this);

    // Erzeuge und verbinde die Buttons
    QPushButton *add = new QPushButton ("Add", this);
    connect (add, SIGNAL (clicked()), SLOT (addClicked()));
    QPushButton *close = new QPushButton ("Close", this);
    connect (close, SIGNAL (clicked()), SLOT (reject()));

    // "Add" wird der Default-Button
    add->setDefault(true);

    // Layout (mit Beschriftungstexten) erzeugen
    // wie gehabt...
}

void AddressDialog::clear()
{
    nameW->clear();
    emailW->clear();
    ageW->setValue (20);
}

void addClicked()
{
    emit newAddressAdded();
    clear();
}
```

In dieser Variante ruft nun ein Klick auf ADD nicht mehr den Slot *accept* auf, der das Fenster schließen würde. Stattdessen wird der private Slot *addClicked* aufgerufen, der ein *newAddressAdded*-Signal schickt und anschließend alle Einträge wieder löscht. Das aufrufende Programm verbindet das Signal *newAddressAdded* mit einem eigenen Slot, der die Werte aus dem Dialogfenster ausliest und den Eintrag mit in das Adressbuch aufnimmt. Diese Änderung sollte sofort auf dem Bildschirm angezeigt werden, damit der Anwender erkennt, dass der Klick auf ADD Auswirkungen hatte. Während das Dialogfenster geöffnet ist, lassen sich die anderen Fenster zwar nicht mehr bedienen, Signale werden aber weiterhin empfangen, und Änderungen an den Widgets in den anderen Fenstern werden sofort

angezeigt. Das Fenster kann nur noch durch den zweiten Button geschlossen werden, den wir nun in CLOSE (Schließen) umbenannt haben, um zu signalisieren, dass das nun seine Hauptaufgabe ist. Der Rückgabewert der *exec*-Methode ist jetzt in jedem Fall *Rejected*, da nur der CLOSE-Button das Fenster schließt.

Im Folgenden werden einige Regeln für die Gestaltung und Aufteilung eines Dialogfensters zusammengefasst, die den Dialog für den Anwender überschaubarer machen, bevor wir uns anhand weiterer Beispiele auch komplexere Dialogfenster ansehen werden.

- Jedes (modale) Dialogfenster sollte am unteren Rand eine Reihe nebeneinander liegender Buttons besitzen. Standardmäßig sind das mindestens die Buttons OK und CANCEL (auf deutsch OK und ABBRECHEN). Die Beschriftungen können auch anders sein, um die Wirkung noch deutlicher werden zu lassen. Insbesondere der OK-Button trägt oft je nach Dialogfenster die Bezeichnung OPEN, SAVE, ADD, CHANGE oder Ähnliches.
- Diese beiden Buttons stehen am rechten Rand, der OK-Button links vom CANCEL-Button. Beide Buttons schließen das Fenster, wobei OK die vorgenommenen Einstellungen übernimmt und CANCEL die Einstellungen verwirft.
- Der OK-Button ist der Default-Button, der automatisch aktiviert wird, wenn die -Taste betätigt wird.
- Die Buttons am unteren Rand werden grafisch (zum Beispiel durch eine Linie) vom Rest des Fensters getrennt.
- Für komplexe Dialoge kommen manchmal noch Buttons hinzu. Ein HELP-Button (deutsch HILFE) steht dann am linken Rand und öffnet ein Hilfefenster zu den Einstellungen. Direkt rechts daneben kann sich ein DEFAULTS-Button (deutsch STANDARD) befinden, der alle Einstellungen auf ihre Standardwerte zurücksetzt.
- Zwischen OK und CANCEL kann noch ein Button APPLY (deutsch ÜBERNEHMEN) stehen, der die getätigten Einstellungen übernimmt und anwendet (es müssen sich direkt Auswirkungen ergeben), das Dialogfenster aber noch nicht schließt. Die mit diesem Button übernommenen Einstellungen können danach auch durch CANCEL nicht mehr rückgängig gemacht werden. Mit diesem Button kann man die Einstellungen testen und direkt weitere Änderungen vornehmen, ohne das Fenster erneut öffnen zu müssen.
- Dialogfenster sollten möglichst vollständig und schnell mit der Tastatur zu bedienen sein. Dazu definiert man möglichst für jedes Feld einen leicht zu merkenden Beschleuniger (-Taste zusammen mit einem Buchstaben oder einer Ziffer). Alle vordefinierten GUI-Elemente sind bereits mit der Tastatur steuerbar. Darauf sollte man auch bei Widgets aus anderen Bibliotheken achten.

3.8.4 Optionsdialoge

Die meisten Programme besitzen eine Reihe von Einstellungen, um das Verhalten des Programms zu beeinflussen. Diese Optionen werden meist in einem eigenen Dialogfenster eingestellt und beim Beenden des Programms in der Konfigurationsdatei gesichert. Die Einstellungen sind nun an (mindestens) drei Stellen gespeichert: in der Konfigurationsdatei, in den Daten der Konfigurationsdatei, die im Speicher abgelegt sind, sowie in den Werten der GUI-Elemente des Optionsdialogs.

In den meisten Fällen empfiehlt es sich, alle Änderungen an den Konfigurationsdateien in einer Klasse zu implementieren. Wenn man diese Klasse von *QDialog* ableitet, kann man auch gleich das Dialogfenster darin unterbringen. Weitere Informationen finden Sie in Kapitel 4.10, *Konfigurationsdateien*.

Hier wollen wir zunächst auf ein paar Gestaltungsbeispiele eingehen, um zu zeigen, wie der Dialog aufgebaut sein kann.

In einem Optionsdialog ist es wichtig, dass die einzelnen Einstellungen zu sinnvollen Gruppen zusammengefasst und übersichtlich auf dem Bildschirm angeordnet sind. Welche GUI-Elemente für die einzelnen Einstellungen geeignet sind, können Sie in Kapitel 3.7, *Überblick über die GUI-Elemente von Qt und KDE*, nachlesen.

Eine Möglichkeit, Gruppen von Einstellungen zusammenzufassen, ist die Umrandung mit einem Rahmen. Als Rahmen benutzen Sie am besten ein *QFrame*-Objekt, bei dem Sie die Rahmenart mit *setFrameStyle* (*QFrame::Box* | *QFrame::Sunken*) und die Liniendicken auf *setLineWidth* (1) und *setMidLineWidth* (0) einstellen. Die einzelnen Einstellungselemente tragen Sie dann als Unter-Widget in den Frame ein. Statt eines *QFrame*-Objekts können Sie auch ein *QGroupBox*-Objekt benutzen oder – speziell wenn *QRadioButton*-Elemente enthalten sein sollen – ein *QButtonGroup*-Objekt, so dass Sie der Gruppe eine Titelbezeichnung geben können.

Ein typischer Dialogaufbau ist zum Beispiel in Abbildung 3.99 dargestellt. Die vier Gruppen sind in einem *QGridLayout*-Objekt angeordnet. Die Unterpunkte in den einzelnen Gruppen sind mit einem *QVBoxLayout*-Objekt angeordnet. Beachten Sie hierbei zum einen unbedingt, dass Sie dabei die Rahmenbreite mit berücksichtigen müssen (siehe Kapitel 3.6.4, *Die Layout-Klassen QVBoxLayout und QGridLayout*). Außerdem ist zu beachten, dass wir hier unabhängige Layout-Hierarchien haben: eine für die Anordnung der Gruppen und innerhalb jeder Gruppe eine eigene, unabhängige Hierarchie. Das oberste Layout-Objekt innerhalb einer Gruppe bekommt das *QFrame*- bzw. *QGroupBox*-Objekt als oberes Widget zugewiesen. Wichtig ist, dass die Layouts der einzelnen Gruppen zuerst erstellt und aktiviert werden sollten, damit die Minimal- und Maximalgrößen der Gruppen festgelegt sind, bevor sie in das Haupt-Layout eingefügt werden. Abbildung 3.100 zeigt nochmals die Layout-Hierarchie für eine Gruppe, Abbildung 3.101 die Layout-Hierarchie für das Dialogfenster.



Abbildung 3-99 Dialog mit zu Gruppen zusammengefassten Optionen

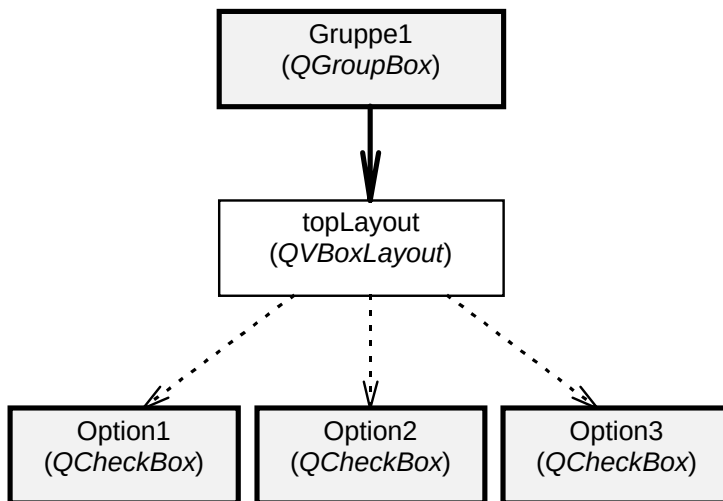


Abbildung 3-100 Layout-Hierarchie in einer Gruppe

Manche Eingabeelemente sind nur dann sinnvoll, wenn in einem anderen Eingabeelement ein bestimmter Wert oder Zustand eingestellt ist. So ist zum Beispiel in Abbildung 3.102 das Eingabefeld für den Wert der Auflösung nur dann aktiv, wenn der *QRadioButton* »Custom« ausgewählt ist. Er wird in den anderen Fällen mit der Methode *setEnabled (false)* deaktiviert. Er erscheint dann grau, ist aber weiterhin vorhanden.

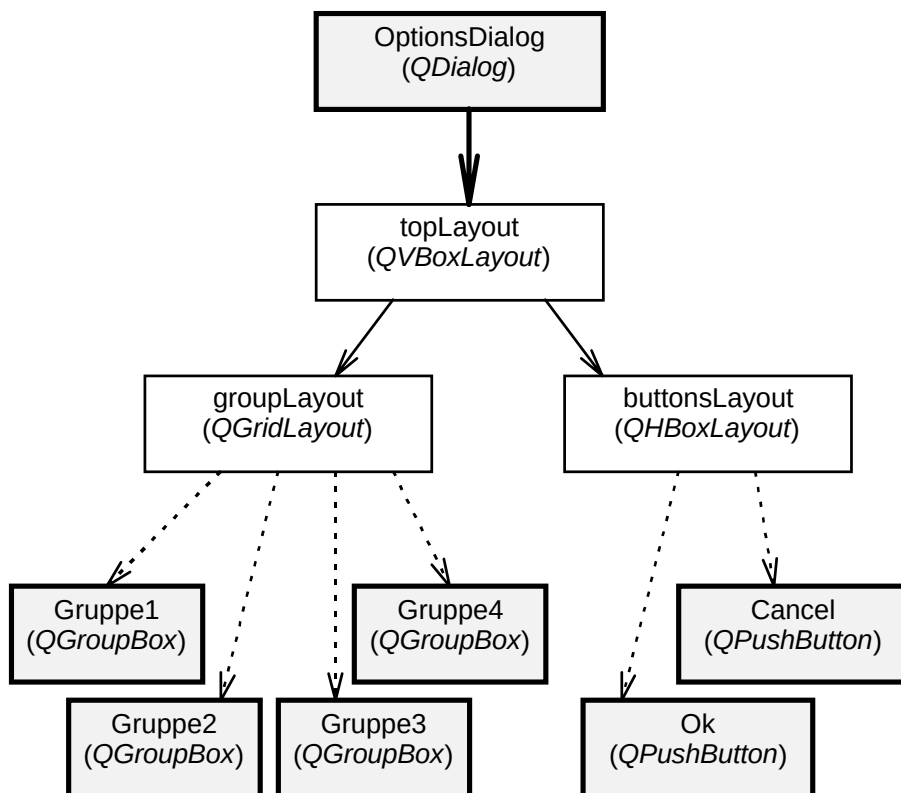


Abbildung 3-101 Layout-Hierarchie des Dialogs

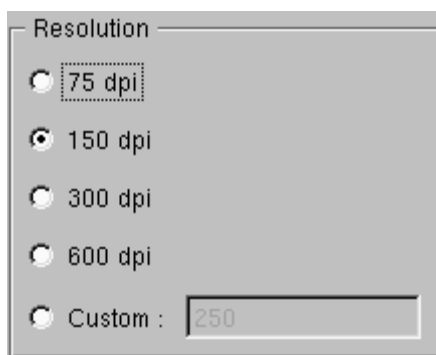


Abbildung 3-102 Abhängigkeiten zwischen Eingabeelementen

Am einfachsten verbindet man in diesem Fall das Signal *toggled (bool)* des *QRadioButton*s »Custom« mit dem Slot *setEnabled (bool)* des Eingabeelements. Ganz zu Beginn des Dialogs ist dann noch der richtige Zustand einzustellen. Das folgende Listing erläutert kurz den Aufbau:

```
QButtonGroup *group =
    new QButtonGroup ("Resolution", this);
QRadioButton *r75 = new QRadioButton("75 dpi", group);
QRadioButton *r150 = new QRadioButton("150 dpi", group);
QRadioButton *r300 = new QRadioButton("300 dpi", group);
QRadioButton *r600 = new QRadioButton("600 dpi", group);
QRadioButton *rcust = new QRadioButton("Custom : ", group);
QLineEdit *custRes = new QLineEdit (group);
connect (rcust, SIGNAL (toggled(bool)),
        custRes, SLOT (setEnabled(bool)));
// setzt 300 dpi als Default
r300->setChecked (true);
// Das Eingabeelement ist also nicht aktiv
custRes->setEnabled (rcust->isChecked());
```

Das Layout in diesem Beispiel ordnet die einzelnen Punkte des Feldes in einem *QVBoxLayout*-Objekt an, wobei das letzte Element wiederum ein *QHBoxLayout*-Objekt mit den beiden Widgets *QRadioButton* und *QLineEdit* ist.

Bei solchen Abhängigkeiten zwischen den Eingabeelementen sind einige Punkte unbedingt zu beachten, um die Übersicht zu gewährleisten:

- Zur Zeit nicht bedienbare Elemente sollten *immer* mit *setEnabled(false)* deaktiviert werden, niemals sollten sie aus dem Dialog entfernt werden. Als Grundregel gilt, dass der Aufbau und die Anordnung eines Dialogfensters immer erhalten bleiben sollen, damit die Einarbeitungszeit kurz bleibt.
- So ist es zum Beispiel nicht ratsam, aus Platzgründen immer nur eines von zwei Einstellungsobjekten zu zeigen, wenn immer nur das eine oder das andere bedient werden kann. Neben den Problemen, die dieses Vorgehen für das Layout mit sich bringt, verliert der Anwender schnell den Überblick über die vorhandenen Einstellungsmöglichkeiten. Daher sollten Sie auch in diesem Fall immer beide Einstellungsobjekte anzeigen und immer nur eines davon aktivieren.
- Abhängige Elemente sollten unmittelbar hinter oder unter dem Element stehen, das sie kontrolliert. Ist das nicht möglich, müssen Sie versuchen, die Abhängigkeit durch andere grafische Effekte (Rahmen, Linien) darzustellen.
- Kein Element sollte von mehr als einem Element abhängig sein. Falls das nicht möglich scheint, überprüfen Sie, ob der Dialogs nicht besser strukturiert werden kann.
- Das Deaktivieren eines Widgets deaktiviert auch alle enthaltenen Unter-Widgets. Wenn also mehrere Elemente von einer Einstellung abhängen, fassen Sie sie am besten in einem *QFrame*- oder *QGroupBox*-Objekt zusammen.

Wenn die Anzahl der einstellbaren Optionen wächst, ist es oft nicht mehr möglich, alle Optionen in einem einfachen Fenster unterzubringen. Mehrere verschiedene Dialogfenster zu benutzen ist aber ebenfalls ungeeignet, da der Anwender dann oftmals mehrere Fenster öffnen muss, bevor er die passende Einstellung gefunden hat.

Die Lösung besteht darin, die Optionen auf verschiedene Seiten aufzuteilen und dem Anwender eine Möglichkeit zu geben, eine der Seiten auszuwählen. Als Standard hat sich dabei ein Karteikartenfenster durchgesetzt, bei dem die Optionen auf Karteikarten stehen, die durch Anklicken eines Reiters (Tab) ausgewählt werden. Qt bietet bereits eine fertige Widget-Klasse `QTabWidget` an, in die ganz einfach Seiten in Form eines `QWidget`-Objekts eingefügt werden (siehe Karteikarten-Widget in Abbildung 3.103). Diese Objekte enthalten in der Regel weitere Unter-Widgets, um die Optionen einzustellen. So enthält beispielsweise die Seite, die dem Tab GRUPPE 2 zugeordnet ist, drei `QCheckBox`-Objekte, die durch ein `QVBoxLayout`-Objekt angeordnet sind. Die Klasse `QTabDialog` enthält ein `QTabWidget`-Objekt als Unter-Widget, verwaltet aber noch zusätzlich Buttons am unteren Rand des Dialogfensters (siehe Abbildung 3.103). Wichtig beim Design eines `QTabDialog`-Fensters ist die Aufteilung der Optionen. Alle Optionen einer Seite müssen unter einem eindeutigen, kurzen Namen zusammengefasst werden können. Zwischen verschiedenen Seiten sollten keine Abhängigkeiten bestehen; jede Seite sollte eine Einheit für sich sein. Freien Platz auf einer Seite lassen Sie am besten unten frei.

Wichtig ist auch die Interpretation der Buttons am unteren Rand: Ihre Wirkung gilt grundsätzlich für *alle* Seiten, auch für die gerade nicht angezeigten. Ein Klick auf CANCEL macht zum Beispiel die Änderungen auf allen Seiten rückgängig. Wirksam werden Änderungen erst, nachdem auf Ok geklickt wurde, und nicht bereits beim Wechseln der Seite.

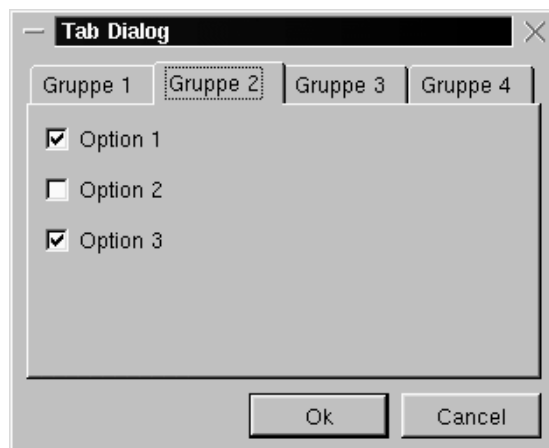


Abbildung 3-103 `QTabDialog` mit vier Karteikarten

Wenn Sie den Dialog anders gestalten wollen, können Sie ihn auch selbst aus einem *QDialog*-Objekt zusammenstellen, in das Sie ein *QTabWidget*-Objekt einfügen. Die Buttons am unteren Fensterrand sowie das Layout müssen Sie dann selbstständig erzeugen.

Ab einer gewissen Zahl von Karteikarten wird der *QTabDialog* wieder unübersichtlich. Dann kann es oftmals sinnvoll sein, die Seiten selbst noch einmal in Gruppen zusammenzufassen. So findet der Anwender schneller die ihn interessierende Seite. Ganz wichtig ist es dabei, dass die Aufteilung der Seiten für den Anwender sehr leicht nachzuvollziehen sein muss. Eine Möglichkeit, ein solches Dialogfenster aufzubauen, besteht beispielsweise darin, die hierarchische Seitenunterteilung in einem Objekt der Klasse *QListView* darzustellen. Der Anwender kann die Teilbereiche, die ihn nicht interessieren, einfach ausblenden. Meist unterteilt man das Fenster in einen linken Bereich mit der Auflistung der Seiten und einen rechten Bereich, in dem die ausgewählte Seite angezeigt wird und die Optionen geändert werden können. Am unteren Rand werden wieder die Buttons zum Schließen des Dialogs angezeigt (siehe Abbildung 3.104).

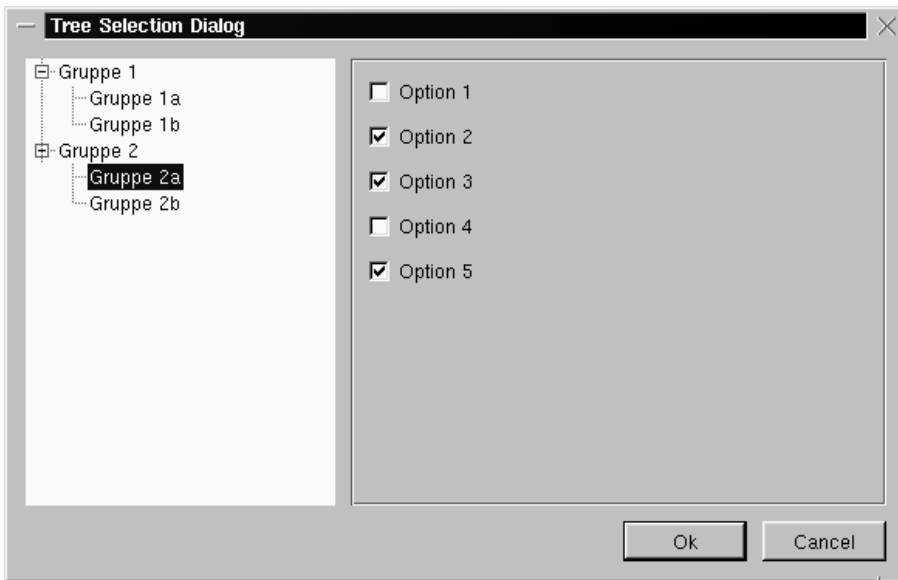


Abbildung 3-104 Auswahl der Seiten mit *QListView*

Beachten Sie folgende Hinweise, wenn Sie einen solchen Dialog erzeugen möchten:

- Das *QListView*-Objekt enthält nur eine Spalte. Die Kopfzeile können Sie durch einen Aufruf von `listView->header()->hide()` ausblenden.

- Auf alle Übergruppen (GRUPPE 1 und GRUPPE 2), die nicht direkt Seiten des Dialogs auswählen, sollten Sie den Methodenaufruf `QListViewItem::setSelectable(false)` anwenden.
- Die Seiten werden in einem `QWidgetStack`-Objekt angezeigt. Mit dem Slot `raiseWidget` kann die ausgewählte Seite angezeigt werden.
- Das Signal `selectionChanged` des `QListView`-Objekts liefert das `QListViewItem`-Objekt zurück, das angeklickt wurde. Um das Widget herauszufinden, das zu dieser Seite gehört, kann man zum Beispiel ein Objekt der Template-Klasse `QPtrDict<QWidget>` benutzen, in dem man die Zuordnung des `QListViewItem`-Zeigers auf das entsprechende `QWidget` speichert.
- `QWidgetStack` ist von `QFrame` abgeleitet. Durch die Methodenaufrufe `setFrameStyle(QFrame::Sunken | QFrame::Box)`, `setLineWidth(1)` und `setMidLineWidth(0)` können Sie den typischen Rahmen erzeugen.
- Auch in diesem Dialog müssen sich die Auswirkungen der Buttons auf *alle* Seiten beziehen, nicht nur auf die gerade angezeigte Seite.

4 Weiterführende Konzepte der Programmierung in KDE und Qt

Nachdem wir in Kapitel 3, *Grundkonzepte der Programmierung in KDE und Qt*, die Struktur eines KDE-Programms besprochen haben, werden wir in diesem Kapitel Techniken besprechen, die für aufwendigere Applikationen nötig sind.

Oftmals kommt man bei speziellen Aufgaben mit den im Qt-Umfang enthaltenen Bedienelementen nicht aus, so dass man selbst eines entwerfen und implementieren muss. Für den Entwurf einer solchen Widget-Klasse sind zunächst Kenntnisse nötig, wie man in ein Fenster zeichnet. In Kapitel 4.1, *Farben unter Qt*, und in Kapitel 4.2, *Zeichnen von Grafikprimitiven*, werden die Grundlagen besprochen. Wie man Bilder im Speicher bearbeitet, um sie erst später anzuzeigen, ist Thema von Kapitel 4.3, *Teilbilder – QImage und QPixmap*. Mit diesen Grundlagen können Sie dann eigene Widgets entwerfen, wie es in Kapitel 4.4, *Entwurf eigener Widget-Klassen*, besprochen wird. Einige Techniken, um ein Flimmern bei der Darstellung von Widgets zu verhindern, lernen Sie in Kapitel 4.5, *Flimmerfreie Darstellung*, kennen.

Wenn Sie eigene Klassen der Allgemeinheit zur Verfügung stellen wollen, sollten Sie eine gute Klassenreferenz erstellen. Hilfsmittel – wie die Programme *kdoc* oder *doxygen* – können Sie dabei unterstützen. In Kapitel 4.6, *Klassendokumentation mit doxygen*, wird ein solches Hilfsmittel besprochen.

Eine ganze Reihe von Klassen für oft benötigte Datenstrukturen – dynamische Arrays, Listen, Hash-Tabellen – sind bereits in der Qt-Bibliothek als Templates enthalten. Diese können Sie in Ihren Programmen benutzen. Kapitel 4.7, *Grundklassen für Datenstrukturen*, enthält eine Einführung in diese Template-Klassen.

Der ASCII-Standard zum Speichern von Texten reicht heute oftmals nicht mehr aus, denn die Anzahl der darstellbaren Zeichen ist beschränkt. Daher hat sich Unicode als Standard etabliert, um in Zukunft ASCII abzulösen. Die Qt-Klasse *QString* speichert Texte bereits im Unicode-Format. Wie dieses Format aufgebaut ist und wie Sie damit in Ihren Programmen umgehen, wird in Kapitel 4.8, *Der Unicode-Standard*, beschrieben.

Moderne Programme sollten die Möglichkeit bieten, die zu verwendende Landessprache auszuwählen. Schließlich wollen Anwender das Programm in der Sprache bedienen, die sie am besten beherrschen. Sowohl KDE als auch Qt bieten dazu einige Möglichkeiten, wie Sie Ihr Programm darauf vorbereiten können. Bereits in Kapitel 2.3.3, *Landessprache: Deutsch*, haben wir diese Möglichkeit kurz angerissen. Eine ausführliche Beschreibung der Techniken finden Sie in Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*.

Jedes größere Programm besitzt in der Regel viele Einstellungsmöglichkeiten, mit denen der Anwender das Programm dem System oder dem eigenen Geschmack gemäß anpassen kann. Diese Daten müssen beim Beenden des Programms abgespeichert und beim nächsten Start wieder eingelesen werden, damit der Anwender die Einstellungen nicht jedes Mal erneut vornehmen muss. KDE bietet zu diesem Zweck bereits einfache Möglichkeiten, Konfigurationsdateien anzulegen und zu ändern. Wie man diese im eigenen Programm einsetzt, wird in Kapitel 4.10, *Konfigurationsdateien*, beschrieben.

Bereits in Kapitel 2.3.4, *Die Online-Hilfe*, waren wir kurz darauf eingegangen, wie man auf einfache Art dem Anwender ein geeignetes Hilfesystem zur Verfügung stellen kann. Kapitel 4.11, *Online-Hilfe*, enthält weitere Informationen, wie Sie Ihr KDE- oder Qt-Programm mit einer Online-Hilfe versehen können. Dazu gehören neben den umfangreichen Hilfedokumenten auch einfache Hilfesysteme wie die so genannten Tool-Tips.

Für Programme, die Zugriff auf eine Zeitbasis benötigen, sind in Qt die Klassen *QDateTime* und *QTimer* definiert, die in Kapitel 4.12, *Timer-Programmierung, Datum und Uhrzeit*, beschrieben werden.

Eine Applikation mit einer grafischen Benutzeroberfläche sollte immer möglichst unmittelbar auf Änderungen an Maus und Tastatur reagieren. Wie man sein Programm so strukturiert, dass dieses gewährleistet ist, wird in Kapitel 4.13, *Blockierungsfreie Programme*, erläutert.

Wie man der Soundkarte Töne entlockt, wird in Kapitel 4.14, *Audioausgabe*, aufgezeigt.

Wie man in Qt und in KDE Daten-Objekte von einer Applikation in eine andere bringen kann, erläutert Kapitel 4.15, *Die Zwischenablage und Drag & Drop*. Diese Konzepte können die Anwendungsmöglichkeiten eines Programms enorm vergrößern, da die Zusammenarbeit mit anderen Programmen ermöglicht wird.

Beim Herunterfahren des X-Servers sollten alle noch laufenden KDE-Programme ihren Zustand sichern, so dass sie beim nächsten Start des X-Servers automatisch wieder hergestellt werden können. Mehr dazu erfahren Sie in Kapitel 4.16, *Session-Management*.

Wie man aus einer Applikation heraus den Drucker ansteuert, beschreibt Kapitel 4.17, *Drucken mit Qt*.

Um plattformunabhängig Dateien lesen und schreiben zu können, bietet Qt einige Klassen an, die in Kapitel 4.18, *Dateizugriffe*, beschrieben werden.

Sowohl Qt als auch KDE bieten eine Reihe von Klassen an, um über ein Netzwerk auf andere Rechner zuzugreifen. Wie Sie diese Klassen in eigenen Programmen nutzen, wird in Kapitel 4.19, *Netzwerkprogrammierung*, beschrieben.

Speziell zur Kommunikation und zum Datenaustausch zwischen KDE-Programmen wurde das *Desktop Communication Protocol (DCOP)* entwickelt. In Kapitel 4.20, *Interprozesskommunikation mit DCOP*, erfahren Sie, wie Sie Nachrichten an andere Programme verschicken und wie Sie Ihr eigenes Programm dafür ausrüsten, Nachrichten von anderen Programmen zu empfangen.

Ein weiteres neues Konzept in KDE ist die Entwicklung von Komponenten, die zur Laufzeit in andere KDE-Programme eingebunden werden können. Kapitel 4.21, *Komponenten-Programmierung mit KParts*, erläutert das Prinzip und gibt Beispiele, wie Sie Komponenten in eigenen Programmen nutzen können.

Beim Anpassen von Programmen von Qt 1.x auf Qt 2.x, sowie von KDE 1.1 auf KDE 2.0, sind einige Punkte zu beachten, da die neueren Bibliotheken nicht vollständig kompatibel zu den alten sind. Wie Sie ein altes Programm anpassen und die neuen Möglichkeiten nutzen, erfahren Sie in Kapitel 4.22, *Programme von Qt 1.x auf Qt 2.x portieren*, und Kapitel 4.23, *Programme von KDE 1.x auf KDE 2.x portieren*.

4.1 Farben unter Qt

Wer schon einmal versucht hat, unter X11 Farben in seinen Programmen zu benutzen, wird die Probleme kennen, die sich ergeben: Viele Grafikkarten bieten nur eine begrenzte Anzahl von Farbwerten, die in einer Palette gespeichert sind, und diese Farben müssen mit Bedacht ausgewählt und mit den anderen Applikationen abgestimmt werden, damit sich auf dem Bildschirm brauchbare Farbkombinationen ergeben.

Qt bietet ein sinnvolles, komfortables und doch effizientes Konzept, um mit diesen Einschränkungen zu arbeiten und auf einfache Weise gute Ergebnisse zu erzielen.

4.1.1 Grundlagen der Farben unter X11

X11 bietet sechs verschiedene Farbtypen an, so genannte *Visuals*. Für einen eingestellten Grafikmodus ist jedoch in den meisten Fällen nur ein Visual nutzbar. Drei Visualtypen sind für Schwarzweiß- bzw. Graustufenmodi vorgesehen, die anderen drei für die Farbdarstellung. Wir wollen uns in unserer Betrachtung auf die drei Farbvisuals beschränken.

- Der Visualtyp *DirectColor* wird bei einem Grafikmodus mit einer sehr geringen Farbanzahl mit festgelegten Farbwerten angeboten, also in der Regel bei acht oder 16 Farben. Grafikkarten, die nur solche Modi bereitstellen, sind inzwischen selten geworden.

- Der Visualtyp *PseudoColor* bietet eine begrenzte Farbpalette mit frei wählbaren Farben. Am häufigsten tritt der Fall von 256 Einträgen in der Farbpalette auf. Dieser Visualtyp ist am kompliziertesten zu handhaben, denn die Einträge müssen sinnvoll gewählt werden. Sie sollten so verwendet werden, dass auch andere Applikationen sie nutzen können, und für Farbwerte, die nicht exakt in der Palette vorhanden sind, muss ein geeigneter Ersatzwert gefunden werden, der sich möglichst wenig von der benötigten Farbe unterscheidet.
- Der Visualtyp *TrueColor* bietet eine hohe Anzahl von fest vorgegebenen Farben, so dass jede benötigte Farbe exakt oder zumindest sehr gut näherungsweise dargestellt werden kann. Dieser Visualtyp umfasst alle Grafikmodi mit fester Palette und mehr als acht Bit pro Pixel. Gängige Werte sind dabei 15 Bit (entspricht 32.768 Farben), 16 Bit (65.536 Farben) und 24 Bit (16.777.216 Farben). Verwechseln Sie also nicht den TrueColor-Modus der Grafikkarte, der nur der 24-Bit-Modus ist, mit dem Visualtyp *TrueColor*.

Für den Visualtyp *PseudoColor* stellt der X-Server einige Möglichkeiten zur Verfügung, um mit den oben angesprochenen Problemen fertig zu werden. Jedem Fenster kann eine eigene Farbpalette zugeordnet werden, die das Programm selbstständig verändern und verwalten kann. Diese Palette wird immer dann aktiviert, wenn sich der Mauszeiger innerhalb des Fensters befindet. Dadurch werden jedoch die Farben der anderen Fenster falsch dargestellt, oftmals ist deren Inhalt sogar nicht mehr lesbar. Das Bildschirmflimmern aufgrund des Palettenwechsels ist auf Dauer auch sehr störend. Qt verzichtet daher auf diese Möglichkeit. Der Benutzer kann jedoch mit dem Kommandozeilenparameter `-cmap` die Verwendung einer eigenen Farbpalette erzwingen.

Statt einer eigenen Palette kann man auch die Defaultpalette benutzen, die sich alle Fenster teilen. In dieser Palette kann man Farben auf zwei verschiedene Arten festlegen (*Color Allocation*). Man kann einen (oder mehrere) Paletteneinträge im *ReadWrite-Modus* anfordern. Diese Einträge (falls es noch genügend freie Einträge gibt) werden dann für das eigene Programm reserviert und können jederzeit auf einen neuen Farbwert gesetzt werden. Da die Anzahl der Paletteneinträge jedoch begrenzt ist und die Einträge nach dem Verfahren »wer zuerst kommt, mahlt zuerst« vergeben werden, werden die freien Einträge bei mehreren farbhungrigen Programmen rasch knapp. Fordert man dagegen einen Farbeintrag im *ReadOnly-Modus* an, so erhält man einen Eintrag zugewiesen, den man selbst nicht mehr verändern kann und der auch von anderen Programmen genutzt werden kann. Die Vorgehensweise des X-Servers ist dabei folgende: Zuerst schaut der Server nach, ob es bereits einen Farbeintrag im Modus *ReadOnly* für genau die geforderte Farbe gibt. Ist das der Fall, so liefert der X-Server diesen Eintrag zurück. Sonst versucht er, einen freien Eintrag in der Farbpalette zu besetzen, und weist ihm die gesuchte Farbe zu. Sind bereits alle Einträge besetzt, so sucht der Server nach einer Farbe im *ReadOnly-Modus* in der Palette, die dem gesuchten Wert möglichst

ähnlich ist. Paletteneinträge im *ReadOnly*-Modus werden übrigens erst dann wieder freigegeben, wenn alle Programme, die diesen Eintrag genutzt haben, ihn wieder dealloziert haben.

Qt verzichtet vollständig auf die Benutzung von Farbeinträgen im *ReadWrite*-Modus. Wenn Sie auf diese Möglichkeiten des X-Servers nicht verzichten wollen, so können Sie natürlich selbst die Funktionen der X-Lib aufrufen. Diese Vorgehensweise hier zu beschreiben würde aber den Rahmen des Buches sprengen.

4.1.2 Farballokation unter Qt

Um dem Programmierer möglichst viel Arbeit bei der Palettenzuordnung abzunehmen, bietet Qt drei verschiedene Möglichkeiten, die Farbhungrigkeit eines Programms festzulegen. Zwei dieser drei Möglichkeiten sind für X-Server jedoch identisch, so dass nur zwei Arten übrig bleiben.

- *NormalColor* und *CustomColor* – diese beiden identischen Optionen sind für Programme vorgesehen, die kaum eigene Farben benutzen (nur Standard-Widgets sowie ein paar Pixmaps für Buttons in der Werkzeugleiste). Qt allokiert die benutzten Farben im *ReadOnly*-Modus aus der Defaultpalette immer dann, wenn sie benötigt werden. Da alle Standard-Widgets auf eine sehr geringe Anzahl von Farben zurückgreifen, sind die benötigten Farben mit großer Wahrscheinlichkeit bereits angelegt und sofort verfügbar.
- *ManyColor* – diese Option ist für Programme vorgesehen, die viele Farben möglichst exakt und schnell benötigen, zum Beispiel zur Darstellung von Fotos oder aufwendigen Farbverläufen. Für *PseudoColor*-Visuals allokiert Qt in diesem Fall 216 Farben im *ReadOnly*-Modus, die im so genannten *Farbwürfel* (*Color Cube*) angeordnet sind. Für jede der drei Grundfarben Rot, Grün und Blau stehen dann sechs verschiedene Helligkeitswerte zur Verfügung. Der Farbwürfel hat den Vorteil, dass man zu jeder benötigten Farbe sehr schnell eine zumindest grob angenäherte Farbe findet. Da auch viele andere Programme (z.B. der Netscape Navigator) den gleichen Farbwürfel verwenden, können die Einträge gemeinsam genutzt werden.

Beachten Sie, dass sich diese beiden Optionen nur für Visuals vom Typ *PseudoColor* unterscheiden. Für *DirectColor* und *TrueColor* werden die Farben immer direkt berechnet. Eine Allokation findet dann nicht statt.

Die benutzte Option wird im Hauptprogramm mit der statischen Methode `void QApplication::setColorSpec(int spec)` festgelegt, bevor das *QApplication*-Objekt erzeugt wird. *NormalColor* ist die Defaulteinstellung. Sie brauchen also diese Methode nur dann aufrufen, wenn Sie ein farbhungriges Programm entwickeln wollen. Das entsprechende Codestück sieht dann beispielsweise so aus:

```
int main (int argc, char **argv )
{
    QApplication::setColorSpec (QApplication::ManyColor);
    QApplication a (argv, argc);
    ...
}
```

Wenn Sie Farbverläufe betrachten, dann werden Sie eventuell die Feststellung machen, dass diese oftmals mit *ManyColor* grober und abgehackter aussehen als mit *NormalColor* oder *CustomColor*. Das passiert aber nur, solange noch genügend freie Farben zur Verfügung stehen. In diesem Fall werden bei *NormalColor* und *CustomColor* möglichst optimale Farben benutzt, während *ManyColor* nur die Farben aus dem Color Cube nutzen kann. Gehen die freien Farben aber zur Neige, müssen bei *NormalColor* oder *CustomColor* oft Farben benutzt werden, die den geforderten Farben nicht einmal entfernt ähnlich sind. Bei *ManyColor* hingegen bleibt die Qualität auch dann unverändert.

Neben dieser Einstellung hat der User noch die Möglichkeit, einige Optionen über Kommandozeilenparameter festzulegen. Sie werden bei der Erzeugung des *QApplication*-Objekts bestimmt (siehe auch Kapitel 3.3, *Grundstruktur einer Applikation*). Speziell zur Farbstrategie gibt es folgende Parameter:

- `-visual TrueColor` erzwingt die Benutzung eines *TrueColor*-Visuals. Wird ein solches Visual nicht angeboten, bricht das Programm mit einer Fehlermeldung ab.
- `-cmap` bewirkt die Benutzung einer eigenen Farbpalette für *PseudoColor*-Visuals. Diese Option kann sinnvoll sein, wenn andere Programme die Defaultfarbpalette blockieren und Qt daher keinen vernünftigen Color Cube allokalieren kann.
- `-ncols anzahl` bewirkt, dass für *PseudoColor*-Visuals und *ManyColor* der Farbwürfel mit *anzahl* Farben allokiert wird. Der Defaultwert ist 216 Farben. Ist *anzahl* 27, so wird ein Farbwürfel mit drei Helligkeitsstufen je Grundfarbe allokiert, für andere Werte wird der Farbwürfel angepasst.

4.1.3 Farben benutzen unter Qt

Für die Definition einer Farbe und die Berechnung des zugehörigen Pixel-Werts gibt es in Qt die Klasse *QColor*. Intern werden die Farben dort im RGB-Format gespeichert, mit acht Bit Genauigkeit für jede der drei Grundfarben Rot, Grün und Blau. Im Konstruktor kann man die Farbe festlegen. So erzeugt zum Beispiel

```
QColor himmelblau (190, 190, 255);
```

ein Objekt, das eine hellblaue Farbe repräsentiert. Die Farbanteile der Grundfarben werden in der Reihenfolge Rot – Grün – Blau angegeben und als Werte im

Bereich von 0 (Farbanteil nicht vorhanden) bis 255 (Farbanteil voll vorhanden) festgelegt. In unserem Beispiel haben wir starke Anteile von Rot und Grün, die der Farbe einen hellen, fast weißen Charakter geben, und den vollen Anteil Blau, der den Ausschlag für das vorherrschend blaue Erscheinungsbild gibt.

Dieses Objekt kann nun in Zeichenoperationen als Farbangabe für *QPen*- oder *QBrush*-Objekte benutzt werden (siehe Kapitel 4.2, *Zeichnen von Grafikprimitiven*). Das folgende Codestück zeichnet eine himmelblaue Linie der Dicke 5:

```
QPen stift (himmelblau, 5);
QPainter painter (this);
painter.setPen (stift);
painter.drawLine (10, 10, 50, 80);
```

Bevor das Zeichnen auf dem Bildschirm beginnen kann, muss zunächst ermittelt werden, welcher Pixelwert für die Darstellung dieser Farbe benutzt wird. Dieser Pixelwert ist beispielsweise der Index in die Farbpalette für *PseudoColor*-Visuals oder die Hardware-Darstellung des Farbwerts auf der Grafikkarte für *TrueColor*-Visuals. Die Ermittlung des Pixelwerts ist unter Umständen etwas aufwendiger. Sie kann entweder sofort nach Festlegung des Farbwerts vorgenommen werden oder erst, wenn der Pixelwert zum Zeichnen benötigt wird (so genannte *Lazy Allocation*). Welche dieser beiden Möglichkeiten benutzt werden soll, kann mit der statischen Methode

```
void QColor::setLazyAlloc (bool enable)
```

festgelegt werden. Die Lazy Allocation ist per Default aktiviert. Daher wird der Pixelwert nur berechnet, wenn die Farbe auch benutzt wird. Wenn man Zeichenoperationen mit mehreren Farben möglichst ohne jede Verzögerung durchführen will, kann man die Lazy Allocation deaktivieren, dann alle benötigten Farben definieren und anschließend die Lazy Allocation wieder aktivieren.

In der Klasse *Qt* sind bereits 19 Standardfarben (17 »normale« Farben und zwei Spezialfarben) als konstante Variablen definiert, die benutzt werden können. Tabelle 4.1 zeigt eine Liste der 17 Farben sowie ihrer Rot-, Grün- und Blauwerte.

Zwei weitere Farbkonstanten mit den Namen *color0* und *color1* sind definiert. Diese beiden Konstanten sind aber keinem RGB-Farbwert zugeordnet, sondern werden zum Zeichnen in eine so genannte Bitmap benutzt, in der jedes Pixel nur zwei Werte (0 oder 1) annehmen kann (siehe Kapitel 4.3, *Teilbilder – QImage und QPixmap*). Alle Pixel, die mit der Farbe *color0* gezeichnet werden, werden auf 0 gesetzt, mit der Farbe *color1* entsprechend auf 1.

Farbname	RGB-Anteile	Farbname	RGB-Anteile
<i>black</i>	(0, 0, 0)	<i>magenta</i>	(255, 0, 255)
<i>white</i>	(255, 255, 255)	<i>yellow</i>	(255, 255, 0)
<i>darkGray</i>	(128, 128, 128)	<i>darkRed</i>	(128, 0, 0)
<i>gray</i>	(160, 160, 164)	<i>darkGreen</i>	(0, 128, 0)
<i>lightGray</i>	(192, 192, 192)	<i>darkBlue</i>	(0, 0, 128)
<i>red</i>	(255, 0, 0)	<i>darkCyan</i>	(0, 128, 128)
<i>green</i>	(0, 255, 0)	<i>darkMagenta</i>	(128, 0, 128)
<i>blue</i>	(0, 0, 255)	<i>darkYellow</i>	(128, 128, 0)
<i>cyan</i>	(0, 255, 255)		

Tabelle 4-1 Farbkonstanten in Qt

Innerhalb von Methoden in einer Klasse, die von *Qt* abgeleitet ist (also auch *QWidget*), können Sie diese Konstanten wie globale Konstanten behandeln. Außerhalb einer solchen Klasse (z.B. in der Funktion *main*) müssen Sie die Klassenspezifikation *Qt::* voranstellen:

```
QPen stift (Qt::cyan, 3);
```

Kleine Icons können oftmals helfen, Applikationen übersichtlicher und intuitiver zu gestalten. Haupteinsatzgebiet ist hierbei die Werkzeugleiste, wie sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*, erläutert wurde. Wenn Sie selbst Icons für Ihr Programm entwerfen, so sollten Sie diese Icons in einer Version mit vielen Farben und in einer Version mit wenigen Farben entwerfen. Auf Systemen mit nur 256 Farben kann dann das zweite Icon gewählt werden. Benutzen Sie für dieses möglichst nur Farben aus der KDE-Iconpalette, um die Anzahl der benötigten Farben gering zu halten. Eine Liste der KDE-Farben und ihrer RGB-Werte finden Sie in Anhang C, *Die KDE-Standardfarbpalette*.

4.1.4 Das HSV-Farbmodell

Neben dem RGB-Modell kann die Klasse *QColor* auch das *HSV-Modell* verarbeiten. Dazu besitzt sie intern Methoden, um die HSV-Werte einer Farbe in RGB-Werte umzuwandeln und umgekehrt.

Im HSV-Modell (HSV steht für *Hue* = Farbton, *Saturation* = Sättigung und *Value* = Helligkeit) wird eine Farbe ebenfalls durch drei Werte charakterisiert. Der erste Wert, *Hue*, liegt im Bereich von 0 bis 360 und gibt an, unter welchem Winkel im Farbkreis sich der Farbton befindet (siehe Abbildung 4.1).

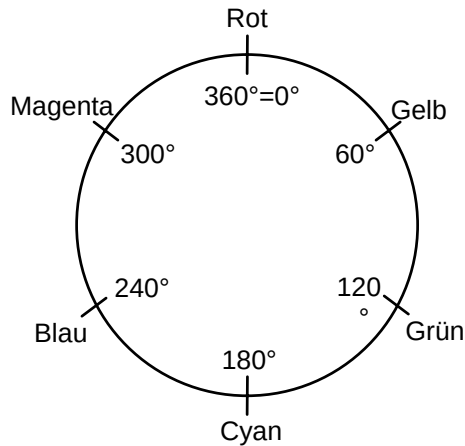


Abbildung 4-1 Der Farbkreis für den Wert Hue

Der zweite Wert, *Saturation*, gibt an, wie bunt die Farbe sein soll. Das entspricht etwa dem Wert, den Sie an Ihrem Fernseher mit der Sättigung einstellen. Der Wert 0 bezeichnet unbunte Farben (Grau), der Wert 255 eine sehr bunte, knallige Farbe. Der dritte Wert, *Value*, gibt die Helligkeit an. Ein Helligkeitswert von 0 bezeichnet Schwarz (unabhängig davon, wie die beiden anderen Werte gesetzt sind), eine Helligkeit von 255 bezeichnet die maximale Helligkeit für die eingestellte Farbe. Die Grundfarben haben HSV-Werte von (0, 255, 255) für Rot, (120, 255, 255) für Grün und (240, 255, 255) für Blau. Die verschiedenen Grautöne von Schwarz bis Weiß haben einen beliebigen Farbton, eine Sättigung von 0 und eine Helligkeit von 0 (Schwarz) bis 255 (Weiß).

Das HSV-Modell hat den Vorteil, dass es intuitiver ist. Intern arbeitet die Klasse *QColor* aber ausschließlich mit dem RGB-Modell und wandelt alle HSV-Werte unmittelbar um.

Um ein Objekt mit HSV-Werten zu setzen, kann man folgenden Konstruktor aufrufen:

```
QColor himmelblau (240, 128, 255, QColor::Hsv);
```

Oder man benutzt die Methode *setHsv*:

```
QColor himmelblau;  
himmelblau.setHsv (240, 128, 255);
```

QColor besitzt weiterhin die beiden Methoden *light* und *dark*, mit denen man zu der aktuellen Farbe eine hellere oder dunklere Farbvariante berechnen lassen

kann. Das ist besonders dann von Nutzen, wenn man verschieden helle Farbtöne für Schatteneffekte, z.B. auf den Rändern von Buttons, benötigt. Die Anwendung dieser Methoden kann beispielsweise so aussehen:

```
QColor mittelgruen (120, 255, 160, QColor::Hsv);
QColor hellgruen = mittelgruen.light (150);
QColor dunkelgruen = mittelgruen.dark (150);
```

Die Farbe in *hellgruen* ist dann um 50 % heller als *mittelgruen*; *dunkelgruen* ist um 50 % dunkler als *mittelgruen*. Zur Berechnung der Farben wird die Helligkeit von *mittelgruen* mit dem Faktor 1,5 multipliziert bzw. durch diesen Faktor geteilt. *hellgruen* erhält daher die HSV-Werte (120, 255, 240), *dunkelgruen* die Werte (120, 255, 107).

4.1.5 Farbkontexte

Oftmals braucht man Farben nur für eine bestimmte Zeit, danach können sie wieder freigegeben und damit anderen Programmen zur Verfügung gestellt werden. Ein typisches Beispiel: Beim Start der Applikation soll zunächst ein farbenprächtiges Logo präsentiert werden. Nach ein paar Sekunden soll dieses Logo wieder verschwinden, und damit werden auch die Farben, die für das Bild allokiert werden mussten, nicht mehr benötigt.

Zu diesem Zweck ermöglicht Qt die Einrichtung so genannter *Allocation Contexts*. Zu jedem *QColor*-Objekt wird gespeichert, in welchem Kontext es angelegt wurde. Am Anfang ist der Kontext 0 aktiv. Legt man nun einen neuen Kontext an (mit der statischen Methode *QColor::enterAllocContext*), so werden alle anschließend allokierten *QColor*-Objekte dem neuen Kontext zugeordnet. Man beendet diesen Kontext mit der Methode *QColor::leaveAllocContext*. Anschließend angelegte *QColor*-Objekte werden wieder dem alten Kontext zugeordnet. Mit der Methode *QColor::destroyAllocContext* kann man schließlich die Farben aus dem eigenen Kontext wieder freigegeben.

Für das oben angesprochene Problem des Logos sieht eine Lösung zum Beispiel wie folgt aus:

```
int myAllocContext = QColor::enterAllocContext ();
...
// hier können nun Farben allokiert und das Logo
// gezeichnet werden
...
QColor::leaveAllocContext ();
```

Nach einer Pause kann dann das Logo wieder vom Bildschirm entfernt werden. Die Farben, die für das Logo benutzt wurden, werden mit folgender Zeile wieder freigegeben:

```
QColor::destroyAllocContext (myAllocContext);
```

4.1.6 Effizienzbetrachtung

Die Kommunikation mit dem X-Server kann unter Umständen sehr langsam sein. Daher sollte die Allokation eines Farbwerts in der Farbpalette möglichst selten benutzt werden. Qt versucht mit mehreren Techniken, die Anzahl der X-Server-Zugriffe zu minimieren:

- Für *TrueColor*- und *DirectColor*-Visuals werden die Pixelwerte direkt berechnet. Eine Allokation einer Farbe ist nicht nötig – und somit ist auch kein Zugriff auf den X-Server erforderlich.
- Für *PseudoColor*-Visuals bei der Einstellung *ManyColor* wird nur beim Start des Programms der Farbwürfel allokiert. Anschließend werden die Farben nur noch aus diesem Würfel benutzt. Auch hier ist kein weiterer X-Server-Zugriff nötig.
- Für *PseudoColor*-Visuals bei der Einstellung *NormalColor* oder *CustomColor* wird eine Farbe nur dann allokiert, wenn es nicht bereits ein Element vom Typ *QColor* mit genau dieser Farbe gibt. Das wird durch eine effiziente Hash-Tabelle überprüft.

Wenn Sie eine spezielle Farbe öfter benötigen, können Sie die Anzahl der Allokationen reduzieren, indem Sie nur einmal ein Objekt der Klasse *QColor* erzeugen und dieses Element immer wieder verwenden. Durch einen speziellen Mechanismus ist es möglich, Farben zu definieren, ohne dass bereits eine Verbindung zum X-Server besteht. Die Allokation des Farbwerts wird in dem Fall wie bei der Lazy Allocation auf den Zeitpunkt des ersten Zugriffs verschoben. Man kann daher problemlos fest definierte Farben als Variable mit dem Zusatz *static* anlegen, wie in folgendem Beispiel:

```
void paintEvent (QPaintEvent *ev) {  
    static QColor himmelblau (190, 190, 255);  
    QPen stift (himmelblau, 1);  
    ...  
}
```

Durch den Zusatz *static* bleibt das *QColor*-Objekt *himmelblau* auch nach dem Ende der Methode *paintEvent* erhalten. Die Allokation findet nur einmal statt.

4.1.7 Farbbenutzung in Applikationen

Nachdem Sie erfahren haben, wie Sie Farben definieren und anwenden können, sollen Sie in diesem Abschnitt zur umsichtigen Benutzung von Farben aufgefordert werden. Der Einsatz von zu vielen Farben und die Missachtung der Farbpsychologie können eine Applikation unübersichtlich und schwer verständlich, in extremen Fällen sogar unbenutzbar machen. Sie sollten daher auf folgende Punkte achten:

- Vermeiden Sie so weit wie möglich, feste Farben zu benutzen. Eines der Ziele des KDE-Projekts ist es, dem Benutzer Applikationen zur Verfügung zu stellen, die von einem zentralen Programm – dem *KDE Control Center* – konfiguriert werden können, auch in der Farbgebung. Wenn ein Benutzer seinen Desktop in Herbstfarben konfiguriert hat, so wird ein Neon-Grün in Ihrem Programm nur schlecht ins Gesamtbild passen. Um auf die eingestellten Farben zurückzugreifen, benutzen Sie einfach die Widget-Palette (siehe Kapitel 4.1.8, *Die Widget-Farbpalette*). Für Dialoge ist es fast immer am günstigsten, wenn Sie die voreingestellten Farben nicht verändern.
- Seien Sie sparsam mit knalligen Farben. Sehr bunte Applikationen sehen vielleicht witzig, aber nur in den seltensten Fällen professionell aus.
- Dunkle Schrift auf hellem Hintergrund hat sich in den letzten Jahren als Standard etabliert.
- Rote Farbtöne erscheinen dem Betrachter näher, blaue weiter entfernt. Blau als Hintergrundfarbe und Rot als Vordergrundfarbe ist daher die natürliche Anordnung, die umgekehrte Anordnung wirkt störend und unruhig.
- Achten Sie auf ausreichenden Kontrast für Schrift. Wenn die Schriftfarbe und die Hintergrundfarbe ähnliche Farbtöne haben, ist der Text nur schwer lesbar. Bedenken Sie insbesondere, dass es durchaus noch Graustufen-Monitore gibt, auf denen Farben in Graustufen umgewandelt werden. Zwei Farben, die sich auf einem Farbmonitor gut voneinander abheben (z.B. Rot und Blau), können auf einem Graustufen-Monitor nahezu identische Helligkeitswerte haben. Wählen Sie also Farben, die auch einen starken Helligkeitsunterschied haben. Den größten Helligkeitsunterschied bieten in jedem Fall die Farben Weiß und Schwarz.
- Beachten Sie auch die intuitive Bedeutung, die Farben tragen: Rot signalisiert Gefahr oder einen Fehler, Blau und Grün sind dagegen Farben, die anzeigen, dass alles in Ordnung ist.

4.1.8 Die Widget-Farbpalette

Alle vordefinierten Widgets von Qt und KDE benutzen für die Darstellung keine festen Farbwerte, sondern verwenden Farben aus einer Liste mit derzeit 14 Einträgen, der so genannten Widget-Palette. Jeder Eintrag repräsentiert dabei eine typische »Rolle« für den Zeichenvorgang von Widgets, wie zum Beispiel die Hintergrundfarbe, die Textfarbe, verschiedene Farbabstufungen für 3D-Kanteneffekte und andere. Da die Palette geändert werden kann, kann die Farbgestaltung der Widgets geändert werden. Man kann so das Aussehen der Applikation an die Bedürfnisse des Programms oder den Geschmack des Anwenders anpassen. In einer KDE-Umgebung ist die Widget-Palette beispielsweise zentraler Bestandteil der so genannten *Themes*. Mit diesen kann der Anwender die Gestaltung seines

Desktops wählen. Anhand dieser Wahl werden die Widget-Paletten aller KDE-Programme entsprechend gesetzt, so dass ein einheitliches Aussehen entsteht.

Jede Widget-Instanz hat eine eigene Widget-Palette, die separat geändert werden kann. Somit ist es auch möglich, die Farben für ein einzelnes Widget gezielt zu ändern. So kann man einzelne Widgets durch leuchtende Farben hervorheben oder von Nachbar-Widgets absetzen.

Verwechseln Sie die oben beschriebene Widget-Farbpalette nicht mit einer Farbpalette der Grafikkarte (*Color Lookup Table*). Die Widget-Farbpalette ist nur eine Liste von Einträgen, die Farbwerte enthalten, auf die ein Widget-Objekt beim Darstellen auf dem Bildschirm zurückgreift.

Struktur der Widget-Farbpalette

Zum Abspeichern der Widget-Palette wird die Klasse *QPalette* benutzt. Jedes Widget enthält ein solches *QPalette*-Objekt, auf das mit der Methode *QWidget::palette()* zugegriffen werden kann. *QPalette* enthält drei Einträge vom Typ *QColorGroup*, die für drei verschiedene Zustände des Widgets zuständig sind. Diese Einträge haben die Bezeichnungen *Active*, *Inactive* und *Disabled*. Jedes der drei *QColorGroup*-Objekte enthält die bereits oben erwähnten 14 Farbeinträge, die jeweils vom Typ *QBrush* sind, also eine Farbe und ein Füllmuster speichern können. Der Zusammenhang ist noch einmal in Abbildung 4.2 dargestellt.

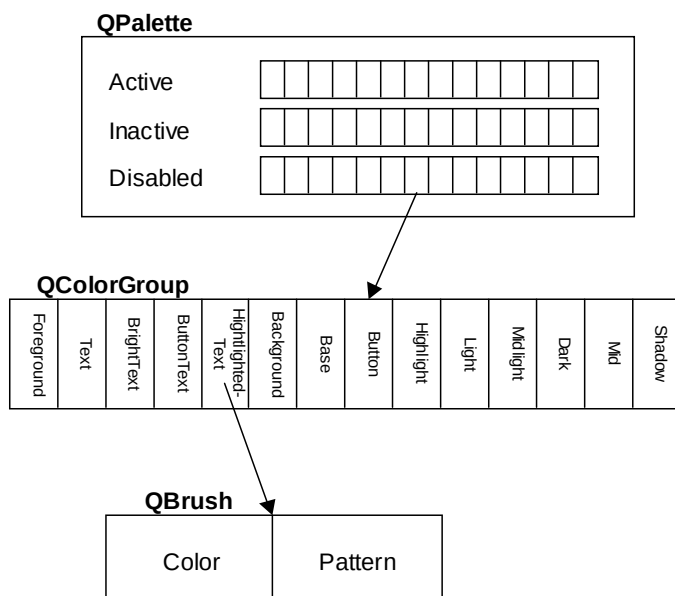


Abbildung 4-2 Aufbau eines *QPalette*-Objekts

Welcher der drei *QColorGroup*-Einträge zum Zeichnen benutzt wird, hängt vom Zustand des Widgets ab. Setzt man ein Widget mit der Methode *QWidget::setEnabled (false)* in einen deaktivierten Zustand, so wird zum Zeichnen das *QColorGroup*-Objekt *Disabled* benutzt. Es enthält farblose, kontrastarme Farben, so dass das Widget auf dem Bildschirm »ausgegraut« erscheint. Alle anderen Widgets, die bedient werden können, benutzen zum Zeichnen die Einträge *Active* und *Inactive*, die kontrastreiche und farbreichere Farbeinträge besitzen. *Active* wird von allen Widgets benutzt, die im gerade aktiven Fenster liegen, also in dem (Toplevel-)Fenster, das den Tastaturfokus besitzt. Die Widgets in allen anderen Fenstern benutzen den Eintrag *Inactive*. Häufig unterscheidet sich ein Farbeintrag in der *Active*-Gruppe nicht oder nur wenig vom entsprechenden Farbeintrag in *Inactive*, so dass die Darstellung in beiden Zuständen gleich oder fast gleich ist. Das typische Motif-Look macht beispielsweise hier keinen Unterschied, im Windows-98-Look sind dagegen die Farben im aktiven Fenster meist eine Nuance heller.

Ein *QColorGroup*-Objekt besitzt 14 Einträge vom Typ *QBrush*, die jeweils verschiedenen »Rollen« (*color role*) beim Zeichnen eines Widgets zugeordnet sind. Tabelle 4.2 enthält eine Liste der Rollen mit einer kurzen Beschreibung des typischen Einsatzgebiets sowie der typischen, zugeordneten Farbe. Welche Farbe tatsächlich benutzt wird, hängt bei KDE vom eingestellten *Theme* ab. Die Einträge lassen sich in drei Bereiche unterteilen: Einträge, die die Farbe zum Zeichnen von Linien und Beschriftungen festlegen, Einträge für die Hintergrundfarbe bzw. -füllmuster und Einträge für die Darstellung von Schattierungseffekten.

Name	Bedeutung und Anwendung	Standardwert
Foreground	Zeichenfarbe für Linien und Polygone, die im Vordergrund stehen (z.B. die Pfeile auf den Buttons eines Rollbalkens)	Schwarz
Text	Allgemeine Textfarbe, zum Beispiel für die Schrift eines <i>QLineEdit</i> -Widgets	Schwarz
BrightText	Textfarbe mit starkem Kontrast zur Hintergrundfarbe, meist identisch mit <i>Text</i>	Schwarz
ButtonText	Textfarbe für die Beschriftung von Buttons	Schwarz
Highlighted-Text	Textfarbe für ausgewählten oder markierten Text; meist hell, da der Hintergrund für ausgewählte Bereiche dunkel ist	Weiß
Background	Allgemeine Farbe (bzw. Füllmuster) für den Hintergrund	Hellgrau
Base	Alternative Hintergrundfarbe, meist für Eingabebereiche; zum Beispiel die Hintergrundfarbe eines <i>QLineEdit</i> -Widgets	Weiß
Button	Füllfarbe (und -muster) für die Fläche eines Buttons, oft identisch zu <i>Background</i> ; oder etwas heller als <i>Background</i>	Hellgrau

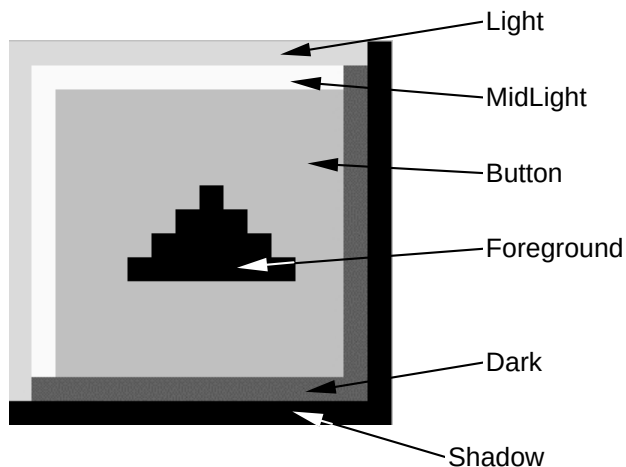
Tabelle 4-2 Die Einträge der Klasse *QColorGroup*

Name	Bedeutung und Anwendung	Standardwert
Highlight	Hintergrundfarbe für ausgewählte oder markierte Bereiche; für Widgets im Stil von Microsoft Windows dunkelblau, Beschriftung erfolgt in der Farbe in <i>HighlightedText</i>	Dunkelblau
Light	Viel heller als <i>Button</i> , für Schattierungseffekte	Weiß
Midlight	Etwas heller als <i>Button</i> , für Schattierungseffekte	Hellgrau
Dark	Dunkler als <i>Button</i> , für Schattierungseffekte	Dunkelgrau
Mid	Etwas dunkler als <i>Button</i> , für Schattierungseffekte	Mittelgrau
Shadow	Sehr viel dunkler als <i>Button</i> , für Schattierungseffekte	Schwarz

Tabelle 4-2 Die Einträge der Klasse *QColorGroup* (Forts.)

Da die Farbeinträge vom Typ *QBrush* sind, können sie nicht nur eine Farbe (*QColor*), sondern auch ein Füllmuster speichern. Das Füllmuster findet aber nur dann Anwendung, wenn beim Zeichnen des Widgets ein flächiger Bereich gezeichnet wird. Beim Zeichnen von Linien und Text wird das Füllmuster ignoriert und nur der Farbwert benutzt. Die Einträge, die typischerweise mit einem Füllmuster genutzt werden, sind *Background*, *Base*, *Button* und *Highlight*.

In den Abbildungen 4.3 und 4.4 ist beispielhaft in einer Vergrößerung gezeigt, wie die Einträge in einem *QColorGroup*-Objekt von Widgets benutzt werden.

Abbildung 4-3 *QColorGroup*-Einträge beim Zeichnen eines Buttons

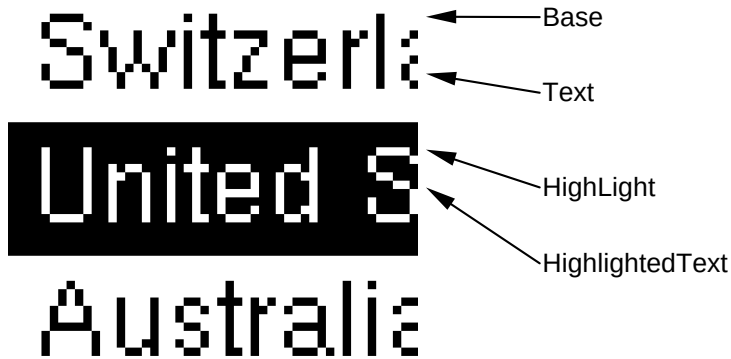


Abbildung 4-4 QColorGroup-Einträge beim Zeichnen eines QListBox-Objekts

Widget-Palette in selbst definierten Bedienelementen

Wenn Sie eigene Widgets entwerfen, sollten Sie beim Darstellen auf dem Bildschirm möglichst fest einprogrammierte Farben vermeiden und nur die Farben aus der Widget-Palette verwenden. So ist gewährleistet, dass Ihr Widget optisch zu den anderen Elementen passt, und dass es in einer KDE-Umgebung sich ebenfalls dem gewählten Theme anpasst. Eine Anleitung und Beispiele dazu finden Sie in Kapitel 4.4, *Entwurf eigener Widget-Klassen*.

Ändern der Widget-Palette eines Widgets

Mit der Methode `QWidget::palette()` können Sie die aktuelle Palette eines Widgets erhalten, allerdings nur als konstante Referenz. Diese Palette können Sie also nicht ändern. (Wenn Sie die Warnungen Ihres Compilers ignorieren, funktioniert das zwar dennoch, aber Sie sollten davon keinen Gebrauch machen.)

Um die Palette eines Widgets zu ändern, setzen Sie mit der Methode `QWidget::setPalette()` ein vollständig neues Paletten-Objekt ein. Es stellt sich jetzt also nur die Frage, wie man ein solches Paletten-Objekt erzeugt.

Eine Möglichkeit ist es, drei `QColorGroup`-Objekte zu erzeugen, jedes mit 14 Einträgen vom Typ `QBrush`, und diese zu einem `QPalette`-Objekt zusammenzufassen. Das ist möglich, jedoch sehr aufwendig, da dann insgesamt 42 Farben festgelegt werden müssten.

Einfacher ist es, eine bereits existierende Palette zu benutzen und in ihr nur einzelne Einträge zu ändern. Mit der statischen Methode `QApplication::palette()` erhalten Sie die Standardpalette für diese Applikation. Mit der Methode `QPalette::setColor()` oder `QPalette::setBrush()` können Sie nun einzelne Einträge in der Palette ändern. Dazu geben Sie als ersten Parameter die Gruppe an (`QPalette::Active`, `QPalette::Inactive` oder `QPalette::Disabled`), als zweiten Parameter die Rolle (`QColor-`

Group::Background, ...) und als dritten Parameter den neuen Wert (als *QColor*-Objekt oder als *QBrush*-Objekt). Sie können den ersten Parameter auch weglassen. In diesem Fall wird der Eintrag in allen drei Gruppen geändert.

Hier sehen Sie als Beispiel den Quelltext für ein *QPushButton*-Widget, das statt des üblichen Grau in Rot dargestellt werden soll (zum Beispiel als Alarm-Knopf):

```
QPushButton *b = new QPushButton ("Alarm", this);
QPalette p = QApplication::palette();
p.setColor (QColorGroup::Button, Qt::red);
b->setPalette (p);
```

Diese Vorgehensweise hat aber einen Nachteil: Ändert man nur einen einzelnen Eintrag, so passen die anderen Einträge oftmals farblich nicht zu dem geänderten. In unserem Fall zum Beispiel ist nun die Fläche des Knopfes rot, die Kanten des Knopfes, die in einem 3D-Effekt abgeschrägt dargestellt werden, sind dagegen noch immer grau. Besser wäre es, wenn auch diese in verschiedenen Abstufungen der Farbe Rot (Hellrot am oberen und linken Rand, Dunkelrot am unteren und rechten Rand) dargestellt würden. Besonders auffällig ist der Nachteil in diesem konkreten Fall, wenn der Motif-Look gewählt wurde. Hier wird nämlich die Fläche des Knopfes im gedrückten Zustand nicht mit dem Eintrag *QColorGroup::Button*, sondern mit dem Eintrag *QColorGroup::Mid* gezeichnet. Somit ist ein nicht gedrückter Knopf rot, ein gedrückter aber grau.

Anstatt nun alle relevanten Einträge der Palette zu ändern, kann man viel einfacher einen Konstruktor von *QPalette* benutzen, dem man die Farbe für die Rollen *Button* und *Background* übergibt. Alle anderen Einträge werden passend aus diesen beiden Farben ermittelt: *Foreground* so, dass es möglichst viel Kontrast zu *Background* hat, und die Schattierungseffekte als dunklere und hellere Farben zu *Button*. Die Einträge in der *Disabled*-Gruppe werden automatisch als farblose, kontrastarme Farben errechnet. Das entsprechende Listing sieht folgendermaßen aus:

```
QPushButton *b = new QPushButton ("Alarm", this);
QColor backgrd =
    QApplication::palette().active().background();
QPalette p (Qt::red, backgrd);
b->setPalette (p);
```

In diesem Fall wählen wir als Farbe für *Button* Rot, als Farbe für *Background* übernehmen wir den Eintrag, der bereits in der Standardpalette gesetzt ist.

Ändern der Widget-Palette aller Widgets

Sie können die Farbgestaltung Ihres Programms auch vollständig verändern, indem Sie die Widget-Palette aller Widgets ändern. Erzeugen Sie dazu ein ent-

sprechendes *QPalette*-Objekt mit den gewünschten Farbeinträgen, und rufen Sie damit die statische Methode *QApplication::setPalette()* auf:

```
// Palette mit hellgrünen und hellblauen Farbtönen
QPalette pal (Qt::green.light (200), Qt::blue.light (180));

// Für alle Widgets setzen
QApplication::setPalette (pal, true);
```

Wenn Sie – wie hier – als zweiten Parameter *true* übergeben, werden auch alle bereits geöffneten Widgets entsprechend geändert, andernfalls betrifft die Palettenänderung nur die Widgets, die in Zukunft erzeugt werden. Als dritten Parameter können Sie außerdem einen Klassennamen angeben. Die Palettenänderung betrifft dann nur Widgets dieser oder einer abgeleiteten Klasse. Geben Sie nichts an – wie hier –, so sind alle Klassen betroffen.

4.1.9 Zusammenfassung

- Wenn Sie eine farbhungrige Applikation entwickeln, die die Farben möglichst unverfälscht wiedergeben soll (z.B. zur Darstellung von Fotos oder komplexen Farbverläufen), so setzen Sie *vor* der Erzeugung des *QApplication*-Objekts (bzw. *KApplication*-Objekts) die Farbstrategie:

```
QApplication::setColorSpec (QApplication::ManyColor);
```

- Eine Farbe legen Sie mit der Klasse *QColor* fest, indem Sie beispielsweise den Konstruktor mit den Werten für die Rot-, Grün- und Blauanteile aufrufen. Anschließend können Sie die Farbe zum Beispiel für die Konstruktion eines Objekts der Klasse *QPen* oder *QBrush* benutzen.
- Folgende Farbkonstanten vom Typ *QColor* sind bereits von Qt definiert und können benutzt werden: *black*, *white*, *darkGray*, *gray*, *lightGray*, *red*, *green*, *blue*, *cyan*, *magenta*, *yellow*, *darkRed*, *darkGreen*, *darkBlue*, *darkCyan*, *darkMagenta*, *darkYellow*. Weiterhin sind die Farbkonstanten *color0* und *color1* definiert, die benutzt werden, um in einem *QBitmap*-Objekt die Pixel auf den Wert 0 oder 1 zu setzen. Außerhalb von Methoden in Klassen, die von der Klasse *Qt* abgeleitet sind, muss man die Klassenspezifikation *Qt::* voranstellen.
- Da die häufige Allokation eines Farbwerts unter Umständen ineffizient sein kann, sollten Sie eine spezielle Farbe, die Sie häufig benutzen, nur einmal in einem *QColor*-Objekt speichern und dann immer auf dieses Objekt zurückgreifen.
- Damit Ihr KDE-Programm von der Farbgebung her zu den Einstellungen passt, sollten Sie möglichst nur die Farben aus der Widget-Palette benutzen und selbst definierte Farben vermeiden.

- Sie können die Farben eines einzelnen Widgets ändern, indem Sie mit `QWidget::setPalette()` eine neue Widget-Palette setzen. Ein entsprechendes `QPalette`-Objekt erzeugen Sie am besten, indem Sie im `QPalette`-Konstruktor zwei Farbwerte für *Button* und *Background* angeben. Die anderen Einträge werden automatisch sinnvoll gewählt.
- Mit `QApplication::setPalette()` können Sie die Widget-Palette aller Widgets ändern lassen (entweder nur für alle neu erzeugten Widgets oder auch für alle bereits bestehenden Widgets). So können Sie die Farbgestaltung der gesamten Applikation anpassen. KDE-Programme sollten darauf verzichten, da diese Palette bereits entsprechend dem gewählten Theme gesetzt wird.

4.1.10 Übungsaufgaben

Übung 4.1

Bestimmen Sie ungefähre RGB- und HSV-Werte für die Regenbogenfarben Rot, Orange, Gelb, Grün, Blau, Violett.

Übung 4.2

Welche Farben verbergen sich hinter den RGB-Werten (255, 255, 128), (80, 80, 80) und (180, 220, 180) sowie hinter den HSV-Werten (30, 220, 100), (330, 200, 80) und (0, 80, 255)?

Übung 4.3

Schreiben Sie ein kurzes Programm, das mit Hilfe der Klasse `QColor` RGB-Werte in HSV-Werte umwandelt und umgekehrt. Geben Sie zu allen Farben aus Übung 4.2 auch die zugehörigen HSV- bzw. RGB-Werte an. Muss das Programm ein Objekt der Klasse `QApplication` (bzw. `KApplication`) erzeugen?

Übung 4.4

Erzeugen Sie eine Reihe von sechs Schaltflächen in den Regenbogenfarben. Benutzen Sie dazu die Farbwerte aus Übung 4.1.

Übung 4.5

Erzeugen Sie eine neue Klasse `QLEDNumber`, abgeleitet von `QLCDNumber`. Diese Klasse soll ebenfalls Zahlen auf dem Bildschirm darstellen, aber in Rot vor schwarzem Hintergrund.

4.2 Zeichnen von Grafikprimitiven

Nachdem in Kapitel 4.1, *Farben unter Qt*, beschrieben wurde, wie man eine Farbe auswählt, wird es nun Zeit, diese Farbe zu benutzen, um Zeichnungen auf dem Bildschirm (oder in einer Pixmap oder auf dem Drucker) auszuführen. Qt stellt dafür einige Klassen zur Verfügung, die verschiedene Aufgaben übernehmen.

Die Klasse *QPaintDevice* repräsentiert ein Objekt, auf dem gezeichnet werden kann. Von dieser abstrakten Klasse werden vier Klassen abgeleitet, die konkrete Geräte ansteuern: *QWidget* ist ein Fenster auf dem Bildschirm, *QPixmap* (und davon nochmals abgeleitet *QBitmap*) ist ein rechteckiger Bildbereich im Speicher des X-Servers, *QPrinter* ist der Drucker, und *QPicture* ist eine Hilfsklasse, die Zeichenoperationen speichert und wiederholen kann.

Die Klasse *QPainter* übernimmt die Zeichenoperationen. Um in ein Objekt der Klasse *QPaintDevice* zu zeichnen, muss man ein neues Objekt der Klasse *QPainter* erzeugen und es dem *QPaintDevice* zuordnen. Dann kann man die Methoden des *QPainter* aufrufen, die die Zeichenoperationen ausführen. Die Operationen von *QPainter* sind sehr umfangreich: Sie reichen vom Zeichnen einzelner Pixel und Linien über Rechtecke, Polygone, Kreise, Ellipsen, Kreissegmente bis zu Texten und Pixmapen. All diese Operationen lassen sich zusätzlich mit Koordinatentransformationen versehen, wodurch die gezeichneten Elemente verschoben, vergrößert, verkleinert, gedreht und gesichert werden können. Hinzu kommen noch sehr mächtige Clipping-Methoden, mit denen man den gezeichneten Bereich auf beliebig geformte Ausschnitte einschränken kann.

Mit Hilfe der Klassen *QPen* und *QBrush* können die Linienart und -dicke sowie die Füllmuster für die Operationen in *QPainter* festgelegt werden.

Ein UML-Diagramm in Abbildung 4.5 verdeutlicht die Zusammenhänge der genannten Klassen zueinander noch einmal. Auf die einzelnen Unterklassen von *QPaintDevice* gehen wir später in Kapitel 4.2.6, *Unterklassen von QPaintDevice*, noch genauer ein.

Das folgende Minimalbeispiel soll das Zusammenspiel der verschiedenen Klassen erläutern:

```
#include <qapplication.h>
#include <qwidget.h>
#include <qpainter.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QWidget *w = new QWidget ();
    w->resize (300, 200);
    w->show();
}
```

```

QPainter p;
p.begin (w);
p.setPen (QPen (Qt::black, 2, Qt::SolidLine));
p.setBrush (QBrush (Qt::black, Qt::DiagCrossPattern));
p.drawEllipse (10, 10, 280, 180);
p.end();

app.setMainWidget (w);
return app.exec ();
}

```

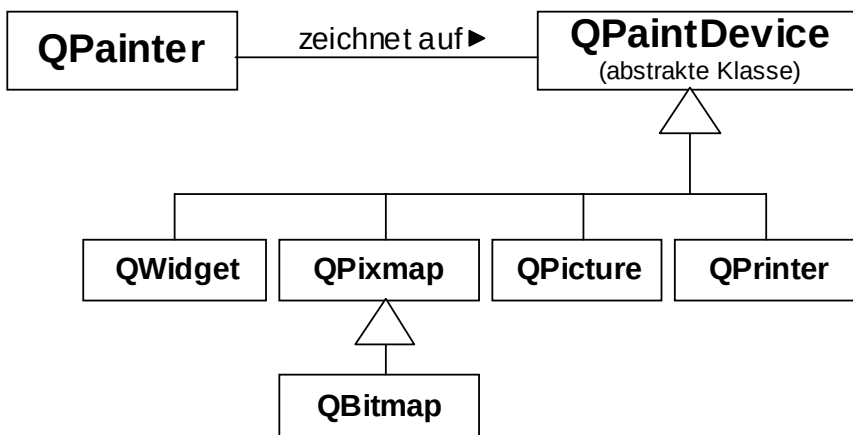


Abbildung 4-5 Zusammenhang zwischen den Klassen

Hinweis: Sie brauchen das Programm nicht zu testen, da es noch einige Mängel enthält. Wir werden weiter unten ein besseres Programm vorstellen.

Das Programm erzeugt zunächst einmal eine Instanz der Klasse *QApplication*. Anschließend wird ein Objekt der Klasse *QWidget* erzeugt, die ja eine Unterklasse von *QPaintDevice* ist. Die Größe des Widgets wird festgelegt, und das Widget wird mit der Methode *show* auf dem Bildschirm dargestellt.

Anschließend erzeugen wir ein Objekt der Klasse *QPainter* und verbinden dieses Objekt mittels der Methode *begin* mit dem *QPaintDevice*. Wir setzen die Linienart und das Füllmuster von *painter* und zeichnen dann mit der Methode *drawEllipse* einen Kreis in das Widget. Dann schließen wir die Zeichenoperation noch mit *end* ab und starten anschließend die Eventschleife, die in diesem Fall eine Endlosschleife ist.

Die Ausgabe, die dieses Programm auf dem Bildschirm erzeugt, ist in Abbildung 4.6 dargestellt. Auf einigen Systemen kann es aber auch sein, dass das Fenster leer bleibt (siehe unten).

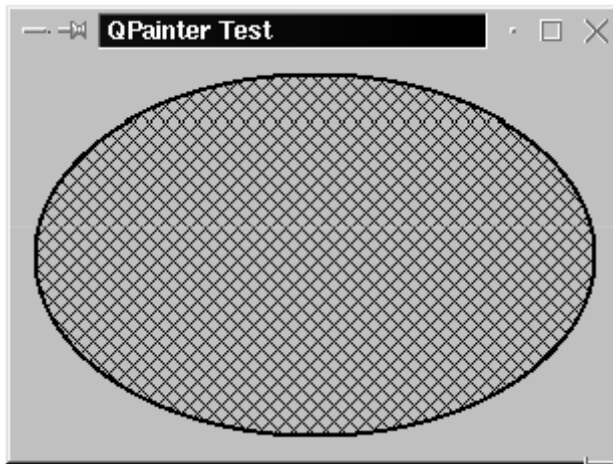


Abbildung 4-6 Ergebnis des Minimalbeispiels

Unser kleines Programm hat zwei entscheidende Probleme:

- Auf manchen Systemen kann es sein, dass der Aufruf der Zeichenroutine bereits stattfindet, bevor der X-Server das Fenster auf dem Bildschirm dargestellt hat. In diesem Fall läuft der Zeichenbefehl ins Leere, das Fenster bleibt auch nachher leer. Es ist noch keine Möglichkeit vorgesehen, dass der Zeichenbefehl erst abgesetzt wird, wenn das Fenster bereit ist.
- Die Zeichenoperationen wirken direkt auf das Widget. Dieses Widget kann aber ganz oder teilweise von einem anderen Fenster verdeckt sein. Wird der verdeckte Teil anschließend wieder aufgedeckt, so wird dieser Teil nicht nachgezeichnet, sondern einfach mit der Hintergrundfarbe gefüllt (siehe Abbildung 4.7).

Um diese beiden Probleme zu beseitigen, ist die normale Vorgehensweise zum Zeichnen in ein Widget daher auch eine andere: Die Zeichenoperationen werden ausschließlich in der Methode *paintEvent* des Widgets ausgeführt. Diese Methode wird automatisch jedes Mal dann aufgerufen, wenn ein Teil des Widgets neu gezeichnet werden muss – zum Beispiel wenn ein Teil des Fensters wieder aufgedeckt wurde, aber auch wenn das Widget zum ersten Mal mit *show* angezeigt wird.

Beachten Sie, dass wir nun nur noch auf Anfragen des X-Servers reagieren. Wir zeichnen nur noch dann in das Widget, wenn wir die Meldung bekommen, dass ein Neuzeichnen nötig geworden ist.

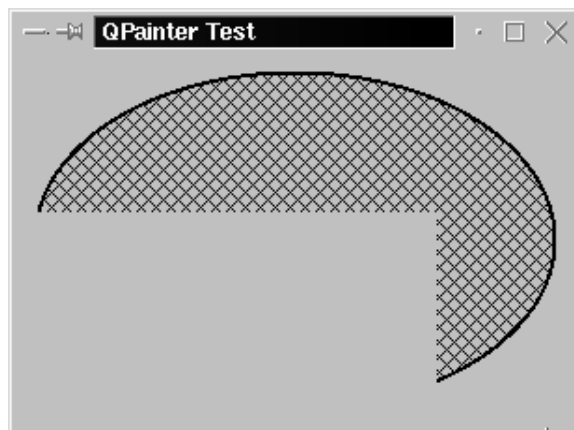


Abbildung 4-7 Kein Neuzeichnen der Ellipse im Minimalprogramm

Im folgenden Kapitel 4.2.1, *QPainter*, werden wir eine entsprechend verbesserte Version unseres Beispielprogramms entwerfen. Mit diesem Programm werden wir der Reihe nach alle Zeichenbefehle, die *QPainter* bietet, vorstellen.

Bei unserem Beispiel lassen wir unser *QPainter*-Objekt immer direkt in ein Fenster, also in ein *QWidget*-Objekt zeichnen. *QPainter* kann aber ebenso in die anderen von *QPaintDevice* abgeleiteten Klassen zeichnen, nämlich in *QPicture*, *QPixmap* und *QPrinter*. In Kapitel 4.2.6, *Unterklassen von QPaintDevice*, schauen wir uns diese Klassen etwas genauer an.

4.2.1 QPainter

Die Klasse *QPainter* übernimmt die Ausführung von Zeichenbefehlen auf einem *QPaintDevice*. Die Zeichenbefehle sind sehr mächtig, und da sie direkt an den X-Server weitergeleitet werden (im Fall von *QWidget* und *QPixmap*), werden sie auf beschleunigten Grafikkarten sehr effizient ausgeführt. *QPainter* bietet eine große Anzahl von Linienarten und Füllmustern an und kann sehr komplexe Transformationen und Clipping-Bereiche anwenden.

Um ein *QPainter*-Objekt zu benutzen, geht man in folgenden Schritten vor:

1. Erzeuge ein *QPainter*-Objekt mit der Adresse des *QPaintDevice* als Argument im Konstruktor.
2. Setze mit den Methoden *setPen* und *setBrush* geeignete Linienarten und Füllmuster, falls nicht die Defaultwerte benutzt werden sollen.
3. Setze gegebenenfalls eine Clipping-Region, eine Transformationsmatrix usw.
4. Führe die Zeichenbefehle aus (*drawPoint*, *drawLine*, *drawEllipse*, *drawText* usw.).
5. Lösche das *QPainter*-Objekt.

Auf einem *QPaintDevice* kann zu einem Zeitpunkt immer nur ein einziges *QPainter*-Objekt aktiv sein. Daher sollte man das *QPainter*-Objekt möglichst sofort löschen, nachdem man die Zeichenoperationen ausgeführt hat.

Alternativ kann man das *QPainter*-Objekt auch ohne Argument im Konstruktor erzeugen. Eine Verbindung zu einem *QPaintDevice* stellt man dann mit der Methode *QPainter::begin* her. Diese Verbindung muss anschließend *unbedingt* mit *QPainter::end* wieder beendet werden. Auf diese Weise kann ein *QPainter*-Objekt für verschiedene Widgets benutzt werden. Da die Werte für Linienart, Füllmuster, Clipping-Region und Transformation bei jedem Aufruf von *begin* auf die Defaultwerte des *QPaintDevice* gesetzt werden, bietet diese Methode keinen Vorteil gegenüber der Möglichkeit, jedes Mal ein neues *QPainter*-Objekt zu erzeugen und anschließend wieder zu löschen.

Die Zeichenoperationen eines *QPainter*-Objekts werden aus Effizienzgründen zunächst in einer Warteschlange gehalten. Beim Beenden des *QPainter* (entweder durch Löschen des *QPainter*-Objekts oder durch die Methode *end*) werden automatisch alle noch gespeicherten Operationen ausgeführt. Wenn Sie eine sofortige Ausführung der noch ausstehenden Operationen erzwingen wollen, rufen Sie die Methode *QPainter::flush ()* auf. Das kann zum Beispiel dann nötig sein, wenn Sie in eine *QPixmap* zeichnen, die Sie benutzen wollen, ohne jedoch den Painter zu beenden.

Für ein paar Experimente, um die Möglichkeiten von *QPainter* zu demonstrieren, benutzen wir das folgende kleine Testprogramm, das ein *QPainter*-Objekt auf ein *QWidget* anwendet. Dazu definieren wir eine neue Klasse, die von *QWidget* abgeleitet ist. In der Methode *paintEvent* führt sie dann die gewünschten Zeichenoperationen aus.

Die Datei *painterwidget.h*:

```
#ifndef _PAINTERWIDGET_H_
#define _PAINTERWIDGET_H_

#include <qwidget.h>
class QPaintEvent;

class PainterWidget : public QWidget
{
    Q_OBJECT

public:
    PainterWidget ();
protected:
    void paintEvent (QPaintEvent *);
};

#endif
```

Die Datei *painterwidget.cpp*:

```
#include "painterwidget.h"
#include <qpainter.h>

PainterWidget::PainterWidget ()
    : QWidget (0, 0)
{
    setFixedSize (200, 100);
}

void PainterWidget::paintEvent (QPaintEvent *)
{
    QPainter p (this);

    // An dieser Stelle werden die
    // Zeichenoperationen eingefügt
}
```

Die Datei *main.cpp*:

```
#include "painterwidget.h"
#include <qapplication.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    PainterWidget *w = new PainterWidget ();
    w->show();

    app.setMainWidget (w);
    return app.exec();
}
```

Kompilieren Sie das Programm wie üblich. Vergessen Sie nicht, die Klassendeklaration in der Datei *painterwidget.h* mit *moc* zu bearbeiten und die daraus entstandene Datei ebenfalls zu kompilieren und einzubinden.

Dieses einfache Programm hat nun bereits einige zusätzliche Features gegenüber unserem Einführungsbeispiel. Das Zeichnen findet jedes Mal beim Aufruf der virtuellen Methode *paintEvent* statt. Damit wird gewährleistet, dass die Zeichnung immer erneut gezeichnet wird, falls ein verdeckter Teil des Fensters aufgedeckt wird, die Fenstergröße geändert wird und auch, sobald das Fenster zum ersten Mal auf dem Bildschirm dargestellt wird.

Das *QPainter*-Objekt wird als lokale Variable in der Methode *paintEvent* erzeugt und beim Verlassen der Methode automatisch wieder gelöscht. Dieses ist die einfachste und am häufigsten verwendete Art, in ein *QWidget* zu zeichnen.

Wir verwenden den an *paintEvent* übergebenen Parameter vom Typ *QPaintEvent* nicht. Daher brauchen wir auch keinen Parameternamen anzugeben. (Würden wir ihn angeben, so bekämen wir die Warnung *unused parameter*.) Außerdem brauchen wir die Header-Datei *qevent.h*, die die Deklaration dieser Klasse enthält, nicht einzubinden, da wir die Klasse nirgends benutzen. Allerdings muss der Datentyp deklariert sein, was in *painterwidget.h* durch die Zeile *class QPaintEvent*; geschieht.

Im Folgenden werden die verschiedenen Grafikprimitive vorgestellt, die das *QPainter*-Objekt zeichnen kann. Fast alle Zeichenmethoden gibt es in zwei Varianten, die sich nur in der Anzahl und Art der Parameter unterscheiden. In der einen Variante werden Koordinaten durch zwei oder vier *int*-Werte angegeben, in der anderen durch Objekte der Klasse *QPoint* oder *QRect*.

Methoden zum Füllen von Rechteckbereichen

Zwei sehr einfache Methoden, *fillRect* und *eraseRect*, dienen zum Füllen eines rechteckigen Fensterbereichs mit einer Farbe oder einem Muster. Im Gegensatz zu den mit *drawRect* gezeichneten Rechtecken, die im übernächsten Abschnitt, *Methoden zum Zeichnen ausgefüllter Primitive*, vorgestellt werden, wird mit diesen Methoden der Rand nicht mit einer Linie umschlossen. Zum Füllen benutzt *fillRect* das Muster, das im *QBrush*-Parameter übergeben wird, *eraseRect* benutzt das Hintergrundmuster. Keine der beiden Methoden benutzt das mit *setBrush* eingestellte Füllmuster (siehe Kapitel 4.2.2, *Linienarten und Füllmuster*). Außerdem werden die angegebenen Koordinaten nicht der eingestellten Transformation unterworfen (siehe Kapitel 4.2.3, *Koordinaten-Transformationen*).

Ein Aufruf von *fillRect*, um ein Rechteck der Breite 100 Pixel und der Höhe 50 Pixel mit der oberen linken Ecke an den Koordinaten (30,40) mit weißer Farbe zu füllen, kann zum Beispiel so aussehen:

```
p.fillRect (30, 40, 100, 50, QBrush (white, SolidFill));
```

Entsprechend können Sie *eraseRect* benutzen, um dieses Rechteck wieder mit der Hintergrundfarbe zu füllen:

```
p.eraseRect (30, 40, 100, 50);
```

Diese beiden Methoden sind sehr einfach, aber auch sehr effizient.

Methoden zum Zeichnen einfacher Strichprimitive

Die folgenden Primitive bestehen nur aus Punkten oder Linien. Zu ihrer Darstellung wird nur der aktuelle Wert des *QPen* berücksichtigt (siehe Kapitel 4.2.2, *Linienarten und Füllmuster*). Es entstehen keine Flächen, die mit dem *QBrush* ausgefüllt werden könnten.

drawPoint (int x, int y)

drawPoint (const QPoint &p)

Zeichnet einen einzelnen Punkt. Auf einem *QWidget*- oder einem *QPixmap*-Objekt entspricht dies immer einem Pixel (unabhängig von der aktuell eingestellten Liniendicke). In Abbildung 4.8 sehen Sie zwei kleine Pixel an den Stellen (60,50) und (140,50).

```
p.drawPoint (60, 50);  
p.drawPoint (140, 50);
```



Abbildung 4-8 *drawPoint*

drawPoints (const QPointArray &a, int index = 0, int npoints = -1)

Zeichnet *npoints* Punkte ab Index *index* aus dem Koordinatenarray *a* (siehe Abbildung 4.9). Mit den Defaultwerten für *index* und *npoints* werden alle Punkte des Arrays gezeichnet.

```
QPointArray a (21);  
for (int i = 0; i <= 20; i++)  
    a.setPoint (i, 50 + 5 * i, 20 + 3 * i);  
p.drawPoints (a);
```

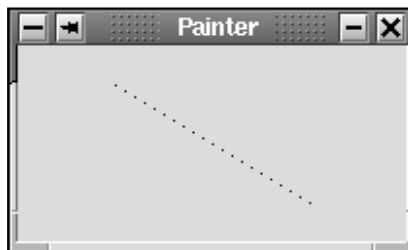


Abbildung 4-9 *drawPoints*

drawLine (int x1, int y1, int x2, int y2)

drawLine (const QPoint &p1, const QPoint &p2)

Zeichnet eine Linie von Punkt (x_1, y_1) zu Punkt (x_2, y_2) bzw. von Punkt p_1 zu Punkt p_2 (siehe Abbildung 4.10).

```
p.drawLine (20, 80, 180, 20);
```

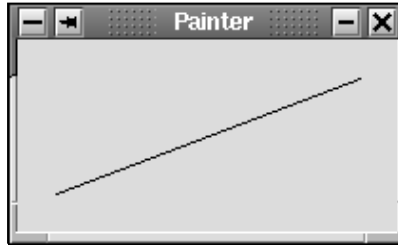


Abbildung 4-10 drawLine

moveTo (int x, int y)

moveTo (const QPoint &p)

lineTo (int x, int y)

lineTo (const QPoint &p)

Bewegt den »aktuellen Punkt« zur Position (x, y) bzw. zu p . *moveTo* setzt dabei nur den aktuellen Punkt neu, *lineTo* zeichnet eine Linie vom letzten aktuellen Punkt zum neuen Punkt. Der aktuelle Punkt wird nur von diesen vier Methoden verändert (siehe Abbildung 4.11).

```
p.moveTo (70, 95);  
p.lineTo (130, 95);  
p.lineTo (130, 35);  
p.lineTo (70, 35);  
p.lineTo (100, 5);  
p.lineTo (130, 35);  
p.lineTo (70, 95);  
p.lineTo (70, 35);  
p.lineTo (130, 95);
```

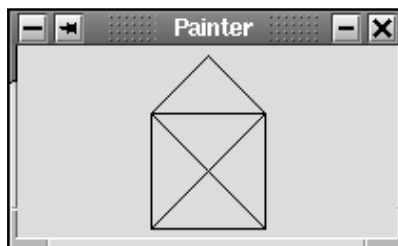


Abbildung 4-11 moveTo und lineTo

drawLineSegments (*const QPointArray &a*, *int index = 0*, *int nlines = -1*)

Zeichnet *nlines* Linien aus den Koordinaten in *a* zwischen den Punkten *index* und *index + 1*, zwischen *index + 2* und *index + 3* usw. Die Linien sind also nicht miteinander verbunden (siehe Abbildung 4.12). Um einen Linienzug zu zeichnen, benutzen Sie am besten die Methode *drawPolyLine*.

```
QPointArray a (20);
for (int i = 0; i < 10; i++) {
    a.setPoint (2 * i, 50 + 10 * i, 20);
    a.setPoint (2 * i + 1, 50 + 10 * i, 80);
}
p.drawLineSegments (a);
```

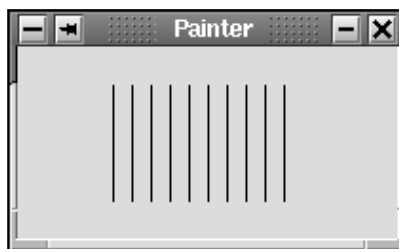


Abbildung 4-12 *drawLineSegments*

drawPolyLine (*const QPointArray &a*, *int index = 0*, *int npoints = -1*)

Zeichnet einen Linienzug, der *npoints* Punkte miteinander verbindet, deren Koordinaten in *a* angegeben sind, beginnend bei *index*. Der erste und letzte Punkt werden nicht miteinander verbunden. Es werden also *npoints-1* Linienstücke gezeichnet (siehe Abbildung 4.13).

```
QPointArray a (20);
for (int i = 0; i < 10; i++) {
    a.setPoint (2 * i, 50 + 10 * i, 20);
    a.setPoint (2 * i + 1, 50 + 10 * i, 80);
}
p.drawPolyline (a);
```

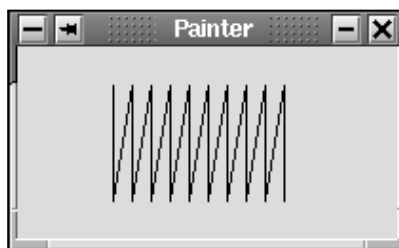


Abbildung 4-13 *drawPolyLine*

drawArc (int x, int y, int w, int h, int a, int alen)

drawArc (const QRect &r, int a, int alen)

Zeichnet ein Ellipsenstück (siehe Abbildung 4.14). Die vollständige Ellipse wird durch das Rechteck aus *x*, *y*, *w* und *h* bzw. aus *r* beschrieben. Der Startwinkel *a* und die Bogenlänge *alen* werden wie bei *drawEllipse* angegeben. Im Gegensatz zu den anderen Ellipsenfunktionen (*drawEllipse*, *drawChord*, *drawPie*) wird bei *drawArc* nur die Ellipsenlinie gezeichnet, es wird kein Bereich ausgefüllt (siehe den nächsten Abschnitt *Methoden zum Zeichnen ausgefüllter Primitive*).

```
p.drawArc (20, 20, 160, 60, 45 * 16, 270 * 16);
```

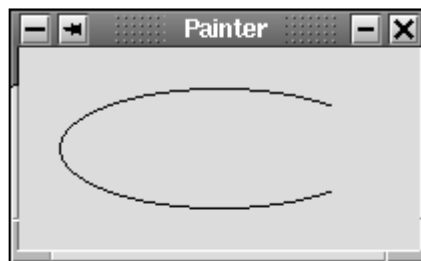


Abbildung 4-14 *drawArc*

drawQuadBezier (const QPointArray &a, int index = 0)

Zeichnet ein Bezierkurven-Segment aus den vier Kontrollpunkten *index*, *index + 1*, *index + 2* und *index + 3* aus *a* (siehe Abbildung 4.15). Eine Bezierkurve ist eine gekrümmte Linie, die den ersten und den vierten Kontrollpunkt miteinander verbindet, wobei sich die Linie in Richtung des zweiten und dritten Kontrollpunktes krümmt. Anders als der Methodenname vermuten lässt, handelt es sich hierbei um eine kubische und nicht um eine quadratische Bezierkurve. Es werden in jedem Fall nur vier Punkte aus *a* benutzt. Wenn Sie eine längere Bezierkurve aus mehreren Segmenten aneinander setzen wollen, können Sie zum Beispiel in einem Array die Punkte ablegen, indem Sie immer nach einem Punkt der Kurve zwei Kontrollpunkte für das nächste Segment, anschließend den nächsten Kurvenpunkt usw. abspeichern und dann diese Methode mit *index = 0*, *index = 3*, *index = 6*, ... aufrufen.

```
QPointArray a;
a.setPoints (4, 25, 80, 75, 20,
             125, 80, 175, 20);
p.drawPoints (a);
p.drawQuadBezier (a);
```

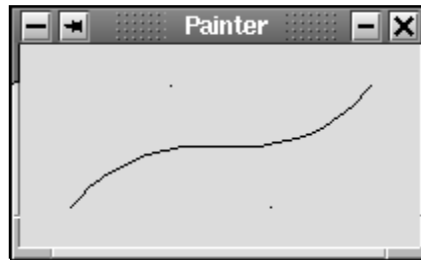



Abbildung 4-15 `drawQuadBezier` (Die Kontrollpunkte wurden ebenfalls eingezeichnet.)

Methoden zum Zeichnen ausgefüllter Primitive

Die folgenden Methoden von `QPainter` zeichnen Primitive, die eine Fläche umranden. Diese Fläche wird mit dem Füllmuster des aktuellen `QBrush` ausgefüllt, anschließend wird der Rand der Fläche mit der Linienart aus dem aktuellen `QPen` nachgezeichnet (siehe Kapitel 4.2.2, *Linienarten und Füllmuster*). Wollen Sie nur die Begrenzungslinie zeichnen, ohne die entstehende Fläche auszufüllen, so setzen Sie einen `QBrush` mit dem Füllmuster `NoBrush`. Wollen Sie nur die Fläche ausfüllen lassen und die Begrenzungslinie nicht zeichnen, so setzen Sie einen `QPen` mit der Linienart `NoPen`.

`drawRect (int x, int y, int w, int h)`

`drawRect (const QRect &r)`

Zeichnet ein Rechteck (siehe Abbildung 4.16). Die obere linke Ecke liegt bei den Pixelkoordinaten (x,y) , das Rechteck hat die Breite w und die Höhe h . Alternativ kann man die Koordinaten auch in einem `QRect`-Objekt angeben.

```
QBrush brush (white);  
p.setBrush (brush);  
p.drawRect (20, 20, 160, 60);
```

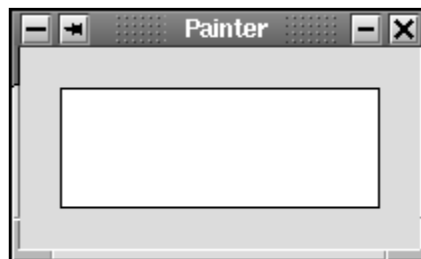


Abbildung 4-16 `drawRect`

drawRoundRect (int x, int y, int w, int h, int xRnd, int yRnd)

drawRoundRect (const QRect &r, int xRnd, int yRnd)

Zeichnet ein Rechteck mit abgerundeten Ecken (siehe Abbildung 4.17). Die Abrundungen bestehen dabei aus Vierteln von Ellipsen. Die Begrenzungsline besteht also aus vier geraden Linienstücken und vier Ellipsenvierteln, die sich jeweils abwechseln. Mit den Parametern *xRnd* und *yRnd* kann man angeben, wie stark die Ecken in x- bzw. y-Richtung abgerundet sein sollen und wie viel Prozent einer Rechteckseite auf die Ellipsenstücke entfallen sollen. Ein Wert von 0 ergibt somit eine rechtwinklige Kante (wie bei *drawRect*), bei einem Wert von 100 besteht das Rechteck nur aus vier Ellipsenvierteln, ist also identisch mit einem Aufruf von *drawEllipse*.

```
QBrush brush (white);  
p.setBrush (brush);  
p.drawRoundRect (20, 20, 160, 60, 19, 50);
```

In diesem Beispiel beträgt die Ellipsenbreite 19% der Breite von 160 Pixel, die Ellipsenhöhe 50% der Höhe von 60 Pixel, in beiden Ausdehnungen also ca. 30 Pixel. Die Rundungen sind daher etwa Viertelkreise.

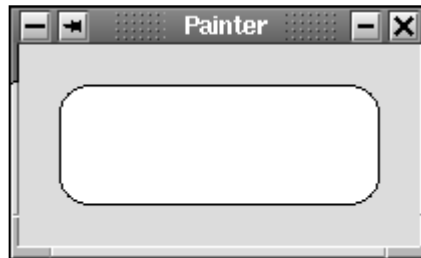


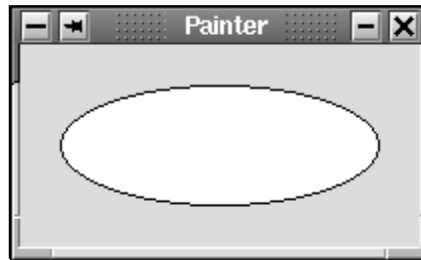
Abbildung 4-17 *drawRoundRect*

drawEllipse (int x, int y, int w, int h)

drawEllipse (const QRect &r)

Zeichnet eine ausgefüllte Ellipse, der vom angegebenen Rechteck begrenzt ist (siehe Abbildung 4.18). Die gezeichnete Ellipse berührt das virtuelle Rechteck an den Mittelpunkten der Seiten. Sind Breite und Höhe gleich groß, so ergibt sich ein Kreis.

```
QBrush brush (white);  
p.setBrush (brush);  
p.drawEllipse (20, 20, 160, 60);
```

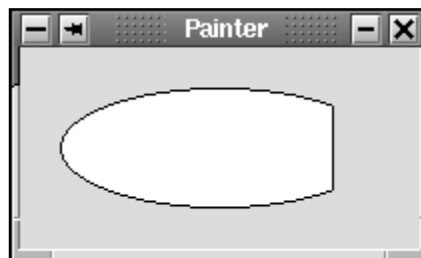
Abbildung 4-18 *drawEllipse*

drawChord (int x, int y, int w, int h, int a, int alen)

drawChord (const QRect &r, int a, int alen)

Zeichnet einen Abschnitt einer Ellipse (siehe Abbildung 4.19). Der Teil der Ellipse wird durch die beiden Winkelparameter *a* und *alen* angegeben. Die Winkel sind dabei in sechzehntel Grad angegeben. Der Anfangs- und der Endpunkt werden durch eine gerade Linie verbunden. *a* gibt den Anfangswinkel an. Ein Wert von 0 bezeichnet den Punkt am rechten Rand der Ellipse. Bei steigenden Werten bewegt sich der Anfangspunkt entlang der Ellipsenkurve gegen den Uhrzeigersinn. Der Punkt am oberen Rand der Ellipse hat somit einen Startwert von 1440 (= 16 * 90°), der Punkt am unteren Rand 4320 (= 16 * 270°). *alen* gibt den Winkel an, den der Bogen überstreicht. Positive Werte geben eine Drehung gegen den Uhrzeigersinn, negative Werte eine Drehung mit dem Uhrzeigersinn an.

```
QBrush brush (white);  
p.setBrush (brush);  
p.drawChord (20, 20, 160, 60, 45 * 16, 270 * 16);
```

Abbildung 4-19 *drawChord*

drawPie (int x, int y, int w, int h, int a, int alen)

drawPie (const QRect &r, int a, int alen)

Zeichnet einen Ausschnitt einer Ellipse, ein so genanntes Tortenstück (siehe Abbildung 4.20). Im Gegensatz zu *drawChord* werden hier Anfangs- und Endpunkt der Ellipse jeweils mit dem Ellipsenmittelpunkt verbunden. Eine typische Anwendung von solchen Ellipsenausschnitten sind beispielsweise Tortendiagramme.

```
QBrush brush (white);
p.setBrush (brush);
p.drawPie (20, 20, 160, 60, 45 * 16, 270 * 16);
```

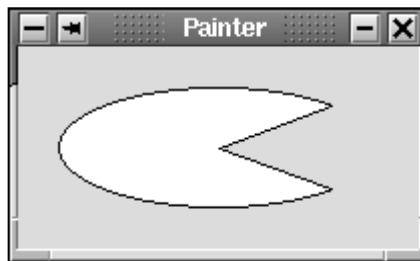


Abbildung 4-20 *drawPie*

drawPolygon (const QPointArray &a, bool winding=false, int index=0, int npoints=-1)

Zeichnet ein ausgefülltes Polygon aus *npoints* Eckpunkten aus *a*, beginnend bei *index* (siehe Abbildung 4.21). Die Defaultwerte benutzen alle Punkte aus *a*. Im Unterschied zu *drawPolyLine* werden hier der letzte und der erste Punkt noch mit einer Linie verbunden, und die so umschlossene Fläche wird ausgefüllt. Mit dem Parameter *winding* kann man festlegen, wie die Fläche ausgefüllt werden soll, wenn sich die Polygonlinie selbst durchdringt. Ist *winding true*, so wird zum Füllen der *Winding-Algorithmus* benutzt. Jedes Pixel wird genau dann gefüllt, wenn sich die Polygonlinie mindestens einmal um diesen Pixel herumwindet. Ist *winding false*, so wird der *Even-Odd-Algorithmus* benutzt, der ein Pixel genau dann füllt, wenn man von diesem Punkt aus zum Äußeren des Polygons die Polygonlinie eine ungerade Anzahl mal kreuzt. Der gewählte Algorithmus wirkt sich in jedem Fall nur dann aus, wenn sich die Polygonlinie selbst überschneidet. Der Even-Odd-Algorithmus ist in den meisten X-Servern effizienter implementiert als der Winding-Algorithmus, daher ist der Even-Odd-Algorithmus als Defaultwert eingestellt.

```
QBrush brush (white);
p.setBrush (brush);
QPointArray a;
a.setPoints (5, 50, 10, 74, 82,
```

```

        12, 38, 88, 38, 26, 82);
p.drawPolygon (a);
a.translate (100, 0);
p.drawPolygon (a, true);

```

Das linke Beispiel in Abbildung 4.21 zeigt den Even-Odd-Algorithmus (die Mitte wird nicht ausgefüllt, da man, um vom Inneren nach außen zu kommen, zwei Polygonlinien überqueren muss), das rechte Beispiel zeigt den Winding-Algorithmus (auch die Mitte ist ausgefüllt).

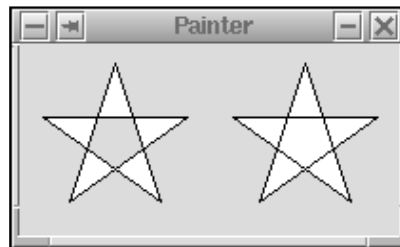


Abbildung 4-21 *drawPolygon* (links Even-Odd-, rechts Winding-Algorithmus)

Methoden zum Zeichnen von Text

Da diese Zeichenmethoden sehr mächtig sind, wollen wir sie hier in einem eigenen Abschnitt behandeln.

drawText (int *x*, int *y*, const char* *str*, int *len* = -1)

drawText (const QPoint &*p*, const char* *str*, int *len* = -1)

Zeichnet den Textstring aus *str* an der Position (*x*,*y*) (siehe Abbildung 4.22). *y* gibt dabei die y-Koordinate der Grundlinie an, während *x* die x-Koordinate des linken Textstrands ist. Es werden maximal *len* Zeichen gezeichnet. Ist *len* -1, so wird der ganze String gezeichnet. Im Gegensatz zur folgenden Methode werden Spezialzeichen wie Tabulatoren, Zeilenumbrüche u.ä. ignoriert, das »&«-Zeichen hat keine besondere Bedeutung.

```

p.drawText (0, 50,
            "Einfacher, unformatierter Text");

```

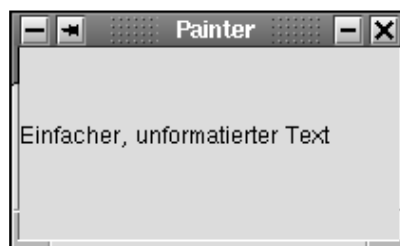


Abbildung 4-22 *drawText* (unformatiert)

drawText (int *x*, int *y*, int *w*, int *h*, int *tf*, const char* *str*, int *len* = -1, QRect **brect* = 0, char ***internal* = 0)

drawText (const QRect &*r*, int *tf*, const char* *str*, int *len* = -1, QRect **brect* = 0, char ***internal* = 0)

Zeichnet den Textstring aus *str* in das durch *x*, *y*, *w* und *h* bzw. durch *r* angegebene Rechteck (siehe Abbildung 4.23). Dabei werden Spezialzeichen wie Zeilenumbrüche (»\n«) und Tabulatoren (»\t«) sowie das »&«-Zeichen interpretiert, sofern diese in den Textformatierungsflags aktiviert sind. Dabei werden maximal *len* Zeichen ausgegeben bzw. der ganze String, falls *len* den Wert -1 hat. Durch den Parameter *tf* (Textformatierung) kann die Platzierung und Darstellung des Textes beeinflusst werden. *tf* entsteht durch die bitweise Oder-Verknüpfung der folgenden Konstanten:

- *AlignLeft*, *AlignRight*, *AlignHCenter* setzen den Text linksbündig, rechtsbündig oder horizontal zentriert.
- *AlignTop*, *AlignBottom*, *AlignVCenter* setzen den Text vertikal an die obere oder untere Rechteckkante bzw. in die Mitte des Rechtecks.
- *AlignCenter* ist die Oder-Verknüpfung von *AlignHCenter* und *AlignVCenter*, stellt den Text also insgesamt im Rechteck zentriert dar.
- *SingleLine* ignoriert alle Zeilenumbrüche im String (»\n«-Zeichen).
- *ExpandTabs* ersetzt alle Tabulator-Zeichen (»\t«-Zeichen) durch einen entsprechend breiten Zwischenraum bis zum Erreichen der nächsten Tabulatorposition. Diese Positionen kann man mit *setTabArray* und *setTabStops* definieren.
- *DontClip* bewirkt, dass der ausgegebene Text nicht am angegebenen Rechteck geclippt wird, also über die Ränder des Rechtecks hinausgehen kann.
- *ShowPrefix* bewirkt, wenn es gesetzt ist, dass das »&«-Zeichen eine Spezialbedeutung bekommt. Dieses Zeichen wird nicht gezeichnet, stattdessen wird das folgende Zeichen unterstrichen dargestellt. Dies wird zum Beispiel benutzt, um in Labels, Buttons und Menüeinträgen das Tastenkürzel zu markieren. Um dennoch ein »&«-Zeichen darzustellen, müssen Sie »&&« in den String einfügen.
- *WordBreak* bewirkt einen automatischen Zeilenumbruch in der Darstellung. Ein Zeilenumbruch wird immer dann eingefügt, wenn die Breite des Rechtecks nicht mehr ausreicht, um das nächste Wort darzustellen.
- *GrayText* stellt den Text halb durchsichtig dar. Dazu wird nur jedes zweite Pixel des Textes gezeichnet. Dieses Flag wird zum Beispiel benutzt, um den Text eines Buttons im Status *Disabled* darzustellen.

Im Parameter *brect* wird das umgebende Textrechteck abgespeichert, falls *brect* nicht 0 ist.

Im Parameter *internal* können Werte der internen Berechnung von *drawText* abgespeichert werden, z. B. die Position von Zeilenumbrüchen, Tabulatorpositionen u. a. Zeichnet man einen komplexen Text mehrmals, kann man diese Werte zwischenspeichern, damit sie nicht jedes Mal neu gezeichnet werden müssen. Wenn Sie 0 als Wert von *internal* benutzen, wird der Parameter ignoriert (normale Anwendung). Wenn Sie einen Zeiger auf einen NULL-Pointer benutzen, so werden im Parameter *internal* die internen Zwischenwerte gespeichert. Diese können dann beim nächsten Aufruf einfach wieder benutzt werden.

```
p.drawText (50, 10, 100, 80,
            AlignCenter | ShowPrefix | WordBreak,
            "Formatierter Text mit "
            "&Unterstrichen unter "
            "&einzelnen Buchstaben && "
            "Zeilenumbruch");
```

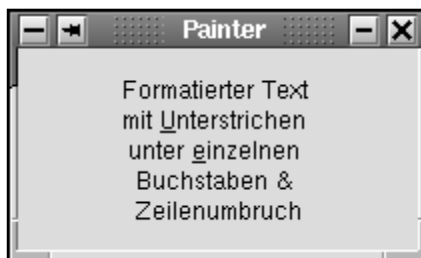


Abbildung 4-23 *drawText* (zentriert, mit Unterstrichen und Zeilenumbruch)

QRect boundingRect

(*int x, int y, int w, int h, int tf, const char* str, int len = -1, char **internal*)

QRect boundingRect

(*const QRect &r, int tf, const char* str, int len = -1, char **internal*)

Berechnet das umgebende Rechteck, das benötigt wird, um den Text *str* zu zeichnen. Die Parameter haben die gleiche Bedeutung wie in der vorhergehenden *drawText*-Methode. Diese Methode wird von *drawText* benutzt, um die korrekte Position des Textes zu ermitteln. Sie kann von Ihnen benutzt werden, um selbst die Textgeometrie zu ermitteln, bevor Sie mit dem Zeichnen des Textes beginnen.

setTabStops (int ts)

int tabStops ()

*setTabArray (int *ta)*

*int *tabArray ()*

Mit den beiden Methoden *setTabStops* und *setTabArray* kann die Tabulatorweite für *drawText* festgelegt werden. Mit *setTabStops* werden Tabulatorpositionen in regelmäßigen Abständen von *ts* Pixel angelegt. Mit *setTabArray* kann man die Tabulatorpositionen auf beliebige Werte in unregelmäßigen Abständen festlegen. *ta* muss dazu ein Array von Pixelpositionen enthalten. Sinnvollerweise ist jeder Wert im Array größer als sein Vorgänger. Als Zeichen für das Ende der Tabulatorliste muss an der letzten Position des Arrays der Wert 0 stehen. Mit den Methoden *tabStops* und *tabArray* kann man die eingestellten Werte wieder auslesen. Beachten Sie, dass das Array *ta* nicht kopiert wird. Es darf also erst freigegeben werden, wenn sichergestellt ist, dass *QPainter* nicht mehr darauf zugreift. Sind sowohl *tabStops* als auch *tabArray* gesetzt, so hat die Einstellung von *tabArray* Vorrang vor den *tabStops*.

QSimpleRichText

Hierbei handelt es sich genau genommen nicht um eine Methode von *QPainter*. Die Klasse *QSimpleRichText* kann aber ebenfalls benutzt werden, um mit einem *QPainter* Text auf den Bildschirm zu malen. Der Text wird dabei als RichText formatiert, kann also Format-Tags enthalten, die das Aussehen und die Anordnung einzelner Textpassagen verändern. Sogar Tabellen und eingebundene Bilder sind möglich. Ein einfaches Beispiel (siehe Abbildung 4.24) soll die Anwendung verdeutlichen. Oftmals können Sie aber auch die Klassen *QTextView* oder *QTextBrowser* benutzen, die bereits fertige Widgets mit Rollbalken und Navigation zur Verfügung stellen.

```
QString text = "<H2><CENTER>Simple RichText"
               "</CENTER></H2>"
               "Formatiert <B>fett</B>, "
               "<I>kursiv</I> und kann auch "
               "Tabellen und Bilder enthalten.";
QSimpleRichText t (text, font());
t.setWidth (width());
t.draw (&p, 0, 0, rect(), palette());
```




Abbildung 4-24 QSimpleRichText

4.2.2 Linienarten und Füllmuster

Die Zeichenroutinen für grafische Elemente, die wir im vorangegangenen Unterkapitel 4.2.1, *QPainter*, kennen gelernt haben, lassen sich in zwei Gruppen unterteilen: Routinen für Elemente, die nur aus Linien bestehen (*drawLine*, *drawQuadBezier* usw.) und Routinen für Elemente, die aus einer Fläche bestehen, die von einer Begrenzungslinie umgeben ist (*drawRect*, *drawEllipse*, *drawPolygon* usw.). Qt definiert zwei zusätzliche Klassen, *QPen* und *QBrush*, in denen man einen speziellen Linienstift bzw. ein spezielles Füllmuster für Flächen definieren kann.

Linienstift QPen

Ein Linienstift *QPen* wird durch drei Werte charakterisiert: Die Linienfarbe (definiert durch ein Objekt der Klasse *QColor*, siehe Kapitel 4.1, *Farben unter Qt*), die Liniendicke (definiert durch einen *int*-Wert) und die Linienart (als eine von sechs Konstanten für die verschiedenen Linienarten). Diese drei Werte kann man im Konstruktor angeben, zum Beispiel:

```
QPen pen (red, 3, DashLine);
```

Diese Zeile definiert einen Linienstift der Farbe Rot, der Liniendicke 3 und der Linienart »gestrichelt«. Dieser Linienstift kann nun in einem *QPainter*-Objekt *p* mit

```
p.setPen (pen);
```

gesetzt werden.

Die Liniendicke wird in der Längeneinheit des *QPaintDevice*-Objekts angegeben. Liniendicke 0 bedeutet dabei, dass die Linie so dünn wie möglich sein soll. Für *QWidget* und *QPixmap* sind daher Liniendicke 0 und Liniendicke 1 identisch, da eine Linie mindestens ein Pixel dick sein muss. Für *QPrinter* dagegen ist Liniendicke 0 eine ganz feine Linie, während Liniendicke 1 eine Linie der Dicke zeichnet, die der eingestellten Druckerauflösung entspricht, standardmäßig $\frac{1}{72}$ Zoll. Abbildung 4.25 zeigt die Liniendicken 0, 1, 2, 3, 5 und 10 in einem *QWidget*-Objekt.

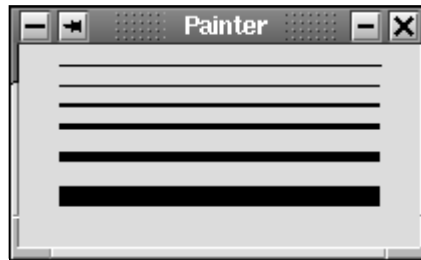


Abbildung 4-25 Verschiedene Liniendicken für QPen

Für die Linienart stehen die Konstanten *NoPen*, *SolidLine*, *DashLine*, *DotLine*, *DashDotLine* und *DashDotDotLine* zur Verfügung. Abbildung 4.26 zeigt die verschiedenen Linienarten (von oben nach unten *SolidLine* bis *DashDotDotLine*). Die am häufigsten benutzte Linienart ist *SolidLine*, die eine durchgezogene Linie zeichnet. *NoPen* kann benutzt werden, um mit *QPainter* eine Fläche auszufüllen, ohne die Umrandung zu zeichnen.

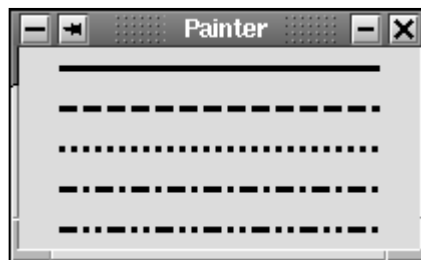


Abbildung 4-26 Verschiedene Linienarten für QPen (*SolidLine*, *DashLine*, *DotLine*, *DashDotLine*, *DashDotDotLine*)

Außer *NoPen* und *SolidLine* zeichnen alle anderen Linienarten unterbrochene Linien. Es gibt zwei Möglichkeiten, wie der Zwischenraum zwischen den gezeichneten Linienstücken dargestellt werden soll. Er kann unverändert erhalten bleiben (die Pixelwerte ändern sich also nicht) oder mit Linienstücken der Hintergrundfarbe gefüllt werden. Im zweiten Fall wird also die Linie vollständig gezeichnet, abwechselnd mit der Linienfarbe und der Hintergrundfarbe. Im *QPainter*-Objekt können Sie dazu mit der Methode *setBackgroundMode* zwischen dem *TransparentMode* (Pixel nicht verändern) und dem *OpaqueMode* (Pixel auf Hintergrundfarbe setzen) wählen. Die Hintergrundfarbe für den *OpaqueMode* können Sie mit der Methode *setBackgroundColor* im *QPainter*-Objekt festlegen.

Das *QPen*-Objekt legt nicht nur die Farbe und Linienart für das Zeichnen von Linien fest, sondern auch die Zeichenfarbe von Text (mit *drawText*). Hierbei wird jedoch nur die Farbe des *QPen*-Objekts benutzt, nicht die Linienart. Ebenso wie

bei Linien kann der Bereich des Textrechtecks, der nicht vom Text beschrieben wird, unverändert bleiben (*TransparentMode*) oder mit der Hintergrundfarbe gefüllt werden (*OpaqueMode*).

Als weitere Eigenschaft des Zeichenstifts kann man festlegen, wie das Ende einer Linie aussehen soll (*capStyle*) und wie der Übergang von einem Liniensegment zum nächsten aussehen soll (*joinStyle*). Diese beiden Einstellungen machen sich nur bei dicken Linien bemerkbar. Abbildung 4.27 zeigt die Auswirkungen der Einstellung *capStyle* für die möglichen Werte *FlatCap*, *SquareCap* und *RoundCap* (von links nach rechts). Der Endpunkt der Linie ist durch ein kleines weißes Kreuz gekennzeichnet. Abbildung 4.28 zeigt die Auswirkungen von *joinStyle* am Übergang von einem Liniensegment zum nächsten mit den Einstellungen *MiterJoin*, *BevelJoin* und *RoundJoin* (von links nach rechts).

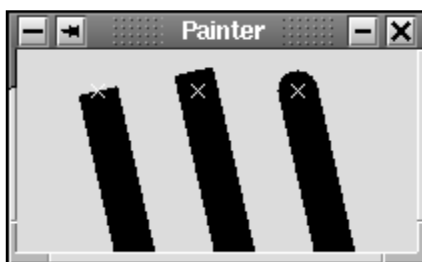


Abbildung 4-27 *capStyle* (*FlatCap*, *SquareCap*, *RoundCap*)

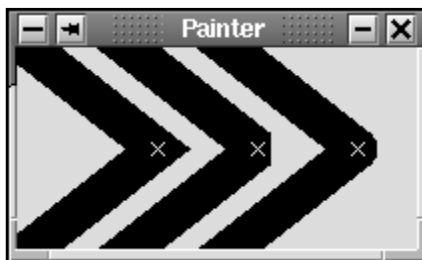


Abbildung 4-28 *joinStyle* (*MiterJoin*, *BevelJoin* und *RoundJoin*)

Füllmuster QBrush

Eine andere Klasse, *QBrush*, definiert ein Füllmuster. Das Füllmuster wird dabei durch zwei Werte charakterisiert: die Farbe (definiert durch ein *QColor*-Objekt, siehe Kapitel 4.1, *Farben unter Qt*) und das Muster (als Auswahl von einem von 16 Mustern). Für das spezielle Muster *CustomPattern* kann zusätzlich noch eine *Pixmap* angegeben werden, die dann als Füllmuster dient.

Im Konstruktor von *QBrush* kann man die Farbe und die Musterart angeben:

```
QBrush brush (blue, DiagCrossPattern);
```

Das so definierte Füllmuster füllt Flächen mit einem Muster von blauen, sich kreuzenden diagonalen Linien. Es wird in einem *QPainter*-Objekt *p* wie folgt angewendet:

```
p.setBrush (brush);
```

Die verschiedenen Konstanten für das Muster sind *NoBrush*, *SolidPattern*, *Dense1Pattern*, *Dense2Pattern*, *Dense3Pattern*, *Dense4Pattern*, *Dense5Pattern*, *Dense6Pattern*, *Dense7Pattern*, *HorPattern*, *VerPattern*, *CrossPattern*, *BDiagPattern*, *FDiagPattern*, *DiagCrossPattern* und *CuttomPattern*. Abbildung 4.29 zeigt Rechtecke, die mit verschiedenen Füllmustern gefüllt sind. Die obere Reihe zeigt von links nach rechts die Muster *SolidPattern*, *Dense1Pattern*, *Dense2Pattern*, *Dense3Pattern*, *Dense4Pattern*, *Dense5Pattern*, *Dense6Pattern* und *Dense7Pattern*. In der unteren Zeile sind von links nach rechts die Muster *HorPattern*, *VerPattern*, *CrossPattern*, *BDiagPattern*, *FDiagPattern* und *DiagCrossPattern* dargestellt.

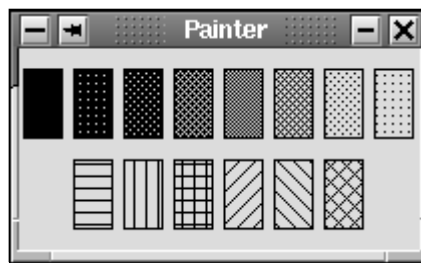


Abbildung 4-29 Verschiedene Füllmuster für QBrush

Die Musterart *CustomPattern* kann im normalen Konstruktor nicht benutzt werden, da zusätzlich noch eine Pixmap definiert werden muss. Zu diesem Zweck gibt es einen Konstruktor, der ein solches *QBrush*-Objekt definiert:

```
QBrush brush (blue, myPixmap);
```

myPixmap ist dabei ein *QPixmap*- (oder *QBitmap*-)Objekt (siehe Kapitel 4.3, *Teilbilder – QImage und QPixmap*). Bei einem *QBitmap*-Objekt (oder einem *QPixmap*-Objekt mit Farbtiefe 1) wird dieses Füllmuster in der angegebenen Farbe an den Stellen dargestellt, an denen die Bitmap den Wert 1 enthält. Für ein *QPixmap*-Objekt mit größerer Farbtiefe wird die gesamte Pixmap als Muster benutzt (mit allen enthaltenen Farben). Die im Konstruktor angegebene Farbe wird in diesem Fall ignoriert. Wir wollen an zwei Beispielen das Prinzip weiter verdeutlichen.

Im ersten Beispiel benutzen wir als Füllmuster ein *QBitmap*-Objekt, in dem wir einen Kreis gezeichnet haben. Wir benötigen einen zusätzlichen Painter *bp*, um das *QBitmap*-Objekt zu zeichnen (*QBitmap* und *QPixmap* sind ebenfalls von *QPaintDevice* abgeleitet). Das Ergebnis des folgenden Listings sehen Sie in Abbildung 4.30.

```
QBitmap b (20, 20);    // erzeugt Bitmap
QPainter bp (b);       // Painter für Bitmap
bp.fillRect (b.rect (), color0); // Bitmap löschen
bp.setPen (color1);
bp.drawEllipse (3, 3, 14, 14); // Kreis zeichnen
p.setBrush (blue, b);   // Bitmap als Füllmuster wählen
p.drawEllipse (rect ()); // ausgefüllte Ellipse zeichnen
```

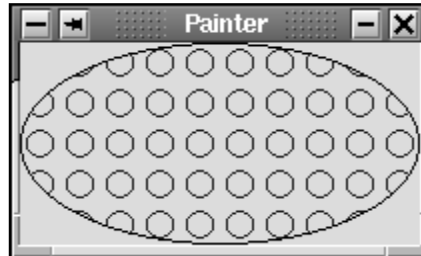


Abbildung 4-30 QBrush mit CustomPattern mit QBitmap-Objekt

Im zweiten Beispiel benutzen wir ein vielfarbiges Bild als Füllmuster. Dazu laden wir ein *QPixmap*-Objekt mit Daten aus einer Bilddatei. Das Ergebnis des folgenden Listings sehen Sie in Abbildung 4.31.

```
QPixmap pix ("kde-logo.gif"); // QPixmap aus Datei laden
p.setBrush (black, pix);      // als Füllmuster wählen
p.drawEllipse (rect ());      // Ellipse füllen
```

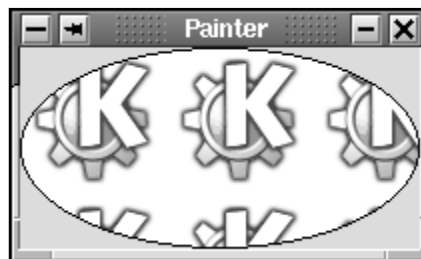


Abbildung 4-31 QBrush mit CustomPattern mit QPixmap-Objekt

Auch für das Füllen von Flächen mit einem Füllmuster, das nicht vollständig deckend ist, können Sie mit der Methode *setBackgroundMode* im *QPainter*-Objekt auswählen, wie die Flächen behandelt werden sollen, die vom Muster nicht gefüllt werden. (Das gilt auch für das Füllen mit *CustomPattern* und einem Bitmap-Füllmuster). Wählen Sie *TransparentMode* aus, um die freien Pixel des Füllmusters nicht zu verändern. Wenn Sie dagegen *OpaqueMode* wählen, werden die freien Pixel auf die Hintergrundfarbe gesetzt.

4.2.3 Koordinaten-Transformationen

Die *QPainter*-Klasse von Qt bietet sehr mächtige Möglichkeiten, um die Koordinaten von Objekten zu transformieren, die gezeichnet werden sollen. Die Palette der Möglichkeiten reicht von einfachen Verschiebungen und Skalierungen bis zu Rotationen und Scherungen. Alle Elemente, die mit den Methoden aus Kapitel 4.2.1, *QPainter*, gezeichnet werden (außer den Methoden *fillRect* und *eraseRect*) unterliegen dabei diesen Transformationen.

View-Transformationen

Einfache Verschiebungen und Skalierungen können mit der so genannten View-Transformation vorgenommen werden. Dazu legen Sie mit der Methode *setWindow* ein neues Koordinatensystem für das Fenster (bzw. das *QPaintDevice*-Objekt) fest. Dazu übergeben Sie der Methode die neuen Koordinaten, die die obere linke Fensterecke haben soll, sowie die Breite und Höhe des Fensters in neuen Koordinaten. So verschiebt zum Beispiel die Ausführung der folgenden Zeile das Koordinatensystem:

```
p.setWindow (100, 50, 800, 1000);
```

Die Koordinaten (100/50) liegen jetzt in der oberen linken Ecke des Fensters (und nicht mehr die Koordinaten (0/0)). Die Ecke unten rechts wird durch die Koordinaten (900/1050) angesprochen. Um also ein Rechteck zu zeichnen, das genau das Fenster ausfüllt, benutzen Sie folgende Zeile:

```
p.drawRect (100, 50, 800, 1000);
```

Das Koordinatensystem ist per Default beim Erzeugen des *QPainter*-Objekts so eingestellt, dass die obere linke Ecke die Koordinaten (0/0) und die untere rechte Ecke die Koordinaten (*width()*,*height()*) hat, wobei *width* und *height* die Breite und Höhe des Fensters in Pixel angeben.

setWindow wird oft benutzt, um den Ursprung des Koordinatensystems oder die Längeneinteilung der Achsen zu ändern. Das folgende Programmstück legt zum Beispiel den Koordinatenursprung in die Mitte des 200 x 100 Pixel großen Fensters und skaliert die Achsen um den Faktor 4. Die obere linke Ecke hat nun die Koordinaten (-400/-200), die untere rechte Ecke die Koordinaten (400/200). Danach zeichnet es eine Ellipse, wie es in Abbildung 4.32 zu sehen ist. (Das Ergebnis ist das gleiche wie in Abbildung 4.18, nur die Art, das Ergebnis zu erreichen, ist anders.)

```
p.setWindow (-400, -200, 800, 400);  
p.drawEllipse (-320, -120, 640, 240);
```

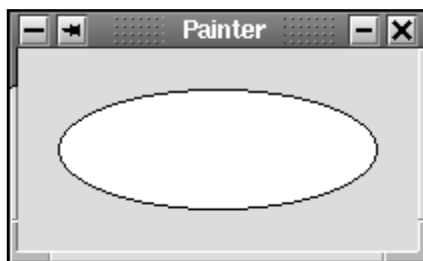


Abbildung 4-32 Zeichnen einer Ellipse nach *setWindow*

Wenn Sie negative Werte für die Höhe oder Breite benutzen, können Sie die Achsen auch in die entgegengesetzte Richtung laufen lassen. Die folgende Zeile legt zum Beispiel den Koordinatenursprung in die untere linke Ecke. Die x-Achse wächst nach rechts, die y-Achse nach oben. Die Achsen entsprechen nun der gängigen Darstellung von mathematischen Funktionen. Eine Skalierung findet dabei nicht statt, d.h. ein Schritt auf einer Achse um 1 bedeutet einen Schritt von einem Pixel im Fenster.

```
p.setWindow (0, height(), width(), -height());
```

Mit der Methode *setViewport* können Sie zusätzlich festlegen, welchen Bildschirmkoordinaten die Koordinaten des Rechtecks aus *setWindow* zugewiesen werden sollen. Standardmäßig sind diese Viewport-Koordinaten auf das ganze Fenster festgelegt, so dass sich die neuen Koordinaten auf das gesamte Fenster beziehen. Man kann mit *setViewport* aber ein anderes Rechteck wählen. Das folgende Beispiel legt das Koordinatensystem so fest, dass die obere linke Ecke des Viewports an den Bildschirmkoordinaten (30/20) die neuen Koordinaten (-100/-100) erhält und die rechte untere Ecke des Viewports an den Bildschirmkoordinaten (50/80) die neuen Koordinaten (100/100) zugewiesen bekommt. (Beachten Sie, dass das Viewport-Rechteck von (30/20) bis (50/80) geht, das Window-Rechteck von (-100/-100) bis (100/100). Die letzten beiden Parameter geben sowohl für *setViewport* als auch für *setWindow* die Breite und Höhe des Rechtecks und nicht die Koordinaten der anderen Ecke an.).

```
p.setViewport (30, 20, 20, 60);  
p.setWindow (-100,-100,200,200);
```

Beachten Sie, dass die Zeichnungen nicht am Viewport-Rechteck abgeschnitten werden. Das Viewport-Rechteck dient ausschließlich zur Umrechnung von Koordinaten. Um Teile der Zeichnung abzuschneiden, müssen Sie stattdessen mit den Methoden *setClipRect* oder *setClipRegion* einen Ausschnitt wählen (siehe Kapitel 4.2.4, *Beschränkung auf Ausschnitte*).

Die Angaben für das Viewport- und das Window-Rechteck wirken gegensätzlich. Wenn Sie zum Beispiel die gleichen Werte für das Viewport- und das Window-Rechteck wählen, hebt sich die Wirkung gerade auf, so als wäre keine View-Transformation gesetzt.

Mit der Methode *setViewXForm* (*false*) können Sie eine eingestellte View-Transformation wieder deaktivieren. Die Methode *hasViewXForm* liefert *true* zurück, falls ein Viewport- oder Window-Rechteck gesetzt worden ist. Mit den Methoden *viewport* und *window* können Sie die eingestellten Rechtecke ermitteln.

World-Transformationen

Die World-Transformationen der *QPainter*-Klasse übertreffen die View-Transformationen in ihrer Mächtigkeit um ein Vielfaches. Auch mit den World-Transformationen können die zu zeichnenden Primitive verschoben und vergrößert oder verkleinert werden. Zusätzlich können die Primitive rotiert und geschert werden, und alle Transformationen sind in beliebiger Reihenfolge miteinander kombinierbar.

Um die verschiedenen Transformationen an Beispielen aufzuzeigen, benutzen wir ein Polygon aus fünf Punkten, das gezeichnet werden soll. Wir ergänzen unser Testprogramm von oben durch das Zeichnen dieses Polygons:

```
p.setBrush (QBrush (white));
p.setPen (QPen (black, 2));
QPointArray a;
a.setPoints (5,
             60, 10,
             60, 90,
             100, 90,
             100, 50,
             140, 10);
p.drawPolygon (a);
```

Abbildung 4.33 zeigt das Ergebnis dieses Programms. Das Polygon ist noch nicht transformiert. Die Methoden zur Festlegung der Transformation werden wir im Folgenden vor dem Befehl *p.drawPolygon (a)* einfügen.

Mit der Methode *QPainter::translate (int dx, int dy)* kann man die Koordinaten der zu zeichnenden Objekte um *dx* nach rechts und *dy* nach unten verschieben. (Negative Werte für *dx* bzw. *dy* bewirken eine Verschiebung nach links bzw. oben.) Die folgende Zeile bewirkt die Verschiebung, die in Abbildung 4.34 zu sehen ist:

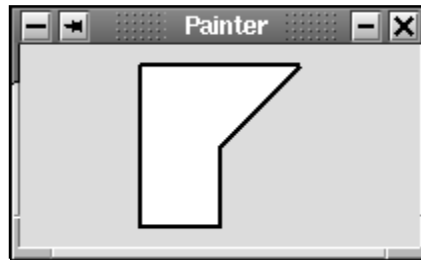


Abbildung 4-33 Das untransformierte Polygon

```
p.translate (20.0, 10.0);
```

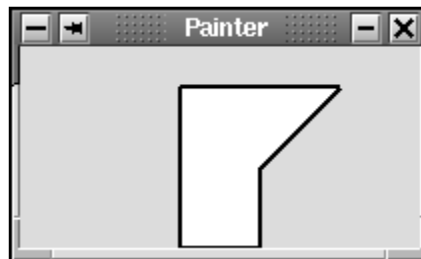


Abbildung 4-34 Verschiebung um 20 Pixel nach rechts und zehn Pixel nach unten

Eine Rotation um einen Winkel von 10° im Uhrzeigersinn um den Koordinatenursprung (die obere linke Ecke) erreichen Sie mit folgender Zeile:

```
p.rotate (10.0);
```

Das Ergebnis sehen Sie in Abbildung 4.35.

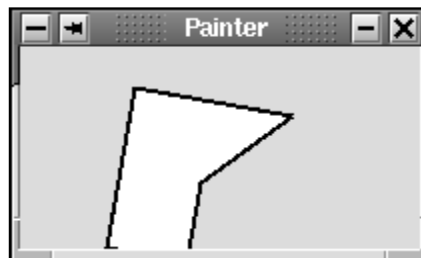


Abbildung 4-35 Rotation um 10° im Uhrzeigersinn um den Koordinatenursprung

Mit der Methode *scale* können Sie für die x- und y-Richtung Skalierungsfaktoren setzen. Die Koordinaten werden dazu mit diesen Werten multipliziert, bevor das Objekt gezeichnet wird. Für unser Beispiel bewirkt die folgende Zeile eine Ausgabe wie in Abbildung 4.36:

```
p.scale (1.5, 0.75);
```

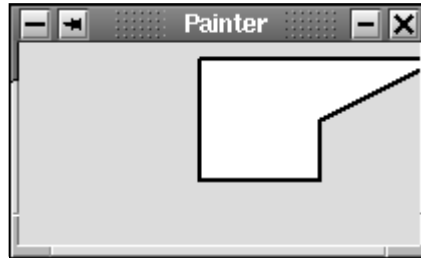


Abbildung 4-36 Vergrößerung in x-Richtung auf 150%, Verkleinerung in y-Richtung auf 75%

Eine Scherung (ein horizontales oder vertikales Verziehen des Objekts) können Sie mit der Methode *shear* erreichen. Den Faktor der Scherung in horizontaler und vertikaler Richtung können Sie in zwei verschiedenen Parametern festlegen. Der erste Parameter legt dabei die Stärke der Scherung in vertikaler Richtung fest (ein positiver Wert verschiebt alle Punkte mit positiver x-Koordinate nach unten, ein negativer Wert nach oben). Der zweite Parameter legt die Scherung in horizontaler Richtung fest (ein positiver Wert verschiebt alle Punkte mit positiver y-Koordinate nach rechts, ein negativer Wert nach links). Unser Beispielpolygon ist in Abbildung 4.37 um den Faktor 0,4 horizontal gesichert dargestellt. Zu jedem Punkt, der dargestellt werden soll, wird das 0,4fache der y-Koordinate zur x-Koordinate hinzuaddiert.

```
p.shear (0.0, 0.4);
```

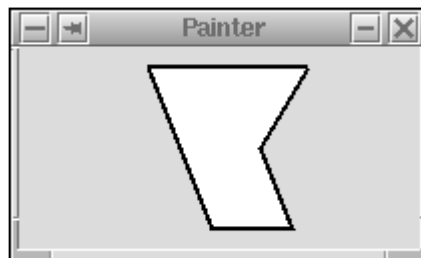


Abbildung 4-37 Scherung in horizontaler Richtung um den Faktor 0,4

Alle Transformationen können auch kombiniert werden. Um beispielsweise unser bisher benutztes Polygon nicht um den Koordinatenursprung, sondern um den Mittelpunkt des Polygons zu drehen, verschieben wir es mit *translate* zunächst so, dass sein Mittelpunkt auf dem Koordinatenursprung liegt. Anschließend rotieren wir es mit *rotate* um den gewünschten Winkel und schieben es mit *translate* wieder an seine Ausgangsposition zurück.

Die Transformationsschritte müssen in der umgekehrten Reihenfolge auf das *QPainter*-Objekt angewendet werden. In unserem Fall erhalten wir die gewünschte Transformation mit den folgenden drei Zeilen (siehe Abbildung 4.38):

```
p.translate (100, 50);  
p.rotate (45);  
p.translate (-100, -50);
```

Die dritte Zeile schiebt das Polygon, dessen Mittelpunkt bei (100,50) liegt, in den Koordinatenursprung. Die zweite Zeile rotiert es um 45° im Uhrzeigersinn. Die erste Zeile verschiebt das Polygon wieder an seine ursprüngliche Position.

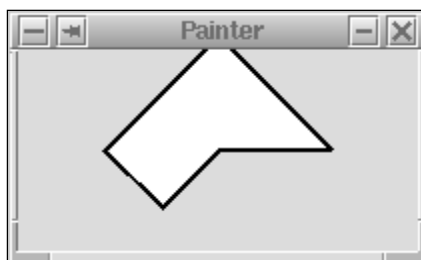


Abbildung 4-38 Rotation um den Mittelpunkt des Polygons

Die Reihenfolge der Transformationen ist wichtig und darf nicht vertauscht werden. So ist es ein Unterschied, ob man zunächst ein Objekt in x-Richtung um den Faktor 2 streckt und es dann um 90° dreht oder ob man die Rotation zuerst ausführt und dann in x-Richtung streckt. Beachten Sie also, dass Sie die Transformationen in der umgekehrten Reihenfolge ihrer Wirkung auf das *QPainter*-Objekt anwenden müssen.

View-Transformationen und World-Transformationen können gleichzeitig benutzt werden. Dabei werden auf die zu zeichnenden Elemente zuerst die World-Transformationen angewandt. Erst danach werden die View-Transformationen berechnet, bevor das Element gezeichnet wird.

Mit der *QPainter*-Methode *setWorldXForm (false)* können Sie alle eingestellten World-Transformationen deaktivieren, und sie mit *setWorldXForm (true)* wieder aktivieren. Alle Transformationen (View- und World-Transformationen) können Sie mit der Methode *resetXForm* löschen.

Die World-Transformationen werden in den *QPainter*-Objekten als 3x3-Matrix von *float*-Werten gespeichert. Diese Matrix ist in einem Objekt der Klasse *QWMatrix* gespeichert. Die Klasse *QWMatrix* bietet ebenfalls die Möglichkeit, Transformationen festzulegen und zu kombinieren. Sie können die Transformationsmatrix eines *QPainter*-Objekts mit der Methode *setWorldMatrix* setzen oder mit *worldMatrix* auslesen. (Sie können mit *setWorldMatrix* die neue Transformati-

onsmatrix direkt einsetzen oder sie mit der vorhandenen Matrix kombinieren.) So können Sie beispielsweise die Transformationsmatrix speichern und für viele *QPainter*-Objekte benutzen, ohne jedes Mal die Transformationen einzeln anzuwenden. Detaillierte Informationen über Koordinatentransformationen finden Sie zum Beispiel in dem Buch *Computer Graphics – Principles and Practice* von den Autoren Foley, van Dam, Fisher und Hughes aus dem Addison-Wesley-Verlag (ISBN 0-201-12110-7).

4.2.4 Beschränkung auf Ausschnitte

Alle gezeichneten Elemente eines *QPainter*-Objekts werden nur innerhalb des verbundenen *QPaintDevice*-Objekts gezeichnet. Was über den Rand hinausgeht, wird abgeschnitten (*Clipping*).

Qt bietet in der Klasse *QPainter* die Möglichkeit, dieses Abschneiden von Objekten noch weiter zu nutzen. Dazu kann der Bereich angegeben werden, in dem gezeichnet werden soll. Alles außerhalb dieses Bereichs wird von Zeichenoperationen nicht verändert. Der Zeichenbereich kann rechteckig oder oval sein, kann aber auch eine komplexe Gestalt haben.

Einen rechteckigen Zeichenbereich kann man sehr einfach mit der Methode *setClipRect* festlegen. Dazu gibt man die Koordinaten des Rechtecks in den Parametern an. Alle Zeichenoperationen werden anschließend nur innerhalb des Rechtecks ausgeführt; das Äußere des Rechtecks bleibt unverändert. Die folgenden zwei Zeilen bestimmen zunächst ein Rechteck von (40/40) bis zur Fensterecke unten rechts. Anschließend wird in das Fenster eine Ellipse gezeichnet, die am oberen und linken Rand über dieses Rechteck hinausragt. An diesen Kanten wird die Ellipse abgeschnitten. Das Ergebnis sehen Sie in Abbildung 4.39. Beachten Sie, dass an den abgeschnittenen Kanten die Begrenzungslinie *nicht* mit dem Zeichenstift nachgefahren wird.

```
p.setClipRect (40, 40, 160, 60);  
p.setBrush (QBrush (white));  
p.setPen (QPen (black, 2));  
p.drawEllipse (10, 10, 180, 80);
```

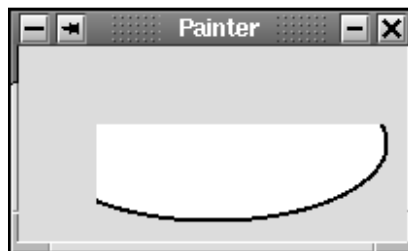


Abbildung 4-39 Rechteckiger Zeichenbereich mit *setClipRect*

Komplexere Regionen werden mit der Methode *setClipRegion* der Klasse *QPainter* festgelegt. Als Parameter bekommt diese Methode ein Objekt der speziellen Klasse *QRegion*, mit der man einen komplexen Bereich festlegen kann.

Im Konstruktor des *QRegion*-Objekts kann man den Bereich als ein Rechteck oder eine Ellipse festlegen. Einen rechteckigen Bereich von (40,40) mit einer Breite von 160 Pixel und einer Höhe von 60 Pixel (wie oben) legen Sie zum Beispiel mit folgender Zeile an:

```
QRegion r (40, 40, 160, 60);  
p.setClipRegion (r);
```

Diese beiden Zeilen haben also die gleiche Wirkung wie der Aufruf der Methode *setClipRect* im oberen Programm.

Einen ovalen Bereich legen Sie zum Beispiel mit der Zeile

```
QRegion r (80, 30, 100, 40, QRegion::Ellipse);
```

an. Die Ellipse hat hier eine Breite von 100 Pixel und eine Höhe von 40 Pixel, und die obere linke Ecke des umgebenden Rechtecks liegt bei den Koordinaten (80/30). Sie können auch ein beliebiges Polygon als Zeichenbereich benutzen, indem Sie einen anderen Konstruktor verwenden:

```
QPointArray a (3, 50, 0, 0, 100, 100, 100);  
QRegion r (a);
```

Der Zeichenbereich ist in diesem Fall ein Dreieck mit den Eckpunkten (50/0), (0/100) und (100/100). Sie können bei diesem Konstruktor noch einen booleschen Wert als zweiten Parameter angeben, der festlegt, ob in selbstdurchdringenden Polygonen die inneren Bereiche nach der Even-Odd-Regel (*false*, Defaultwert) oder nach der Winding-Regel (*true*) ausgefüllt werden (siehe auch die Methode *drawPolygon* in Kapitel 4.2.1, *QPainter*, im Abschnitt *Methoden zum Zeichnen ausgefüllter Primitive*).

Objekte der Klasse *QRegion* können durch die Methoden *unite*, *intersect*, *subtract* und *eor* miteinander verknüpft werden, um neue, komplexe Bereiche zu definieren. Das Ergebnis der Methode *unite* ist der Bereich, der durch die Vereinigung der beiden Teilbereiche entsteht. *intersect* hat die Schnittmenge beider Bereiche als Ergebnis. *subtract* zieht den zweiten Bereich vom ersten ab, und *eor* liefert den Bereich zurück, der nur von genau einem der beiden Ausgangsbereiche überdeckt wird. Jede der vier Methoden hat als Parameter genau ein *QRegion*-Objekt, das mit dem aktuellen Objekt verknüpft wird, dessen Methode aufgerufen wird. Beide *QRegion*-Objekte bleiben unverändert. Das Ergebnis der Verknüpfung wird als Rückgabewert der Methode in einem neuen *QRegion*-Objekt zurückgeliefert. Diese Verknüpfung wollen wir anhand von zwei Beispielen näher erläutern. Im ersten Beispiel vereinigen wir zwei kreisförmige Bereiche:

```

p.fillRect (0, 0, 200, 100,
            QBrush (black, DiagCrossPattern));
QRegion c1 (0, 0, 100, 100, QRegion::Ellipse);
QRegion c2 (100, 0, 100, 100, QRegion::Ellipse);
QRegion c3 = c1.unite (c2);
p.setClipRegion (c3);
p.setBrush (QBrush (white));
p.setPen (QPen (black, 2));
p.drawEllipse (10, 10, 180, 80);

```

Die erste Zeile füllt das Fenster mit einem Muster, damit die unveränderten Bereiche leichter zu erkennen sind. Die *QRegion*-Objekte *c1* und *c2* enthalten jeweils einen Kreis. *c3* ist die Vereinigung der beiden Kreise. Beim Zeichnen der Ellipse wird nun nur der Teil gezeichnet, der innerhalb von mindestens einem der Kreise liegt. Das Ergebnis sehen Sie in Abbildung 4.40.

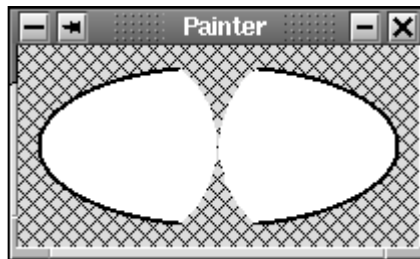


Abbildung 4-40 Vereinigung von zwei kreisförmigen Bereichen

In unserem zweiten Beispiel beginnen wir mit einem Bereich, der das ganze Fenster enthält. Von diesem Bereich ziehen wir ein kleineres Rechteck am oberen Rand ab und fügen ein darin enthaltenes, noch kleineres Rechteck wieder hinzu. Als Ergebnis erhalten wir einen Zeichenbereich, der das ganze Fenster bis auf einen Streifen aus drei schmalen Rechtecken enthält. In diesem Beispiel benutzen wir keine zusätzlichen Variablen für die *QRegion*-Objekte, sondern wenden die Verknüpfungsmethoden direkt an. Das Ergebnis sehen Sie in Abbildung 4.41.

```

p.fillRect (0, 0, 200, 100,
            QBrush (black, DiagCrossPattern));
QRegion c1 = QRegion (0, 0, 200, 100)
    .subtract (QRegion (50, 0, 100, 40))
    .unite (QRegion (60, 0, 80, 30));
p.setClipRegion (c1);
p.setBrush (QBrush (white));
p.setPen (QPen (black, 2));
p.drawEllipse (10, 10, 180, 80);

```

Auf diese Weise lassen sich sehr komplexe Zeichenbereiche definieren.

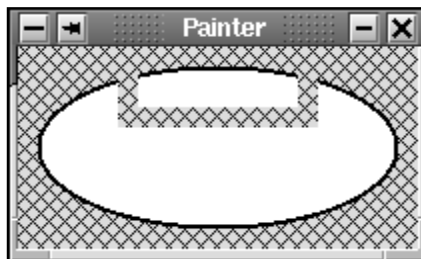


Abbildung 4-41 Verknüpfung von drei rechteckigen Bereichen

4.2.5 Effizienzbetrachtungen

Das Zeichnen von Grafikprimitiven in ein *QWidget*- oder *QPixmap*-Objekt übernimmt in den meisten Fällen vollständig der X-Server. Mit einem geeigneten Grafikkartentreiber kann man so eine sehr hohe Zeichengeschwindigkeit erreichen. Die einfachen Strichprimitive werden dabei in ein einziges X-Server-Kommando umgesetzt, die ausgefüllten Primitive meist in zwei Kommandos: zuerst in eines zum Füllen der Fläche und danach in ein weiteres Kommando, um die Begrenzungslinie zu zeichnen. Hat man *NoBrush* oder *NoPen* gewählt, so entfällt einfach das entsprechende X-Kommando.

Eine Ausnahme von dieser Umsetzung in einfache X-Kommandos ist zum einen die Methode *drawQuadBezier*. In ihr wird die Bezierkurve in eine ausreichend feine *PolyLine* umgewandelt, also in eine Kette von geraden Linien. Diese wird dann (in einem einzelnen X-Kommando) dargestellt. Zum anderen werden alle runden Primitive (*drawArc*, *drawEllipse*, *drawChord*, *drawPie*) sowie das Rechteck (*drawRect* und *drawRoundRect*) in einen feinen Polygonzug umgewandelt, sobald eine Rotation oder Scherung angewendet werden soll. Dann lassen sich diese Primitive nämlich nicht mehr vom X-Server einfach darstellen. Für die *QPaintDevice*-Klassen *QPrinter* und *QPicture* wird diese Umsetzung in Linienzüge immer vorgenommen.

4.2.6 Unterklassen von *QPaintDevice*

QPaintDevice repräsentiert ein Objekt, auf das man einen *QPainter* anwenden kann. Hierbei handelt es sich um eine abstrakte Klasse, von der keine Objekte erzeugt werden können. Stattdessen gibt es in Qt die vier abgeleiteten Klassen *QWidget*, *QPixmap*, *QPrinter* und *QPicture*.

Zu einem Zeitpunkt kann auf einem *QPaintDevice*-Objekt immer nur maximal ein *QPainter*-Objekt aktiv sein. Wenn Sie ein weiteres *QPainter*-Objekt verbinden wollen, so gibt Qt eine Fehlermeldung aus. Die häufigste Anwendung eines *QPainter*-Objekts sieht daher so aus, dass Sie *QPainter* und *QPaintDevice* verbinden, ein paar Zeichenkommandos ausführen und anschließend die Verbindung wieder beenden.

Qt definiert eine globale Funktion *bitBlt* (Abkürzung für »bit block transfer«), mit der man einen Ausschnitt eines *QPaintDevice* in ein anderes *QPaintDevice* kopieren kann. Eine genauere Beschreibung der Funktion finden Sie in Kapitel 4.3, *Teilbilder – QImage und QPixmap*. Für das *QPaintDevice*, das als Quelle für den kopierten Bereich dient, gibt es allerdings eine Einschränkung: Es muss sich um ein internes Device handeln. Zu den internen Devices, die man also auch wieder auslesen kann, zählen nur *QPixmap* und *QWidget*. *QPrinter* und *QPicture* dagegen können nicht ausgelesen werden. Sie können also nicht als Quelle dienen. Als Ziel eines *bitBlt*-Funktionsaufrufs sind jedoch alle *QPaintDevices* zugelassen.

Mit der Klasse *QPaintDeviceMetrics* können Sie zusätzliche Angaben über die Abmessungen eines *QPaintDevice* erhalten. Dazu erzeugen Sie einfach ein *QPaintDeviceMetrics*-Objekt mit dem gewünschten *QPaintDevice* als Parameter im Konstruktor und können dann z.B. über die Methoden *heightMM* und *widthMM* die Höhe und Breite in Millimetern erfragen (die für *QPixmap* und *QWidget* allerdings nur ungefähre Angaben sind). Eine Anwendung kann zum Beispiel so aussehen:

```
QPrinter prn;
QPaintDeviceMetrics pdm (&prn);
printf ("Papierabmessungen: %d x %d mm",
        pdm.widthMM (), pdm.heightMM ());
```

Das kurze Beispiel gibt die Papierbreite und -höhe auf dem Bildschirm aus, die zur Zeit auf dem Drucker eingestellt sind.

Im Folgenden wollen wir kurz die vier von *QPaintDevice* abgeleiteten Klassen betrachten.

QWidget

Die Klasse *QWidget*, die bereits in Kapitel 3.2, *Die Fensterklasse – QWidget*, genauer betrachtet wurde, stellt einen rechteckigen Bildschirmbereich dar. Wie auf allen *QPaintDevice*-Objekten kann man mit Hilfe der Klasse *QPainter* auf dem Device zeichnen. Da Fenster auf dem Bildschirm von anderen Fenstern überdeckt sein können, reicht ein einmaliges Zeichnen auf dem Bildschirm nicht aus. Alle Zeichenoperationen auf ein *QWidget* sollten daher innerhalb der virtuellen Methode *paintEvent* stattfinden. Diese Methode wird immer dann aufgerufen, wenn ein Bereich des Widgets sichtbar wird. Im übergebenen Parameter sind die Koordinaten des Bereichs abgelegt, der sichtbar geworden ist, so dass Sie nur diesen Bereich neu zeichnen müssen. Wie Sie eigene Widgets entwerfen und programmieren, können Sie in Kapitel 4.4, *Entwurf eigener Widget-Klassen*, nachlesen.

QWidget ist ein internes *QPaintDevice*, d.h. es kann als Quelle einer *bitBlt*-Operation dienen. Teile des Widgets, die dabei von anderen Fenstern verdeckt sind, ergeben einen undefinierten Bereich beim Kopieren.

QPixmap

QPixmap ist ein Speicherbereich im Speicher des X-Servers, der sich genauso verhält wie ein *QWidget*, nur dass er nicht sichtbar ist. Meist wird dieser Speicherbereich sogar im Bildschirmspeicher auf der Grafikkarte abgelegt, sofern dort noch genügend freier Speicher vorhanden ist. Dann kann die Grafikbeschleunigung beim Zeichnen wirksam werden, und das Kopieren mit *bitBlt* von einem Widget in eine *Pixmap* und umgekehrt läuft vollständig innerhalb des Bildschirmspeichers ab.

Eine genauere Beschreibung der Klasse *QPixmap* finden Sie in Kapitel 4.3, *Teilbilder – QImage und QPixmap*.

QPixmap ist ebenfalls ein internes *QPaintDevice*, kann also als Quelle einer *bitBlt*-Operation dienen.

QPrinter

QPrinter repräsentiert einen Drucker. In der Unix-Version von Qt ist er als Postscript-Device ausgelegt, in der Version für Microsoft Windows wird der ausgewählte Druckertreiber verwendet. Die Ausgabe kann über ein Programm an den Drucker geschickt (Default ist *lpr*) oder in einer Datei abgelegt werden. Einige Beispiele für die Anwendung finden Sie in Kapitel 4.17, *Drucken mit Qt*.

QPrinter ist ein externes Device, kann also nicht als Quelle für eine *bitBlt*-Operation dienen. Schließlich ist ein Drucker ja kein Scanner.

QPicture

QPicture ist ein Device, das alle Zeichenoperationen, die mit einem *QPainter* ausgeführt werden, speichert und bei Bedarf wieder abrufen kann. Die gespeicherten Daten können auch in einer Datei abgespeichert oder aus einer Datei wieder eingelesen werden (in einem plattformunabhängigen Binärformat). Ein typisches Beispiel hierfür kann so aussehen:

```
QPicture pic;
QPainter painter;
painter.begin (&pic);
// Hier können Zeichen-Operationen auf p stehen
painter.end ();
if (!pic.save ("output.dat"))
    debug ("Datei konnte nicht geschrieben werden!");;
```

Dieses Programmfragment speichert alle Operationen, die zwischen *painter.begin* und *painter.end* aufgerufen wurden, in der Binärdatei *output.dat* ab. (Die Dateinendung ist beliebig, eine Standardendung für Qt-Picture-Dateien gibt es nicht.)

Ein anderes Programm (auf einem anderen Rechner, sogar mit einem anderen Betriebssystem) könnte zum Beispiel mit folgendem Programmstück diese Datei

wieder einlesen und die gespeicherten Zeichenoperationen in einem Widget ausführen lassen:

```
QPicture pic;  
if (!pic.load ("output.dat"))  
    debug ("Datei konnte nicht gelesen werden!");  
QPainter paint;  
paint.begin (&myWidget);  
paint.drawPicture (pic);  
paint.end ();
```

Anstatt die Daten eines *QPicture*-Objekts in einer Datei zu speichern, können Sie auch mit den Methoden *data* und *size* direkt auf die Daten zugreifen und sie in einem anderen *QPicture*-Objekt mit *setData* setzen. Anstatt die Daten in eine Datei zu schreiben, können Sie sie so zum Beispiel auch über eine Internet-Verbindung schicken.

4.2.7 Übungsaufgaben

Übung 4.6

Zeichnen Sie mit einem *QPainter* die Grafiken aus den Abbildungen 4.42, 4.43 und 4.44.

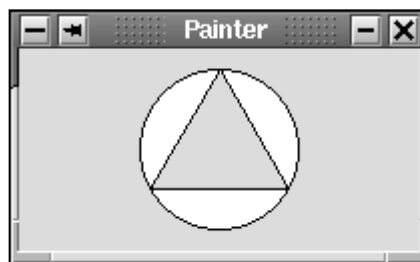


Abbildung 4-42 Unausgefülltes Dreieck in einem Kreis

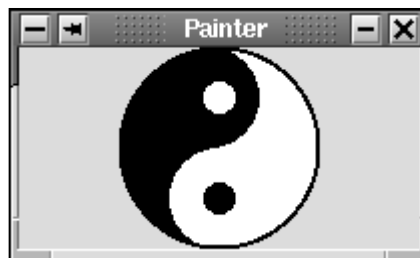


Abbildung 4-43 Das Yin-Yang-Symbol



Abbildung 4-44 Ein Button mit abgerundeten Ecken

4.3 Teilbilder – QImage und QPixmap

Oftmals ist es beim Zeichnen nötig, Teilbilder nicht direkt auf den Bildschirm zu zeichnen, sondern zunächst in einen Speicherbereich. So kann man beispielsweise eine Zeichnung zunächst vollständig ausführen und anschließend das ganze im Speicher befindliche Teilbild mit einem einzigen Befehl auf den Bildschirm kopieren. Diese Technik wird zum Beispiel benutzt, um ein kurzes Flimmern auf dem Bildschirm zu verhindern, wenn ein gezeichnetes Objekt ein anderes kurzzeitig übermalt. Genauer über diese Anwendung erfahren Sie in Kapitel 4.5, *Flimmerfreie Darstellung*.

Qt besitzt zwei Klassen, *QImage* und *QPixmap*, in denen Teilbilder im Speicher abgelegt werden können. Der grundlegende Unterschied ist der Ort, an dem das Bild gespeichert wird. Während *QImage* einen Speicherbereich in Ihrem Programm belegt, auf den Sie direkten Zugriff haben, wird das Bild von *QPixmap* im X-Server gespeichert. Ihr Programm besitzt nur eine Kennung. Um mit dem Bild zu arbeiten, muss Ihr Programm Befehle an den X-Server senden, die die entsprechende Kennung enthalten. Dieser grundlegende Unterschied und die daraus entstehenden Konsequenzen werden in den beiden nächsten Kapiteln, 4.3.1, *QImage*, und 4.3.2, *QPixmap*, beschrieben. Kapitel 4.3.3, *Effizienzbetrachtungen*, soll Ihnen eine Orientierung bieten, welche der beiden Klassen Sie für eine spezielle Aufgabe benutzen sollten und wie Sie langsame Berechnungen vermeiden. Dort wird auch die Hilfsklasse *KImageIO* aus der KDE-Bibliothek vorgestellt, die mit Hilfe von Shared Memory noch einmal eine deutliche Geschwindigkeitssteigerung erreichen kann.

Oftmals sollen Teile eines Bildes durchsichtig sein, so dass der Hintergrund an diesen Stellen erhalten bleibt. Sowohl *QPixmap* als auch *QImage* bieten diese Möglichkeit an. Eine genaue Beschreibung finden Sie in Kapitel 4.3.4, *Transparenz in Teilbildern*.

4.3.1 QImage

Ein Objekt der Klasse *QImage* speichert die Daten eines Bildes im Hauptspeicher der Applikation. Die Farbtiefe kann dabei aus drei verschiedenen Werten ausgewählt werden, die angeben, wie viele Bit pro Pixel gespeichert werden sollen. Farbtiefe 1 bedeutet, dass ein Pixel durch ein einziges Bit repräsentiert wird. Es kann daher den Wert 0 oder 1 haben. Bei einer Farbtiefe von 8 kann ein Pixel einen Wert zwischen 0 und 255 besitzen. Das *QImage*-Objekt besitzt dazu eine Tabelle, in der jedem Wert ein RGB-Wert zugeordnet werden kann. Bei einer Farbtiefe von 32 Bit pro Pixel wird der Farbwert direkt in TrueColor-Qualität abgespeichert. Ein Pixel erhält acht Bit für den Rot-Anteil, acht Bit für den Grün-Anteil und acht Bit für den Blau-Anteil der Farbe. Die verbleibenden acht Bit pro Pixel bleiben frei. Sie können aber auch genutzt werden, um als so genannter Alpha-Wert für eine beliebige Transparenz zwischen undurchsichtig und durchsichtig zu sorgen. (Ein Wert von 0 für die Transparenz bedeutet vollständige Durchsichtigkeit. Ein Wert von 255 bedeutet volle Deckwirkung. Beim Anzeigen eines *QImage*-Objekts mit Alpha-Kanal gibt es verschiedene Techniken, um diese Transparenz zu erreichen. Eine genaue Beschreibung finden Sie in Kapitel 4.3.4, *Transparenz in Teilbildern*.) Eine Farbtabelle ist bei der Farbtiefe 32 nicht möglich, aber auch nicht nötig.

Sie können ein *QImage*-Objekt erzeugen, indem Sie im Konstruktor die Breite, *width*, die Höhe, *height*, und die Farbtiefe, *depth*, angeben. In zwei weiteren, optionalen Parametern können Sie festlegen, wie viele Farben die Farbtabelle besitzen soll (*numColors*) und wie die Bitreihenfolge für die Farbtiefe 1 Bit pro Pixel sein soll (*bitOrder*). *numColors* ist nur sinnvoll für eine Farbtiefe von 1 oder 8, denn für eine Farbtiefe von 32 wird keine Farbtabelle angelegt. Wenn Sie hier den Wert 0 angeben (das ist auch der Default-Wert), bekommt die Farbtabelle zwei Einträge für die Farbtiefe 1 bzw. 256 Einträge für die Farbtiefe 8. Der Wert von *bitOrder* ist nur für eine Farbtiefe von einem Bit pro Pixel sinnvoll. Geben Sie in diesem Fall den Wert *QImage::BigEndian* (das Pixel links auf dem Bildschirm entspricht dem Bit 7 in einem Byte) oder *QImage::LittleEndian* (das Pixel links auf dem Bildschirm entspricht Bit 0 in einem Byte) an. Für alle anderen Farbtiefen benutzen Sie *QImage::IgnoreEndian*.

Die folgende Zeile erzeugt ein *QImage*-Objekt, *img*, mit einer Breite von 300 Pixel, einer Höhe von 150 Pixel und einer Farbtiefe von 8 Bit. Die Farbpalette erhält die standardmäßigen 256 Einträge:

```
QImage img (300, 150, 8);
```

Der Konstruktor erzeugt bereits den Speicher für die Farbtabelle und für den Bildbereich. Sowohl die Farbtabelle als auch der Bildbereich bleiben aber uninitialized, enthalten also in der Regel zufällige Daten.

Da die Bilddaten auf dem Heap der Applikation angelegt werden, kann die Applikation direkt auf diese Daten zugreifen, also sehr effizient ein einzelnes Pixel auf einen anderen Wert setzen oder die Farbe eines Pixels auslesen. *QImage* ist also immer dann ganz besonders gut geeignet, wenn Sie die Farbe jedes Pixels einzeln setzen wollen. Typische Anwendungen sind beispielsweise aufwendige Farbverläufe, bei denen jedes Pixel einzeln berechnet wird, berechnete Bilder wie Mandelbrot-Mengen oder fotorealistische Ray-Tracer. Auch zum Speichern von Bilddaten in einer Datei und zum Wiedereinlesen ist *QImage* gut geeignet.

Die Bilddaten werden im Speicher zeilenweise abgelegt, jede Zeile von links nach rechts. Bei einer Farbtiefe von einem Bit pro Pixel repräsentiert jedes Byte acht Pixel. Je nach Einstellung von *bitOrder* im Konstruktor ist dabei das Pixel ganz links das Bit 7 (*BigEndian*) oder Bit 0 (*LittleEndian*). Wenn die Breite einer Zeile keine durch acht teilbare Zahl ist, bleiben die letzten Bits im letzten Byte frei, so dass eine neue Zeile wieder mit einem neuen Byte beginnt. Bei einer Farbtiefe von acht Bit entspricht jedes Pixel einem Byte. Bei einer Farbtiefe von 32 Bit wird ein Pixel durch vier Byte angegeben. *QImage* enthält ein Array von Zeigern, die den Anfang jeder Zeile im Speicher angeben. Mit der Methode *scanLine(i)* können Sie die Anfangsadresse der Zeile *i* erfragen (wobei *i* die Werte von 0 bis *height*-1 annehmen kann).

Anstatt aufwendige Adressberechnungen eines Pixels selbst durchzuführen, können Sie auch auf die Methoden *pixel*, *pixelIndex* und *setPixel* zurückgreifen. *pixel(x, y)* liefert den RGB-Wert des Pixels an den Koordinaten (*x/y*) zurück. *pixelIndex(x, y)* liefert den Index in die Farbtabelle des Pixels an den Koordinaten (*x/y*) zurück. Diese Methode funktioniert nur für ein *QImage*-Objekt mit einer Farbtiefe von einem oder acht Bit pro Pixel. Bei 32 Bit pro Pixel gibt es keine Farbtabelle; das Ergebnis der Methode ist daher undefiniert. Mit der Methode *setPixel(x, y, value)* können Sie die Farbe des Pixels an den Koordinaten (*x/y*) ändern. Für *QImage*-Objekte mit Farbtabelle (also Farbtiefe 1 oder 8) ist *value* der Index in die Farbtabelle. Bei einem *QImage*-Objekt ohne Farbtabelle (also Farbtiefe 32) enthält *value* die Farbe in RGB-Anteilen.

Um die Rot-, Grün- und Blau-Anteile einer Farbe abzuspeichern, wird die Struktur *QRgb* benutzt. Diese Struktur belegt vier Byte und ist kompatibel zu einer 32-Bit-Integerzahl. Mit der Hilfsfunktion *qRgb* können Sie eine solche Struktur erzeugen, indem Sie als Parameter die Werte für Rot, Grün und Blau übergeben (im Wertebereich 0 bis 255). Der Rückgabewert der Funktion ist eine *QRgb*-Struktur, die diesen Farbwert repräsentiert. Mit den Hilfsfunktionen *qRed*, *qGreen* und *qBlue* können Sie die entsprechenden Farbanteile aus einer *QRgb*-Struktur wieder auslesen. Die *QRgb*-Struktur wird besonders bei der Bearbeitung von *QImage*-Objekten der Klasse *QColor* bevorzugt, da sie viel weniger Overhead besitzt.

Das folgende Programmstück erzeugt ein *QImage*-Objekt von 50x50 Pixel, das einen radialen Farbverlauf von Blau (in der Mitte) nach Weiß (in den Ecken) enthält:

```
QImage img (50, 50, 32);
int x, y;
// Länge der Diagonale des Quadrats
double diagonale = sqrt (2.0) * 50;
for (y = 0; y < 50; y++)
    for (x = 0; x < 50; x++)
    {
        // Koordinaten relativ zur Mitte des Quadrats
        double rx = x - 24.5;
        double ry = y - 24.5;
        // Abstand von der Mitte des Quadrats
        double abstand = sqrt (rx * rx + ry * ry);
        // Abstandswert als int von 0 bis 255
        int abstandInt = abstand / diagonale * 2 * 255;
        // Farbwert ermitteln:
        // Blau-Anteil immer maximal, Rot und Grün steigen
        // von der Mitte (0) bis zu den Ecken (255)
        QRgb farbe = qRgb (abstandInt, abstandInt, 255);
        // Pixel eintragen
        img.setPixel (x, y, farbe);
    }
```

Dieser eindimensionale Farbverlauf kann auch sehr gut mit einem *QImage*-Objekt der Farbtiefe 8 gespeichert werden, indem die Farbtabelle den Farbverlauf enthält und im *QImage*-Objekt nur der Abstand eingetragen wird. Mit der Methode *setColor* kann ein Eintrag in der Farbtabelle geändert werden:

```
QImage img (50, 50, 8);
// Zunächst Farbtabelle erstellen mit Farbverlauf
// von Blau (Eintrag 0) bis Weiß (Eintrag 255)
for (int i = 0; i < 256; i++)
    img.setColor (i, qRgb (i, i, 255));
int x, y;
// Länge der Diagonale des Quadrats
double diagonale = sqrt (2.0) * 50;
for (y = 0; y < 50; y++)
    for (x = 0; x < 50; x++)
    {
        // Koordinaten relativ zur Mitte des Quadrats
        double rx = x - 24.5;
        double ry = y - 24.5;
        // Abstand von der Mitte des Quadrats
        double abstand = sqrt (rx * rx + ry * ry);
        // Abstandswert als int von 0 bis 255
```

```
int abstandInt = abstand / diagonale * 2 * 255;
// Abstand als Index auf die Farbtabelle
// in das Bild eintragen
img.setPixel (x, y, abstandInt);
}
```

Die hier angestellten Berechnungen laufen alle im eigenen Prozess ohne Zugriff auf den X-Server ab. Da es sich nur um »Zahlenschiebereien« handelt, ist dieser Vorgang recht effizient.

Um dieses Teilbild auf dem Bildschirm anzuzeigen, muss es zunächst zum X-Server übertragen werden. Dazu ist unter Umständen eine aufwendige Farbumrechnung nötig. Benutzt der X-Server einen TrueColor- oder RealColor-Grafikmodus, so sind diese Umrechnungen sehr einfach: Jeder Pixelwert im *QImage*-Objekt wird direkt in einen Wert für die Grafikkarte umgerechnet. Benutzt der Grafikmodus jedoch selbst eine Farbtabelle (*Color Lookup Table*), muss zu jedem Pixel im *QImage*-Objekt ein möglichst geeigneter Farbwert in der Farbtabelle des X-Servers gefunden werden. Auch wenn das *QImage*-Objekt eine Farbtiefe von acht Bit benutzt, also selbst eine Farbtabelle besitzt, müssen die Einträge dieser Farbtabelle geeigneten Einträgen in der Farbtabelle des X-Servers zugeordnet werden. Nähere Informationen über die Farbverwaltung von Qt finden Sie in Kapitel 4.1, *Farben unter Qt*. Die Effizienz von *QImage*-Objekten wird in Kapitel 4.3.3, *Effizienzbetrachtungen*, genauer untersucht.

Ein *QImage*-Objekt kann mit der Methode *QPainter::drawImage* auf den Bildschirm gezeichnet werden. In dieser Methode können Sie die Koordinaten angeben, an denen die linke obere Ecke des *QImage*-Objekts liegen soll. Außerdem können Sie die Ausgabe auf einen Teil des *QImage*-Objekts einschränken lassen. Mit der folgenden Zeile zeichnen Sie das oben berechnete Objekt *img* auf dem Bildschirm in das Widget *widget* an der Position (80/70):

```
QPainter p;
p.begin (widget);
p.drawImage (80, 70, img);
p.end();
```

Soll dagegen der Ausschnitt ab den Koordinaten (10/15) mit der Breite von 30 und der Höhe von 20 Pixel ausgegeben werden, so benutzen Sie:

```
QPainter p;
p.begin (widget);
p.drawImage (80, 70, img, 10, 15, 30, 20);
p.end();
```

Die Methode *drawImage* wandelt das *QImage*-Objekt (bzw. den gewählten Ausschnitt) in ein *QPixmap*-Objekt um, gibt es dann auf dem Bildschirm aus und löscht das *QPixmap*-Objekt wieder. Alle Transformationen, die im *QPainter*-

Objekt eingetragen sind (Vergrößerungen, Verschiebungen, Rotationen, Scheitungen), werden auch auf das *QImage*-Objekt angewandt. Die Berechnungen sind dabei zum Teil so aufwendig, dass die Ausgabe deutlich langsamer werden kann. Seien Sie also möglichst sparsam mit der Ausgabe transformierter *QImage*-Objekte.

Der Inhalt eines *QImage*-Objekts kann direkt aus einer Grafikdatei eingelesen werden. Am einfachsten geben Sie dazu im Konstruktor als einzigen Parameter den Dateinamen an. Qt öffnet diese Datei, versucht, das Dateiformat zu erkennen, und liest die Daten aus der Datei in das *QImage*-Objekt ein. Die Breite und Höhe sowie die Farbtiefe werden automatisch erzeugt. Alternativ kann man auch in einem bereits erzeugten *QImage*-Objekt die Methode *load* benutzen. Bei *load* kann man zusätzlich als zweiten Parameter das Dateiformat angeben. Das Format wird durch einen String festgelegt. Die am häufigsten benutzten Werte sind hier *GIF*, *JPG*, *BMP*, *XBM*, *XPM* und *PNG*. Wenn Sie einen Null-Zeiger als Formatangabe verwenden, versucht Qt, das Format selbstständig zu erkennen. Welche Dateiformate Qt automatisch erkennt, hängt unter anderem davon ab, welche beim Kompilieren der Qt-Bibliotheken eingebunden wurden. Insbesondere *JPG*, *GIF* und *PNG* können ausgeschlossen sein. Verlassen Sie sich also nicht unbedingt darauf, dass diese Formate auf allen Systemen problemlos gelesen werden können.

Analog zum Laden können Sie mit der Methode *save* ein *QImage*-Objekt als Grafikdatei speichern. Dazu legen Sie im ersten Parameter den Dateinamen fest, im zweiten wie bei *load* das Grafikformat. Das Grafikformat muss hier angegeben werden. Alle Grafikformate, die auch bei *load* verwendet werden können, sind hier erlaubt. Eine Ausnahme bildet das GIF-Format. Da die Firma Unisys ein Patent auf die Komprimierung von Bilddateien mit dem LZW-Verfahren besitzt, ist es in einigen Ländern nicht legal, GIF-Dateien zu erstellen, ohne eine Lizenz von Unisys eingeholt zu haben. Das Schreiben einer GIF-Datei ist daher in Qt nicht möglich.

Sie können eigene Dateiformate erkennen und dekodieren und kodieren lassen, indem Sie je eine Funktion zum Lesen und eine Funktion zum Schreiben implementieren. Diese Funktion erhält als Parameter ein Objekt der Klasse *QImageIO*. Mit der Methode *QImageIO::fileName* kann man in den Funktionen den Dateinamen erfragen. Mit *QImageIO::setImage* kann man das dekodierte Bild festlegen (beim Einlesen), mit *QImageIO::image* das zu kodierende Bild erhalten (beim Schreiben). Diese beiden Funktionen müssen Sie mit der statischen Methode *QImageIO::defineIOHandler* registrieren lassen. Dateien in Ihrem Format werden dann automatisch in *QImage::load* und *QImage::save* verwendet, wenn Sie das entsprechende Format angegeben haben.

Für Dateien im XPM-Format gibt es zusätzlich ein besonderes Verfahren, um sie in das Programm einzubinden. Das Dateiformat einer XPM-Datei ist ein reiner

ASCII-Text, der ein syntaktisch korrektes C-Programm darstellt. (Öffnen Sie zum Beispiel einmal eine XPM-Datei mit einem normalen Texteditor.) Eine XPM-Datei kann daher mit `#include` in den Quelltext eingebunden werden. In diesem Quelltext wird eine Variable definiert, die ein Array von Strings enthält. Der Variablenname ist normalerweise der Dateiname ohne die Endung `.xpm`, sofern die Datei nicht nachträglich umbenannt wurde. Diese Strings enthalten in kodierter Form die Bildinformationen. (Die Kodierung ist dabei meist sehr einfach: Je nach benutzter Farbanzahl wird ein Pixel von einem oder zwei Zeichen repräsentiert.) Diese Variable können Sie ebenfalls als Parameter für einen *QImage*-Konstruktor verwenden. Diese Methode hat den Vorteil, dass das Programm das Bild nicht von der Festplatte lesen muss. Insbesondere kann es keine Probleme mit falschen Pfaden oder Lesefehlern geben. Ihr Nachteil ist, dass die kodierte Form des Bildes in der ausführbaren Datei gespeichert ist. Das belegt den Hauptspeicher doppelt: zum einen für das kodierte Bild, zum anderen für das *QImage*-Objekt. Diese Möglichkeit sollte also nur für kleine Bilder verwendet werden. Das folgende Stück Programmcode erzeugt ein *QImage*-Bild *img* aus der XPM-Datei *exit.xpm*:

```
// Einbinden der XPM-Datei
// Diese Datei muss dazu im aktuellen Verzeichnis
// oder im Include-Pfad liegen.
#include "exit.xpm"
....

// Erzeugen des QImage-Objekts
QImage img (exit);
```

4.3.2 QPixmap

Im Gegensatz zur Klasse *QImage* wird ein Teilbild eines *QPixmap*-Objekts nicht im Speicher der Applikation abgelegt, sondern im Speicher des X-Servers. Da das Programm und der X-Server auf zwei verschiedenen Rechnern laufen können, ist der Unterschied durchaus relevant.

Moderne Grafikkarten haben einen viel größeren Bildspeicher, als für die Anzeige nötig ist. So benötigt die Grafikkarte zum Speichern des Bildschirminhalts bei einer Auflösung von 1280x1024 Pixel bei 32-Bit-TrueColor etwas über 5 MByte an Speicher. Viele Grafikkarten besitzen aber bereits 16 MByte Grafikspeicher und mehr. Die Daten von *QPixmap*-Objekten werden von X-Servern daher oft in dem noch verbleibenden Grafikspeicher abgelegt. Dadurch kann ein *QPixmap*-Objekt extrem schnell angezeigt werden, da die Bilddaten nur innerhalb des Grafikspeichers verschoben werden müssen. Das erledigen die meisten Grafikkarten inzwischen selbstständig, so dass der Hauptprozessor überhaupt nicht belastet wird. Da der Datenbus auf einer Grafikkarte oftmals auch sehr breit und sehr schnell getaktet ist, sind die Übertragungsraten innerhalb des Grafikspeichers enorm hoch.

Auf ein *QPixmap*-Objekt können die gleichen Zeichenoperationen ausgeführt werden wie auf ein Widget. Da viele Grafikkarten auch hier schon spezielle Grafikprozessoren besitzen, um die grundlegenden Zeichenoperationen (Linien, Polygone, Flächen) selbstständig ohne Hilfe des Hauptprozessors auszuführen, werden diese Operationen auf *QPixmap*-Objekten, deren Daten im Grafikspeicher stehen, ebenfalls enorm beschleunigt.

Auch wenn der freie Grafikspeicher für die Daten eines *QPixmap*-Objekts nicht mehr ausreicht, kann man weitere *QPixmap*-Objekte anlegen. Die zugehörigen Daten werden dann im Hauptspeicher des X-Servers angelegt. Die Verarbeitung der Daten ist in diesem Fall zwar nicht mehr so schnell, die Funktionalität ist aber in keiner Weise eingeschränkt.

Ein *QPixmap*-Objekt enthält also nicht die Daten des Bildes, sondern nur eine Kennung, mit der es die Bilddaten auf dem X-Server ansprechen kann. Der direkte Zugriff auf einzelne Pixel ist damit sehr ineffizient. Um zum Beispiel ein einzelnes Pixel eines *QPixmap*-Objekts zu ändern, sind einige Zeilen notwendig:

```
QColor c (128, 0, 0); // ein dunkles Rot
QPainter p;           // Painter erstellen
p.begin (&pix);       // mit QPixmap pix verbinden
p.setPen (c);          // Farbe setzen
p.drawPoint (80, 30);  // einen Punkt zeichnen
p.end();              // Verbindung wieder aufheben
```

Zumindest die Methoden *setPen* und *drawPoint* sind für jedes einzelne Pixel aufzurufen, das gezeichnet werden soll. Ein Auslesen eines einzelnen Pixels aus einem *QPixmap*-Objekt ist nicht möglich.

Da *QPixmap* von *QPaintDevice* abgeleitet ist, kann man allerdings auch aufwendige Zeichenbefehle in einem *QPixmap*-Objekt ausführen lassen. Für jede Zeichenoperation sind dabei nur ein paar Befehle nötig, um dem X-Server mitzuteilen, was er zeichnen soll.

Im Konstruktor eines *QPixmap*-Objekts können Sie die Breite, die Höhe und die Farbtiefe des Teilbildes angeben. Für die Farbtiefe sind allerdings nur zwei Werte erlaubt: Die Farbtiefe 1 (jedes Pixel wird mit einem Bit dargestellt und kann die Werte 0 oder 1 besitzen) oder die Farbtiefe des Grafikmodus des X-Servers. Je nach Grafikmodus kann das eine Farbtiefe von 8, 15, 16, 24 oder 32 Bit sein. Nur einer der Werte ist natürlich möglich, denn der Grafikmodus hat eine feste Farbtiefe. Wenn Sie als Wert für die Farbtiefe -1 benutzen – das ist auch der Default-Wert –, wird automatisch der aktuell benutzte Wert ausgewählt. Diesen Wert können Sie auch mit der statischen Methode *QPixmap::defaultDepth* ermitteln.

Die Daten eines *QPixmap*-Objekts können mit der globalen Funktion *bitBlt* in ein anderes *QPaintDevice*-Objekt kopiert werden, also zum Beispiel in ein Widget, in ein anderes *QPixmap*-Objekt oder auf ein *QPrinter*-Objekt, also den Drucker. Von

einem Widget lässt sich ebenfalls mit der *bitBlt*-Funktion ein Bereich in ein *QPixmap*-Objekt kopieren. Bereiche eines Widgets, die dabei von anderen Fenstern verdeckt sind, ergeben dabei beim Kopieren einen undefinierten Teil im *QPixmap*-Objekt.

Das folgende Programmstück zeichnet einen roten Kreis der Größe 20x20 Pixel in ein *QPixmap*-Objekt und kopiert dieses Bild fünfmal nebeneinander in das *QWidget*-Objekt *widget*:

```
QPixmap pix (20, 20);
QPainter p;
p.begin (&pix);    // QPainter-Objekt zeichnet in pix
p.setBrush (red);  // Füllfarbe Rot
p.setPen (NoPen);  // keine Randlinie
p.drawEllipse (0, 0, 20, 20); // Kreis zeichnen
p.end();

// fünf Kopien in widget hineinkopieren
for (int i = 0; i < 5; i++)
    bitBlt (widget, i * 20, 0, &pix);
```

Die Parameter der *bitBlt*-Funktion haben folgende Bedeutung: Der erste Parameter bezeichnet das *QPaintDevice*, in das die Daten geschrieben werden sollen, also das Ziel der Kopie. Die Stelle, an der die Daten eingefügt werden sollen, wird in den nächsten beiden Parametern angegeben. Der vierte Parameter gibt an, aus welcher Quelle die Daten kopiert werden.

Die *bitBlt*-Funktion besitzt noch weitere Parameter, die in unserem Beispiel jedoch mit ihren Default-Werten benutzt wurden: Die Parameter fünf bis acht – jeweils *int*-Werte – geben den Ausschnitt der Quelle an, der kopiert wird. Er ist durch die x- und y-Koordinate der oberen linken Ecke, die Breite und die Höhe definiert. Der neunte Parameter gibt die Art an, wie die Quelldaten mit den Zieldaten verknüpft werden sollen. Der Default-Wert *CopyROP* ist dabei der am häufigsten benutzte Wert. Die Daten des Ziels werden dabei einfach durch die Daten der Quelle überschrieben. Es gibt eine ganze Reihe weiterer Werte, die aber in der Praxis so gut wie keine Bedeutung haben. Der zehnte Parameter legt fest, ob eine Maske, die bei einem *QPixmap*-Objekt gesetzt werden kann, benutzt werden soll. Diese Maske kann beliebige Bereiche des *QPixmap*-Objekts als transparent definieren. Genauer zur Transparenz erfahren Sie in Kapitel 4.3.4, *Transparenz in Teilbildern*. Da wir hier keine Maske gesetzt hatten, ist der Wert dieses Parameters nicht relevant. Wir benutzen hier den Default-Wert *false*.

QPixmap wird oft benutzt, um eine aufwendige Zeichnung zunächst im Speicher auszuführen und dann in einem Schritt anzeigen zu lassen. So lässt sich ein kurzes Flimmern auf dem Bildschirm vermeiden. Mehr dazu erfahren Sie in Kapitel 4.5, *Flimmerfreie Darstellung*. Auf diese Weise kann aber auch der Inhalt eines

Widgets zwischengespeichert werden. Beim nächsten Aufruf von *paintEvent* kann dann der Inhalt des *QPixmap*-Objekts direkt wieder in das Widget kopiert werden.

Ebenso wie bei *QImage* kann man ein *QPixmap*-Objekt auch mit der *QPainter*-Methode *drawPixmap* in ein Widget oder ein anderes *QPainterDevice*-Objekt zeichnen lassen. In diesem Fall werden ebenfalls alle in *QPainter* definierten Transformationen auf das *QPixmap*-Bild angewandt. Diese Umrechnung ist aber sehr aufwendig. Da der X-Server keine Routinen für das Rotieren oder Vergrößern von *QPixmap*-Daten besitzt, müssen die Daten des *QPixmap*-Objekts zunächst vom X-Server in die Applikation kopiert, dort transformiert, wieder in den X-Server übertragen und dann ausgegeben werden. Dieses Vorgehen ist daher noch ineffizienter als die Ausgabe eines *QImage*-Objekts mit Transformationen. Sind im *QPainter*-Objekt keine Transformationen aktiviert, benutzt *drawPixmap* die Funktion *bitBlt*, ist also doch sehr effizient.

Für den Spezialfall einer *Pixmap* mit der Farbtiefe von einem Bit gibt es die Klasse *QBitmap*, die von *QPixmap* abgeleitet ist. Das Hauptanwendungsgebiet für die Klasse *QBitmap* ist die Festlegung einer Transparenzmaske für *QWidget*- oder *QPixmap*-Elemente. Da ein *QBitmap*-Objekt eine Farbtiefe von 1 besitzt, kann jedes Pixel den Wert 0 oder 1 annehmen. Diese Werte sind jedoch nicht einer Farbe zugeordnet. Stattdessen bedeutet der Wert 0, dass das *QPixmap*-Objekt bzw. das *QWidget*-Objekt an dieser Stelle durchsichtig ist. Ein Wert von 1 bedeutet, dass es undurchsichtig ist, dass also die Farbe dargestellt wird, die im *QPixmap*-Objekt oder im *QWidget*-Objekt angegeben ist. Sowohl für *QPixmap* als auch für *QWidget* wird eine Maske mit der Methode *setMask* festgelegt. Wichtig ist dabei, dass das *QBitmap*-Objekt, das als Maske dienen soll, die gleiche Breite und Höhe wie das *QPixmap*- bzw. *QWidget*-Objekt hat. Näheres über Transparenz erfahren Sie in Kapitel 4.3.4, *Transparenz in Teilbildern*.

Die Klasse *QPixmap* enthält zwei Methoden, um ein *QImage*-Objekt in ein *QPixmap*-Objekt umzuwandeln und umgekehrt. Die Methode *convertFromImage* überträgt die Daten eines *QImage*-Objekts in den X-Server und speichert sie im *QPixmap*-Objekt, für das diese Methode aufgerufen wurde. Der erste Parameter ist dabei das *QImage*-Objekt, das übertragen werden soll. Der zweite Parameter enthält Informationen über die Art der Umsetzung. Hier können Sie eine der folgenden Optionen oder-verknüpft einsetzen:

- *AutoColor*, *ColorOnly* oder *MonoOnly* – Wenn Sie *MonoOnly* wählen, werden die Bilddaten im *QImage*-Objekt in ein Schwarzweiß-Bild mit der Farbtiefe 1 umgewandelt. Für *ColorOnly* werden die Farben in den Bilddaten auf die vorhandenen Farben des Grafikmodus umgesetzt, den der X-Server nutzt. Im Modus *AutoColor* wird das Bild genau dann in ein Bild mit Farbtiefe 1 umgewandelt, wenn auch das *QImage*-Objekt die Farbtiefe 1 hatte, sonst in die Farbtiefe des Grafikmodus. *AutoColor* ist der Default-Modus.

- *DiffuseDither*, *OrderedDither* oder *ThresholdDither* – Mit einer dieser Einstellungen legen Sie fest, wie Zwischenfarben simuliert werden sollen, wenn die Farbtiefe des Grafikmodus geringer ist als die des *QImage*-Objekts. *ThresholdDither* ist die einfachste Variante. Hier wird jedem Pixel einfach die nächstliegende Farbe zugeordnet. Bei Farbverläufen kommt es dabei zu Sprüngen im Verlauf. Die beiden Werte *DiffuseDither* und *OrderedDither* versuchen, diese Sprünge dadurch zu mildern, dass nebeneinander liegende Pixel leicht unterschiedliche Farbwerte bekommen. Das menschliche Auge kann eng aneinander liegende Farben nicht mehr unterscheiden. Die Farbe erscheint dann wie eine Zwischenfarbe zwischen den einzelnen Pixelfarben. Auf diese Weise kann man also auch Zwischentöne erzeugen. *DiffuseDither* verteilt dabei den Farbfehler nach einer Berechnungsvorschrift, während *OrderedDither* ein regelmäßiges Muster benutzt. Die besten Ergebnisse erzielt man meist mit *DiffuseDither*, das auch der Default-Modus ist. Dieses Verfahren ist aber auch das langsamste.
- *DiffuseAlphaDither*, *OrderedAlphaDither* oder *ThresholdAlphaDither* – Mit einer dieser Einstellungen legen Sie fest, wie der Alpha-Kanal – also die Transparenzmaske – des *QImage*-Objekts in eine *QBitmap*-Maske für das entstehende *QPixmap*-Objekt umgesetzt werden soll. Während im Alpha-Kanal die Transparenz eines Pixels durch einen Wert von 0 bis 255 angegeben werden kann, kann die Maske des *QPixmap*-Objekts nur zwischen durchsichtig und undurchsichtig unterscheiden. Mit dem Wert *ThresholdAlphaDither* werden alle Pixel mit einem Transparenzwert größer als 127 als undurchsichtig dargestellt. Nur Pixel mit einem Transparenzwert kleiner oder gleich 127 werden im *QPixmap*-Objekt durchsichtig. *DiffuseAlphaDither* und *OrderedAlphaDither* versuchen, einen Mittelwert zwischen völlig transparent und völlig undurchsichtig durch eine Verteilung von durchsichtigen und undurchsichtigen Pixeln zu simulieren. Auch hier kann man zwischen einem verteilten oder einem geordneten Dither-Verfahren wählen. Der Default-Wert für die Dither-Strategie beim Alpha-Kanal ist *ThresholdAlphaDither*.
- *PreferDither* oder *AvoidDither* – Wenn Sie *AvoidDither* wählen, testet die Methode vor der Konvertierung, wie viele verschiedene Farben im *QImage*-Objekt enthalten sind. Soll ein Bild in einen Grafikmodus mit maximal 256 Farben umgesetzt werden und hat das Originalbild mehr als 256 verschiedene Farben, wird das Bild mit der eingestellten Dither-Strategie angepasst. Hat es weniger Farben, versucht die Methode zunächst, alle benötigten Farben direkt zu erhalten. Wählen Sie dagegen die Einstellung *PreferDither*, so wird nicht nachgezählt, wie viele verschiedene Farben das *QImage*-Objekt hat, sondern es wird in jedem Fall die Dither-Strategie angewendet.

Der Rückgabewert der Methode *convertFromImage* ist vom Typ *bool*. Ist er *true*, so war die Konvertierung erfolgreich. Ist er dagegen *false*, so war im X-Server nicht genügend Speicher frei, um das *QPixmap*-Objekt anzulegen.

Die Methode *convertToImage* der Klasse *QPixmap* übernimmt die Umsetzung in die andere Richtung. Das im *QPixmap*-Objekt enthaltene Bild wird in ein *QImage*-Objekt umgewandelt und über den Rückgabewert der Methode an den Aufrufer geliefert. Hierbei sind keine weiteren Parameter nötig. Die Farbtiefe des erzeugten *QImage*-Objekts ist 1, wenn auch das *QPixmap*-Objekt eine Farbtiefe von 1 hat. Sie ist 8, wenn das *QPixmap*-Objekt eine Farbtiefe zwischen zwei und acht Bit pro Pixel hat. Hat das *QPixmap*-Objekt eine Farbtiefe größer als 8, so wird ein *QImage*-Objekt mit einer Farbtiefe von 24 Bit erzeugt. Da also das erzeugte *QImage*-Objekt immer mindestens ebenso viele Farben darstellen kann wie das ursprüngliche *QPixmap*-Objekt, geht keine Information verloren. Die Umwandlung ist deutlich schneller als bei *convertFromImage*, da die Umwandlung der Farbwerte hier trivial ist. Ein Farbanpassung durch Dithering ist nicht nötig.

QPixmap enthält – genau wie *QImage* – die beiden Methoden *save* und *load*, um eine Grafikdatei in das *QPixmap*-Objekt einzulesen bzw. das Bild des Objekts in einer Datei zu speichern. Intern wird dabei zuerst ein *QImage*-Objekt eingelesen und dann mit *convertFromImage* umgewandelt bzw. ein *QImage*-Objekt mit *convertFromImage* erstellt und dann gespeichert.

4.3.3 Effizienzbetrachtungen

Aufgrund ihrer unterschiedlichen Konzepte haben *QImage* und *QPixmap* unterschiedliche Aufgabengebiete:

QImage ist besonders dann effizient, wenn Berechnungen auf die Bilddaten angewendet werden sollen (wenn zum Beispiel die Helligkeit oder der Kontrast verändert oder ein Filter angewendet werden soll) oder wenn die Bilddaten geladen oder gespeichert werden. *QImage* kann die Bilder in TrueColor speichern und bearbeiten, auch wenn der Grafikmodus das nicht unterstützt. Es kann die Transparenz in weichen Stufen benutzen (siehe Kapitel 4.3.4, *Transparenz in Teilbildern*). Diese weiche Transparenz lässt sich allerdings nicht unmittelbar anzeigen.

QPixmap ist besonders effizient, wenn Bilddaten dargestellt werden sollen, wenn sie kopiert oder verschoben werden sollen und wenn Zeichnungen von Linien- und Flächenprimitiven ausgeführt werden sollen. *QPixmap* wird auch oft als Hilfsspeicher benutzt, um ein Flimmern beim Zeichnen eines Widgets zu vermeiden (siehe Kapitel 4.5, *Flimmerfreie Darstellung*). *QPixmap* hat immer die Farbtiefe und die Farbtabelle des Grafikmodus. Bei der Darstellung sind also keine Umrechnungen nötig. *QPixmap* kann ebenfalls transparente Pixel enthalten, indem eine Maske definiert wird. Ein Pixel kann dann aber nur vollständig durchsichtig oder vollständig undurchsichtig sein.

Aufwendig sind vor allem Konvertierungen von einer Klasse in die andere. Die Bilddaten müssen in jedem Fall von der Applikation zum X-Server bzw. vom X-Server zur Applikation übertragen werden. Wenn die Verbindung zum X-Server über eine langsame Internet-Verbindung läuft, ist die Übertragung eines großen Bildes extrem aufwendig. Diese Konvertierungen sollten also nach Möglichkeit so selten wie möglich benutzt werden.

Zusätzlich zur Übertragung der Daten müssen bei der Konvertierung von *QImage* nach *QPixmap* auch noch geeignete Farben für die Darstellung gewählt werden. Das ist relativ unproblematisch, wenn der X-Server einen TrueColor-Grafikmodus benutzt, da dann jeder RGB-Wert einfach umgerechnet werden kann. Benutzt der X-Server dagegen einen Grafikmodus mit nur 256 Farben aus einer Farbpalette, kann die Umrechnung aufwendiger sein. Qt benutzt hier einige beschleunigende Techniken (siehe auch Kapitel 4.1, *Farben unter Qt*). In diesem Fall gilt, dass ein *QImage*-Objekt mit der Farbtiefe 8 schneller konvertiert werden kann als eines mit der Farbtiefe 32. Für eine schnelle Konvertierung sollten Sie also die Farbtiefe 8 wählen. Das bedeutet für Sie in der Regel aber deutlich mehr Aufwand, um eine geeignete Farbtabelle zu bestimmen.

Sie können auch die Konvertierungseinstellungen wählen, wenn Sie eine besonders schnelle Konvertierung benötigen. Wählen Sie *ThresholdDither* | *ThresholdAlphaDither* | *AvoidDither*, wenn Sie eine besonders schnelle Konvertierung benötigen, für die Sie auch Abstriche an der Qualität in Kauf nehmen. Für eine gute Qualität bei aufwendigerer Berechnung benutzen Sie *DiffuseDither* | *DiffuseAlphaDither* | *PreferDither*.

In vielen Fällen kommen Sie um eine Konvertierung von *QImage* nach *QPixmap* nicht herum, wenn ein berechnetes Bild angezeigt werden soll. Es stellt sich nun die Frage, ob das Bild als *QImage* erhalten bleiben soll oder ob man es einmal in ein *QPixmap*-Objekt umwandelt und das *QImage*-Objekt löscht. In den meisten Fällen ist die zweite Variante die bessere: Da ein dargestelltes Bild oft wiederhergestellt werden muss – zum Beispiel wenn es verdeckt war und anschließend aufgedeckt wird –, kann ein *QPixmap*-Objekt diese Zeichenoperation mit Hilfe der Funktion *bitBlt* sehr schnell ausführen. Eine Ausnahme von dieser Faustregel bilden sehr große *QImage*-Bilder, von denen immer nur ein kleiner Ausschnitt auf dem Bildschirm dargestellt werden soll – zum Beispiel eine Landkarte, bei der man den angezeigten Teil mit einem Rollbalken verschieben kann. Da der Speicherplatz im X-Server begrenzt sein kann, sollten keine allzu großen *QPixmap*-Bilder angelegt werden. Auch die Übertragung des ganzen *QImage*-Bildes kann sehr viel Zeit in Anspruch nehmen. Günstiger ist es daher, in diesem Spezialfall das *QImage*-Bild beizubehalten und für jede Zeichenoperation die Methode *QPainter::drawImage* zu benutzen, in der man den Ausschnitt angeben kann, der gezeichnet werden soll. Hier ist auch ein Kompromiss möglich: Sie können ein *QPixmap*-Objekt erzeugen, das die Größe des anzuzeigenden Bereichs hat. In die-

ses Objekt zeichnen Sie immer den Bereich, der gerade angezeigt werden soll. Wenn der Bildschirminhalt wiederhergestellt werden soll, kann einfach das *QPixmap*-Objekt benutzt werden. Soll der Inhalt verschoben werden, muss zunächst das *QPixmap*-Bild mit einem anderen Ausschnitt des *QImage*-Objekts gefüllt werden. (Um einen Ausschnitt des *QImage*-Bildes in das *QPixmap*-Objekt zu kopieren, können Sie eine überladene Variante der Funktion *bitBlt* benutzen, die als Quelle ein *QImage*-Objekt und als Ziel ein *QPixmap*-Objekt erlaubt. Die ersten acht Parameter haben die gleiche Bedeutung wie in der Funktion *bitBlt* zwischen *QPaintDevice*-Objekten; der neunte Parameter enthält die Angaben zur Konvertierung, wie sie in *QPixmap::convertFromImage* benutzt werden.)

Eine weitere Beschleunigung bei der Konvertierung von *QPixmap* in *QImage* und umgekehrt kann die Klasse *KImageIO* bieten, die in den KDE-Bibliotheken definiert ist. Falls das eigene Programm und der X-Server auf dem gleichen Rechner laufen – und das trifft auf 90% der Unix-Systeme zu –, wird dabei zur Übertragung der Daten Shared Memory benutzt, also ein Speicherbereich im Hauptspeicher des Rechners, den Programm und X-Server gemeinsam nutzen. Diese Übertragung ist unter Umständen deutlich schneller als die herkömmliche Übertragung über eine Socket-Verbindung. Ob Shared Memory aber auch tatsächlich benutzt wird, hängt davon ab, ob Programm und X-Server auch tatsächlich auf dem gleichen Rechner laufen und das Unix-System sowie der X-Server Shared Memory unterstützen. Ist das nicht der Fall, greift *KImageIO* auf die Socket-Verbindung zurück. Ein kurzes Programmstück soll erläutern, wie *KImageIO* benutzt werden kann:

```
// Ein Objekt der Klasse KImageIO anlegen
// Das können Sie beispielsweise in der main-Funktion
// oder im Konstruktor Ihrer Hauptklasse erledigen
// und dieses Objekt immer wieder benutzen.
KPixmapIO *kpix = new KPixmapIO ();

// Hier wird ein QImage-Bild angelegt und mit
// Daten gefüllt
QImage image;
....

// Konvertieren...
QPixmap p = kpix->convertToPixmap (image);

// ... und anzeigen
bitBlt (widget, 0, 0, p);
```

Für den Einsatz der Klasse *KImageIO* müssen Sie ein *KApplication*-Objekt erzeugt haben. Ein *QApplication*-Objekt reicht nicht aus, da *KImageIO* auf einige Methoden von *KApplication* zurückgreift.

Auf der CD, die dem Buch beiliegt, ist ein Beispielprogramm enthalten, das den Effekt demonstriert. Im Beispiel wird ein Bild langsam eingeblendet und wieder ausgeblendet. (Dazu wird die Helligkeit aller Pixel gleichmäßig verringert, um dunklere Bilder zu erhalten.) Dazu werden 16 Zwischenbilder vom Typ *QImage* berechnet, die dann der Reihe nach auf den Bildschirm kopiert werden. Dazu muss ein Zwischenbild natürlich in ein *QPixmap*-Objekt umgewandelt werden. Sie können im Beispielprogramm wählen, ob diese Umwandlung über das normale *QPixmap::convertFromImage* oder über *KImageIO::convertToPixmap* erfolgen soll. Falls auf Ihrem System Shared Memory genutzt wird, wird ein deutlicher Geschwindigkeitsunterschied sichtbar.

4.3.4 Transparenz in Teilbildern

Sowohl die Klasse *QPixmap* als auch die Klasse *QImage* bietet die Möglichkeit, einzelne Pixel des Bildes transparent darzustellen. Wird dieses Bild auf den Bildschirm gezeichnet, bleiben an den Stellen, an denen transparente Pixel im Bild stehen, die ursprünglichen Pixel des Bildschirms erhalten.

In der Klasse *QPixmap* wird die Transparenz durch ein Objekt der Klasse *QBitmap* (also eine zusätzliche Pixmap mit Farbtiefe 1) festgelegt. Dazu benutzen Sie die Methode *QPixmap::setMask*. Das *QBitmap*-Objekt muss die gleiche Breite und Höhe wie das *QPixmap*-Objekt haben, für das Sie die Transparenz definieren wollen. Im *QBitmap*-Objekt können Sie Zeichenoperationen mit der Klasse *QPainter* ausführen. Benutzen Sie dazu die Zeichenfarben *color0* und *color1*. Mit diesen Farben setzen Sie die Bits auf den Wert 0 bzw. 1. Ein 0-Pixel in der Maske bedeutet dabei, dass das zugehörige Pixel im *QPixmap*-Objekt durchsichtig ist, ein 1-Pixel in der Maske bedeutet, das zugehörige Pixel ist nicht durchsichtig.

Solange Sie keine Maske gesetzt haben, sind alle Pixel des *QPixmap*-Objekts undurchsichtig. Ein *QPixmap*-Objekt ohne Maske kann meist sehr viel schneller gezeichnet und bearbeitet werden. Sie sollten daher nur dann eine Maske benutzen, wenn es nötig ist.

Das folgende Programm erzeugt ein *QPixmap*-Objekt mit drei waagerechten roten Linien auf weißem Grund. Anschließend wird eine Maske gesetzt, bei der nur eine Kreislinie der Dicke 5 als undurchsichtig gesetzt ist, alle anderen Pixel sind durchsichtig. Das Ergebnis dieser Maske sehen Sie in Abbildung 4.45.

```
QPainter p;  
  
// QPixmap mit drei waagerechten roten Linien  
QPixmap pix (50, 50);  
pix.fill (white);  
p.begin (&pix);  
p.setPen (QPen (red, 3));  
p.drawLine (0, 15, 50, 15);
```

```
p.drawLine (0, 25, 50, 25);  
p.drawLine (0, 35, 50, 35);  
p.end ();  
  
// Maske mit einem Kreis in color1, Rest color0  
QBitmap mask (50, 50);  
mask.fill (color0);  
p.begin (&mask);  
p.setPen (QPen (color1, 5));  
p.setBrush (NoBrush);  
p.drawEllipse (0, 0, 50, 50);  
p.end ();  
  
pix.setMask (mask);
```

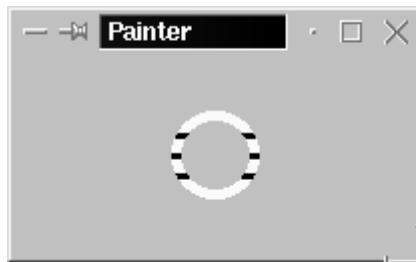


Abbildung 4-45 Ergebnis einer kreisförmigen Maske mit `setMask`

Beachten Sie, dass die Maske fertig gezeichnet sein muss, bevor `setMask` aufgerufen wird. Eine nachträgliche Änderung ist nur möglich, indem man die Maske durch eine neue ersetzt. Insbesondere müssen Sie `QPainter::end` aufrufen, bevor Sie die Maske eintragen, um zu gewährleisten, dass alle Zeichnungen in die Maske ausgeführt wurden. Ansonsten kann es passieren, dass einige Zeichenoperationen so verzögert ausgeführt werden, dass sie nicht mehr in der Maske des `QPixmap`-Objekts eingetragen werden.

Die Klasse `QWidget` enthält übrigens ebenfalls die Methode `setMask`. Sie wird genauso benutzt wie bei `QPixmap`. Alle Pixel in der Maske, die den Farbwert `color0` besitzen, erscheinen im Fenster durchsichtig. An dieser Stelle sieht man also das Pixel des darunter liegenden Fensters bzw. des Hintergrundbildes. Auf diese Weise kann man beispielsweise Fenster realisieren, die nicht rechteckig sind. In der Regel benutzen Sie für solche Fenster im Konstruktor als Parameter `WFlags` den Wert `WStyle_Customize | WStyle_NoBorder`, damit der Window-Manager keine Dekoration (also keinen Rahmen und keine Titelzeile) am Fenster zeichnet. Diese Darstellung von durchsichtigen Fenstern ist extrem aufwendig, insbesondere wenn ein Fenster mit gesetzter Maske verschoben werden soll. Verwenden Sie sie also wirklich nur an sinnvollen Stellen.

Die Klasse *QPixmap* hat eine Methode *createHeuristicMask*, die eine Maske für das zur Zeit in *QPixmap* gespeicherte Bild generiert. Dazu wird die Farbe einer Ecke ermittelt, und dann werden ausgehend von den vier Ecken alle Pixel als transparent definiert, die diese Farbe besitzen. Wenn *QPixmap* ein Objekt auf einem einfarbigen Hintergrund enthält, erzeugt diese Methode in den meisten Fällen eine gute Maske, bei der nur das Objekt undurchsichtig ist, während der einfarbige Hintergrund transparent ist. Die Berechnung dieser Maske ist jedoch sehr aufwendig: Zuerst muss das *QPixmap*-Objekt in ein *QImage*-Objekt umgewandelt werden, um die einzelnen Farbwerte auslesen zu können. Anschließend wird die Maske mit zum Teil aufwendigen Algorithmen aus dem *QImage*-Objekt generiert. (Dazu wird die Methode *QImage::createHeuristicMask* benutzt.) Diese Maske wird dem *QPixmap*-Objekt zugewiesen. Diese Methode sollte nur auf kleine *QPixmap*-Bilder angewandt werden, zum Beispiel auf Icons. Sie funktioniert nur bei einfarbigen Hintergründen gut. Bei Farbverläufen, wie sie bei vielen Icons vorkommen, ist die generierte Maske meist unbrauchbar.

Viele Dateiformate für Bilder können ebenfalls einzelne Pixel in den Bildern als transparent definieren, so zum Beispiel die Grafikformate GIF, PNG und XPM. Wenn Sie solche Bilder aus einer Datei in ein *QPixmap*-Objekt einlesen (indem Sie den Dateinamen im Konstruktor von *QPixmap* angeben oder die Methode *QPixmap::load* benutzen), wird automatisch die Maske für das Bild erzeugt und gesetzt. Nicht alle Grafikprogramme können allerdings Bilder mit transparenten Pixeln erzeugen und abspeichern. Sie sollten also vorher testen, ob Ihr Grafikprogramm dazu in der Lage ist. Insbesondere für Icons ist es dann sehr sinnvoll, die durchsichtigen Bereiche schon beim Zeichnen des Icons festzulegen und mit abzuspeichern.

Die Klasse *QImage* bietet ebenfalls die Möglichkeit, die Transparenz jedes Pixels anzugeben. Dazu benutzt sie die verbleibenden acht Bit, den so genannten Alpha-Kanal, die bei einer Farbtiefe von 32 Bit nicht von den Farbanteilen Rot, Grün und Blau benutzt werden. (Auch bei einer Farbtiefe von acht Bit können Sie Transparenz benutzen. Tragen Sie dazu in die Farbtabelle Farben mit Transparenzwerten ein.) In diesen acht Bit werden Werte von 0 bis 255 gespeichert, die einen weichen Übergang von völlig durchsichtig (Wert 0) bis völlig undurchsichtig (Wert 255) ermöglichen.

Standardmäßig wird der Alpha-Kanal eines *QImage*-Objekts nicht genutzt. Die Transparenzinformation in den freien acht Bit wird ignoriert, und alle Pixel gelten als nicht durchsichtig. Um den Alpha-Kanal zu nutzen, rufen Sie die Methode *setAlphaBuffer (true)* auf. Beachten Sie aber, dass der Inhalt des Alpha-Kanals nicht initialisiert ist. Die Transparenzwerte der Pixel sind also zunächst undefiniert und müssen von Ihnen explizit gesetzt werden.

Um einem Pixel bzw. einem Eintrag in der Farbtabelle eine Farbe und einen Transparenzwert zuzuweisen, erzeugen Sie eine *QRgb*-Struktur mit der globalen

Funktion *qRgba (int red, int green, int blue, int alpha)*. Diese Struktur können Sie zum Beispiel mit *setPixel* in ein Pixel eintragen (für Farbtiefe 32) oder mit *setColor* einem Eintrag der Farbtabelle zuweisen (für Farbtiefe 8).

Für die Anzeige eines *QImage*-Objekts mit Alpha-Kanal muss das *QImage*-Objekt mit der Methode *QPixmap::convertFromImage* in ein *QPixmap*-Objekt umgewandelt werden. Das übernimmt die Methode *QPainter::drawImage* automatisch. Da aber die Maske eines *QPixmap*-Objekts keine halb durchsichtigen Pixel erzeugen kann, wird hier die Maske in der Regel mit *ThresholdAlphaDither* erzeugt. Somit werden alle Pixel durchsichtig dargestellt, die einen Alpha-Wert größer als 127 besitzen; alle anderen sind undurchsichtig. Das Ergebnis bei weichen Transparenzen ist daher sehr schlecht. Das Ergebnis sehen Sie im oberen Teil von Abbildung 4.46 am Ende dieses Abschnitts.

Um bessere Ergebnisse zu erzielen, können Sie das *QImage*-Objekt selbst in ein *QPixmap*-Objekt umwandeln. Dazu geben Sie in der Methode *QPixmap::convertFromImage* die Strategie *DiffuseAlphaDither* an. In diesem Fall werden für Alpha-Werte zwischen 0 und 255 mehr oder weniger Pixel in der Umgebung als durchsichtig oder undurchsichtig dargestellt. Haben beispielsweise alle Pixel den Alpha-Wert 128, so trägt genau jedes zweite Pixel in der Maske des generierten *QPixmap*-Objekts den Wert 0. Durch diese Rasterung wird zumindest ansatzweise eine Halbtransparenz ermöglicht. Das Ergebnis einer solchen Umwandlung sehen Sie in Abbildung 4.46 in der Mitte.

Das folgende Programm erzeugt ein *QImage*-Objekt, in dem alle Pixel schwarz sind. Die Transparenz der Pixel nimmt aber von links nach rechts zu, so dass die Pixel am linken Rand undurchsichtig sind, die Pixel am rechten Rand völlig durchsichtig. Dieses Bild wird zum einen durch die Methode *QPainter::drawImage* dargestellt, zum anderen, indem es von Hand in ein *QPixmap*-Objekt umgewandelt und dann mit *QPainter::drawPixmap* dargestellt wird. Die zweite Möglichkeit erzeugt ein deutlich besseres Ergebnis als die erste.

```
QImage img (100, 20, 32);
// Alpha-Kanal aktivieren
img.setAlphaBuffer (true);
for (int y = 0; y < 20; y++)
    for (int x = 0; x < 100; x++)
    {
        // Pixel in Schwarz mit sinkendem Wert für Alpha
        // setzen, also mit immer größerer Transparenz
        QRgb color = qRgba (0, 0, 0, 255 - x * 256 / 100);
        img.setPixel (x, y, color);
    }

// Zeichnen in widget mit drawImage
QPainter p;
```

```
p.begin (widget);
p.drawImage (20, 10, img);

QPixmap pix;
pix.convertFromImage (img, DiffuseAlphaDither);
p.drawPixmap (20, 40, pix);
p.end();
```

Wenn Sie die Transparenz wirklich so exakt wie möglich wiedergeben wollen, müssen Sie die Pixelfarben, die sich beim Schreiben eines halb transparenten Pixels auf den Untergrund ergeben, selbst errechnen. Diese Berechnung führen Sie am besten innerhalb von zwei *QImage*-Objekten durch. Den Transparenzwert in der Farbe eines Pixels kann man mit der globalen Funktion *qAlpha* aus der *QRgb*-Struktur ermitteln.

Die folgende Funktion *drawImageTransparent* zeichnet das *QImage*-Objekt *source* in das *QImage*-Objekt *dest* ein – unter voller Berücksichtigung der Transparenz im Alpha-Kanal von *source*. Dabei wird die obere linke Ecke von *source* an die Position (*x*/*y*) verschoben. *dest* muss dabei eine Farbtiefe von 32 besitzen. Durch die Halbtransparenz können nämlich neue Farben entstehen, die vorher noch nicht im Bild enthalten waren. Bei einer Farbtiefe von 8 wären diese Farben mit großer Wahrscheinlichkeit nicht in der Farbtabelle enthalten. *dest* darf keinen eigenen Alpha-Kanal besitzen.

```
bool drawImageTransparent (QImage &dest, int rx, int ry,
                          const QImage &source)
{
    // Nur möglich, falls dest die Farbtiefe 32 hat
    if (dest.depth() != 32 || dest.hasAlphaBuffer())
        return false;
    // Für jedes Pixel der Quelle
    for (y = 0; y < source.height(); y++)
        for (int x = 0; x < source.width(); x++)
            // Falls Pixel innerhalb von dest liegt
            if (dest.valid (x + rx, y + ry))
            {
                QRgb d = dest.pixel (x + rx, y + ry);
                QRgb s = source.pixel (x, y);
                int a = qAlpha (s);
                QRgb resultColor =
                    qRgb ((qRed (s)*a + qRed (d)*(255-a))/255,
                        (qGreen(s)*a + qGreen(d)*(255-a))/255,
                        (qBlue (s)*a + qBlue (d)*(255-a))/255);
                dest.setPixel (x + rx, y + ry, resultColor);
            }
    return true;
}
```

Die folgende Funktion *drawImageToPainter* zeichnet (unter Benutzung von *drawImageTransparent*) ein *QImage*-Objekt in ein *QWidget*- oder *QPixmap*-Objekt. Dabei wird der gesamte Wertebereich des Transparenzwerts ausgenutzt. Der Funktion werden das *QImage*-Objekt *source*, ein *QPaintDevice*-Objekt, *dest*, das das gewünschte Ziel angibt, und die Position (*x/y*) übergeben, an der das Bild gezeichnet werden soll. Die Funktion holt zunächst den aktuellen Inhalt an der Stelle, an der das Bild nachher liegen soll, in ein *QPixmap*-Objekt. Dieses *QPixmap*-Objekt wird in ein *QImage*-Objekt umgewandelt, in das dann mit der Funktion *drawImageTransparent* das andere *QImage*-Objekt hineingezeichnet wird. Das Ergebnis wird dann wieder in das *QPaintDevice*-Objekt *dest* zurückgeschrieben. Da der Inhalt von *dest* ausgelesen wird, kann *dest* nur ein *QPixmap*- oder ein *QWidget*-Objekt sein. *QPainter* oder *QPicture* sind nicht erlaubt, da sie nicht ausgelesen werden können.

Das Ergebnis dieser selbst geschriebenen Funktion sehen Sie in Abbildung 4.46 unten.

```
bool drawImageToPainter (const QImage &source,
                        QPaintDevice *dest,
                        int x, int y)
{
    // Falls dest nicht QWidget oder QPixmap,
    // kann der momentane Inhalt nicht ausgelesen werden
    // => Fehler
    if (dest->isExtDev())
        return false;

    int w = source.width();
    int h = source.height();
    QPixmap pix (w, h);
    // Hintergrund in das QPixmap-Objekt kopieren
    bitBlt (&pix, 0, 0, dest, x, y, w, h);

    // in QImage umwandeln
    QImage destImg = pix.convertToImage();
    // evtl. auf Farbtiefe 32 umwandeln
    if (destImg.depth() != 32)
        destImg = destImg.convertDepth(32);

    // Beide QImage-Objekte verknüpfen
    drawImageTransparent (destImg, 0, 0, source);

    // Ergebnis wieder in QPixmap umwandeln
    pix.convertFromImage (destImg);
    // und wieder zeichnen
    bitBlt (dest, x, y, &pix);
    return true;
}
```



Abbildung 4-46 *drawImage*, *convertFromImage* und *drawImageToPainter*

4.4 Entwurf eigener Widget-Klassen

Wenn Sie ein grundlegend neues Bedienelement entwerfen wollen, so gehen Sie dabei am besten folgendermaßen vor:

1. Entwerfen Sie die Deklaration von geeigneten Signalen für die Klasse Ihres Widgets, mit denen Ihr Widget Zustandsänderungen melden soll.
2. Entwerfen Sie die Deklaration von Methoden, mit denen der Zustand des Widgets durch das Programm gesetzt werden kann. Überlegen Sie, welche dieser Methoden sinnvollerweise als Slots deklariert werden sollten (üblicherweise parameterlose Methoden wie *clear* oder *reset*).
3. Überlegen Sie, welche Klasse Sie als Basisklasse benutzen. Oftmals existieren bereits Klassen, die eine Teilfunktion Ihres Problems implementieren. Beispielsweise kommt *QPushButton* für alles in Frage, was zwei Zustände besitzen soll, *QTableView* für Widgets mit tabellenartigem Aufbau (auch mit nur einer Zeile oder Spalte) und verschiebbarem Inhalt, *QScrollView* für ein Fenster, das einen verschiebbaren Ausschnitt einer größeren Darstellung anzeigen soll, sowie *QMultiLineEdit* für mehrzeilige Texte, die man markieren kann. Gibt es kein solches Widget, so benutzen Sie *QWidget* als Basisklasse.
4. Überschreiben Sie alle Event-Methoden, die für die Funktionalität von Bedeutung sind. Diese Methoden sorgen für die Interaktivität Ihres Widgets. Mit diesen Methoden reagieren Sie beispielsweise auf die Maus oder die Tastatur, insbesondere aber auch auf die Anforderung des X-Servers, das Fenster oder Teile davon neu zu zeichnen. Die wichtigsten Event-Methoden werden in Kapitel 4.4.1, *Event-Methoden*, beschrieben.
5. Überschreiben Sie die Methoden *QWidget::sizeHint* und *QWidget::sizePolicy*, so dass das Widget problemlos in ein Layout eingefügt werden kann. *sizeHint* sollte dabei eine gute Standardgröße für das Widget angeben. Das kann eventuell vom Inhalt abhängig sein, der dargestellt werden soll. *sizePolicy* legt fest, wie sich das Widget verhalten soll, wenn die Größe des übergeordneten Fensters verändert wird. Nähere Informationen hierzu finden Sie in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*.

4.4.1 Event-Methoden

Über die Event-Methoden wird einem Objekt der Klasse *QObject* (oder einer abgeleiteten Klasse) mitgeteilt, dass eine Veränderung stattgefunden hat, auf die das Objekt eventuell reagieren soll. Die Event-Methoden werden durch ihre Ursache oder Bedeutung bezeichnet, gefolgt vom Wort »Event«. Beispiele sind *paintEvent*, *mousePressEvent*, *timerEvent*. Die meisten Event-Methoden sind erst ab der Klasse *QWidget* definiert. *QObject* selbst enthält nur die beiden Event-Methoden *timerEvent* und *childEvent*.

Allen Event-Methoden wird ein Zeiger auf ein Event-Objekt übergeben, in dem wichtige Informationen enthalten sind, was den Event ausgelöst hat. Jede Event-Methode hat eine Event-Klasse, die die spezifischen Informationen speichert. So übergibt man beispielsweise an die *keyPressEvent*- und *keyReleaseEvent*-Methoden ein Objekt der Klasse *QKeyEvent*; die Methoden *mousePressEvent*, *mouseMoveEvent*, *mouseReleaseEvent* und *mouseDoubleClickEvent* erhalten ein Objekt der Klasse *QMouseEvent*; die Methode *timerEvent* bekommt ein Objekt der Klasse *QTimerEvent* und so weiter. Alle Event-Klassen sind von der Klasse *QEvent* abgeleitet. Diese Basisklasse enthält eine Variable vom Typ *int*, in der die Art des Events abgespeichert wird, so wie die Methode *type*, die genau den Wert dieser Variablen zurückerliefert. Die Methode *QObject::event* erhält alle Events, die an das Objekt abgesendet wurden. Als Parameter erhält sie einen Zeiger auf das Objekt mit den Event-Informationen von der Klasse *QEvent*. Sie fragt mit der Methode *QEvent::type* ab, um was für einen Event es sich handelt, und ruft je nach Event eine der Event-Methoden auf. Dabei führt *event* einen Type-Cast auf die entsprechende Unterklasse des Event-Objekts aus. So kann innerhalb der Event-Methode auf alle Informationen zugegriffen werden.

Wenn Sie eine eigene Widget-Klasse entwerfen wollen, müssen Sie meist einige der folgenden Methoden in Ihrer Klasse überschreiben, um die Funktionalität Ihres Widgets zu implementieren. Oft reicht es aus, die Methode *paintEvent* sowie die Methoden für die Maus-Events und die Tastatur-Events zu überschreiben. Für Spezial-Widgets können auch einige der anderen Methoden interessant sein.

paintEvent

Dieser Event ist für das Zeichnen des Widgets zuständig. Er wird immer aufgerufen, wenn das Widget oder ein Teil davon neu gezeichnet werden muss. Das ist zum Beispiel unmittelbar nach einem Aufruf von *show* der Fall, aber auch, wenn ein Teil des Widgets von einem anderen Fenster verdeckt war und nun aufgedeckt wird.

In dieser Methode implementieren Sie normalerweise, wie der Inhalt Ihres Widgets gezeichnet werden soll. Daher muss diese Methode in der Regel überschrieben werden.

Sie können davon ausgehen, dass der zu zeichnende Bereich vom X-Server bereits mit der Hintergrundfarbe bzw. dem Hintergrundbild gefüllt worden ist. In der *paintEvent*-Methode erzeugen Sie in der Regel ein *QPainter*-Objekt für das Widget, mit dem Sie die Zeichenbefehle ausführen lassen, die das Widget mit Inhalt füllen. Die Zeichnungen wirken sich dabei automatisch nur auf den Bereich aus, der neu gezeichnet werden muss. Der Rest des Widgets bleibt unverändert.

Eine übliche Implementierung der *paintEvent*-Methode sieht so aus:

```
void MyWidget::paintEvent (QPaintEvent *) {
    QPainter p (this);
    ...
    // Zeichenroutine, basierend auf p
    ...
}
```

Wie Ihr Widget gezeichnet wird, hängt von mehreren Faktoren ab. Zum einen bestimmen natürlich die Daten, die Ihr Widget anzeigen soll, was dargestellt wird. Wenn Ihr Widget den Tastaturfokus tragen kann, so sollte der Anwender bereits aus der Darstellung ablesen können, ob das Widget den Fokus trägt oder nicht. Eingabeelemente zeichnen zum Beispiel einen Text-Cursor. Schaltflächen stellen meist eine gestrichelte Linie dar. Weiterhin sollte man an der Darstellung des Widgets erkennen, ob es mit der Methode *setEnabled (false)* deaktiviert worden ist. In diesem Fall sollte das Widget kontrastärmer und grau gezeichnet werden.

Benutzen Sie beim Zeichnen möglichst keine festen Farben, sondern versuchen Sie so weit wie möglich, mit den Farben aus der Widget-Palette auszukommen. Die Methode *QWidget::colorGroup* liefert Ihnen das zur Zeit aktive *QColorGroup*-Objekt aus der Widget-Palette zurück. Die Widget-Palette enthält drei Farbgruppen: *active*, *inactive* und *disabled*. *colorGroup* wählt automatisch die Gruppe, die dem aktuellen Zustand entspricht (siehe auch Kapitel 4.1.8, *Die Widget-Farbpalette*).

Die Darstellung kann weiterhin vom eingestellten Style abhängen. Wenn Sie ein Widget implementieren, das dem Anwender schon von anderen Programmen her bekannt sein dürfte, müssen Sie darauf achten, ob das Widget zum Beispiel in Motif anders dargestellt wird als in Microsoft Windows. Die Methode *QWidget::style* liefert ein Objekt der Klasse *QStyle* zurück. Diese Klasse enthält bereits eine Reihe von Methoden, mit denen typische Zeichenoperationen vorgenommen werden können, die sich in den verschiedenen Styles unterscheiden. Benutzen Sie möglichst diese Methoden, um Ihr Widget automatisch auch für neue Styles so darstellen zu lassen, dass es ins Gesamtbild passt. Weitere Informationen erhalten Sie in der Qt-Online-Dokumentation zur Klasse *QStyle*.

Es kann bei aufwendigen Zeichnungen sinnvoll sein, die vollständige Zeichnung zunächst in ein *QPixmap*-Objekt zu schreiben und anschließend den Inhalt des *QPixmap*-Objekts in das Widget zu kopieren. So vermeidet man ein kurzes Flimmern auf dem Bildschirm. Nähere Informationen dazu erhalten Sie in Kapitel 4.5, *Flimmerfreie Darstellung*.

In den meisten Fällen ist es ausreichend, in *paintEvent* das ganze Widget neu zu zeichnen. Alle Zeichenoperationen, die nicht den neu zu zeichnenden Bereich betreffen, werden automatisch vom X-Server unterdrückt. Bei sehr aufwendigen Zeichenoperationen kann es aber sinnvoll sein, diese Zeichenbefehle bereits vorher zu unterdrücken, um die Datenübertragung zum X-Server gering zu halten. Für einfache Widgets lohnt sich der Aufwand jedoch meist nicht. Mit der Methode *QPaintEvent::region* können Sie den neu zu zeichnenden Bereich erfragen. Sie erhalten als Rückgabewert ein *QRegion*-Objekt, das den exakten Bereich angibt. Mit *QRegion::contains* können Sie nun beispielsweise prüfen, ob Ihr Objekt in diesem Bereich liegt. Im folgenden Beispiel wird der Zeichenbefehl für das Rechteck nur dann ausgeführt, wenn es im neu zu zeichnenden Bereich ist.

```
void MyWidget::paintEvent (QPaintEvent *ev)
{
    QPainter p (this);

    QRect r (20, 20, 160, 60);
    // Testen, ob im neu zu zeichnenden Bereich
    if (ev->region().contains (r))
    {
        // Ja, dann zeichnen
        p.setPen (QPen (black, 0));
        p.setBrush (white);
        p.drawRect (r);
    }
}
```

Achtung: Wenn wir für die Umrandungslinie des Rechtecks eine Liniendicke größer als 1 Pixel wählen, so muss das beim Test mit berücksichtigt werden. Die Abmessungen des Rechtecks werden dadurch ja größer.

Für ein einfaches Rechteck wie hier lohnt sich dieser Aufwand sicher nicht, da dieser Befehl nur wenige Daten an den X-Server überträgt. Ein besonders datenlastiger Befehl ist dagegen *QPainter::drawImage*, mit dem Sie *QImage*-Objekte zeichnen lassen (siehe Kapitel 4.3.1, *QImage*). Für diesen Fall lohnt sich der Aufwand des zusätzlichen Tests in der Regel immer.

mousePressEvent, mouseReleaseEvent, mouseMoveEvent, mouseDoubleClickEvent, wheelEvent

Mit diesen Events teilt Qt einem Widget mit, dass eine für das Widget relevante Mausaktion stattgefunden hat. Dabei wird *mousePressEvent* aufgerufen, sobald eine der Maustasten gedrückt wurde und sich der Mauszeiger innerhalb des Widgets befand. Die Methode wird nicht aufgerufen, wenn der Mauszeiger dabei in einem der Unter-Widgets war, da in diesem Fall der Event an das Unterfenster geschickt wird.

Wenn Ihr Widget auf Maus-Events reagieren soll, überschreiben Sie einige oder alle dieser Methoden. Innerhalb der Methode können Sie dann Fallunterscheidungen vornehmen, welche Taste gedrückt oder losgelassen wurde, an welcher Stelle sich der Mauszeiger befand usw. Sollte der Maus-Event den Zustand Ihres Widgets ändern, so rufen Sie gegebenenfalls die Methode *QWidget::repaint* auf, um den neuen Zustand auf dem Bildschirm darzustellen. Geben Sie dabei möglichst den Ausschnitt an, in dem sich die Zustandsänderung auswirkt, damit nicht das ganze Widget neu gezeichnet werden muss. Rufen Sie anschließend die nötigen Signalmethoden auf, um die Zustandsänderung den angeschlossenen Objekten mitzuteilen. Testen Sie möglichst, ob sich der Zustand wirklich verändert hat. Ein Neuzeichnen und das Aussenden eines Signals kann unter Umständen lange dauern, es sollte von daher nicht unnötig vorgenommen werden.

Sobald eine Maustaste gedrückt und gehalten wird, gehen alle folgenden Maus-Events an dieses Widget, auch wenn sich der Mauszeiger aus dem Widget herausbewegt. Eine Bewegung des Mauszeigers bei gedrückter Taste führt zu einem Aufruf von *mouseMoveEvent*. Wird eine Maustaste losgelassen, so wird *mouseReleaseEvent* aufgerufen. Erst wenn alle Maustasten wieder losgelassen wurden, können spätere Maus-Events an ein anderes Widget geschickt werden.

Standardmäßig wird die Methode *mouseMoveEvent* nur dann aufgerufen, wenn während der Mausbewegung mindestens eine der Maustasten gedrückt ist. Auf diese Weise erzeugt eine normale Mausbewegung keine unnötigen Events. Ist Ihr Widget darauf angewiesen, die Bewegung der Maus auch ohne gedrückte Maustaste nachzuvollziehen, so müssen Sie für Ihr Widget die Methode *setMouseTracking (true)* aufrufen. Dann erhalten Sie bei jeder Mausbewegung, bei der sich der Mauszeiger innerhalb des Widgets befindet, einen *mouseMoveEvent*.

Allen Methoden wird ein Objekt der Klasse *QMouseEvent* übergeben. Diese Klasse enthält eine Reihe von Informationen über den Zeitpunkt des Maus-Events, die mit verschiedenen Methoden ermittelt werden können. Mit der Methode *pos* können Sie die Position des Mauszeigers zum Zeitpunkt des Events – relativ zur oberen linken Ecke des Widgets – erfragen. Mit der Methode *globalPos* erhalten Sie die Position relativ zur linken oberen Bildschirmcke. Beachten Sie, dass die Position des Mauszeigers auch außerhalb des Widgets liegen kann, wenn die Maus nach dem Drücken einer Taste noch bewegt wird. Mit der Methode *button*

können Sie die Maustaste ermitteln, die den Event ausgelöst hat. Mögliche Rückgabewerte sind *LeftButton*, *RightButton*, *MidButton* oder *NoButton*. Nur einer dieser Werte kann hier zurückgegeben werden. Werden zwei Maustasten »gleichzeitig« gedrückt, werden zwei *mousePressEvent*s erzeugt. Bei einem *mouseMoveEvent* ist der Rückgabewert von *button* grundsätzlich *NoButton*. Mit der Methode *state* können Sie den Zustand der Maustasten und der Tasten ,  und  auf der Tastatur zum Zeitpunkt des Events – genauer gesagt unmittelbar vor dem Event – ermitteln. Der Rückgabewert ist eine Oder-Kombination der Werte *LeftButton*, *RightButton*, *MidButton*, *ShiftButton*, *AltButton* und *ControlButton*. Der Zustand der Tasten ist oftmals wichtig, um zu entscheiden, welche Aktion auszuführen ist. So kann zum Beispiel eine gedrückte -Taste bedeuten, dass eine Datei kopiert anstatt verschoben werden soll. Eine gedrückte -Taste soll oft bewirken, dass eine Markierung bis zur aktuellen Mausposition ausgedehnt werden soll. Da der Zustand, den *state* zurückliefert, unmittelbar vor dem Event ermittelt wurde, ist beispielsweise bei einem *mousePressEvent*, der durch die linke Maustaste erzeugt wurde, das Bit von *LeftButton* im Rückgabewert von *state* noch nicht gesetzt. Sie können auch die Methode *stateAfter* benutzen, die den Zustand der Maustasten und Tasten *nach* dem Event-Ereignis zurückliefert.

Unter bestimmten Umständen kann es vorkommen, dass ein Widget den einen oder anderen Maus-Event nicht bekommt. So sollten Sie sich nicht immer darauf verlassen, dass zu einem *mousePressEvent* auch ein *mouseReleaseEvent* gesendet wird. Solch ein Fehl-Event tritt meist dann auf, wenn Ihre Applikation ein modales Dialogfenster (*QDialog* oder *QDialog*) öffnet. Alle weiteren Maus-Events für das Widget werden dann ignoriert und nicht weitergeleitet. Sie sollten also darauf achten, dass Sie ein modales Dialogfenster möglichst als Reaktion auf ein *mouseReleaseEvent* und nicht auf ein *mousePressEvent* öffnen.

Ein weiterer Maus-Event wird der Methode *mouseDoubleClickEvent* geliefert. Diese Methode wird immer dann aufgerufen, wenn eine Maustaste innerhalb kurzer Zeit zweimal betätigt wird. Die erste Betätigung löst einen Aufruf von *mousePressEvent* aus, die zweite einen Aufruf von *mouseDoubleClickEvent*. Eine dritte würde wieder *mousePressEvent* aufrufen, eine vierte wieder *mouseDoubleClickEvent* usw. Wie klein der Abstand zwischen zwei Mausklicks sein muss, damit *mouseDoubleClickEvent* ausgelöst wird, kann mit der statischen Methode *QApplication::setDoubleClickInterval* eingestellt werden.

Die Standardimplementierung von *mouseDoubleClickEvent* ruft nur die Methode *mousePressEvent* auf, so dass der Doppelklick genau wie zwei einzelne Klicks behandelt wird. Da ein Doppelklick für den Anwender oft nicht leicht einzugeben ist, ist als Vorgabe für KDE-Programme festgelegt, dass der Doppelklick möglichst nicht benutzt werden soll. Ein einzelner Klick auf ein Objekt soll bereits eine Aktion auslösen. Dadurch sind KDE-Programme von der Bedienungsführung her mit fast allen Internet-Browsern konsistent, bei denen ebenfalls das

Anklicken eines Links die Verfolgung dieses Links bewirkt. Sie sollten sich möglichst an diese Vorgabe halten, um eine einheitliche Bedienung aller KDE-Programme zu gewährleisten. In diesem Fall lassen Sie `mouseDoubleClickEvent` einfach unverändert. Wollen Sie dagegen Doppelklicks speziell behandeln, überschreiben Sie die Methode `mouseDoubleClickEvent`. Beachten Sie aber dabei, dass vor der Ausführung dieser Methode bereits einmal die Methode `mousePressEvent` ausgeführt wurde. Eventuell müssen Sie die Aktion, die in dieser Methode ausgeführt wurde, wieder rückgängig machen. Wollen Sie verhindern, dass ein Doppelklick aus Versehen zweimal die Aktion eines einzelnen Klicks auslöst, überschreiben Sie die Methode `mouseDoubleClickEvent` und führen in dieser Methode nichts aus. So wird der zweite Klick des Doppelklicks einfach ignoriert.

Oftmals müssen Sie in Ihrem Widget zwischen einem Klicken (*Click*) und einem Ziehen (*Drag*) unterscheiden. Beim Klicken wird die Maustaste nur kurz gedrückt und sofort wieder losgelassen. Beim Ziehen wird die Maustaste gedrückt, die Maus bewegt und dann an der Zielposition die Taste wieder losgelassen. Da aber auch beim Klicken die Maus aus Versehen ein kleines Stück bewegt werden kann, ist die Unterscheidung nicht leicht. Am besten unterscheidet man Klicken und Ziehen durch die Entfernung, die mit der Maus zurückgelegt wurde, und die Zeitspanne, die zwischen dem Drücken und Loslassen der Maustaste verstrichen ist. Gängig ist dabei eine Entfernung von mehr als zehn Pixel oder eine Zeitspanne von mehr als 300 Millisekunden. Ist eine der Bedingungen überschritten, wird die Aktion ausgeführt, die mit dem Ziehen verknüpft ist.

Eine typische Implementierung der Maus-Event-Methoden wollen wir hier kurz vorstellen. In diesem Fall wird zwischen dem Ziehen und dem Klicken mit der linken Maustaste unterschieden. Die rechte Maustaste öffnet ein Kontextmenü. Die mittlere Maustaste hat keine Bedeutung. Die Struktur der Klasse könnte beispielsweise so aussehen:

```
class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget (QWidget *parent = 0, const char *name = 0);
    ~MyWidget () {}
    ....
protected:
    void mousePressEvent (QMouseEvent *ev);
    void mouseMoveEvent (QMouseEvent *ev);
    void mouseReleaseEvent (QMouseEvent *ev);
    ....
private:
    bool leftDown; // Ist true, wenn linke Maustaste gedrückt
    bool dragging; // Ist true, wenn gezogen statt
                  // geklickt wird
```

```

    QTime timer;
    QPoint startPoint;
    QPopupMenu *kontext;
}

MyWidget::MyWidget (QWidget *parent, const char* name)
    : QWidget (parent, name), leftDown (false)
{
    ....
}

void MyWidget::mousePressEvent (QMouseEvent *ev)
{
    if (leftDown) return; // Wenn linke Taste gedrückt,
                          // hat die rechte keine Auswirkung
    if (ev->button() == RightButton)
        kontext->exec (ev->globalPos());
    else if (ev->button() == LeftButton)
    {
        // Linke Maustaste gedrückt:
        // Zuerst Klicken annehmen (dragging auf false)
        // und Zeitpunkt und Position merken
        leftDown = true;
        dragging = false;
        timer.start();
        startPoint = ev->pos();
    }
}

void MyWidget::mouseMoveEvent (QMouseEvent *ev)
{
    // Nur bei gedrückter linker Maustaste relevant
    if (!leftDown) return;
    if (!dragging)
        if (timer.elapsed() >= 300 ||
            Q_ABS (startPoint.x() - ev->pos().x()) > 10 ||
            Q_ABS (startPoint.y() - ev->pos().y()) > 10)
            dragging = true;
    if (dragging)
    {
        // Eventuell bereits eine Aktion ausführen, zum
        // Beispiel ein Objekt auf dem Bildschirm
        // verschieben
    }
}

void MyWidget::mouseReleaseEvent (QMouseEvent *ev)
{
    // Nur linke Maustaste relevant

```

```
if (!leftDown || ev->button() != LeftButton) return;

// Falls die Maus nicht bewegt worden ist, aber mehr
// als 300 ms vergangen sind, muss dragging auf true
// gesetzt werden. Dazu rufen wir einfach nochmals
// mouseMoveEvent auf.
mouseMoveEvent (ev);

leftDown = false;
if (dragging)
    // Aktion für das Ziehen ausführen
else
    // Aktion für das Klicken ausführen
}
```

Auch das zusätzliche Scroll-Rädchen, das viele Mäuse inzwischen bieten, kann unter Qt genutzt werden. Eine Bewegung des Rads bewirkt einen Aufruf der Event-Methode *wheelEvent*. Im übergebenen Event-Objekt der Klasse *QWheelEvent* ist neben den Methoden *state*, *pos* und *globalPos* (identisch mit den gleichnamigen Methoden in *QMouseEvent*) noch eine weitere Methode implementiert: Mit der Methode *delta* können Sie ermitteln, um wie viel das Rad seit dem letzten Aufruf von *wheelEvent* gedreht worden ist. Ein positiver Wert bedeutet dabei, dass das Rad vom Anwender weg bewegt wurde, ein negativer Wert, dass das Rad auf den Anwender zu bewegt wurde. Da Mäuse mit Rad immer mehr Verbreitung finden, empfiehlt es sich, diesen Event zu implementieren, sofern es sinnvoll ist. Achten Sie jedoch auf jeden Fall darauf, dass Ihr Widget auch mit einer Maus ohne Rad den vollen Funktionsumfang bietet. Überschreiben Sie die Methode *wheelEvent*, und rufen Sie in dieser Methode die Methode *accept* des übergebenen *QWheelEvent*-Objekts auf.

keyPressEvent, keyReleaseEvent

Eine Eingabe an der Tastatur wird einem Widget über die beiden Event-Methoden *keyPressEvent* und *keyReleaseEvent* mitgeteilt. Dabei erhält dasjenige Widget, das den Tastaturfokus besitzt, den Event (siehe Kapitel 3.2.3, *Die wichtigsten Widget-Eigenschaften*, Abschnitt *Tastaturfokus*).

Jedes Widget sollte möglichst vollständig (wenn auch vielleicht etwas unkomfortabler) über die Tastatur gesteuert werden können. Bei der Entwicklung eines neuen Widgets sollten Sie diese Vorgabe berücksichtigen. Daher sollten Sie eine oder beide dieser Event-Methoden überschreiben.

Der X-Server bietet eine sehr mächtige Kontrolle über die Tastatur. Die Methode *keyPressEvent* wird aufgerufen, sobald eine Taste gedrückt wird. Weitere *keyPressEvent*s werden generiert, wenn der Anwender die Taste lange Zeit gedrückt hält, so dass die automatische Tastenwiederholung einsetzt. Wird die Taste wieder losgelassen, wird die Methode *keyReleaseEvent* aufgerufen. Auf diese

Weise kann die Applikation auch testen, ob mehrere Tasten gleichzeitig gedrückt sind, indem sie über die gedrückten und losgelassenen Tasten Buch führt. Jede Taste der Tastatur erzeugt einen Event, also auch die Tasten `⇧`, `Strg`, `Alt`, `Esc`, die Funktionstasten `F1` bis `F12`, die Cursor-Tasten usw.

Die beiden Methoden erhalten als ersten Parameter ein Objekt der Klasse `QKeyEvent`, das nähere Informationen zum Event enthält. Mit der Methode `QKeyEvent::key` können Sie ermitteln, welche Taste den Event ausgelöst hat. Den `int`-Wert, den diese Methode zurückliefert, können Sie mit Konstanten vergleichen, die in Qt definiert sind. Diese Konstanten tragen zum Beispiel die Namen `Key_A` bis `Key_Z` (für die Tasten `A` bis `Z`), `Key_0` bis `Key_9` (für die Tasten `0` bis `9`), `Key_F1` bis `Key_F12` (für die Funktionstasten `F1` bis `F12`), `Key_Left`, `Key_Right`, `Key_Up` und `Key_Down` (für die Cursor-Tasten). Eine vollständige Liste der Konstanten finden Sie in der Datei `qnamespace.h`. Diese Konstanten sind innerhalb der Klasse `Qt` definiert, von der alle wichtigen Klassen abgeleitet sind. Innerhalb von Methoden können Sie diese Konstanten daher ohne Zusatz benutzen; außerhalb (z.B. in globalen Funktionen wie der `main`-Funktion) müssen Sie die Klassenspezifikation `Qt::` voranstellen.

Die folgende Implementierung der Methoden `keyPressEvent` und `keyReleaseEvent` könnte zum Beispiel in einem Action-Spiel dazu dienen, immer den aktuellen Zustand aller Cursor-Tasten zu ermitteln:

```
class GameWidget : public QWidget
{
    Q_OBJECT
public:
    GameWidget (QWidget *parent = 0, const char *name = 0);
    ~GameWidget () {}

protected:
    void keyPressEvent (QKeyEvent *ev);
    void keyReleaseEvent (QKeyEvent *ev);

private:
    bool up, down, left, right;
}

GameWidget::GameWidget (QWidget *parent, const char *name)
    : QWidget (parent, name),
      up (false), down (false), left (false), right (false)
{
    ....
}

void GameWidget::keyPressEvent (QKeyEvent *ev)
{

```



```

ev->accept ();
switch (ev->key())
{
    case Key_Left : left = true; break;
    case Key_Right: right = true; break;
    case Key_Up    : up = true; break;
    case Key_Down  : down = true; break;
    default: ev->ignore();
}
}

void GameWidget::keyReleaseEvent (QKeyEvent *ev)
{
    ev->accept ();
    switch (ev->key())
    {
        case Key_Left : left = false; break;
        case Key_Right: right = false; break;
        case Key_Up    : up = false; break;
        case Key_Down  : down = false; break;
        default: ev->ignore();
    }
}

```

Diese Implementierung enthält bereits die beiden Methoden *QKeyEvent::accept* und *QKeyEvent::ignore*. Durch diese Methoden entscheiden Sie, ob Sie den Tastatur-Event benutzen konnten (*accept*) oder nicht (*ignore*). Dieser Zustand wird im *QKeyEvent*-Objekt gespeichert. Wenn Sie den Event nicht benutzt haben, so wird er an das Vater-Widget weitergereicht. Vergessen Sie also nicht, *accept* aufzurufen, wenn Sie einen Event benutzt haben, da er sonst auch noch auf andere Widgets wirken könnte.

Das *QKeyEvent*-Objekt besitzt weiterhin die Methode *ascii*, mit der Sie ermitteln können, welches ASCII-Zeichen durch den Tastendruck erzeugt wird. Diese Methode leistet sehr gute Dienste, wenn Sie eine Texteingabe über die Tastatur im Widget erlauben wollen. Für die Buchstaben, Zahlen, Sonderzeichen und einige Steuerzeichen (zum Beispiel die -Taste oder die -Taste) wird der normale ASCII-Code geliefert. Für Sondertasten, denen kein ASCII-Code zugeordnet ist (z.B. für die Funktionstasten  bis , , ,  und die Cursor-Tasten), wird der Wert 0 zurückgeliefert. Einige Zeichen – insbesondere länderspezifische Zeichen – werden durch zwei Tasten erzeugt, zum Beispiel ñ, Â oder é. Bei der Eingabe eines solchen Zeichens wird ein Tastatur-Event erzeugt, der in *ascii* den korrekten ASCII-Wert enthält und in *key* den Wert 0 (da es nicht eine eindeutige Taste war, die dieses Zeichen erzeugte). Beachten Sie, dass Sie solche Zeichen in Linux nicht eingeben können, wenn Sie im X-Server als Tastaturmodus *No Dead Key* gewählt haben.

Die Methode `QKeyEvent::text` liefert Ihnen das eingegebene Zeichen als `QString` zurück. Das hat den Vorteil, dass Sie auch alle Unicode-Zeichen mit dieser Methode erfragen können.

Die -Taste sowie + haben in Qt die Spezialbedeutung, dass man mit ihnen den Tastaturfokus zwischen den Widgets wechseln kann. Deshalb werden Tastatur-Events, die die Tabulator-Taste betreffen, in der Regel nicht an ein Widget weitergeleitet. Wenn Ihr Widget die Tabulator-Events benötigt – zum Beispiel ein Widget mit Texteingabe, in das man auch Tabulatoren einfügen kann –, überschreiben Sie am besten die virtuelle Methode `focusNextPrevChild` in Ihrem Widget. Wenn diese Methode `false` zurückliefert, wird der Tabulator-Event an das Widget weitergeleitet. Eine Implementierung kann beispielsweise so aussehen:

```
class MyWidget : public QWidget
{
    ....
protected:
    bool focusNextPrevChild (bool) {return false;}

    // Jetzt erhalten die Tastatur-Event-Methoden
    // auch Tabulator-Events
    void keyPressEvent (QKeyEvent *ev);
}
```

Wenn Ihr Widget das einzige Unter-Widget ist, das den Tastaturfokus erhalten kann – weil es zum Beispiel das einzige Widget ist oder alle anderen Widgets die `focusPolicy` von `NoFocus` besitzen –, werden Tabulator-Events automatisch an das Widget weitergeleitet. In diesem Fall brauchen Sie die Methode `focusNextPrevChild` nicht zu überschreiben.

Bedenken Sie, dass ein Dialog eventuell nicht mehr vollständig über die Tastatur bedient werden kann, wenn ein Widget die Tabulator-Events für sich beansprucht. Es ist dann nicht mehr möglich, andere Widgets mit der -Taste zu wählen, wenn dieses Widget einmal den Fokus erhalten hat. In diesem Fall kann der Fokus nur noch mit der Maus oder mit einem Accelerator ( zusammen mit einer anderen Taste) gewechselt werden.

focusInEvent, focusOutEvent

Diese beiden Event-Methoden werden aufgerufen, wenn das Widget den Tastaturfokus erhält bzw. verliert. Das übergebene Objekt der Klasse `QFocusEvent` enthält keine weiteren relevanten Informationen.

Wenn Ihr Widget auf Tastatur-Events reagieren soll, dann sollte am Aussehen des Widgets erkennbar sein, ob es den Tastaturfokus besitzt oder nicht. So kann der Anwender erkennen, in welchem Widget eine Texteingabe wirkt. Der Unterschied im Aussehen sollte bereits in `paintEvent` realisiert sein. Meist reicht es daher, in bei-

den Methoden die Methode *repaint* des Widgets aufzurufen, die das gesamte Widget neu zeichnet. Das ist auch die Default-Implementierung dieser Methoden, so dass Sie sie in der Regel nicht überschreiben müssen.

moveEvent, resizeEvent

Die Methode *moveEvent* wird aufgerufen, wenn ein Widget bewegt wird. Bei einem Toplevel-Widget kann dies zum Beispiel durch den Anwender geschehen, der das Fenster an der Titelleiste verschoben hat. Für Unter-Widgets ist in der Regel ein Aufruf der Methode *move* oder *setGeometry* Auslöser für diesen Event. Diese Methoden können beispielsweise vom Layout-Management aufgerufen worden sein. Das übergebene *QMoveEvent*-Objekt enthält die alte Position des Widgets, die mit der Methode *oldPos* ermittelt werden kann, sowie die neue Position, die Sie mit *pos* erhalten.

Das Verschieben eines Widgets hat in der Regel keine Relevanz für das Widget. Falls Teile des Widgets neu gezeichnet werden müssen, wird bereits automatisch *paintEvent* aufgerufen. Diese Methode müssen Sie also nur in Spezialfällen überschreiben.

Die Methode *resizeEvent* wird bei einer Größenänderung des Widgets aufgerufen. Bei einem Toplevel-Widget kann diese Änderung durch den Anwender erfolgt sein, für Unter-Widgets durch die Aufrufe der Methoden *resize* oder *setGeometry*. In der Regel reicht es aus, in dieser Methode die Methode *repaint* des Widgets aufzurufen, um das Widget erneut zu zeichnen. Dies ist auch die Default-Implementierung, so dass Sie diese Methode nur selten überschreiben müssen.

Wenn Sie zusätzliche Berechnungen mit der neuen Größe durchführen müssen, können Sie diese Methode überschreiben. Aus dem übergebenen Objekt der Klasse *QResizeEvent* können Sie mit der Methode *oldSize* die vorherige Größe und mit der Methode *size* die neue Größe des Widgets ermitteln. Wenn Ihr Widget Unter-Widgets enthält, können Sie in dieser Methode beispielsweise die Position und Größe dieser Unter-Widgets von Hand anpassen, so wie es in Kapitel 3.6.2, *Anordnung der Widgets im resize-Event*, beschrieben wird.

enterEvent, leaveEvent

Diese Event-Methoden werden aufgerufen, sobald die Maus in den rechteckigen Bereich des Widgets eintritt bzw. diesen Bereich verlässt. Sie können diese Methoden zum Beispiel überschreiben, falls ein Widget sein Aussehen ändern soll, wenn sich die Maus über ihm befindet. Die Default-Implementierung bewirkt nichts.

dragEnterEvent, dragMoveEvent, dragLeaveEvent, dropEvent

Diese Event-Methoden werden speziell für Drag&Drop benutzt. Nähere Informationen hierzu finden Sie in Kapitel 4.15.2, *Drag&Drop*.

showEvent, hideEvent, closeEvent

Diese Spezial-Event-Methoden werden Sie nur selten überschreiben müssen. Die Methoden *showEvent* und *hideEvent* werden aufgerufen, nachdem das Widget mit *show* angezeigt oder mit *hide* wieder versteckt wurde. Wenn Sie zum Beispiel verhindern wollen, dass ein Fenster versteckt werden kann, überschreiben Sie *hideEvent* und führen darin die Methode *show* aus.

Die Methode *closeEvent* wird aufgerufen, wenn der Anwender auf den X-Button (meist am linken Rand der Titelzeile eines Toplevel-Widgets) klickt bzw. den Befehl SCHLIESSEN aus dem Fenstermenü (das Icon am linken Rand der Titelzeile) wählt. Sie können bestimmen, was in diesem Fall geschehen soll, indem Sie diese Methode überschreiben. Das Event-Objekt der Klasse *QCloseEvent* enthält zwei Methoden, *accept* und *ignore*. Mit diesen Methoden setzen Sie ein Flag innerhalb des Objekts, das nach der Rückkehr aus der Methode ausgewertet wird. Wenn Sie *accept* aufrufen (wie es die Default-Implementierung der Methode *closeEvent* macht), wird das Fenster nach der Rückkehr mit der Methode *hide* versteckt. Wenn Sie die Methode *closeEvent* überschreiben und die Methode *ignore* des übergebenen Objekts aufrufen, bleibt das Fenster unverändert bestehen.

Wenn Sie das Fenster vollständig löschen wollen, können Sie die Methode zum Beispiel folgendermaßen implementieren:

```
void MyWidget::closeEvent (QCloseEvent *ev)
{
    ev->ignore();
    delete this;
}
```

Auf diese Weise wird das Fenster gelöscht. Dazu muss es natürlich mit *new* auf dem Heap angelegt worden sein. Beachten Sie unbedingt, dass Sie nach dem Löschen mit *delete this* nicht mehr auf Objektvariablen und virtuelle Methoden zugreifen dürfen, da das Widget im Speicher nicht mehr existiert. Achten Sie auch darauf, dass beim Löschen des letzten Fensters das Programm beendet werden muss, da der Anwender sonst keine andere (reguläre) Möglichkeit mehr hat, das Programm zu beenden.

Die Klasse *KMainWindow*, die ein Hauptfenster eines KDE-Programms erzeugt, benutzt diese Event-Methode in der beschriebenen Art. Beim Schließen des letzten Fensters wird automatisch die Applikation beendet. Ausführlichere Informationen darüber, wie Sie in dieser Klasse das Schließen eines Fensters noch genauer kontrollieren können, finden Sie in Kapitel 3.51, *Ableiten einer eigenen Klasse von KMainWindow*.

4.4.2 Beispiel: Festlegen einer eindimensionalen Funktion

Als Beispiel für ein selbst definiertes GUI-Element wollen wir nun ein Widget entwerfen, das die Festlegung einer eindimensionalen Funktion mit Werte- und Definitionsbereich von 0 bis 1 ermöglicht (siehe Abbildung 4.47). Solche Funktionen werden z.B. für die Gammakorrektur, Farb-, Helligkeits- und Kontraständerungen eingesetzt.

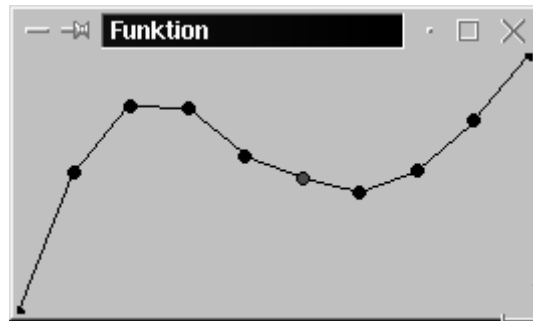


Abbildung 4-47 Widget zur Einstellung einer eindimensionalen Funktion

Das Widget soll eine im Konstruktor festgelegte Anzahl von Stützstellen mit gleichem Abstand haben, zwischen denen linear interpoliert wird. Jede Stützstelle wird durch einen kleinen Kreis repräsentiert, der mit der Maus nach oben oder unten verschoben werden kann. Ebenso sollen die Stützstellen mit den Pfeiltasten verschoben werden können.

Gehen wir die fünf Schritte zum Entwurf eines Widgets der Reihe nach durch:

1. Es gibt mehrere Möglichkeiten, wie das Widget eine Änderung durch ein Signal mitteilen könnte. Es könnte den Index und den neuen Wert der geänderten Stützstelle liefern. Dieses Vorgehen ist aber ungünstig, falls gleichzeitig mehrere Stützstellen geändert werden, z.B. durch ein Reset o. Ä. Das Widget könnte im Signal ein Array mit den Werten der Stützstellen liefern. Eine dritte Möglichkeit wäre, ein Signal zu benutzen, das gar keine Parameter hat. In diesem Fall muss der Slot, der mit dem Signal verbunden wird, die Werte der Stützstellen z.B. über eine eigene Methode der Klasse besorgen. In unserem Fall wollen wir die zweite Variante implementieren. Die Deklaration des Signals sieht dann etwa wie folgt aus:

```
signals:  
    void changed (int m, const double *val);
```

2. Die Anzahl der Stützstellen soll im Konstruktor festgelegt werden und danach nicht mehr zu ändern sein. Über eine Methode *setValues* sollen alle Stützstellen aus einem Array von Werten festgelegt werden können. Außerdem soll es

noch drei einfache Methoden geben, um die Stützstellen auf Standardwerte zu legen: *linear* setzt die Stützstellen auf eine Gerade mit der Steigung 1, *negativlinear* auf eine Gerade mit der Steigung -1 und *gamma* auf eine Gamma-korrekturkurve zu einem Parameter *g*. Diese drei Methoden werden als Slots implementiert, so dass sie sehr leicht z.B. durch einen Button oder einen Schieberegler aufgerufen werden können. Wir erhalten somit die Deklaration der Slots und Methoden, die als *public* definiert werden:

```
public:
    // Setzen von einem Wert oder allen Werten
    void setValue (int n, double v);
    void setValues (const double *val);
    // Abfrage der Werte und der Stützstellenanzahl
    const double *values ();
    int number ();
public slots:
    // Initialisieren auf drei verschiedene Arten
    void linear ();
    void negative_linear ();
    void gamma (double g);
```

3. Keines der bereits in Qt oder KDE definierten Widgets eignet sich als Basis-klasse. Wir wählen also *QWidget* als Basis.
4. Wir implementieren neue Versionen der Event-Routinen für *paintEvent*, *mousePressEvent*, *mouseMoveEvent*, *mouseReleaseEvent*, *keyPressEvent*, *keyReleaseEvent*, *focusInEvent*, *focusOutEvent* und *resizeEvent*. Die Stützstellen werden im Widget durch kleine Kreise, so genannte Handles, dargestellt, die durch gerade Linien verbunden sind. Unabhängig vom neu zu zeichnenden Ausschnitt werden hier im Beispiel alle Zeichenbefehle zum X-Server geschickt, da sie sehr einfach sind und kaum Aufwand bedeuten. Eines der Handles ist aktiv, die anderen sind inaktiv. Das aktive Handle wird in Rot dargestellt, falls das Widget den Tastaturfokus hat, sonst in der normalen Textfarbe. Mit den Pfeiltasten  und  oder durch einen Mausklick kann man das aktive Handle auswählen. Damit es nicht zu ungewollten Effekten kommt, werden die Pfeiltasten  und  ignoriert, während die Maustaste noch gedrückt ist. Eine Bewegung der Maus mit gedrückter Taste nach oben oder unten, die Pfeiltasten  und  sowie die Tasten für  und  ändern den Wert der aktiven Stützstelle. Dabei wird die Darstellung auf dem Bildschirm aktualisiert, und das Signal, das die Änderung anzeigt, wird gesendet. Die Methoden für *focusInEvent* und *focusOutEvent* sorgen dafür, dass das aktive Handle nur dann hervorgehoben dargestellt wird, wenn unser Widget den Tastaturfokus hat. Anstatt allerdings das komplette Widget neu zu zeichnen, wird nur das aktive Handle neu dargestellt.

5. Da die Handles durch kleine Kreise mit einem Durchmesser von sieben Pixel dargestellt werden, sollten sie einen Mindestabstand von zehn Pixel voneinander haben. Als vernünftige Breite für das Widget legen wir also $10 * (\text{Anzahl Stützstellen} - 1)$ fest. Da das Widget möglichst quadratisch sein sollte, legen wir die Höhe auf den gleichen Wert fest. Damit steht der Rückgabewert von *sizeHint* fest. Als *sizePolicy* legen wir fest, dass die von *sizeHint* angegebene Größe das Minimum darstellt, das Widget aber gern sowohl in der Breite als auch in der Höhe verfügbaren Platz nutzt. *sizePolicy* liefert also für Breite und Höhe *MinimumExpand* zurück.

Hier folgt nun das komplette Listing für unser Widget:

```
class Function: public QWidget {
    Q_OBJECT

public:
    Function (int n, QWidget *parent=0, const char *name=0);
    ~Function ();

    void setValue (int n, double v);
    void setValues (const double *val);
    const double *values () {return m_values;}
    int number () {return m_number;}
    QSize sizeHint ();
    QSizePolicy sizePolicy ();

public slots:
    void linear ();
    void negative_linear ();
    void gamma (double g);

signals:
    void changed (int m, const double *val);

protected:
    void paintEvent (QPaintEvent *ev);
    void resizeEvent (QResizeEvent *ev);
    void keyPressEvent (QKeyEvent *ev);
    void mousePressEvent (QMouseEvent *ev);
    void mouseMoveEvent (QMouseEvent *ev);
    void mouseReleaseEvent (QMouseEvent *ev);
    void focusInEvent (QFocusEvent *ev);
    void focusOutEvent (QFocusEvent *ev);

private:
    QPoint position (int n);
    void setActualIndex (int i);
    void drawHandle (int i, QPainter *p);
    double *m_values;
```

```

    int m_number;
    int actualIndex;
    bool isChanging;
};

Function::Function (int n, QWidget *parent, const char *name)
    : QWidget (parent, name)
{
    if (n < 2) n = 2;
    m_number = n;
    m_values = new double [n];
    for (int i = 0; i < n; i++)
        m_values [i] = 0.0;
    isChanging = false;
    actualIndex = 0;
}

Function::~Function () {
    delete [] m_values;
}

void Function::linear () {
    for (int i = 0; i < m_number; i++)
        m_values [i] = ((double) i) / (m_number - 1);
    repaint ();
    emit changed (m_number, m_values);
}

void Function::negative_linear () {
    for (int i = 0; i < m_number; i++)
        m_values [i] = 1.0 - ((double) i) / (m_number - 1);
    repaint ();
    emit changed (m_number, m_values);
}

void Function::gamma (double g) {
    for (int i = 0; i < m_number; i++)
    {
        double x = ((double) i) / (m_number - 1);
        m_values [i] = pow (x, g);
    }
    repaint ();
    emit changed (m_number, m_values);
}

QPoint Function::position (int n) {
    if (n < 0 || n >= m_number || m_number < 2)
        return QPoint (0, 0);
    return QPoint ((width () - 1) * n / (m_number - 1),

```



```
        (int) ((height () - 1) *
        (1.0 - m_values [n])));
    }

void Function::drawHandle (int i, QPainter *p) {
    if (hasFocus () && i == actualIndex)
        p->setBrush (red);
    else
        p->setBrush (colorGroup ().foreground ());
    p->drawEllipse (QRect (position (i) - QPoint (3, 3),
        QSize (7, 7)));
}

void Function::setActualIndex (int i) {
    if (i < 0 || i >= m_number || i == actualIndex)
        return;
    int oldIndex = actualIndex;
    QPainter p (this);
    actualIndex = i;
    drawHandle (oldIndex, &p);
    drawHandle (actualIndex, &p);
}

void Function::paintEvent (QPaintEvent *) {
    QPainter p(this);
    p.setPen (colorGroup().foreground ());
    p.moveTo (position (0));
    int i;
    for (i = 1; i < m_number; i++)
        p.lineTo (position (i));
    for (i = 0; i < m_number; i++)
        drawHandle (i, &p);
}

void Function::resizeEvent (QResizeEvent *) {
    repaint ();
}

void Function::keyPressEvent (QKeyEvent *ev) {
    switch (ev->key ()) {
        case Key_Left:
            if (actualIndex > 0)
                setActualIndex (actualIndex - 1);
            break;
        case Key_Right:
            if (actualIndex < m_number - 1)
                setActualIndex (actualIndex + 1);
            break;
        case Key_Up:
```

```

        setValue (actualIndex, m_values [actualIndex] + 0.01);
        break;
    case Key_Down:
        setValue (actualIndex, m_values [actualIndex] - 0.01);
        break;
    case Key_PageUp:
        setValue (actualIndex, m_values [actualIndex] + 0.1);
        break;
    case Key_PageDown:
        setValue (actualIndex, m_values [actualIndex] - 0.1);
        break;
    default:
        ev->ignore ();
    }
}

void Function::mousePressEvent (QMouseEvent *ev) {
    if (ev->button () != LeftButton)
        return;
    int index = (ev->x () * (m_number - 1) + width () / 2) / width ();
    QPoint pos = position (index);
    if (ev->x () < pos.x () - 3 || ev->x () > pos.x () + 3 ||
        ev->y () < pos.y () - 3 || ev->y () > pos.y () + 3)
        return;
    setActualIndex (index);
    isChanging = true;
}

void Function::mouseMoveEvent (QMouseEvent *ev) {
    if (!isChanging)
        return;
    setValue (actualIndex, 1.0 - ((double) ev->y () / height ());
}

void Function::mouseReleaseEvent (QMouseEvent *ev) {
    if (!isChanging || ev->button () != LeftButton)
        return;
    isChanging = false;
}

void Function::focusInEvent (QFocusEvent *) {
    drawHandle (actualIndex, &QPainter (this));
}

void Function::focusOutEvent (QFocusEvent *) {
    drawHandle (actualIndex, &QPainter (this));
}

QSize Function::sizeHint () {

```

```

    return QSize ((m_number - 1) * 10, (m_number - 1) * 10);
}

QSizePolicy Function::sizePolicy () {
    return QSizePolicy (MinimumExpanding, MinimumExpanding);
}

void Function::setValue (int index, double val) {
    if (val < 0.0) val = 0.0;
    if (val > 1.0) val = 1.0;
    if (index < 0 || index >= m_number || m_values [index] == val)
        return;
    m_values [index] = val;
    int x1 = position (QMAX (index - 1, 0)).x ();
    int x2 = position (QMIN (index + 1, m_number - 1)).x ();
    repaint (QRect (x1, 0, x2 - x1 + 1, height ()));
    emit changed (m_number, m_values);
}

```

4.4.3 Beispiel: Dame-Brett

Als zweites Beispiel wollen wir ein Widget implementieren, das ein Dame-Spielbrett mit Steinen auf dem Bildschirm darstellt und die Möglichkeit bietet, einzelne Steine oder Felder auszuwählen (siehe Abbildung 4.48). Es soll zeigen, wie man eine neue Widget-Klasse von der Klasse `QTableView` ableitet.

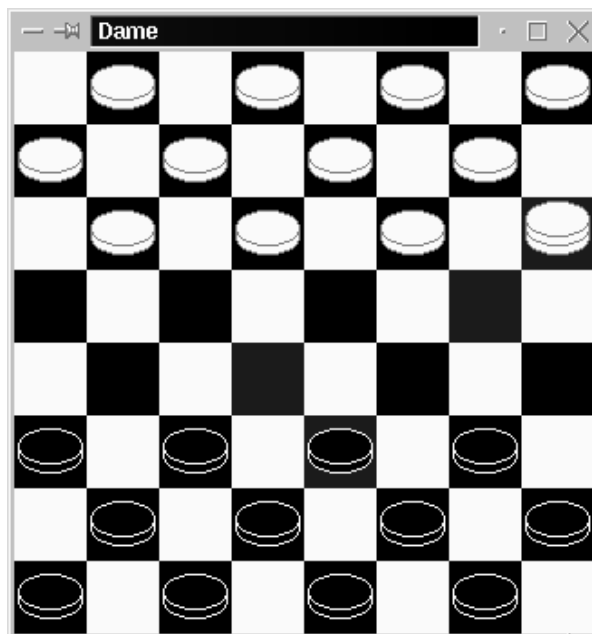


Abbildung 4-48 Dame-Spielbrett

Im KDE-Projekt gibt es übrigens bisher noch kein Programm für das Dame-Spiel. Wenn Sie also ein solches Programm entwickeln möchten, können Sie gern das hier beschriebene Widget benutzen. Mit geringfügigen Änderungen kann man dieses Widget auch zur Darstellung anderer Spiele wie Schach, Tic Tac Toe, Vier Gewinnt, Schiffeversenken oder Käsekästchen benutzen.

Ein Dame-Spiel ist folgendermaßen aufgebaut: Gespielt wird auf einem 8x8 Felder großen Brett mit abwechselnd schwarzen und weißen Feldern. Die weißen Felder sind für das Spiel nicht relevant. Auf den schwarzen Feldern können Spielsteine stehen, die entweder schwarz oder weiß sind und die entweder ein normaler Stein oder eine Dame sind.

Unser Widget soll nur die Darstellung des Spielbretts und der Figuren realisieren. Die Funktionalität der erlaubten Züge soll außerhalb des Widgets implementiert sein. Das Widget soll die Möglichkeit bieten, Felder oder Steine per Maus oder Tastatur auszuwählen und diese dann hervorgehoben darzustellen. Da es aber nicht entscheiden kann, welche Markierungen zulässig sind, leitet es die Tastatur- und Maus-Events nur geringfügig verarbeitet durch Signale weiter. Über zusätzliche Methoden können die Steine auf dem Spielfeld bewegt und einzelne Felder als markiert dargestellt werden.

Gehen wir auch hier alle fünf Schritte der Reihe nach durch:

1. Der Anwender kann mit der Maus auf das Spielfeld klicken oder die Tastatur benutzen. Diese Events wollen wir etwas aufbereiten und die Daten per Signal der Außenwelt mitteilen. Angeschlossene Slots können dann entscheiden, wie auf diese Events zu reagieren ist, indem sie beispielsweise Felder markiert darstellen lassen oder Steine auf dem Brett bewegen. Damit Felder markiert, aber auch einzelne Steine mit der Maus von einem Feld zum anderen gezogen werden können, benötigen wir drei Signale für die Meldung der drei Maus-Events: *mousePressEvent*, *mouseMoveEvent* und *mouseReleaseEvent*. Die Signale liefern dabei nicht die Mausposition, sondern das Feld, auf dem die Maus gerade steht. Für die Maus-Events wird nur die linke Maustaste berücksichtigt. Andere Maustasten haben keine Wirkung. Bei Tastatur-Events ist es nur wichtig festzustellen, wenn eine Taste betätigt wird, und nicht, wann sie losgelassen wird. Daher reicht hier ein Signal, das den Tastatur-Code und das erzeugte ASCII-Zeichen übergibt. So kann der angeschlossene Slot sehr einfach auf bestimmte Buchstaben oder auf die Pfeiltasten reagieren. Die Deklaration unserer Signalmethoden sieht nun so aus:

```
signals:
    void pressedAt (int x, int y);
    void movedTo (int x, int y);
    void releasedAt (int x, int y);
    void keyPressed (int key);
```

2. Wir benutzen eine Methode, die alle Markierungen löscht. Diese Methode bekommt den Namen *resetAllMarks* und wird als Slot definiert. Weiterhin gibt es eine Methode, *setField*, mit der wir einen Stein auf ein Feld des Bretts setzen können. Eine weitere Methode, *setMark*, dient dazu, ein Feld des Bretts hervorgehoben darzustellen. Unsere Methoden, die den Zustand des Widgets ändern, haben nun folgende Deklaration:

```
public:
    enum Piece {Empty, BlackMan, WhiteMan,
                BlackKing, WhiteKing};
    void setField (int x, int y, Piece contents);
    void setMark (int x, int y, bool mark);
public slots:
    void resetAllMarks ();
```

3. Als Basisklasse benutzen wir die Klasse *QTableView*, da diese Klasse bereits viel Funktionalität für die Darstellung von Tabellen enthält. So werden automatisch Rollbalken zum Widget hinzugefügt, wenn der Platz nicht ausreichen sollte. Das Innere des Widgets ist in Zellen unterteilt, die mit der virtuellen Methode *paintCell* einzeln gezeichnet werden können. In unserem Beispiel entspricht eine Zelle einem Feld auf dem Spielbrett.
4. Um die Maus- und Tastatur-Events verarbeiten und weiterleiten zu können, müssen wir die Event-Methoden *mousePressEvent*, *mouseMoveEvent*, *mouseReleaseEvent* und *keyPressEvent* überschreiben. Die Methode *keyReleaseEvent* benötigen wir nicht. Zum Zeichnen des Inhalts überschreiben wir die virtuelle Methode *paintCell* der Klasse *QTableView*. Diese Methode wird innerhalb der Methode *paintEvent* automatisch für jede Zelle aufgerufen, die vom Neuzeichnen betroffen ist. Die Methode *paintEvent* brauchen wir nicht mehr zu überschreiben, da sie bereits in *QTableView* geeignet überschrieben wurde.
5. Für die Größe eines Feldes auf dem Spielbrett wählen wir 40 Pixel. Die empfohlene Gesamtgröße, die *sizeHint* zurückgeben soll, ergibt bei einem 8x8 Felder großen Brett 320x320 Pixel. Als *sizePolicy* legen wir für Höhe und Breite den Wert *Maximum* fest, da es keinen Sinn macht, das Spielfeld größer zu machen – die einzelnen Felder vergrößern sich dabei nicht –, das Spielfeld aber durchaus auch in einem kleineren Widget bedient werden kann, indem man die Rollbalken des Fensters nutzt.

Das gesamte Listing für unsere Widget-Klasse sieht folgendermaßen aus:

```
class DraughtBoard : public QTableView {
    Q_OBJECT

public:
    DraughtBoard(QWidget *parent=0, const char *name=0);
    ~DraughtBoard() {}
```

```

enum Piece {Empty, BlackMan, WhiteMan,
            BlackKing, WhiteKing};
void setField (int x, int y, Piece contents);
Piece field (int x, int y);
void setMark (int x, int y, bool mark);

QSize sizeHint ()
{return QSize (320, 320);}
QSizePolicy sizePolicy ()
{return QSizePolicy (Maximum, Maximum);}

public slots:
    void resetAllMarks ();

signals:
    void pressedAt (int x, int y);
    void movedTo (int x, int y);
    void releasedAt (int x, int y);
    void keyPressed (int key);

protected:
    void paintCell (QPainter *p, int x, int y);
    void mousePressEvent (QMouseEvent *ev);
    void mouseMoveEvent (QMouseEvent *ev);
    void mouseReleaseEvent (QMouseEvent *ev);
    void keyPressEvent (QKeyEvent *ev);

private:
    Piece boardData [8][8];
    bool marks [8][8];
};

DraughtBoard::DraughtBoard (QWidget *parent,
                             const char *name)
: QTableView (parent, name)
{
    setTableFlags (Tbl_autoScrollBars | Tbl_smoothScrolling);
    setCellWidth (40);
    setCellHeight (40);
    setNumRows (8);
    setNumCols (8);

    int i, j;
    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++) {
            boardData [i][j] = Empty;
            marks [i][j] = false;
        }
}

```

```

}

void DraughtBoard::setField (int x, int y, Piece contents) {
    if (x < 1 || x > 8 || y < 1 || y > 8 ||
        field (x, y) == contents)
        return;
    boardData [x - 1][y - 1] = contents;
    updateCell (x - 1, y - 1, false);
}

DraughtBoard::Piece DraughtBoard::field (int x, int y) {
    if (x < 1 || x > 8 || y < 1 || y > 8)
        return Empty;
    else
        return boardData [x - 1][y - 1];
}

void DraughtBoard::setMark (int x, int y, bool mark) {
    if (x < 1 || x > 8 || y < 1 || y > 8 ||
        marks [x - 1][y - 1] == mark)
        return;
    marks [x - 1][y - 1] = mark;
    updateCell (x - 1, y - 1, false);
}

void DraughtBoard::resetAllMarks () {
    int i, j;
    for (i = 1; i <= 8; i++)
        for (j = 1; j <= 8; j++)
            setMark (i, j, false);
}

void DraughtBoard::paintCell (QPainter *p, int y, int x) {
    int w = cellWidth (x);
    int h = cellHeight (y);
    QColor bg;
    QPen line;
    QBrush coin;
    if (marks [x][y]) bg = blue;
    else if ((x + y) % 2) bg = black;
    else bg = white;
    p->fillRect (0, 0, w, h, bg);

    if (boardData [x][y] == Empty) return;

    if (boardData [x][y] == BlackMan ||
        boardData [x][y] == BlackKing)
        {coin = black; line = white;}
    else

```

```

    {coin = white; line = darkGray;}

    int w1 = w / 20;
    int w2 = w - 2 * w1;
    bool king = (boardData [x][y] == BlackKing ||
                 boardData [x][y] == WhiteKing);
    p->setBrush (coin);
    p->setPen (NoPen);
    p->drawEllipse (w1, h * 5 / 16, w2, h / 2);
    p->drawRect (w1, h * (king ? 5 : 7) / 16,
                w2, h * (king ? 2 : 1) / 8);

    p->setPen (line);
    p->drawEllipse (w1, h * (king ? 1 : 3) / 16,
                  w - 2 * w1, h / 2);
    p->drawArc (w1, h * 5 / 16, w2, h / 2, 2880, 2880);
    if (king)
        p->drawArc (w1, h * 3 / 16, w2, h / 2, 2880, 2880);
    p->drawLine (w1, h * (king ? 5 : 7) / 16,
                w1, h * 9 / 16);
    p->drawLine (w - w1 - 1, h * (king ? 5 : 7) / 16,
                w - w1 - 1, h * 9 / 16);
}

void DraughtBoard::mousePressEvent (QMouseEvent *ev) {
    emit (pressedAt (findCol (ev->x ()), findRow (ev->y ())));
}

void DraughtBoard::mouseMoveEvent (QMouseEvent *ev){
    emit (movedTo (findCol (ev->x ()), findRow (ev->y ())));
}

void DraughtBoard::mouseReleaseEvent (QMouseEvent *ev) {
    emit (releasedAt (findCol (ev->x ()), findRow (ev->y ())));
}

void DraughtBoard::keyPressEvent (QKeyEvent *ev) {
    emit (keyPressed (ev->key ()));
}

```

4.4.4 Beispiel: Das Anzeigefenster eines Grafikprogramms

Das Programm *KAWDraw* (*K-Addison-Wesley-Draw*) war ursprünglich als Beispielprogramm für dieses Buch gedacht. Am Ende war es aber zu komplex, um ausführlich in allen Einzelheiten erläutert zu werden.

KAWDraw ist ein Editor für Vektorgrafiken (siehe Abbildung 4.49), ähnlich wie *Killustrator* oder *Corel Draw*. Man kann einfache Grafikprimitive im Dokument

platzieren und verschieben, in der Größe ändern oder rotieren. Als Elemente stehen Linien, Rechtecke, Ellipsen, Texte und Bilder (Pixmaps) zur Verfügung. Außerdem kann man beliebige Linienarten und Füllmuster wählen. Die Elemente können zu Gruppen zusammengefasst werden.

Der Code ist zu umfangreich, um ihn hier abzdrukken, aber er ist auf der CD-ROM enthalten, die dem Buch beiliegt. Das Programm ist unter der Lizenz GPL veröffentlicht, so dass Sie den Code für eigene freie Programme benutzen können.

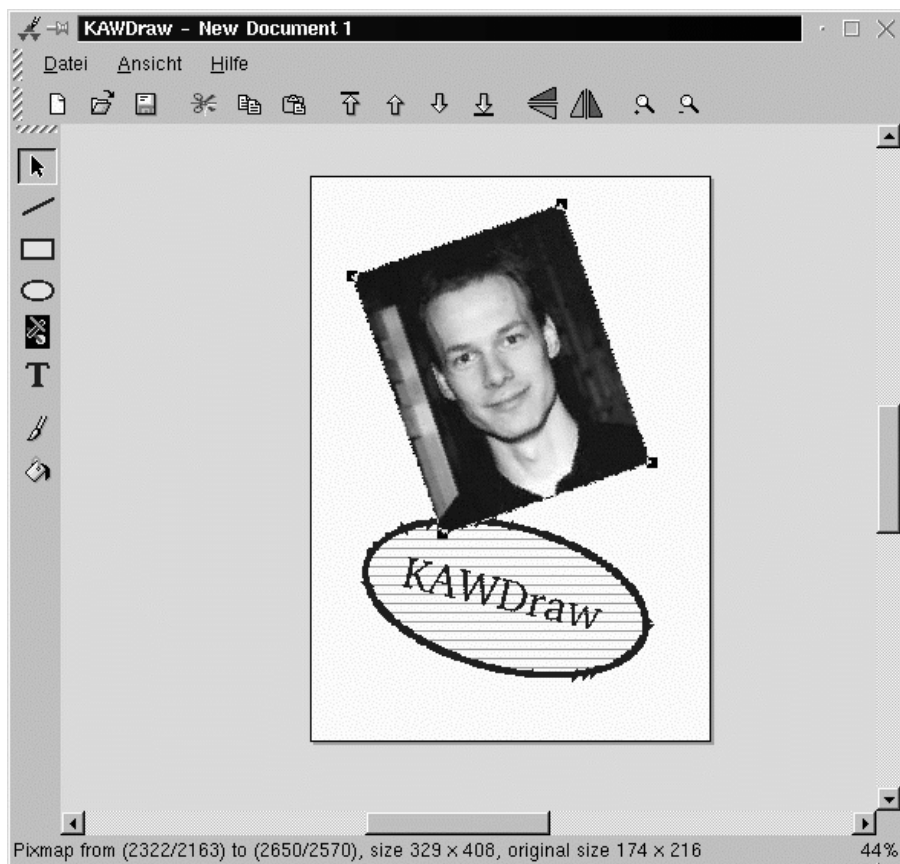


Abbildung 4-49 KAWDraw, in der Mitte die selbst definierte Anzeigeklasse

Der Anzeigebereich des Programms (siehe Kapitel 3.5.3, *Der Anzeigebereich*) ist eine selbst definierte Klasse, denn keine der KDE- oder Qt-Klassen bietet die nötige Funktionalität. Wir wollen hier nur kurz auf die Besonderheiten dieser Klasse eingehen:

Das Widget kann sich in verschiedenen Modi befinden, je nachdem, welche Aktion der Anwender gerade ausführt:

- *Auswählen und Verschieben* – Dieses ist der Grundmodus. Er wird angezeigt durch den ausgewählten Pfeil in der Werkzeugleiste am linken Rand.
- *Zeichnen einfacher Objekte (Linien, Rechtecke, Ellipsen)* – Durch Anwählen des entsprechenden Icons in der Werkzeugleiste wird dieser Modus aktiv. In diesem Modus zieht der Anwender ein Rechteck auf dem Bildschirm auf, in das das gewünschte Objekt dann eingefügt wird.
- *Einfügen von Text und Bildern* – In diesem Modus klickt der Anwender auf eine Stelle des Dokuments, an der er den Text oder das Bild platziert haben möchte. Dann öffnet sich ein Dialog, in dem der Anwender den Text und die Schriftart wählen bzw. die Bilddatei öffnen kann.

Je nach Modus reagiert das Anzeigefenster anders auf Maus-Events. Betrachten wir hier einmal den Modus *Auswählen und Verschieben*. Er ist der komplexeste der drei Modi. In diesem Modus wird unterschieden zwischen »Klicken« (d.h. Drücken und Loslassen der linken Maustaste an der gleichen Position) und »Ziehen« (d.h. Drücken der linken Maustaste, Bewegen der Maus, und Loslassen an einer anderen Stelle):

- *Klicken ohne Sondertaste* – Klickt der Anwender auf ein Objekt des Dokuments, so ist dieses anschließend ausgewählt (z.B. zum Ausschneiden oder Kopieren in die Zwischenablage). Alle vorher ausgewählten Objekte sind nun nicht mehr ausgewählt. Klickt er auf einen freien Bereich des Dokuments, wird die Auswahl aufgehoben: Kein Objekt ist mehr ausgewählt.
- *Klicken mit gedrückter `[Strg]`-Taste* – In diesem Fall wird die Auswahl erweitert. Das Objekt, auf das der Anwender klickt, ändert seinen Zustand von »nicht ausgewählt« auf »ausgewählt« oder umgekehrt. Sind andere Objekte ausgewählt, so bleiben sie es. Ein Klicken mit gedrückter `[Strg]`-Taste auf einem freien Bereich des Dokuments hat keine Auswirkung.
- *Ziehen ohne Sondertaste* – Ist die Maus beim Drücken der linken Maustaste auf einem nicht ausgewählten Objekt, so wird dieses Objekt ausgewählt (alle anderen sind nicht mehr ausgewählt) und wird mit der Maus zusammen verschoben. Ist die Maus dagegen beim Drücken auf einem ausgewählten Objekt, so wird dieses Objekt sowie alle anderen ausgewählten Objekte mit der Maus verschoben. Ist die Maus auf keinem Objekt, so wird ein rechteckiger Rahmen gezogen und alle in diesem Rahmen liegenden Objekte (und nur diese) sind ausgewählt.
- *Ziehen mit gedrückter `[Strg]`-Taste* – Hier verhält sich das Hauptfenster ähnlich wie beim Ziehen ohne Sondertaste. Allerdings wird die Auswahl in jedem Fall nur erweitert.

- *Ziehen eines Markierungspunkts* – Um alle ausgewählten Objekte wird das umschließende Rechteck mit einer gestrichelte Linie eingezeichnet. Ist genau ein Objekt ausgewählt, so werden in den Ecken dieses Rechtecks zusätzlich Markierungspunkte eingezeichnet. Zieht der Anwender einen dieser Markierungspunkte, so ändert er damit die Größe des Objekts.

Beim Verschieben eines oder mehrerer Objekte kann zusätzlich noch die -Taste gedrückt werden. In diesem Fall geschieht die Verschiebung nur horizontal oder vertikal, aber nicht diagonal.

Wie Sie sehen, sind die Fälle recht komplex, die bei der Auswertung der Maus-Events unterschieden werden müssen. Man könnte die Komplexität verringern, indem man diesen Modus in zwei verschiedene Modi aufteilt: Einen Modus zum Auswählen und einen Modus zum Verschieben. Allerdings ist der oben beschriebene Standard inzwischen sehr weit verbreitet. Der Anwender ist es gewohnt, im gleichen Modus auswählen und markieren zu können.

Besondere Beachtung muss man auch der Unterscheidung von *Klicken* und *Ziehen* widmen: Wann wollte der Anwender etwas anklicken und ist dabei nur etwas mit der Maus verrutscht? Wann wollte er etwas um wenige Pixel verschieben? Die Klasse in *KAWDraw* benutzt dazu das folgende Kriterium: Wurde die Maustaste länger als 300 ms gedrückt oder wurde dabei ein Weg von mehr als zehn Pixeln zurückgelegt, wird die Aktion als *Ziehen* interpretiert, sonst als *Klicken*.

Die anderen beiden Modi – *Zeichnen einfacher Objekte* und *Einfügen von Text und Bildern* – sind einfacher. In ihnen werden nicht so viele verschiedene Fälle unterschieden.

Eine zusätzliche Eigenschaft gilt in allen Modi: Wird die Maus bei gedrückter Maustaste aus dem Fenster herausbewegt, so verschiebt sich der betrachtete Ausschnitt automatisch in die Richtung des Mauszeigers. Dazu prüfen wir in der Methode *viewportMouseMoveEvent* (die hier statt *mouseMoveEvent* benutzt wird), ob sich der Mauszeiger außerhalb des Fensters befindet. In diesem Fall starten wir einen Timer, der in regelmäßigen Zeitintervallen die Methode *autoScroll* aufruft, die den Ausschnitt verschiebt. Der Timer wird gestoppt, sobald die Maustaste wieder losgelassen wird oder der Mauszeiger wieder innerhalb des Fensters ist.

Wie Sie sehen, muss dieses Widget also eine Vielzahl von Unterscheidungen treffen, wie die einzelnen Maus-Events zu behandeln sind. Wir wollen die Techniken dazu hier nicht weiter erläutern. Sie können sich jedoch im Listing ansehen, wie die einzelnen Unterscheidungen vorgenommen werden.

Die Klasse *DrawWindow* – so nennen wir unsere selbst definierte Klasse für den Anzeigebereich – ist von der Klasse *QScrollView* abgeleitet, die bereits die Arbeit übernimmt, einen Ausschnitt aus einer größeren Grafik auszuwählen. Dieser Ausschnitt kann mit Rollbalken verschoben werden. *QScrollView* interpretiert die

Maus-Events selbst, weshalb wir die Event-Methoden hier nicht überschreiben wollen. Dafür ruft *QScrollView* seinerseits die virtuellen Methoden *viewportMouseEvent*, *viewportMouseMoveEvent* und *viewportMouseReleaseEvent* auf, die ganz analog zu den entsprechenden Maus-Event-Methoden genutzt werden können. Diese Methoden überschreiben wir nun in unserer Klasse *DrawWindow*.

Ebenso überschreiben wir nicht die Methode *paintEvent*, sondern die Methode *drawContentsOffset*. Diese Methode wird von der Klasse *QScrollView* immer dann aufgerufen, wenn ein Teil des Ausschnitts neu gezeichnet werden muss. Dabei gibt die Methode *drawContentsOffset* in zwei Parametern an, an welcher Stelle der Ausschnitt liegt, der angezeigt werden soll. Anhand dieses Offsets bestimmen wir, um wie viel wir die Objekte, die wir zeichnen wollen, verschieben müssen, damit gerade der gewünschte Ausschnitt gezeichnet wird.

Zum Zeichnen des Ausschnitts bedienen wir uns eines verbreiteten Tricks, um das Neuzeichnen möglichst effizient zu machen: Wir zeichnen zunächst alles in ein *QPixmap*-Objekt und kopieren den Inhalt dieses Objekts auf den Bildschirm. Falls nun ein Teil des Bildschirms wiederhergestellt werden soll, brauchen wir nicht alle Zeichenoperationen zu wiederholen. Es reicht, wenn wir den Inhalt des *QPixmap*-Objekts erneut auf den Bildschirm kopieren. In unserem Fall machen wir das *QPixmap*-Objekt insgesamt doppelt so hoch und doppelt so breit wie das Fenster, das den Ausschnitt anzeigt, so dass das *QPixmap*-Objekt im Normalfall in jeder Richtung um 50% über das Fenster hinausragt. So kann der Ausschnitt um bis zu 50% in jede Richtung verschoben werden, ohne dass es nötig wäre, die Elemente erneut zu zeichnen. Es muss nur ein anderer Teil des *QPixmap*-Objekts auf den Bildschirm kopiert werden. Erst wenn der Ausschnitt um mehr als 50% in eine Richtung verschoben wird, muss das *QPixmap*-Objekt neu gezeichnet werden.

Wir gehen hier nicht weiter auf die Details im Listing ein. Sie können sich bei Interesse das Listing des Programms auf der CD-ROM anschauen, die dem Buch beiliegt. Dieses Listing enthält viele Kommentare, die Ihnen hoffentlich die Orientierung in der recht komplexen Klasse erleichtern.

4.5 Flimmerfreie Darstellung

Wird ein Widget neu gezeichnet, kann es zu einem Flimmern kommen, wenn einzelne Pixel oder ganze Bereiche mehrmals mit verschiedenen Farben übermalt werden. Der schnelle Farbwechsel wirkt störend, insbesondere bei Widgets, die sehr oft neu gezeichnet werden müssen, zum Beispiel ein Rollbalken, der mit der Maus gezogen wird. Es gibt eine Reihe von Techniken, um dieses Flimmern zu vermeiden oder zumindest zu verringern.

4.5.1 Hintergrundfarbe

Für jedes Widget können Sie mit der Methode *setBackgroundMode* eine Hintergrundfarbe oder mit *setBackgroundPixmap* ein Hintergrundbild festlegen. Der X-Server sorgt in diesem Fall dafür, dass alle neu zu zeichnenden Bereiche mit dieser Farbe bzw. diesem Bild ausgefüllt werden, bevor die Event-Methode *paintEvent* aufgerufen wird.

Wenn Sie ein eigenes Widget entwickeln, bestimmen Sie die Farbe, die als Hintergrundfarbe benutzt werden soll, die also den größten Teil des Widgets ausfüllen wird. Die Default-Einstellung für die Hintergrundfarbe ist der Eintrag *Background* aus der Widget-Farbtabelle (siehe Kapitel 4.1.8, *Die Widget-Farbpalette*). Sie ist in der Regel hellgrau. Viele Widgets, die sich von ihrer Umgebung abheben sollen, benutzen aber beispielsweise den Eintrag *Base*, der normalerweise die Farbe Weiß enthält, so zum Beispiel *QLineEdit*, *QMultiLineEdit* oder *QListBox*.

Wenn Sie ebenfalls einen anderen Paletteneintrag des Widgets als Hintergrundfarbe benutzen wollen, stellen Sie diese Hintergrundfarbe mit *setBackgroundMode* ein. Die Alternative, die neu zu zeichnende Fläche selbst mit der gewünschten Farbe zu füllen, würde ein Flimmern verursachen, da zuerst die eingestellte Hintergrundfarbe und anschließend die von Ihnen gewünschte Hintergrundfarbe gezeichnet würde.

Wenn Ihr Widget keine einheitliche Hintergrundfarbe besitzt – beispielsweise wie ein Schachbrett etwa gleich viele schwarze und weiße Bereiche –, können Sie das Flimmern reduzieren, indem Sie *setBackgroundMode* mit dem Wert *NoBackground* aufrufen. So weisen Sie den X-Server an, einen neu zu zeichnenden Bereich nicht mit einer Farbe zu füllen, sondern ihn zunächst unverändert zu lassen. In der Methode *paintEvent* können Sie nun den Hintergrund von Hand für die verschiedenen Bereiche füllen.

4.5.2 Begrenzung des Zeichenausschnitts

Oft wird ein Pixel oder ein Bereich innerhalb einer komplexen Zeichnung mehrfach in verschiedenen Farben gezeichnet. Als Beispiel soll hier die folgende *paintEvent*-Methode eines Widgets dienen, das die Grafik aus Abbildung 4.50 zeichnet:

```
void MyWidget::paintEvent (QPaintEvent *) {
    QPainter p (this);
    p.setPen (NoPen);
    p.setClipRect (ev->rect());
    int size = 45;
    for (int i = 0; i < 10; i++)
    {
        QRect part (100 - size, 50 - size, 2 * size, 2 * size);
```

```

// Schwarzes Quadrat zeichnen
p.setBrush (black);
p.drawRect (part);
// Weißen Kreis zeichnen
p.setBrush (white);
p.drawEllipse (part);
// size durch Wurzel 2 teilen
size = size * 707 / 1000;
}
}

```



Abbildung 4-50 Zeichnung mit mehrfach übermalten Bereichen

In diesem Widget wird zunächst ein schwarzes Quadrat gezeichnet, darin ein weißer Kreis, darin ein kleineres schwarzes Rechteck, darin wieder ein weißer Kreis usw. Der mittlere Bereich wird dabei sehr oft neu gezeichnet, immer abwechselnd in Schwarz und Weiß. Das führt zu einem starken Flimmern, das sehr störend wirkt, insbesondere, wenn das Widget oft neu gezeichnet wird.

Um zu verhindern, dass der mittlere Bereich jedes Mal übermalt wird, beschränken wir den Ausschnitt, der gezeichnet werden soll, mit *setClipRegion*. Mit dieser Methode sparen wir jeweils die Mitte der Zeichnung aus. So zeichnet jeder Befehl nur den Bereich, der nachher nicht mehr verändert wird. Das Listing dazu sieht so aus:

```

void paintWidget::paintEvent (QPaintEvent *) {
    QPainter p (this);

    p.setPen (NoPen);
    int size = 45;
    for (int i = 0; i < 10; i++)
    {
        QRect part (100 - size, 50 - size, 2 * size, 2 * size);
        p.setBrush (black);
        // Die nächstkleinere Ellipse aussparen
        p.setClipRegion (QRegion (ev->rect()).
            subtract (QRegion (part, QRegion::Ellipse)));
        p.drawRect (part);
    }
}

```

```

    p.setBrush (white);
    size = size * 707 / 1000;
    QRect newPart (100 - size, 50 - size,
                   2 * size, 2 * size);
    // Das nächstkleinere Rechteck aussparen
    p.setClipRegion (QRegion (ev->rect()).
                     subtract (QRegion (newPart)));
    p.drawEllipse (part);
}
}

```

Das Zeichnen in einen gewählten Ausschnitt ist langsamer, was aber meist nicht ins Gewicht fällt. Das Flimmern während des Zeichnens ist vollständig beseitigt.

4.5.3 Zeichnen in ein *QPixmap*-Objekt

Eine gängige und auch sehr einfache Lösung besteht darin, die Zeichnung zunächst unsichtbar innerhalb eines *QPixmap*-Objekts durchzuführen, um anschließend das gesamte Objekt mit einem Befehl auf den Bildschirm zu kopieren. Dabei wird jedes Pixel im neu zu zeichnenden Bildschirmausschnitt genau einmal neu gezeichnet.

Da *QWidget* und *QPixmap* beide von *QPaintDevice* abgeleitet sind, kann man in beide mit einem Objekt der Klasse *QPainter* zeichnen. Beim Umstellen auf dieses Konzept ist daher nicht viel zu ändern. Gehen wir beispielsweise von einer *paintEvent*-Methode aus, die folgende Gestalt hat:

```

void MyWidget::paintEvent (QPaintEvent *ev)
{
    QPainter p (this);
    p.setClipRect (ev->rect());

    // Zeichenoperationen mit p ausführen
    ....
}

```

Unsere neue Methode, die zunächst in ein *QPixmap*-Objekt zeichnet, sieht dann so aus:

```

void MyWidget::paintEvent (QPaintEvent *ev)
{
    // QPixmap mit passender Größe anlegen
    QPixmap pix (ev->rect()->size());

    // p zeichnet jetzt in pix, nicht in this. Es übernimmt
    // dabei aber die Einstellungen aus this, z.B.
    // Zeichensatz, Hintergrundfarbe oder -bild usw.
    QPainter p (&pix, this);
}

```

```

// Koordinaten auf diesen Ausschnitt umrechnen
p.setWindow (ev->rect());
// Pixmap mit Hintergrund füllen
p.eraseRect (ev->rect());

// Zeichenoperationen mit p ausführen
....

// Sicherstellen, dass alle Zeichenoperationen vor dem
// Kopieren ausgeführt wurden
p.flush();

// Inhalt in das Widget kopieren
bitBlt (this, ev->rect().topLeft(), &pix);
}

```

In unserem Beispiel von oben sieht die *paintEvent*-Methode also so aus:

```

void MyWidget::paintEvent (QPaintEvent *ev)
{
    QPixmap pix (ev->rect()->size());
    QPainter p (&pix, this);

    p.setWindow (ev->rect());
    p.eraseRect (ev->rect());

    p.setPen (NoPen);
    p.setClipRect (ev->rect());
    int size = 45;
    for (int i = 0; i < 10; i++)
    {
        QRect part (100 - size, 50 - size, 2 * size, 2 * size);
        // Schwarzes Quadrat zeichnen
        p.setBrush (black);
        p.drawRect (part);
        // Weißen Kreis zeichnen
        p.setBrush (white);
        p.drawEllipse (part);
        // size durch Wurzel 2 teilen
        size = size * 707 / 1000;
    }

    p.flush();
    bitBlt (this, ev->rect().topLeft(), &pix);
}

```

Auch diese Lösung verhindert das Flimmern vollständig. Da Zeichnungen in ein *QPixmap*-Objekt meist ebenso schnell durchgeführt werden können wie Zeichnungen in ein Widget, ist die Verzögerung hier minimal. Der Nachteil der

Methode ist, dass zusätzlicher Speicher im X-Server benötigt wird, um die Daten des *QPixmap*-Objekts abzulegen. Besonders bei sehr großen Widgets kann der X-Server dann deutlich langsamer werden. In einigen Fällen kann der Speicher des X-Servers sogar begrenzt sein, so dass kein *QPixmap*-Objekt mehr angelegt werden kann.

4.5.4 Bewegte Objekte mit *QCanvas*

Bei einem Action-Spiel – wie zum Beispiel Pacman oder Asteroids – hat man viele, sich schnell bewegende Objekte. Diese Objekte flimmerfrei auf dem Bildschirm darzustellen ist oft schwierig. Wird nämlich ein Objekt bewegt, muss zunächst das Objekt gelöscht und der Bildschirminhalt an dieser Stelle wieder rekonstruiert werden. Anschließend wird das Objekt an der neuen Position gezeichnet. Dabei wird das Objekt kurz unsichtbar. Wenn man dazu noch eine Reihenfolge der Objekte beachten muss, weil einige Objekte andere überdecken, kann die Berechnung sehr aufwendig werden.

Einen ganz anderen Weg geht dabei die Klasse *QCanvas*. Diese Klasse zeichnet das Widget nicht jedes Mal neu, wenn sich ein Objekt bewegt hat, sondern in regelmäßigen Zeitintervallen, meist kleiner als 25 ms, so dass die Bewegungen ruckfrei erscheinen. Wird die Position eines Objekts verändert, merkt sich die Klasse die neue Position. Wenn das nächste Bildschirm-Update ansteht, teilt sie das Fenster in mehrere kleine Rechtecke auf. Bei jedem Rechteck wird entschieden, ob sich der Inhalt verändert hat. Ist dies der Fall, wird dieses Rechteck in einem *QPixmap*-Objekt neu gezeichnet und anschließend auf den Bildschirm kopiert. Auf diese Weise können alle Objekte flimmerfrei bewegt werden.

Dieses Konzept ist optimal geeignet, wenn viele Objekte dargestellt werden sollen, die sich schnell bewegen, z.B. in Spielen oder großflächigen gezeichneten Animationen, oder wenn die dargestellten Objekte vom Anwender mit der Maus gezogen werden.

Dargestellt werden können Objekte der Klasse *QCanvasItem* oder Unterklassen davon. Es gibt bereits fertige Unterklassen zum Zeichnen von Linien, Rechtecken, Polygonen, Ellipsen, Text oder Pixmaps. Sie können weitere Objekte darstellen lassen, indem Sie eine eigene Unterklasse von *QCanvasItem* bilden.

Die Klasse *QCanvas* selbst verwaltet nur die Objekte, die dargestellt werden sollen. Angezeigt werden sie mit der Widget-Klasse *QCanvasView*, die einen bestimmten Ausschnitt aus einem *QCanvas*-Objekt darstellt. Sie können mehrere *QCanvasView*-Objekte auf das gleiche *QCanvas*-Objekt zugreifen lassen, um so verschiedene Ausschnitte darzustellen.

Die Online-Referenz der Qt-Bibliothek enthält genauere Informationen zur Nutzung dieser Klassen. Dort finden Sie auch ein sehr gutes Beispielprogramm.

4.6 Klassendokumentation mit *doxygen*

Eine Klasse, die nicht oder nur schlecht dokumentiert ist, kann von anderen Programmierern außer dem Entwickler der Klasse kaum genutzt werden. Nur in den seltensten Fällen sind die Methoden- und Parameternamen eindeutig genug, um alle Missverständnisse auszuschließen. Sich durch den Quellcode einer Klasse zu quälen, um herauszufinden, wie die Klasse zu benutzen ist, ist meist auch nicht möglich. Kommentare im Quelltext können da weiterhelfen. Noch besser ist in jedem Fall eine eigenständige Dokumentation der Klasse. Das bedeutet aber zum einen erheblichen Mehraufwand für den Entwickler der Klasse, und weiterhin ist es oftmals nur schwer möglich, Klasse und Dokumentation gleichzeitig immer auf dem aktuellsten Stand zu halten. Jede Änderung an der Klasse muss unmittelbar auch in der Dokumentation vermerkt werden.

Eine Lösung des Problems bieten Systeme, bei denen der Entwickler der Klasse den Kommentar direkt in den Quellcode der Klasse schreibt. Mit Hilfe eines Tools werden diese Kommentare dann aus dem Quellcode extrahiert und zu einem übersichtlichen Dokument zusammengestellt. Querverweise innerhalb der Dokumentation – zu anderen Methoden der Klasse oder auch zu anderen Klassen – können dabei ebenfalls automatisch hergestellt werden. Auch die exzellente Klassendokumentation der Qt-Bibliothek wurde mit einem solchen Tool erstellt.

Wenn Sie von Ihnen entwickelte Klassen anderen Programmieren zur Verfügung stellen wollen, sollten Sie ebenfalls ein solches Tool benutzen. Aber auch für Klassen, die Sie nur in eigenen Projekten einsetzen, ist ein solches Tool praktisch. So können Sie sich auch nach Wochen und Monaten mit wenigen Mausklicks in Erinnerung rufen, wie Ihre Klasse zu benutzen ist.

Für die Programmiersprache JAVA hat sich das Tool *JavaDoc* durchgesetzt, und auch für C++ sind inzwischen einige Programme verfügbar, die die Dokumentation aus dem Quelltext extrahieren, beispielsweise *DOC++* oder *cocoon*. Auch im KDE-Projekt ist ein eigenes Tool *KDoc* zu diesem Zweck entwickelt worden. Es scheint jedoch so, als würde dieses Tool kaum noch weiterentwickelt, obwohl es noch eine Reihe von Schwachstellen und Bugs enthält. Stattdessen setzt sich das Tool *doxygen* immer mehr durch, da es nahezu vollständig kompatibel zu *KDoc*, aber sehr viel ausgereifter ist und ein Vielfaches an Möglichkeiten bietet. Außerdem liegt *doxygen* eine ausgezeichnete und ausführliche Dokumentation bei, während *KDoc* nur mit einem kleinen erklärenden Beispiel ausgestattet ist. Zur Zeit wird die Klassendokumentation der KDE-Bibliotheken noch mit *KDoc* erstellt. Falls Sie also zu *KDoc* kompatibel bleiben müssen, so können Sie in Kapitel 4.6.6, *Kompatibilität zu KDoc*, nachlesen, was Sie dazu beachten sollten.

4.6.1 Installation von *doxygen*

Die Installation von *doxygen* ist in der Regel unproblematisch. *doxygen* benutzt einige Hilfsklassen der Qt-Bibliothek. Aus diesem Grund muss die Qt-Bibliothek bereits installiert sein. *doxygen* kann dabei sowohl die Version Qt 1.44, als auch Qt 2.0 oder neuere Versionen benutzen.

Die CD, die diesem Buch beiliegt, enthält *doxygen* in der Version 1.2.1 als Quelltext und als vorkompilierte Version für Linux (mit statisch gelinkter Qt-Bibliothek). Auch die Dokumentation zum Programm liegt bei. Die jeweils aktuellste Version von *doxygen* sowie Informationen zum Programm können Sie auf der *doxygen*-Homepage unter <http://www.stack.nl/~dimitri/doxygen/> erhalten.

Wenn Sie die Quelltext-Version selbst kompilieren wollen, müssen Sie die Qt-Bibliothek bereits installiert haben. Dazu kann sowohl eine alte Qt 1.44-Bibliothek als auch Qt 2.0 oder neuer benutzt werden. Ebenso müssen die Programme *flex*, *bison*, *make* und *perl* installiert sein, was aber bei nahezu allen Linux-Distributionen bereits standardmäßig der Fall ist. Um den vollen Funktionsumfang auszuschöpfen, sollten Sie ebenfalls *LaTeX*, *GhostScript* und das *Graph Visualization Toolkit* installiert haben. Auch das ist bei fast allen Linux-Distributionen bereits geschehen.

Nachdem Sie das *doxygen*-Paket entpackt haben, können Sie es mit dem bekannten Dreizeiler erstellen und installieren:

```
% ./configure
% make
% make install
```

Anschließend ist *doxygen* einsatzbereit.

4.6.2 Format der *doxygen*-Kommentare im Quelltext

Da wir die Dokumentation in den Quelltext einfügen wollen, darf diese die Kompilierbarkeit des Programms natürlich nicht beeinflussen. Daher wird sie in C++-Kommentaren abgelegt. Um *doxygen* zu signalisieren, dass dieser Kommentar zur Dokumentation gehören soll, beginnt ein solcher Kommentar entweder mit `»/**«` (für mehrzeiligen Text) oder mit `»/*!«` (für nur eine einzelne Zeile). Syntaktisch handelt es sich also um einen normalen Kommentar, den der Compiler überspringt.

Alternativ darf ein Dokumentationskommentar auch mit `»/*!»` oder `»/*!«` beginnen, so wie es in der Source-Dokumentation bei Qt gemacht wird. Um kompatibel zu *KDoc* zu bleiben, sollten Sie allerdings darauf verzichten.

Die Dokumentation für *doxygen* wird in der Regel in den Header-Dateien `»*.h«` untergebracht. Die Code-Dateien `»*.cpp«` bleiben unverändert. Ein *doxygen*-Kom-

mentar steht dabei *immer unmittelbar vor* der Deklaration der Klasse oder Methode, die näher beschrieben werden soll. Zwischen dem Kommentar und dem Beginn der Deklaration dürfen sich nur Zeilenvorschübe, Leerzeichen und Tabulatoren befinden.

Die Dokumentierung für eine Klasse und eine ihrer Methoden kann beispielsweise folgendermaßen aussehen:

```
/** This class does everything you ever dreamed of. It
    proves that P = NP, calculates the largest prime ever
    found and can even wash the dishes!

    Have a lot of fun with it!
*/
class Everything
{
public:
    /** This is the constructor of our nice class. It
        allocates 15 TByte memory as a hash table.
    */
    Everything::Everything ();
    ....
};
```

doxygen wird aus dieser Datei zwei Beschreibungen extrahieren, die erste zur Klasse *Everything*, die zweite zum Konstruktor der Klasse. Ganz analog können auch die anderen Methoden mit einer Beschreibung versehen werden. Aber nicht nur Klassen und Methoden werden von *doxygen* erfasst. Auf die gleiche Art können Sie Attribut-Variablen, globale Variablen, globale Funktionen und Aufzählungstypen (*enum*) mit einem Dokumentationstext versehen.

Der Kommentartext ist hier – wie meist üblich – in englischer Sprache verfasst. Falls Sie Ihre Klassen ins Internet stellen wollen, so sollten Sie sich daran halten. Schreiben Sie die Klassen dagegen nur für den Eigengebrauch, ist das natürlich nicht nötig.

Die Zeilenumbrüche in einem *doxygen*-Kommentar werden ignoriert. Um einen echten Absatz zu formatieren, fügen Sie einfach eine Leerzeile ein. Die Beschreibung der Klasse *Everything* besteht daher aus zwei Absätzen, die Beschreibung des Konstruktors nur aus einem Absatz.

Weiterhin ist es erlaubt, jede Zeile eines längeren Kommentars mit einem Stern »*« zu beginnen, so wie es insbesondere viele C-Programmierer gewohnt sind. *doxygen* ignoriert einen Stern als erstes Zeichen einer Zeile. Der folgende Kommentar führt also zum selben Ergebnis wie der oben angegebene:

```
/** This class does everything you ever dreamed of. It
 * proves that P = NP, calculates the largest prime ever
 * found and can even wash the dishes!
 *
 * Have a lot of fun with it!
 */
```

Statt vor der Deklaration kann ein *doxygen*-Kommentar auch hinter der Deklaration eines Bezeichners stehen. Dazu lassen Sie den *doxygen*-Kommentar mit `»/**<«` bzw. `»/// beginnen. Besonders häufig wird das bei Aufzählungstypen verwendet, um die einzelnen Werte zu dokumentieren:`

```
/** This enumeration type is used to report
    the probability of the result. */
enum ResultState
{
    Correct,    ///< guaranteed to be right
    Likely,     ///< right with probability >80%
    Maybe,      ///< right with probability >50%
    Unknown,    ///< maybe wrong, maybe right
    Incorrect   ///< guaranteed to be wrong
}
```

4.6.3 Erzeugen der Dokumentation

Nachdem Sie Ihr Listing so mit Dokumentationskommentaren versehen haben, können Sie *doxygen* diese Informationen aus den Quellen extrahieren lassen. Diese einfachen Techniken reichen bereits aus, um eine Klasse komfortabel mit einer vollständigen und übersichtlichen Dokumentation zu versehen. *doxygen* kann die Dokumentation dabei in vielen verschiedenen Formaten erzeugen. Die am häufigsten benutzten sind dabei *HTML* und *PDF*, die ein sehr effizientes Navigieren in der Dokumentation erlauben, sowie *LaTeX* und *PostScript*, um ein gedrucktes Handbuch zu erzeugen.

In einer Konfigurationsdatei können Sie genau festlegen, welche Dateien durchsucht und welche Dokumentationen erzeugt werden sollen. Eine Konfigurationsdatei kann sich auf beliebig viele Quelltext-Dateien beziehen, die auch in verschiedenen Verzeichnissen liegen können. So können Sie die Dokumentation für ein ganzes Projekt – zum Beispiel eine Bibliothek mit einer Vielzahl von Klassen – mit einer einzigen Konfigurationsdatei erzeugen lassen. Sie müssen daher zunächst diese Konfigurationsdatei erzeugen. Starten Sie dazu *doxygen* mit dem Parameter `»-g«`:

```
% doxygen -g
```

Dadurch wird automatisch das Grundgerüst der Konfigurationsdatei mit dem Namen *Doxyfile* im aktuellen Verzeichnis angelegt.

Dieses Gerüst können Sie nun mit einem beliebigen Texteditor ändern oder ergänzen. In der Regel reicht es, wenn Sie die Zeilen *PROJECT_NAME* und *INPUT* ergänzen, da die anderen Werte bereits sinnvoll vordefiniert sind.

Bei *PROJECT_NAME* tragen Sie den Namen des Projekts ein, also zum Beispiel den Namen der Bibliothek, die dokumentiert werden soll. Bei *INPUT* geben Sie an, welche Dateien nach einer Dokumentation durchsucht werden sollen. (*INPUT* befindet sich etwa in der Mitte der Konfigurationsdatei.) Sie können hier einzelne Dateien angeben, optional mit relativer oder absoluter Pfadangabe. Diese Vorgehensweise eignet sich am besten, wenn Sie nur eine oder ein paar Klassen verarbeiten lassen. Wollen Sie dagegen viele Klassen verarbeiten, so können Sie bei *INPUT* auch ein oder mehrere Verzeichnisse angeben, die vollständig nach dokumentierten Dateien durchsucht werden sollen. Bei *FILE_PATTERNS* können Sie nun zusätzlich einschränken, welche Dateitypen durchsucht werden sollen. Liegen zum Beispiel alle Dokumentationen in Header-Dateien mit der Dateierweiterung *.h* im aktuellen Verzeichnis, so können Sie folgende Einstellungen benutzen:

```
INPUT          = .
FILE_PATTERNS  = *.h
```

Nachdem Sie diese Ergänzungen in der Konfigurationsdatei *Doxyfile* vorgenommen haben, können Sie die Dokumentation automatisch erzeugen lassen:

```
% doxygen Doxyfile
```

doxygen untersucht nun in einem ersten Schritt alle angegebenen Dateien auf enthaltene C++-Definitionen (Klassen, Methoden, Aufzählungstypen usw.). Anschließend werden im aktuellen Verzeichnis (bzw. in dem Verzeichnis, das in der Konfigurationsdatei unter *OUTPUT_DIRECTORY* angegeben wurde) die Unterverzeichnisse *html*, *latex*, *man* und *rtf* erzeugt. In diese Unterverzeichnisse werden anschließend die erzeugten Dokumentationsdateien im entsprechenden Format abgelegt.

Das Verzeichnis *html* enthält also die Dokumentation im HTML-Format. Die Startseite lautet *./html/index.html*. Von dieser Seite aus kann man sich über einige Links zu der Dokumentation der einzelnen Klassen vorarbeiten. In Abbildung 4.51 sehen Sie, wie die erzeugte HTML-Dokumentation unserer Beispielsklasse *Everything* aussieht.

Im Verzeichnis *latex* befindet sich die gleiche Dokumentation im LaTeX-Format. Zusätzlich wird die Datei *Makefile* in diesem Verzeichnis erzeugt, mit deren Hilfe Sie sehr einfach Handbücher in den Formaten *DVI*, *PS* und *PDF* erzeugen können. Tabelle 4.3 zeigt die entsprechenden Aufrufe von *make*. Dabei werden einige Hilfsprogramme benutzt, die auf den meisten Linux-Systemen aber bereits installiert sind.

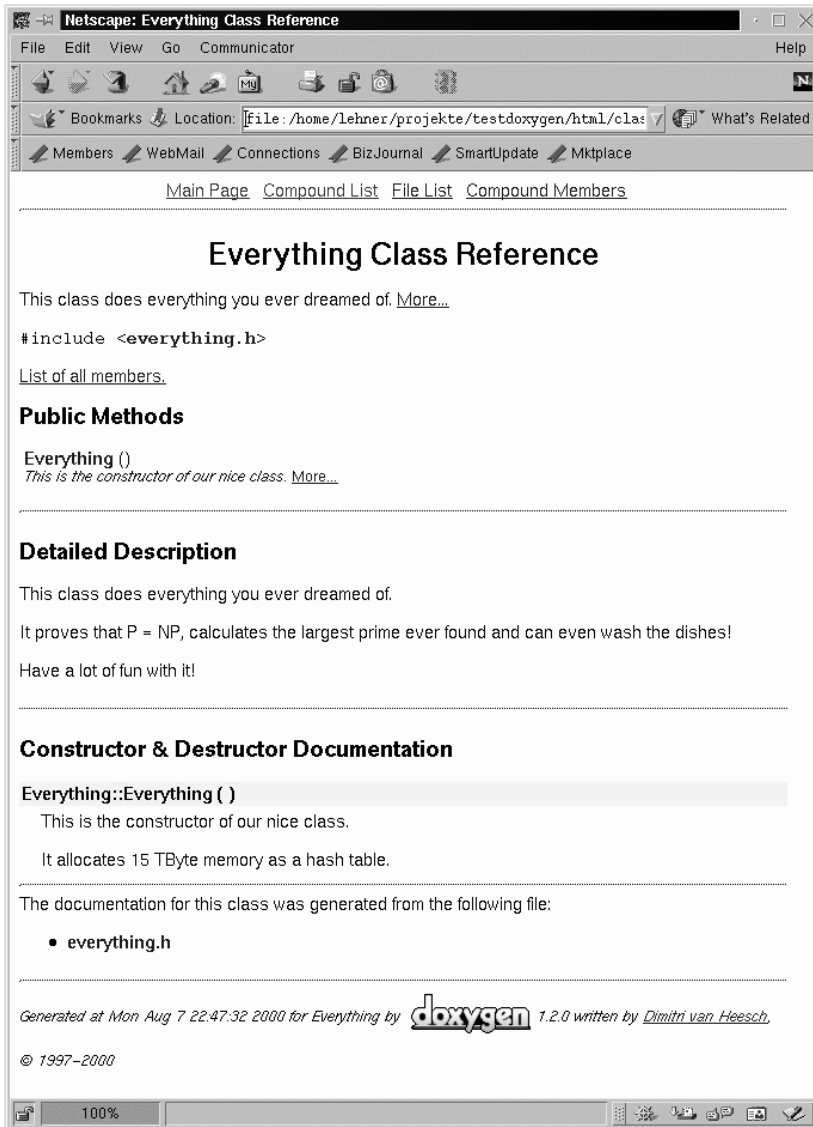


Abbildung 4-51 Die von doxygen erzeugte Klassendokumentation im HTML-Format

Im Verzeichnis `man` ist die Dokumentation im Manual-Page-Format abgelegt. Damit Sie diese Dokumentation nutzen können, müssen Sie entweder dieses Verzeichnis in die Umgebungsvariable `MANPATH` eintragen, oder die erzeugten Dateien in ein Verzeichnis im `MANPATH` verschieben.

Im Verzeichnis *rft* befindet sich die Dokumentation im Rich-Text-Format. Dieses kann zum Beispiel mit vielen Textverarbeitungsprogrammen eingelesen und weiterverarbeitet werden. In der aktuellen *doxygen*-Version 1.2.1 scheinen aber noch einige Fehler bei der Generierung dieses Formats vorhanden zu sein.

Standardmäßig erzeugt *doxygen* die Dokumentation in allen Formaten. Wenn Sie sie nur in einem der Formate benötigen, so schalten Sie die anderen Formate in der Konfigurationsdatei *Doxyfile* einfach aus.

Aufruf	erzeugtes Format
make	DVI (Device Independent)
make ps	PS (PostScript)
make pdf	PDF (Acrobat Reader)
make ps_2on1	PS, zwei Seiten auf ein Blatt
make pdf_2on1	PDF, zwei Seiten auf ein Blatt

Tabelle 4-3 Erzeugen verschiedener Dokumentationsformate im Unterverzeichnis *latex*

Die Konfigurationsdatei enthält noch viele weitere Einstellungsmöglichkeiten. Nähere Informationen finden Sie in den Kommentaren der Konfigurationsdatei selbst sowie im *doxygen*-Handbuch.

4.6.4 Querverweise

Eine der besonderen Stärken von *doxygen* ist die automatische Erzeugung von Querverweisen, auch *cross referencing* genannt: Die von *doxygen* erzeugten HTML- und PDF-Dateien enthalten Links (Querverweise), die der Anwender anklicken kann, um zur Dokumentation anderer Elemente zu gelangen. Das LaTeX- und das PostScript-Dokument enthalten statt der Links Verweise auf die Seitennummer des Elements. So kann der Leser der Dokumentation sehr einfach innerhalb der Dokumentation navigieren. Bei der Manual-Page-Dokumentation werden die Querverweise nicht dargestellt.

doxygen erkennt in den von Ihnen geschriebenen Dokumentationstexten automatisch Bezeichner, zu denen es ebenfalls eine Dokumentation erzeugt hat. Diese Bezeichner werden dann zu den entsprechenden Querverweisen, ohne dass Sie selbst eine entsprechende Angabe machen müssten. Überall, wo Sie einen Klassen-, Methoden- oder Funktionsnamen benutzen, der ebenfalls dokumentiert ist, wird ein solcher Querverweis angelegt. Hier ein Beispiel:

```
/** This class simply does nothing. If you want to do
    more, have a look at class Everything.*/
class Nothing
{
    ....
};
```


In der Dokumentation zu dieser neuen Klasse *Nothing* wird nun das Wort *Everything* als Link auf die Hauptseite der Klasse *Everything* angelegt. Sie können diese Begriffe also wie hier in den Text einbeziehen.

Für die Querverweise auf Methoden gibt es mehrere Formate, die Sie verwenden können. Für Methoden innerhalb der gleichen Klasse reicht es, den Methodennamen und ein leeres Klammernpaar anzugeben. Leerzeichen, Tabulatoren oder Zeilenumbrüche zwischen diesen sind dabei nicht erlaubt. Für Verweise auf Methoden anderer Klassen benutzen Sie das Format `<Klassenname>::<Methodenname>`. In diesem Fall können Sie die Klammern sogar weglassen.

Falls es mehrere überladene Methoden (auch Konstruktoren) mit dem gleichen Namen gibt und Sie auf eine spezielle Methode verweisen wollen, so können Sie hinter dem Methodennamen auch in Klammern die Datentypen der Argumente angeben. *doxygen* erzeugt dann automatisch einen Link auf die richtige Methode. Hier folgt ein Beispiel für diesen Fall:

```
/** Class Everything has two constructors. Always use
    Everything(QString), since Everything(int,bool) is
    just für internal purposes! */
```

doxygen erkennt nicht nur C++-Elemente, sondern auch Dateinamen von Dateien, deren Inhalt ebenfalls in der Dokumentation enthalten ist, URLs (wie beispielsweise »<http://www.trolltech.com/>«) oder E-Mail-Adressen. Auch diese werden automatisch als Hyperlink angelegt.

Falls *doxygen* einmal zu arbeitswütig ist und Querverweise an Stellen erkennt, an denen sie nicht beabsichtigt sind, so setzen Sie einfach vor das entsprechende Wort ein Prozentzeichen (%). *doxygen* löscht dieses Zeichen aus dem Text, erzeugt aber keinen Querverweis mehr.

Sehr interessant ist weiterhin die Möglichkeit, Querverweise zu anderen Bibliotheken erzeugen zu lassen, die nicht im gleichen Projekt liegen. So können Sie insbesondere Links zur Dokumentation der Qt- und KDE-Bibliotheken erzeugen lassen. Dazu benötigen Sie für jede Bibliothek – genauer gesagt für jedes Dokumentationsprojekt – eine so genannte Tag-Datei. Wenn Sie selbst die Dokumentation einer Bibliothek mit *doxygen* erstellen lassen, so erreichen Sie das ganz einfach, indem Sie in der Konfigurationsdatei `Doxyfile` in der Zeile `GENERATE_TAGFILE` den Dateinamen der zu erzeugenden Tag-Datei ergänzen. Soll zum Beispiel die Dokumentation zu unserer Klasse *Everything* auch in andere Dokumentationen als Querverweis aufgenommen werden, so lautet diese Zeile beispielsweise:

```
GENERATE_TAGFILE      = everything.tag
```

Beim Erzeugen der Dokumentation wird dann neben den Dokumentationsdateien auch die Tag-Datei `everything.tag` erzeugt. Sie enthält genaue Informationen, welche Klassen, Methoden usw. dokumentiert wurden.

Schwieriger wird es dagegen, wenn der Quelltext mit den Kommentaren nicht mehr zur Verfügung steht, sondern nur noch die erzeugten HTML-Dateien. Auch daraus kann man mit Hilfe des Programms *doxytag* eine Tag-Datei erzeugen. Für die Qt-Dokumentation sieht der Aufruf folgendermaßen aus:

```
% doxytag -t qt.tag /usr/lib/qt/html
```

Nachdem Sie nun Tag-Dateien für alle Bibliotheken erzeugt haben, für die Sie Querverweise in Ihrer eigenen Dokumentation benutzen wollen, können Sie diese Dateien in der Konfigurationsdatei *Doxyfile* in der Zeile *TAGFILES* angeben. Beim Erzeugen der Dokumentation werden nun diese Dateien ebenfalls eingelesen, und Begriffe, die in diesen Tag-Dateien aufgeführt sind, werden ebenfalls als Querverweise in der Dokumentation eingebunden. So können Sie zum Beispiel mit folgender Zeile auch alle Querverweise auf Qt-Klassen erzeugen lassen:

```
TAGFILES          = qt.tag
```

Schwierig ist es aber, anzugeben, auf welche URLs diese Querverweise zeigen sollen. Da die Dokumentation unter Umständen in ein anderes Verzeichnis verschoben oder auf einem WWW-Server abgelegt wird, liegen die Dokumentationsdateien der Begriffe, auf die verwiesen wird, oft an anderen Stellen als zum Zeitpunkt der Dokumentationserzeugung.

Sie können die Verzeichnisse fest vorgeben, indem Sie in *Doxyfile* hinter dem Namen einer Tag-Datei ein Gleichheitszeichen und danach das Verzeichnis oder die URL der Dokumentation anhängen. Für die Qt-Klassen können Sie zum Beispiel die Dokumentation auf der Homepage der Firma Trolltech wählen:

```
TAGFILES          = qt.tag=http://doc.trolltech.com
```

Der Nachteil ist hier natürlich, dass beim Aufruf des Links die neue Seite aus dem Internet geladen werden muss. Besser ist es, wenn die Dokumentation auf der heimischen Festplatte genutzt werden kann. Außerdem kann es Probleme geben, wenn die Version von Qt sich ändert und daher eine neue Dokumentation auf den Server gestellt wird.

Alternativ kann man auch die Dokumentation zunächst ohne feste Pfade erzeugen lassen. Dazu löschen Sie zunächst wieder die Angabe des Pfades:

```
TAGFILES          = qt.tag
```

doxygen erzeugt nun automatisch im Unterverzeichnis `html` das ausführbare Perl-Skript *installdox*. Mit Hilfe dieses Skripts kann der Anwender die echten Pfade einfügen lassen. Der Aufruf sieht dann beispielsweise folgendermaßen aus:

```
% installdox -l qt.tag@usr/lib/qt/html
```

Anschließend sind alle Querverweise korrekt erzeugt.

4.6.5 Zusätzliche Formatierungen

Wie wir bereits gesehen haben, sorgt *doxygen* automatisch für einen Zeilenumbruch. Einen neuen Absatz in der Dokumentation erzeugt man durch eine Leerzeile im Kommentar. Auch Querverweise legt *doxygen* automatisch an. Allein diese sehr einfachen Möglichkeiten reichen bereits aus, um gute Dokumentationen zu erzeugen.

Zusätzlich bietet *doxygen* nun aber noch eine große Anzahl von Möglichkeiten, um das Aussehen der erzeugten Dokumentation zu verbessern und die Struktur zu organisieren. Dazu nutzt *doxygen* eine große Anzahl von so genannten Tags. Diese Tags beginnen mit einem Backslash »\« oder einem At-Zeichen »@«. Welche der beiden Varianten Sie benutzen, bleibt vollständig Ihrem Geschmack überlassen. Sie können sie auch gemischt verwenden. Wir werden im Weiteren das Backslash-Zeichen verwenden, da es im Quelltext meist übersichtlicher aussieht. Wenn Sie kompatibel zu *KDoc* sein wollen, sollten Sie dagegen unbedingt das At-Zeichen benutzen.

An dieser Stelle sollen nur einige besonders interessante Möglichkeiten vorgestellt werden. Eine vollständige Übersicht enthält die Dokumentation zu *doxygen*.

Aufzählungen

Sie können sehr einfach eine Aufzählung erzeugen, indem Sie jeden Punkt der Aufzählung mit einem Minuszeichen beginnen lassen. *doxygen* erzeugt daraus automatisch Aufzählungspunkte. Sogar verschachtelte Aufzählungen sind hierbei möglich, indem Sie die eingeschachtelte Aufzählung einfach ein Stück weiter einrücken. Die Fähigkeiten unserer Klasse *Everything* kann man beispielsweise so aufführen:

```
/** This class does everything you ever dreamed of.
```

```
    It can:
```

- prove that $P = NP$
- calculate the largest prime
 ever found
- wash the dishes
 - cups
 - plates

```
- knives  
- and much, much more!
```

```
Have a lot of fun with it! */
```

Achten Sie aber unbedingt darauf, dass Aufzählungspunkte auf der gleichen Stufe um die gleiche Kombination aus Tabulatoren und Leerzeichen eingerückt sind. Im Zweifelsfall benutzen Sie für die Einrückung ausschließlich Leerzeichen.

Alternativ können Sie Aufzählungen mit dem Spezial-Tag `@li` oder mit den HTML-Tags `<UL>` und `<LI>` erstellen.

Hervorhebung einzelner Wörter

Eine Reihe von Tags dient zur Hervorhebung einzelner Wörter, zum Beispiel von Namen von Parametern, Datentypen oder Dateinamen. Die wichtigsten sind `\b` (*bold* = fett), `\e` (*emphasized* = hervorgehoben, kursiv) und `\c` (*code* = Nichtproportionalsschrift). Dieses Tag wirkt sich nur auf das unmittelbar folgende Wort aus, das durch ein Leerzeichen oder ein Satzzeichen abgeschlossen ist. Es hat sich allgemein eingebürgert, `\e` für Parameternamen zu benutzen und `\c` für vordefinierte Datentypen und Dateinamen. `\b` sollte besonderen Blickfängern vorbehalten bleiben. Ein kleines Beispiel soll die Anwendung verdeutlichen:

```
/** This method uses \e x and \e y as coordinates  
    of the start point. The result is a \c double value  
    that represents the distance. The configuration file  
    \c config.dat is used.
```

```
\b Attention: If \e x or \e y are negative, the  
program may dump core! */
```

Eingebettete Listings

Wenn Sie Listings – zum Beispiel den typischen Aufruf einer Methode – in die Dokumentation aufnehmen wollen, so benutzen Sie das Tag `\code`, um den Anfang zu markieren, und `\endcode` für das Ende. Diese beiden Tags müssen jeweils auf einer eigenen Zeile stehen. Der Text zwischen diesen beiden Tags wird wörtlich zitiert, d.h. Zeilenumbrüche werden an genau den gleichen Stellen wie im Kommentar vorgenommen, Einrückungen bleiben erhalten, und andere Tags werden nicht interpretiert. Außerdem wird eine nichtproportionale Schrift gewählt und Syntax-Highlighting benutzt. Querverweise werden aber weiterhin eingefügt. (Wollen Sie auch diese unterdrücken, so benutzen Sie stattdessen `\verbatim` und `\endverbatim`.) Hier folgt ein einfaches Beispiel, das die typische Anwendung der Klasse *Everything* beschreibt.

```
/** This class does everything you ever dreamed of.
```

```
You can use it like this:
```

```
\code
    Everything *ev = new Everything ();
    int x = ev->calculateTheAnswer();
    if (x != 42)
        cerr << "Douglas Adams was wrong!";
\endcode
*/
```

Auch innerhalb des Listings wird ein Stern (*) am Anfang der Zeile entfernt. Achten Sie also darauf, dass keine der Zeilen im Listing versehentlich mit einem Stern beginnt (Dereferenzierung eines Zeigers, Multiplikation). In diesem Fall fügen Sie einen weiteren Stern ein.

Falls Sie im Quelltext Tabulatoren für die Einrückung des Kommentars benutzt haben, so achten Sie darauf, dass Sie die Tabulatorweite in *Doxyfile* in der Zeile *TAB_SIZE* korrekt eingestellt haben.

Absatzformate für Klassen und Methoden

Bei der Dokumentation von Klassen und Methoden hat sich ein Standard für den Aufbau etabliert. Dabei gibt es einige Standardabsätze, die auch in *doxygen* mit speziellen Tags festgelegt werden können.

Nach einer allgemeinen Beschreibung enthalten Klassen einen eigenen Absatz mit dem Namen und der E-Mail-Adresse des Entwicklers (*\author*) sowie einen Absatz mit der Versionsnummer (*\version*). Bei Methoden schließt sich an die allgemeine Beschreibung ein Absatz mit der Auflistung der einzelnen Parameter (*\param*) und ihrer Bedeutung an, gefolgt von einem Absatz über die Bedeutung des Rückgabewerts (*\return*). Oftmals folgt bei beiden noch ein Absatz mit Querweisen zu verwandten Klassen oder Methoden (*\sa* oder synonym *\see*).

Für diese Absatz-Tags gilt, dass sie als Erstes in der Zeile stehen müssen. (Nur Leerschritte und Tabulatoren sowie ein einzelner Stern dürfen vor ihnen stehen.) Ihr Geltungsbereich gilt grundsätzlich für den ganzen Absatz, also für den Rest der Zeile sowie für alle folgenden Zeilen bis zu einer Leerzeile oder einer Zeile mit einem weiteren Absatz-Tag.

Eine Besonderheit weist das Tag *\param* auf, mit dem die Bedeutung eines einzelnen Parameters beschrieben wird. Das erste Wort nach dem Tag muss der Name des Parameters sein (nicht der Datentyp), der Rest des Absatzes ist die Beschreibung. *doxygen* überprüft nicht, ob es tatsächlich einen Parameter mit diesem Namen gibt. Die Unterscheidung dient ausschließlich zur Formatierung.

Die Anwendung dieser Tags wollen wir hier wieder an unserer Klasse *Everything* verdeutlichen:

```

/** This class does everything you ever dreamed of.
    \author Burkhard Lehner <Burkhard.Lehner@gmx.de>
    \version 0.99 Beta 2
    \sa Nothing, Something
*/
class Everything
{
    ....
    /** calculates the answer
        \param u the complete universe
        \param c the mouse that is observing
            the calculation
        \return should be 42
        \sa DeepThought, calculateTheQuestion()
    */
    int calculateTheAnswer (Universe u, Mouse o);
    ....
}

```

doxygen unterstützt eine große Anzahl weiterer Absatz-Tags. Eine Liste aller Tags finden Sie im Handbuch.

Bilder

Ein Bild sagt mehr als tausend Worte. Daher bietet *doxygen* mit Hilfe des Tags `\image` die Möglichkeit, Bilder in die Dokumentation einzubinden. Dazu müssen Sie in der Konfigurationsdatei *Doxyfile* in der Zeile *IMAGE_PATH* den Pfad des Verzeichnisses angeben, in dem die Bilder liegen. *doxygen* kopiert die benutzten Bilder dann automatisch mit in das Dokumentationsverzeichnis.

Das `\image`-Tag muss auf einer eigenen Zeile stehen. Dahinter steht das Format (entweder *html* oder *latex*) und anschließend der Dateiname der Bilddatei.

In die HTML-Dokumentation können alle Dateiformate eingebunden werden, die ein Browser darstellen kann, also beispielsweise PNG, GIF oder JPG. In die LaTeX-, PostScript- und PDF-Dokumentation können dagegen zur Zeit nur EPS-Dateien eingebunden werden.

Wollen Sie das Bild sowohl in der HTML- als auch der LaTeX-Dokumentation zeigen lassen, so stellen Sie es in beiden Formaten zur Verfügung, und verwenden Sie zweimal das `\image`-Tag:

```

/** This dialog provides user input of name and
    address. Here is a screenshot of the resulting
    window:
    \image html screenshot.png
    \image latex screenshot.eps
*/

```

doxygen kopiert nun automatisch die Bilddateien aus dem Verzeichnis, das in *Doxyfile* unter *IMAGE_PATH* angegeben ist, in das entsprechende Dokumentationsverzeichnis.

HTML-Tags

Neben den speziellen *doxygen*-Tags können Sie auch eine Vielzahl von HTML-Tags benutzen, wie beispielsweise `<UL>`, `<OL>`, `<H1>`, `<HR>`, `<TABLE>` usw. All diese Tags werden von *doxygen* analysiert und interpretiert. Auch für die Dokumentation in den anderen Formaten werden entsprechende Formatierungen erzeugt. Eine genaue Liste der HTML-Tags, die *doxygen* versteht, finden Sie im Handbuch. So können Sie der fertigen Dokumentation den letzten Schliff geben.

Inhalt der Startseite

Um den Text für die erste Seite der LaTeX-Dokumentation bzw. für die Index-Seite der HTML-Dokumentation festzulegen, gibt es das spezielle Tag */mainpage*. Der Text des Abschnitts, der diesem Tag folgt, wird auf diese Seite gesetzt. Alle Formatierungen sind hier erlaubt, und Querverweise werden ebenfalls automatisch hergestellt. Es empfiehlt sich, auf dieser Seite eine Zusammenfassung der wichtigsten Eigenschaften der dokumentierten Bibliothek zu geben.

Sie können den Kommentar mit dem */mainpage*-Tag in jede beliebige Quelltext-Datei einfügen. Liegen mehrere */mainpage*-Tags vor, wird nur eines benutzt. Achten Sie also darauf, dieses Tag insgesamt nur einmal zu benutzen. Falls Sie beispielsweise eine Bibliothek dokumentieren, die viele gleich wichtige Klassen enthält, können Sie eine eigene Datei anlegen, die nur den Kommentar mit dem */mainpage*-Tag enthält. Diese Datei müssen Sie natürlich in der Konfigurationsdatei ebenfalls in der Zeile *INPUT* eingeben.

4.6.6 Kompatibilität zu *KDoc*

Die Klassendokumentation der KDE-Bibliotheken wird bislang noch mit dem Programm *KDoc* erstellt, obwohl *KDoc* nur einen Bruchteil der Möglichkeiten von *doxygen* bietet. *doxygen* versteht nahezu uneingeschränkt die Syntax von *KDoc*-Kommentaren. Ein Umstieg von *KDoc* auf *doxygen* ist daher in der Regel problemlos möglich. Wenn Ihre Kommentare weiterhin mit *KDoc* verarbeitet werden sollen, gibt es einige Punkte, die Sie beachten sollten:

- Während in *doxygen* Tags sowohl mit dem Zeichen »\« als auch mit »@« beginnen können, erkennt *KDoc* nur Tags, die mit »@« beginnen.
- Sie sollten auf die Tags verzichten, die von *doxygen*, aber nicht von *KDoc* unterstützt werden. Beschränken Sie sich daher möglichst auf die Tags *@short*, *@author*, *@version*, *@li*, *@param*, *@return* und *@see*. (*@see* und *@sa* sind in *doxygen* synonym verwendbar.)

- Die Syntax des *@image*-Tags ist bei *doxygen* und *KDoc* unterschiedlich: *doxygen* verlangt zusätzlich die Angabe `html` bzw. `latex`.
- Die Tags zur Hervorhebung einzelner Wörter (wie z.B. *@e* und *@c*) funktionieren in *KDoc* nicht.
- Während *doxygen* Querverweise automatisch bildet, müssen Sie in *KDoc* Querverweise mit dem Tag *@ref* markieren.
- *KDoc* versteht und interpretiert – im Gegensatz zu *doxygen* – keine HTML-Tags. Diese werden sogar bei der Erzeugung einer HTML-Dokumentation wörtlich zitiert, sind also unwirksam. Verzichten Sie also möglichst vollständig auf HTML-Tags, wenn Sie kompatibel bleiben wollen. Einzige Ausnahme: die Tags `<pre>` und `</pre>`.
- Da *KDoc* die Tags *@code* und *@endcode* zum Zitieren von Listings nicht versteht, kann man auf die HTML-Tags `<pre>` und `</pre>` zurückgreifen.
- *KDoc* erzeugt nicht automatisch eine Aufzählung, wenn mehrere Zeilen mit einem Minus-Zeichen beginnen. Benutzen Sie stattdessen das *@li*-Tag. Geschachtelte Aufzählungen sind in *KDoc* gar nicht möglich.

4.7 Grundklassen für Datenstrukturen

Qt benutzt intern eine Reihe von Datenstrukturen, die auch dem Programmierer zur Verfügung stehen. So braucht dieser das Rad nicht jedes Mal neu zu erfinden, wenn er eine verkettete Liste oder eine Hash-Tabelle erzeugen will. Stattdessen kann er einfach auf die Qt-Klassen und -Templates zurückgreifen.

Es stellt sich immer die Frage, ob man auf die Qt-Klassen oder auf die Templates der STL (*Standard Template Library*) zurückgreifen soll. In beiden sind etwa die gleichen Datenstrukturen vorhanden, sie sind jedoch völlig inkompatibel. Zu dieser Frage wurden schon einige hitzige Diskussionen in den Mailing-Listen des KDE-Projekts geführt. Wie so oft gibt es keine »richtige« Lösung, und was man einsetzt, hängt stark von Geschmack des Programmierers ab. Im Folgenden werden einige Argumente aufgelistet, die diese Design-Entscheidung erleichtern sollen:

- Ein Großteil des Codes für die Qt-Klassen muss auf jeden Fall zum Programm gelinkt werden, da Qt selbst darauf zurückgreift. Benutzt man in seinem Programm außerdem die STL-Templates, so muss zusätzlicher Code hinzugelinkt werden.
- Der Programmierer kann ohne Einschränkungen voraussetzen, dass die Qt-Klassen zur Verfügung stehen, da Qt zwangsläufig auf dem Rechner installiert sein muss, auf dem das Programm laufen soll. STL ist zwar ebenfalls ein Stan-

dard auf C++-Compilern geworden, allerdings gibt es immer noch ältere Compiler, die eine zum Teil abweichende Schnittstelle zur STL besitzen.

- Eine Reihe von darstellenden Klassen in Qt nimmt die Argumente in Form von Qt-Klassen entgegen. Speichert man intern die Daten in STL-Templates, muss vor dem Aufruf eine Umwandlung in Qt-Klassen erfolgen, die das Programm ineffizienter machen.
- Wählt man die Qt-Klassen, so ist ein Umstieg auf eine andere grafische Oberfläche (z.B. *XForms*, *Gtk* usw.) schwierig, da auch die internen Datenstrukturen geändert werden müssen. Will man also von Qt möglichst unabhängig bleiben, sollte man sein Programm am besten strikt in einen *Daten verarbeitenden Teil* (zum Beispiel mit STL-Templates) und einen *darstellenden Teil* (auf Qt-Basis) unterteilen.

4.7.1 Gemeinsame Daten

Aus Effizienzgründen wird in vielen Qt-Klassen auf ein Kopieren von Daten so weit wie möglich verzichtet und stattdessen mit einer Referenz auf dieselben Daten gearbeitet. Dieses Prinzip nennt man *gemeinsame Datennutzung* (*shared data*). Intern werden gemeinsame Daten durch einen Referenzzähler implementiert (siehe Abbildung 4.52).

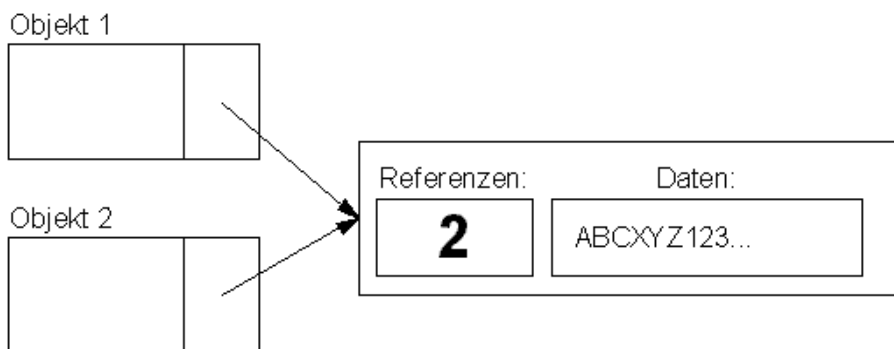


Abbildung 4.52 Zwei Objekte mit gemeinsamer Referenz und Referenzzähler

Wird ein neues Objekt als Kopie eines bestehenden Objekts angelegt (über den Copy-Konstruktor oder den Zuweisungsoperator), so wird nur der Referenzzähler um 1 erhöht. Das neue Element erhält eine Referenz auf das gleiche Datenelement. Wird eines der Objekte gelöscht, wird der Referenzzähler um 1 verringert. Ist der Referenzzähler 0, wird auch das gemeinsame Datenobjekt gelöscht.

Man unterscheidet zwischen *implizit gemeinsamen Daten* und *explizit gemeinsamen Daten*. Bei implizit gemeinsamen Daten bemerkt der Programmierer nicht, dass zwei Elemente auf das gleiche Datenelement verweisen können. Sobald die

Daten in einem Element geändert werden sollen, das sein Datenelement mit anderen Objekten teilt (Referenzzähler > 1), wird zunächst das gemeinsame Datenelement kopiert. Das zu ändernde Objekt erhält nun ein eigenes Datenelement (Referenzzähler jetzt 1), das es beliebig verändern kann, ohne dass die anderen Objekte mit verändert würden. Bei explizit gemeinsamen Daten muss der Programmierer selbst darauf achten, dass er beim Ändern der Daten eventuell andere Objekte mit verändert. Diese Klassen haben in Qt in der Regel jedoch eine Methode *detach*, die eine neue Datenkopie anlegt, falls der Referenzzähler größer als 1 ist. Wenn man also im Zweifelsfall diese Methode vor einer Veränderung aufruft, ist man auf der sicheren Seite.

Ein Beispiel für implizit gemeinsame Daten ist die Qt-Klasse *QString*, die Textstrings speichert. Betrachten wir folgendes Programmstück:

```
QString a, b;  
a = "Alles neu";  
b = a;    // a und b zeigen auf die gleichen Daten  
a += " macht der Mai";
```

Nach der Zuweisung von *a* an *b* teilen sich beide die Textdaten. Der Referenzzähler des Strings ist nun 2. Bevor jedoch an *a* eine Veränderung vorgenommen wird (Anhängen eines Textstücks), werden die Daten kopiert, und *a* erhält die neue Kopie. Somit hat am Ende dieses Programmstücks *a* den Wert »Alles neu macht der Mai«, während *b* nach wie vor den Wert »Alles neu« enthält.

Ein Beispiel für eine Datenstruktur mit explizit gemeinsamen Daten ist das Template *QArray <type>*. Folgendes Beispiel soll das verdeutlichen:

```
QArray <int> a, b;  
a.fill (2, 10); // a ist ein Array mit 10 Elementen,  
               // die mit dem Wert 2 gefüllt werden  
b = a;         // a und b benutzen die gleichen Daten  
a[3] = 42;     // Wert wird in a UND b verändert  
a.detach ();  
a[4] = 42;     // Wert wird nur in a verändert
```

Die Variablen *a* und *b* sind hier Arrays von *int*-Werten und greifen auf explizit gemeinsame Daten zu. Eine Änderung eines Wertes in *a* ändert also die gemeinsamen Daten und somit auch *b*. Ein Aufruf von *detach* für *a* bewirkt jedoch, dass das gemeinsame Datenelement kopiert wird und *a* eine eigene Kopie erhält. Die zweite Änderung in *a* hat also keine Auswirkung mehr auf *b*. Beachten Sie, dass *detach* das Datenelement nur kopiert, wenn der Referenzzähler größer als 1 ist. Auch hier wird jeder unnötige Kopieraufwand vermieden.

Beispiele für Klassen mit explizit gemeinsamen Daten sind *QArray <type>*, *QByteArray*, *QBitArray* und *QImage*. Die Klassen *QString*, *QPalette*, *QPen*, *QBrush* und *QPixmap* sind Beispiele für implizit gemeinsame Daten.

4.7.2 Container-Klassen

Als Container-Klassen werden Klassen bezeichnet, die Elemente eines Datentyps speichern und verwalten können. Die Anzahl der Elemente ist dabei meist flexibel, so dass man zur Laufzeit weitere Elemente einfügen oder entfernen kann. Intern werden die Elemente in einer Datenstruktur gespeichert. Fast alle Container-Klassen von Qt sind als Templates realisiert, so dass Sie selbst festlegen können, welchen Datentyp die Elemente besitzen, die gespeichert werden sollen.

Die Container-Klassen lassen sich in zwei Gruppen unterteilen: Die *Datencontainer* speichern die Elemente ab, die *Zeigercontainer* dagegen speichern nur Zeiger auf die Elemente. Dementsprechend sind die Anwendungsgebiete verschieden:

Datencontainer werden hauptsächlich für Grundtypen verwendet, z.B. für Zahlenwerte, *QString*-Objekte oder einfache Klassen. Als Voraussetzung muss gegeben sein, dass eine Zuweisung (*operator=*) sowie eine Initialisierung (Copy-Konstruktor) für diesen Datentyp definiert sind. Das ist insbesondere für die Klasse *QObject* und damit automatisch auch für alle abgeleiteten Klassen nicht der Fall; diese können also nicht in einem Datencontainer gespeichert werden. Wollen Sie im Container außerdem nach einem bestimmten Element suchen, so muss auch der Gleichheitsoperator (*operator==*) definiert sein. Sie können mit einem Datencontainer einen Zeigercontainer »simulieren«, indem Sie als Datentyp der Elemente einen Zeigertyp (also z.B. *QObject**) angeben. Es empfiehlt sich jedoch meist, den entsprechenden Zeigercontainer zu verwenden, da dieser mehr Methoden für die Verwaltung zur Verfügung stellt.

Zeigercontainer speichern nur die Adresse der verwalteten Elemente. An den Datentyp werden daher keine besonderen Ansprüche gestellt. Die Objekte müssen allerdings an einer anderen Stelle angelegt werden (in der Regel mit *new*). Ein Zeigercontainer kann in zwei verschiedenen Modi betrieben werden: Ist der Container im Modus *autoDelete*, so übernimmt er die spätere Speicherfreigabe der Elemente (mit *delete*), wenn die Elemente aus dem Container entfernt werden oder der ganze Container gelöscht wird. Ist er nicht in diesem Modus, so muss der Programmierer selbst die Speicherfreigabe der Elemente durchführen, wenn er Speicherlecks vermeiden will. Zeigercontainer sind sehr gut geeignet für Objekte von komplexeren Klassen, deren Copy-Konstruktor nicht definiert oder sehr aufwendig ist, z.B. gilt das auch für alle von *QObject* abgeleiteten Klassen.

Tabelle 4.4 zeigt einen Überblick über die definierten Containerklassen in Qt.

Templatename	Typ	Funktion	interne Realisierung
<i>QArray</i> <typ>	Datencontainer	dynamisches Array	Array mit Elementen fester Größe
<i>QVector</i> <typ>	Zeigercontainer	dynamisches Array	Array von Zeigern
<i>QValueList</i> <typ>	Datencontainer	lineare Liste	doppelt verkettete Liste
<i>QList</i> <typ>	Zeigercontainer	lineare Liste	doppelt verkettete Liste
<i>QMap</i> <typ1, typ2>	Datencontainer	Zuordnungstabelle, Schlüsseltyp typ1	binärer Rot-Schwarz-Baum
<i>QDict</i> <typ>	Zeigercontainer	Zuordnungstabelle, Schlüsseltyp QString	Hash-Tabelle
<i>QAsciiDict</i> <typ>	Zeigercontainer	Zuordnungstabelle, Schlüsseltyp char*	Hash-Tabelle
<i>QIntDict</i> <typ>	Zeigercontainer	Zuordnungstabelle, Schlüsseltyp int	Hash-Tabelle
<i>QPtrDict</i> <typ>	Zeigercontainer	Zuordnungstabelle, Schlüsseltyp void*	Hash-Tabelle

Tabelle 4-4 Container-Klasse von Qt

Wie Sie der Tabelle entnehmen können, gibt es drei grundsätzliche Datentypen (dynamisches Array, lineare Liste und Zuordnungstabelle), jeweils in den Typen Datencontainer und Zeigercontainer.

Die Datenstruktur *dynamisches Array* ermöglicht den Zugriff auf ein einzelnes Element mit Hilfe einer Index-Nummer. Dieser Zugriff ist sehr effizient. Ineffizienter ist es dagegen, einen Teil (oder alle) Elemente zu verschieben, um ein einzelnes Element einzufügen oder zu löschen. Im Gegensatz zu einem normalen C- und C++-Array hat dieser Container den Vorteil, dass die Größe jederzeit geändert werden kann (was allerdings bei großen Arrays uneffizient ist) und dass ein Zugriff mit einer Indexnummer, die außerhalb der Grenzen liegt, zu einer Warnung führt (und eben nicht zu einem Programmabsturz, der die Fehlersuche erschwert).

Die Datenstruktur *lineare Liste* bietet die Möglichkeit, sich in der Kette der Elemente vorwärts und rückwärts zu bewegen, um auf einzelne Elemente zuzugreifen. Ein Einfügen oder Löschen von Elementen an beliebigen Stellen der Liste ist sehr effizient möglich.

Die Datenstruktur *Zuordnungstabelle* ist speziell für eine effiziente Suche ausgelegt. In diesem Container werden die Elemente zusammen mit einem Suchschlüssel abgelegt. Die Datencontainer-Klasse *QMap* ermöglicht dabei beliebige

Suchschlüsseltypen; die einzige Voraussetzung ist, dass der Typ einen Vergleich (*operator<*) implementiert hat. Mögliche Typen sind beispielsweise *int*, *double* oder *QString*. Die Zeigercontainer-Klassen sind im Schlüsseltyp dagegen eingeschränkt. Als Suchschlüssel sind hier Texte möglich (*QString* oder *char**), ganze Zahlen (*int*) oder Adressen (*void**).

Neben den oben beschriebenen Datentypen hält Qt noch einige Spezialdatentypen bereit: *QQueue* realisiert einen FIFO-Speicher (Warteschlange, Typ Zeigercontainer), *QStack* einen LIFO-Speicher (Kellerspeicher, Typ Zeigercontainer).

Die Klasse *QByteArray* ist eine abgeleitete Klasse des Datentyps *QArray <char>*. Sie wird häufig benutzt, um einen Speicherblock abzulegen. *QString* ist wiederum eine Unterklasse von *QByteArray* und speichert einen Text in der herkömmlichen C-Art als Array von *char*-Zeichen, der durch ein 0-Zeichen abgeschlossen wird. (Nähere Informationen zu *QString* finden Sie in Kapitel 4.7.4, *Die String-Klassen – QString und QStringRef*.)

Für die Container-Objekte selbst (sowohl Datencontainer als auch Zeigercontainer) sind der Zuweisungsoperator und der Copy-Konstruktor definiert. Da die Datenstrukturen jedoch umfangreich sein können, sollte man von ihnen möglichst nicht Gebrauch machen. Wenn Sie beispielsweise in einer Methode ein Container-Objekt als Parameter entgegennehmen wollen, so tun Sie das am besten als Referenz. Schreiben Sie also statt

```
void MyObject::myMethod (QArray <int> x)
```

besser:

```
void MyObject::myMethod (const QArray <int> &x)
```

Besondere Vorsicht ist bei Zeigercontainern im Modus *autoDelete* geboten. Wird von einem solchen Zeigercontainer-Objekt eine Kopie gemacht, wird nur die zugrunde liegende Datenstruktur – die die Adressen der Elemente enthält – kopiert, nicht aber die Elemente selbst. Wird nun eines der Objekte wieder gelöscht, werden auch die Elemente mit *delete* freigegeben; das andere Container-Objekt verweist aber noch auf diese Objekte. Spätestens beim Löschen des zweiten Container-Objekts kommt es zwangsläufig zum Programmabsturz.

Es ist problemlos möglich, als Typ einer Containerklasse eine andere Containerklasse einzusetzen. So können Sie bei Bedarf durchaus eine Hash-Tabelle (*QDict*) benutzen, die Zeiger auf Listen (*QList*) enthält, wobei jede Liste Zeiger auf ein Objekt der Klasse *MyObject* enthält. Eine Variable *myHashTable* von diesem Typ wird beispielsweise so deklariert:

```
QDict <QList <MyObject> > myHashTable;
```

Beachten Sie dabei unbedingt, dass Sie ein Leerzeichen zwischen die beiden schließenden spitzen Klammern »> >« setzen müssen, da der Compiler sie sonst als Right-Shift-Operator auffasst und einen Syntaxfehler melden.

Wir wollen uns nun die verschiedenen Datenstrukturen der Containerklassen etwas genauer anschauen und uns den Einsatz in eigenen Programmen anhand von Beispielen verdeutlichen.

Das dynamische Array – *QArray* und *QVector*

Mit dem Klassen-Template *QArray* <type> kann man ein Objekt erzeugen, das ein Array von Objekten vom Typ *type* verwaltet. Die Anzahl der Elemente wird dabei im Konstruktor festgelegt, kann jedoch auch nachträglich mit der Methode *resize* geändert werden. Die Elemente liegen hintereinander im Speicher, so dass der Zugriff mit Zeigerarithmetik wie bei einem C-Array durchgeführt werden kann. Auch der *[]*-Operator wurde so überladen, dass er das Element an der Stelle des Index zurückliefert.

Die möglichen Datentypen für *QArray* sind eingeschränkt. Zugelassen sind die elementaren Datentypen (*int*, *double*, ...) und Strukturen. Klassen können nur dann verwendet werden, wenn Sie keine Konstruktoren, keinen Destruktor und keine virtuellen Methoden besitzen. Der Grund für diese Einschränkung ist, dass die Datenelemente als einfache Datenblöcke kopiert und angelegt werden. Konstruktoren und Destrukturen werden daher nicht aufgerufen, die virtuelle Methodentabelle wird nicht initialisiert.

QArray besitzt eine Reihe von Methoden, mit denen man das Array manipulieren kann. Wie bereits oben beschrieben, kann man mit der Methode *resize* die Anzahl der enthaltenen Elemente verändern. Dazu wird intern ein neues Array mit passender Größe angelegt, und die Daten werden kopiert. Je mehr Daten enthalten sind, desto länger dauert diese Operation, so dass man möglichst nicht allzu oft die Größe ändern sollte.

Mit der Methode *fill* kann man das ganze Array mit einem Element füllen lassen. Im ersten Parameter übergibt man dabei das Element, das in alle Array-Positionen kopiert werden soll. Der zweite Parameter ist optional. Mit ihm kann man eine neue Größe des Arrays festlegen.

Mit *find* kann man ein Element im Array suchen. Der Rückgabewert ist der Index des Elements oder -1, falls das Element nicht vorhanden ist. Der optionale zweite Parameter der Methode *find* legt fest, ab welchem Index gesucht werden soll. Die Methode *contains* ermittelt, wie oft ein Element im Array vorhanden ist, und liefert als Rückgabewert die Anzahl.

Auch der *==*-Operator wurde überladen. Mit ihm kann man zwei *QArray*-Objekte vergleichen, die auf dem gleichen Typ basieren. Sie gelten dabei als gleich, wenn sie gleich viele Elemente enthalten und wenn ihr Inhalt (auf Bit-Ebene) identisch ist.

QArray benutzt explizit gemeinsame Daten. Wenn Sie also ein *QArray*-Objekt einem anderen zuweisen, benutzen beide das gleiche Daten-Array. Mit der Methode *detach* können Sie bewirken, dass der Inhalt kopiert wird, so dass zwei unabhängige *QArray*-Objekte entstehen (siehe auch Kapitel 4.7.1, *Gemeinsame Daten*).

Als Beispiel für den Einsatz von *QArray* wollen wir hier in einem *QArray*-Objekt die ersten fünf Primzahlen einfügen, sie in umgekehrter Reihenfolge ausgeben und anschließend nach dem Element 7 suchen.

```
QArray<int> primliste (5); // Array mit fünf Elementen
// Primzahlen mit dem []-Operator einfügen
primliste[0] = 2;
primliste[1] = 3;
primliste[2] = 5;
primliste[3] = 7;
primliste[4] = 11;
// Elemente rückwärts ausgeben
for (int i = primliste.count() - 1; i >= 0; i--)
    cout << primliste[i] << endl;
// Nach Element 7 suchen
if (primliste.find(7) != -1)
    cout << "7 ist enthalten" << endl;
else
    cout << "7 ist nicht enthalten ??? Fehler!" << endl;
```

Die Qt-Bibliothek enthält bereits zwei Klassen, *QByteArray* und *QBitArray*, die von *QArray* abgeleitet sind. *QByteArray* ist dabei weit gehend mit *QArray<char>* identisch. Es speichert also Datenelemente von der Größe eines Bytes. *QByteArray* wird oft benutzt, um Datenblöcke – beispielsweise aus einer Datei – zu speichern und zu bearbeiten. Es wird unter anderem auch in der Klasse *QBuffer* eingesetzt (siehe Kapitel 4.18, *Dateizugriffe*).

QBitArray speichert Datenelemente, die nur ein Bit umfassen. Dabei werden die Elemente dicht hintereinander gepackt, so dass sie auch tatsächlich nur ein Bit Speicher pro Element verbrauchen. *QBitArray* wird meist eingesetzt, wenn man eine große Anzahl von *bool*-Variablen speichern will. Eine *bool*-Variable belegt bei den meisten Compilern den gleichen Speicherplatz wie eine *int*-Variable, also meist vier Byte. *QArray<bool>* würde also 32mal so viel Speicher belegen wie *QBitArray* für die gleiche Anzahl von Elementen.

Eine weitere vordefinierte Array-Klasse ist *QPointArray*, die von *QArray<QPoint>* abgeleitet ist. In einem solchen Objekt kann man eine Menge von Koordinatenpunkten abspeichern. Diese Klasse wird als Parameter für einige Zeichenoperationen in *QPainter* benutzt (siehe Kapitel 4.2.1, *QPainter*). Sie bietet auch eine Reihe von Methoden, um die gespeicherten Punkte zu manipulieren.

Der entsprechende Zeigercontainer für dynamische Arrays ist die Klasse *QVector*. Sie legt im Gegensatz zu *QArray* nicht die Elemente selbst ab, sondern Zeiger auf Elemente. Bei *QVector* gibt es keine Einschränkungen für die Zeigertypen, die gespeichert werden können. Als einfaches Beispiel wollen wir hier noch einmal unser Primzahlbeispiel von oben benutzen. Beachten Sie hier, dass wir die zu speichernden Objekte selbst mit *new* anlegen müssen. Das Freigeben mit *delete* übernimmt *QVector* für uns, da wir *setAutoDelete (true)* benutzen.

```
// Array mit fünf Zeigern auf int-Werte
QVector<int> primliste (5);
// Beim Entfernen von Elementen oder im Destruktor
// von QVector werden automatisch die Elemente gelöscht,
// wenn autoDelete auf true gesetzt wird:
primliste.setAutoDelete (true);
// Primzahlen mit dem []-Operator einfügen
// Die Werte werden mit new angelegt
primliste[0] = new int (2);
primliste[1] = new int (3);
primliste[2] = new int (5);
primliste[3] = new int (7);
primliste[4] = new int (11);
// Elemente rückwärts ausgeben
for (int i = primliste.count() - 1; i >= 0; i--)
    cout << *(primliste [i]) << endl;
// Nach Element 7 suchen
if (primliste.find (7) != -1)
    cout << "7 ist enthalten" << endl;
else
    cout << "7 ist nicht enthalten ??? Fehler!" << endl;
```

In diesem einfachen Beispiel wird klar, dass *QVector* für einfache Datentypen nicht gut geeignet ist: Der Speicherbedarf ist ungleich höher als bei *QArray*, da wir neben den Daten auch noch für jedes Element einen Zeiger speichern müssen. *QVector* wird daher eher für komplexe Datentypen eingesetzt, insbesondere für Klassen, die nicht in *QArray* benutzt werden können, da sie einen Konstruktor, Destruktor oder virtuelle Methoden besitzen.

Die lineare Liste – *QValueList* und *QList*

Eine lineare Liste wird oft eingesetzt, wenn zur Laufzeit des Programms häufig Elemente hinzugefügt oder entfernt werden, denn diese Operationen sind in einer linearen Liste sehr effizient implementiert. Qt definiert auch für diese Datenstruktur zwei verschiedene Container-Templates: *QValueList* ist ein Datencontainer und speichert die Elemente selbst ab, während *QList* ein Zeigercontainer ist, also nur eine Liste von Zeigern verwaltet, die auf die gespeicherten Elemente zeigen. Tabelle 4.5 zeigt die wichtigsten Methoden, die *QValueList* und *QList* definieren. Beachten Sie aber, dass die Parameter für diese Methoden für

QValueList und *QList* zum Teil unterschiedlich sind. Zusätzlich ist in der Tabelle noch das Laufzeitverhalten angegeben. Ein Laufzeitverhalten von $O(1)$ bedeutet, dass die Operation in konstanter Zeit ausgeführt wird, $O(n)$ bedeutet, dass die Ausführungszeit proportional zur Anzahl der Elemente steigt.

Methodenname	Bedeutung	Laufzeit
<i>count</i>	liefert Anzahl der gespeicherten Elemente	$O(1)$
<i>clear</i>	löscht alle Elemente	$O(n)$
<i>append</i>	fügt ein Element am Ende an	$O(1)$
<i>prepend</i>	fügt ein Element am Anfang an	$O(1)$
<i>insert</i>	fügt ein Element an beliebiger Stelle ein	$O(1)$
<i>remove</i>	löscht ein Element an beliebiger Stelle	$O(1)$
<i>find</i>	sucht nach einem Element	$O(n)$
<i>contains</i>	zählt Anzahl der Vorkommen eines Elements	$O(n)$
<i>at</i>	liefert das x-te Element der Liste	$O(n)$

Tabelle 4-5 Zugriffsmethoden von *QValueList* und *QList*

Der Zugriff auf ein *QValueList*-Objekt geschieht ausschließlich über Iterator-Objekte (siehe Kapitel 4.7.3, *Iterator-Objekte*). So erwarten die Methoden *insert* und *remove* ein Iterator-Objekt zur Angabe der Stelle, an der eingefügt oder entfernt werden soll. Weiterhin definiert *QValueList* einige Operatoren, um einfacher mit Listen umgehen zu können. Sie können z.B. Listen mit *operator+* aneinander hängen oder mit *operator<<* einzelne Elemente an die Liste anhängen. Das folgende Listing zeigt den typischen Umgang mit *QValueList*-Objekten:

```
QValueList<int> l1; // leere Liste für int-Zahlen
l1.append(2);      // eine Zahl einfügen
l1.append(3);      // eine weitere Zahl anhängen
l1 << 5 << 7 << l1; // drei Zahlen anhängen
QValueList<int> l2; // weitere Liste
l2 << l3 << l7;

// zwei Listen aneinander hängen
QValueList<int> l3 = l1 + l2;

// eine Liste von vorn nach hinten durchlaufen
// Dazu zunächst Iterator-Objekt erzeugen
QValueListIterator<int> it;
// in einer for-Schleife alle Elemente besuchen
for (it = l3.begin(); it != l3.end(); ++it)
{
    // Zugriff mit operator*
    int value = *it;
    qDebug("Wert = %d", value);
}
```

Die Template-Klasse *QList* <type> enthält innerhalb der doppelt verketteten Liste nur Zeiger auf die Elemente, die in der Liste abgelegt sind. Sie stellt daher keine Ansprüche an den Datentyp, ist daher auch problemlos für Objekte der Klasse *QObject* oder einer abgeleiteten Klasse geeignet.

Jedes *QList*-Objekt enthält selbst einen Verweis auf ein »aktuelles Element«. Dieser Verweis wird durch fast jede Methode des Zugriffs neu gesetzt, also beim Einfügen, Löschen oder Suchen nach einem Element. Mit den Methoden *next* und *prev* kann man diesen Verweis um ein Element nach hinten bzw. vorn versetzen, mit *first* und *last* auf das erste bzw. letzte Element. Auf diese Weise kann man die Liste durchlaufen. Mit der Methode *current* erhält man einen Zeiger auf das aktuelle Element. Gibt es kein aktuelles Element (weil die Liste leer ist oder weil *prev* beim ersten oder *next* beim letzten Element der Liste aufgerufen wurde), so wird der Null-Zeiger zurückgeliefert.

Neben dieser Möglichkeit, alle Elemente zu durchlaufen, kann man auch für *QList*-Objekte ein Iterator-Objekt erzeugen, über das man auf einzelne Elemente zugreift (siehe Kapitel 4.7.3, *Iterator-Objekte*). Diese Iteratoren haben aber einen geringeren Stellenwert als bei *QValueList*.

Mit *remove* (ohne Parameter) wird das aktuelle Element aus der Liste entfernt. Ist *autoDelete* gesetzt, wird das Objekt mit *delete* freigegeben. Wenn Sie bei *remove* eine ganze Zahl als Parameter benutzen, wird das Objekt an der entsprechenden Position gelöscht.

Mit *take* wird ein Objekt aus der Liste entfernt und an den Aufrufer zurückgegeben. Auch wenn *autoDelete* gesetzt ist, wird das Element hier *nicht* mit *delete* freigegeben. (Das würde auch nicht viel Sinn machen, da das zurückgelieferte Element dann nicht mehr existieren würde.) Auch *take* kann man ohne Parameter benutzen (dann wird das aktuelle Element entfernt) oder mit einem ganzzahligen Parameter verwenden (dann wird das Element an dieser Position entfernt).

In der Liste kann auch nach einem Objekt gesucht werden. Mit der Methode *find* wird das erste Auftreten eines Elements gesucht, das den gleichen Inhalt wie das Element hat, das als Parameter übergeben wurde. (Dazu wird getestet, ob die Elemente bezüglich des Gleichheitsoperators gleich sind.) Mit *findNext* wird das nächste Auftreten des Elements nach dem aktuellen Element gesucht.

Die Suche kann effizienter durchgeführt werden, wenn man die Elemente nicht bezüglich des Gleichheitsoperators vergleicht, sondern nach einem bestimmten Zeiger sucht. Die entsprechenden Methoden heißen *findRef* und *findNextRef*. Mit den Methoden *contains* bzw. *containsRef* können Sie auch die Anzahl der Vorkommen eines Objekts zählen lassen.

Einige der Operatoren von *QValueList* sind für *QList* nicht implementiert. So können Sie *QList*-Objekte nicht aneinander hängen, und Sie können auch nicht einen einzelnen Zeiger mit *operator<<* anhängen lassen.

Das folgende einfache Beispiel soll die Anwendung des *QList*-Templates verdeutlichen. Dazu fügen wir in eine Liste, die Pointer auf *int*-Werte enthalten kann, die ersten fünf Primzahlen ein. Diese Liste lassen wir anschließend rückwärts ausgeben, und dann suchen wir in der Liste nach der Zahl 7.

```
QList<int> primliste;
// primliste verwaltet den Speicher der Elemente
primliste.setAutoDelete(true);
// Neue int-Elemente erzeugen und einfügen
primliste.append(new int(2));
primliste.append(new int(3));
primliste.append(new int(5));
primliste.append(new int(7));
primliste.append(new int(11));
// Elemente rückwärts ausgeben
primliste.last();
while(primliste.current())
{
    cout << *(primliste.current()) << endl;
    primliste.prev();
}
// Nach Element 7 suchen, dazu zunächst die Zahl
// 7 in einer Variablen speichern, damit wir
// einen Zeiger darauf benutzen können.
// Beachten Sie auch, dass wir hier nicht
// containsRef benutzen können, da die 7 in der
// Liste und die 7 in der Variablen a an
// verschiedenen Speicherstellen stehen.
int a = 7;
if(primliste.contains(a) != 0)
    cout << "7 ist enthalten" << endl;
else
    cout << "7 ist nicht enthalten ??? Fehler!" << endl;

// Anlegen einer Liste von QPushButton-Objekten
QList<QPushButton> buttonList;
for(int i = 1; i < 10; i++)
{
    QString text = QString::number(i) + ". Button";
    buttonList.append(new QPushButton(text, 0));
}
```

Sobald die Variable *primliste* in unserem Programm gelöscht wird, werden automatisch auch alle enthaltenen Zahlen mit *delete* freigegeben, da *autoDelete* gesetzt ist.

Wie Sie an diesem Beispiel sehen, ist *QList* nicht besonders gut für einfache Datentypen wie *int* oder *bool* geeignet, da man extra mit *new* eigene Speicherstellen auf dem Heap anlegen muss. Wenn Sie allerdings als Objekttyp komplexere Strukturen oder Klassen – insbesondere *QObject*, *QWidget* oder ähnliche Klassen – benutzen, ist das Template *QList* sehr praktisch, so wie es auch im Listing angewendet wird.

Qt definiert auch zwei weitere Listenklassen, die bereits konkrete Datentypen speichern: *QStringList* ist eine Unterklasse von *QValueList* *<QString>*. *QStrList* dagegen ist eine Unterklasse von *QList* *<char>*. Hierin werden also Zeiger auf *char* gespeichert, die jeweils den Anfang eines C-Strings darstellen. Beide Datentypen speichern also eine Liste von Strings, wobei *QStringList* Unicode-Strings verwalten kann, *QStrList* dagegen nur ASCII-Strings. *QStringList* benötigt dafür aber mehr Speicherplatz.

Keller und Schlange – *QStack* und *QQueue*

Diese beiden Klassen-Templates, *QStack* *<type>* und *QQueue* *<type>*, erzeugen Listen von Zeigern auf den Datentyp *type*.

Der Zugriff geschieht bei *QStack* mit den Methoden *push*, *pop* und *top*, die die Liste als Kellerspeicher (LIFO-Speicher) nutzen. Sie fügen ein neues Element mit *push* zum Stack hinzu. *pop* entfernt das zuletzt eingefügte Element wieder und liefert es zurück. *top* liefert das zuletzt eingefügte Element zurück, ohne es zu entfernen. Mit *isEmpty* können Sie testen, ob der Stack noch Elemente enthält. Typisches Anwendungsgebiet sind Algorithmen auf Backtracking-Basis.

Bei *QQueue* werden die Elemente in einer Warteschlange (FIFO-Speicher) organisiert. Mit *enqueue* können Sie ein Element hinten in die Schlange einreihen, mit *dequeue* das vorderste Element aus der Schlange entfernen und zurückliefern lassen. Mit *head* können Sie das vorderste Element einsehen, ohne es zu entfernen. Mit *isEmpty* können Sie testen, ob die Schlange leer ist. Die typische Anwendung dieser Klasse ist das Ablegen von noch zu bearbeitenden Aufgaben.

Die Zuordnungstabelle – *QMap* und *QDict*

In einer Zuordnungstabelle werden alle Elemente zusammen mit einem Suchschlüssel abgelegt. Die Hauptoperation, die man auf einer Zuordnungstabelle durchführt, ist die Suche nach einem Element anhand seines Schlüssels. Diese Operation ist daher besonders effizient implementiert. Innerhalb des Containers haben die Einträge keine eindeutige Reihenfolge (anders als etwa bei der linearen Liste oder dem Array). Hier folgen zunächst drei Beispiele für die Anwendung einer Zuordnungstabelle:

- Für die Übersetzung von Text-Strings aus einer Sprache in eine andere. In diesem Fall ist der Suchschlüssel ein String (der zu übersetzende Text); der Elementtyp ist ebenfalls ein String (der übersetzte Text).
- Für die Zuordnung von Hilfetexten zu einer Anzahl von QWidget-Objekten. Der Schlüsseltyp ist hier »Zeiger auf QWidget-Objekt« (der Zeiger auf das jeweilige Widget), der Elementtyp ist ein String (der Hilfetext für dieses Widget).
- Als Speicher sparender Ersatz für ein »dünn besetztes« Array, also ein Array, in dem nur sehr wenige Felder tatsächlich mit relevantem Inhalt gefüllt sind. Suchschlüssel ist hier der Datentyp *int* (die Position des Elements); der Elementtyp ist der Datentyp des »simulierten« Arrays.

Sie müssen bei einer Zuordnungstabelle beachten, dass Sie nicht zwei verschiedene Elemente mit dem gleichen Suchschlüssel in die Zuordnungstabelle eintragen. In diesem Fall würde bei der Suche immer nur eines der Elemente zurückgeliefert, der Zugriff auf das zweite Element wäre nicht mehr möglich. Die Suchschlüssel müssen also eindeutig sein.

Auch für die Containerart Zuordnungstabelle stellt Qt zwei verschiedene Varianten zur Verfügung: Der Datencontainer ist in *QMap* <schlüsseltyp, elementtyp> implementiert, der Zeigercontainer in den Klassen *QDict* <elementtyp>, *QAsciiDict* <elementtyp>, *QIntDict* <elementtyp> und *QPtrDict* <elementtyp>.

Methoden	Bedeutung	Laufzeit
<i>count</i>	liefert die Anzahl der enthaltenen Elemente	O(1)
<i>insert</i>	fügt neues Schlüssel/Element-Paar ein	O(log n)
<i>remove</i>	entfernt Element	O(log n)
<i>replace</i>	ersetzt Element zu einem Schlüssel durch ein anderes	O(log n)
<i>find</i>	sucht Element zu einem Schlüssel, liefert Iterator	O(log n)
<i>operator[]</i>	sucht Element zu einem Schlüssel, liefert das Element	O(log n)
<i>contains</i>	prüft, ob Suchschlüssel vorhanden ist	O(log n)
<i>clear</i>	löscht die Zuordnungstabelle	O(n)

Tabelle 4-6 Methoden von *QMap*

Für den Datencontainer *QMap* kann man sowohl den Schlüsseltyp als auch den Elementtyp fast beliebig wählen. Für beide Datentypen muss nur die Zuweisung (*operator=*) und die Initialisierung (Copy-Konstruktor) definiert sein. Zwei Elemente des Schlüsseltyps müssen außerdem mit dem *operator<* verglichen werden können. Als Schlüsseltyp kommen damit auf jeden Fall die Typen *int* und *double* in Frage, außerdem alle Zeigertypen sowie *QString* und *QCString*. Wollen Sie Objekte einer eigenen Klasse als Suchschlüssel verwenden, müssen Sie nur *operator<* definieren. *operator>* und *operator==* werden nicht unbedingt benötigt.

Die interne Realisierung der *QMap*-Klasse ist ein binärer Schwarz-Rot-Suchbaum, also ein Suchbaum mit Balancierungsmechanismen. Dadurch ist gewährleistet, dass auch bei sehr vielen Elementen die Suche nach einem Schlüssel sehr effizient ist.

Tabelle 4.6 enthält die wichtigsten Methoden im Umgang mit *QMap*-Containern.

Um alle Einträge in einem *QMap*-Container der Reihe nach zu durchlaufen, können Iteratoren benutzt werden (siehe Kapitel 4.7.3, *Iterator-Objekte*).

Das folgende Listing zeigt exemplarisch, wie man mit einem *QMap*-Objekt arbeitet. Dabei wollen wir zu einem Namen einer Person ihr Geburtsdatum abspeichern. Der Suchschlüsseltyp ist daher *QString*, der Elementtyp *QDate*.

```
// Container anlegen
QMap <QString, QDate> geburtstage;

// Container mit Daten füllen,
// entweder mit insert...
geburtstage.insert ("Donald E. Knuth",
                   QDate (1938, 1, 10));
// ...oder mit operator[]
geburtstage ["Steven Jobs"] = QDate (1955, 2, 25);

// Abfragen, ob Suchschlüssel vorhanden
if (geburtstage.contains ("Bill Gates"))
{...}

// Iterator-Objekt anlegen
QMap <QString, QDate>::Iterator it;

// Nach einer Person suchen
it = geburtstage.find ("Steven Jobs");
if (it == geburtstage.end())
    // nicht gefunden
else
    // gefunden

// Oder Zugriff per operator[]
// Achtung: Falls der Schlüssel noch nicht existierte,
// wird er mit einem QDate-Objekt angelegt, das vom
// parameterlosen QDate-Konstruktor erzeugt wird (ergibt
// ein ungültiges Datum).
QDate datum = geburtstage ["Steven Jobs"];

// Nun alle gespeicherten Personen durchlaufen und alle
// an einem Freitag geborenen ausgeben
for (it = geburtstage.begin();
```

```

        it != geburtstage.end(); ++it)
    {
        if (it.data().dayOfWeek() == 5)
            qDebug ("Freitagskind %s",
                    it.key().ascii());
    }

```

Beachten Sie, dass die umgekehrte Zuordnung von Geburtstagen zu Personen nicht mit einer Zuordnungstabelle gelöst werden sollte, da mehrere Personen den gleichen Suchschlüssel (nämlich das gleiche Geburtsdatum) besitzen können.

Die Zeigercontainer der Zuordnungstabelle werden in Qt in den Template-Klassen *QDict*, *QAsciiDict*, *QIntDict* und *QPtrDict* erzeugt. Im Gegensatz zu *QMap* nimmt jede dieser Klassen nur einen Datentyp entgegen, der den Elementtyp festlegt. Der Schlüsseltyp ist automatisch durch die verwendete Klasse festgelegt, wie es in Tabelle 4.7 zu sehen ist.

Template	Schlüsseltyp
<i>QDict</i>	<i>QString</i> (Unicode-String)
<i>QAsciiDict</i>	<i>char *</i> (ASCII-String, mit 0-Zeichen abgeschlossen)
<i>QIntDict</i>	<i>int</i>
<i>QPtrDict</i>	<i>void *</i> (Zeiger auf ein beliebiges Objekt)

Tabelle 4-7 Schlüsseltypen der Zeiger-Container von Zuordnungstabellen

Die interne Realisierung ist in jedem Fall eine Hash-Tabelle. Über den Schlüssel lässt sich sehr schnell der passende Wert in der Hash-Tabelle finden. Die zu speichernden Elemente sind in einem Array abgelegt, und aus dem Schlüssel wird der Index berechnet, unter dem das Schlüssel/Wert-Paar gespeichert werden soll. In den Qt-Hash-Tabellen ist jeder Eintrag des Arrays eine Liste, so dass auch mehrere Paare pro Array-Position eingefügt werden können. So enthält ein Objekt der Klasse *QIntDict* <*QWidget*> eine Hash-Tabelle mit Paaren, die aus einer *int*-Zahl als Schlüssel und einem Zeiger auf ein *QWidget*-Objekt als Element bestehen.

Im Konstruktor einer Hash-Tabellen-Klasse müssen Sie die Größe des Arrays angeben. Der Default-Wert ist 17. Die Hash-Tabelle arbeitet dann optimal, wenn es zu möglichst wenig Kollisionen kommt, also jeder Array-Eintrag höchstens ein Element enthält. Daher sollte diese Größe mindestens der Anzahl der Elemente entsprechen, die Sie als Maximum erwarten. Besser ist es, den erwarteten Maximalwert noch einmal um 50% zu überschreiten. Jeder Array-Eintrag ist nur ein Zeiger auf die Liste der Elemente, belegt also je nach Prozessor vier oder acht Byte. Es ist also nicht besonders kritisch, wenn Sie den Wert zu groß wählen. Auch wenn Sie mehr Elemente in der Hash-Tabelle speichern, als das Array Ein-

träge hat, funktioniert die Hash-Tabelle weiterhin. Sie ist nur nicht mehr so effizient. Die Array-Größe kann auch nachträglich noch mit der Methode *resize* verändert werden. Weil diese *resize*-Operation aber aufwendig ist, da alle Paare aus der alten in die neue Hash-Tabelle kopiert werden müssen, sollten Sie von dieser Methode nur in Ausnahmefällen Gebrauch machen.

Die Größe des Arrays sollte möglichst eine Primzahl sein, da sich so in der Regel bessere Verteilungen innerhalb des Arrays ergeben. Tabelle 4.8 enthält Primzahlen verschiedener Größenordnungen, jeweils etwa mit einem Faktor von 2 von einer Zahl zur nächsten, die Sie benutzen können.

3	5	11	17	37	67	131
257	521	1.031	2.053	4.099	8.209	16.411
32.771	65.537	131.101	262.147	524.309	1.048.583	2.097.169
4.194.319	8.388.617	16.777.259	33.554.467	67.108.879	134.217.757	268.435.459

Tabelle 4-8 Primzahlen verschiedener Größenordnungen

Alle Klassen besitzen die gleichen Methoden wie die in Tabelle 4.6 aufgelisteten Methoden von *QMap*. Eine Ausnahme ist hier die Methode *contains*, die nicht vorhanden ist. Sie ist aber auch nicht nötig, denn für den Fall, dass ein Schlüssel nicht existiert, liefern *find* und auch der *operator[]* einen Null-Zeiger zurück. Das Laufzeitverhalten der Hash-Tabellen ist ebenfalls besser als das von *QMap*: Falls die Hash-Tabelle groß genug gewählt wurde und die Anzahl der Kollisionen vernachlässigbar bleibt, ist das Verhalten aller Einfüge-, Entfernen- und Suchmethoden $O(1)$.

Im Gegensatz zu *QMap* können Sie in den Hash-Tabellen Einträge nur mit *insert* hinzufügen, nicht aber mit dem *operator[]*. Die Suche nach einem Element mit einem bestimmten Schlüssel geschieht entweder mit *find* oder mit dem *operator[]*. Wurde kein Eintrag mit diesem Schlüssel gefunden, wird ein Null-Zeiger zurückgeliefert.

Mit *remove* kann man ein Schlüssel/Wert-Paar (identifiziert durch den Schlüssel) entfernen lassen. Ist *autoDelete* aktiviert, wird der Wert dabei mit *delete* gelöscht. Mit *take* können Sie einen Wert zu einem Schlüssel erfragen und dabei gleichzeitig das Schlüssel/Wert-Paar entfernen. Der Wert wird dabei nicht mit *delete* freigegeben, auch wenn *autoDelete* aktiv sein sollte. Mit *replace* können Sie schließlich einem Schlüssel einen neuen Wert zuordnen. Falls es noch kein Paar mit diesem Schlüssel gibt, wird das Paar eingefügt.

In einem kleinen Beispiel wollen wir eine Hash-Tabelle benutzen, die einem Zeiger – in diesem Fall einem Zeiger auf ein Widget – einen String zuordnet. So etwas kann beispielsweise benutzt werden, wenn mehrere Signale mit einem Slot

verbunden sind. Mit der Methode *sender* kann man im Slot ermitteln, von welchem Objekt das Signal ausgesendet wurde. Diesen Zeiger wollen wir als Schlüssel für unsere Hash-Tabelle benutzen:

```
QPushButton *b1, *b2, *b3, *b4;

QPtrDict<QString> hash (7); // Array mit 7 Einträgen
// Einfügen von Schlüssel/Wert-Paaren
hash.insert (b1, "Eins");
hash.insert (b2, "Zwei");
hash.insert (b3, "Drei");
hash.insert (b4, "Vier");

// Suche nach dem zugehörigen String zu einem Zeiger
cout << hash [b3] << endl; // Gibt "Drei" aus
```

Bei nur vier Elementen lohnt sich der Einsatz einer Hash-Tabelle sicherlich nicht. Da aber die Zugriffszeit auf eine Hash-Tabelle auch bei steigender Anzahl von Einträgen in etwa konstant bleibt, ist eine Hash-Tabelle ab 20 Einträgen sehr effizient im Vergleich zu anderen Strukturen. Wählen Sie dabei aber eine ausreichende Größe der Hash-Tabelle.

Der Cache – *QCache*

Die Template-Klassen *QCache*, *QAsciiCache*, *QIntCache* und *QPtrCache* erzeugen genau wie die entsprechenden *QDict*-Klassen eine Hash-Tabelle mit Schlüssel/Wert-Paaren. Jedem Paar kann hier aber ein Kostenwert zugeordnet werden. Die Gesamtkosten des Cache-Objekts können auf einen Wert begrenzt werden. Wird dieser Wert überschritten, werden die Paare gelöscht, die am längsten nicht mehr referenziert wurden. Zusätzlich kann man den Einträgen Prioritäten geben, so dass zuerst Einträge mit geringerer Priorität aus dem Cache entfernt werden.

Diese Klassen werden meist dann eingesetzt, wenn man Objekte für einen schnellen Zugriff im Speicher halten möchte, nachdem sie einmal eingelesen wurden. Sie werden zur Zeit bereits benutzt, um einen Cache für *Pixmap*-Dateien anzulegen. Jede Grafik, die bereits einmal eingelesen wurde, wird in einem *QAsciiCache*-Objekt mit Werten vom Typ *QPixmap** (Zeiger auf *QPixmap*) gespeichert. Wird später einmal die gleiche Datei angefordert, kann das *QPixmap*-Objekt direkt aus dem Cache benutzt werden. Der Kostenwert für einen Eintrag ist hier der benötigte Speicherplatz.

Beachten Sie, dass alle Klassen immer mit aktiviertem *autoDelete* arbeiten. Das Cache-Objekt löscht seine Einträge selbstständig bei Bedarf. Beachten Sie außerdem beim Erfragen eines Objekts, dass ein Zeiger auf den Wert eines Paares, den das Cache-Objekt zurückliefert, nach einiger Zeit ungültig sein könnte, da das Paar gelöscht wurde. Benutzen Sie diesen Zeiger also nur so lange, bis Sie den nächsten Eintrag in den Cache schreiben.

4.7.3 Iterator-Objekte

Für die Objekte der Container-Klassen definiert Qt zusätzliche Klassen-Templates, die einzig den Zweck haben, alle Elemente dieses Objekts zu durchlaufen. Diese Objekte heißen *Iterator-Objekte*, und die Klassen werden konsequent durch Anhängen von »Iterator« an den Klassennamen benannt. Will man zum Beispiel alle Elemente eines *QIntDict*-Objekts durchlaufen (um sie zum Beispiel in einer Datei zu speichern), konstruiert man ein Objekt der Klasse *QIntDictIterator*, das man dem *QIntDict*-Objekt zuordnet. Das *QIntDictIterator*-Objekt bietet jetzt Methoden, um das nächste Element auszuwählen oder auf das ausgewählte Element zuzugreifen.

Zu jedem Container-Objekt können beliebig viele Iterator-Objekte gleichzeitig aktiv sein. Sie arbeiten unabhängig voneinander.

Die Arbeitsweise der Iterator-Objekte unterscheidet sich (aus Kompatibilitätsgründen) zwischen Datencontainern und Zeigercontainern. Daher werden die beiden Typen getrennt voneinander beschrieben.

Iteratoren von Datencontainern

Die Datencontainer von Qt (*QValueList*, *QMap*) sind in Hinblick auf die Syntax des Zugriffs stärker an STL angelehnt. Entsprechend sind auch die Iteratoren wie in der STL definiert. Die Datencontainer benutzen Iteratoren für zwei Aufgaben: zum Durchlaufen aller Elemente im Container, und zur Angabe einer Position im Container. Wir wollen hier an einfachen Beispielen die Vorgehensweise beim Arbeiten mit Iteratoren für Datencontainer erläutern.

```
// Zunächst ein QValueList-Objekt anlegen und füllen
QValueList <QString> zahlen;
zahlen << "Null" << "Eins" << "Zwei" << "Drei";

// Nun ein Iterator-Objekt anlegen
// Datentyp ist der in der Template-Klasse
// QValueList <QString> definierte Datentyp Iterator
QValueList <QString>::Iterator it;

// In einer for-Schleife nun alle Elemente durchlaufen
for (it = zahlen.begin(); it != zahlen.end(); ++it)
{
    // Zugriff auf das Element, auf das der Iterator
    // aktuell zeigt, über den operator*
    QString zahl = *it;
}

// Finden eines Elements in der Liste
it = zahlen.find ("Zwei");
// Falls nicht gefunden, hat der Iterator den Wert
```

```

// zahlen.end()
if (it == zahlen.end())
    qDebug ("Nicht vorhanden!");
else
{
    QString zahl = *it;
    // Element, auf das der Iterator zeigt, entfernen.
    // Achtung: Der Iterator ist nach dieser Operation
    // ungültig. Deshalb setzen wir ihn hier auf den
    // Rückgabewert von remove, der auf das nachfolgende
    // Element zeigt.
    it = zahlen.remove (it);
}

```

Beachten Sie bitte, dass nur der Präfix-Operator für Iteratoren definiert ist (***++it***), aber nicht der Postfix-Operator (***it++***).

Besondere Vorsicht ist geboten, wenn Sie gleichzeitig in einer Schleife den gesamten Container durchlaufen wollen und dabei einige Elemente, die spezielle Kriterien erfüllen, löschen wollen. Das folgende Beispiel funktioniert nicht:

```

// Dieses Beispiel überspringt einige Elemente
QValueList <QString>::Iterator it;
for (it = zahlen.begin(); it != zahlen.end(); ++it)
{
    if (Bedingung)
        it = zahlen.remove (it);
}

```

Der Grund liegt darin, dass beim Löschen *it* auf das nachfolgende Element gesetzt wird, es aber anschließend nochmals mit ***++it*** um ein Element verschoben wird.

Stattdessen benutzen Sie das folgende Schema:

```

// Das ist die korrekte Lösung
QValueList <QString>::Iterator it = zahlen.begin();
while (it != zahlen.end())
{
    if (Bedingung)
        it = zahlen.remove (it);
    else
        ++it;
}

```

Für konstante Container-Objekte können Sie nur den *ConstIterator* verwenden, nicht aber den normalen Iterator. Konstante Container-Objekte treten besonders häufig auf, wenn ein Container an eine Methode als konstante Referenz übergeben wird. Hier sehen Sie ein Beispiel für die Anwendung von *ConstIterator*.

```
int MyClass::calculateSum (const QList<int> &list)
{
    int result = 0;
    QList<int>::ConstIterator it;
    for (it = list.begin(); it != list.end(); ++it)
        result += *it;
    return result;
}
```

Iteratoren von Zeigercontainern

Die Zeigercontainer (*QList*, *Q*Dict*, *Q*Cache*) besitzen nicht die Methoden *begin* und *end*, die Anfangs- und End-Iteratoren liefern. Für diese Klassen definieren Sie stattdessen ein Iterator-Objekt und übergeben den Container als Parameter im Konstruktor. Der erzeugte Iterator zeigt dann direkt auf das erste Element des Containers. Die Namen der Iteratorklassen werden aus dem Namen der Containerklasse durch Anhängen von »Iterator« gebildet. So ist beispielsweise die Iteratorklasse zu *QIntDict* die Klasse *QIntDictIterator*.

Für die Zeigercontainer werden die Iteratoren ausschließlich zum Durchlaufen aller Elemente benutzt. Die Position eines Elements innerhalb des Containers wird auf andere Weise angegeben (durch die aktuelle Position bei *QList* und durch den Suchschlüssel bei den *Q*Dict*-Klassen).

Der häufigste Einsatz eines Iterator-Objekts ist das einfache Durchlaufen durch alle Elemente eines Container-Objekts. Das soll im Folgenden beispielhaft für die *QDict*-Container-Klasse gezeigt werden. Es gibt mehrere Möglichkeiten, ein Programmstück zu formulieren. Die vier gängigsten Arten werden wir hier anführen. Welche Sie wählen, bleibt ganz Ihrem Geschmack überlassen.

- Möglichkeit 1 (benutzt die Methode *current* und den ++-Operator):

```
QList<QWidget> dict; // dict sei bereits gefüllt
QListIterator<QWidget> it (dict); // Iterator-Objekt
QWidget *w;
for (; w = it.current (); ++it) {
    // bearbeite w
}
```

- Möglichkeit 2 (benutzt die Methode *toFirst* und den ++-Operator):

```
QDict<QWidget> dict;
QDictIterator<QWidget> it (dict);
for (QWidget *w = it.toFirst(); w; w = ++it) {
    // bearbeite w
}
```

- Möglichkeit 3 (benutzt den *cast*-Operator und den *++*-Operator):

```
QDict <QWidget> dict;
QDictIterator <QWidget> it (dict);
QWidget *w;
while (it) {
    w = it;
    ++it;
    // bearbeite w
}
```

- Möglichkeit 4 (benutzt den überladenen *()*-Operator):

```
QDict <QWidget> dict;
QDictIterator <QWidget> it (dict);
QWidget *w;
while (w = it()) {
    // bearbeite w
}
```

Die erste Möglichkeit ist sicherlich die Version, die auch für einen ungeübteren Qt-Programmierer am leichtesten zu durchschauen ist. Die anderen Möglichkeiten sind zum Teil kürzer. Beachten Sie, dass nur der Präfix-*++*-Operator definiert ist, nicht aber der Postfix-*++*-Operator. *++it* ist also erlaubt, *it++* dagegen nicht.

4.7.4 Die String-Klassen – *QString* und *QCString*

Qt bietet mit den Klassen *QString* und *QCString* sehr mächtige Klassen für den Umgang mit Texten. Die Klasse *QCString* enthält dabei ASCII-Text im C-Format, also mit einem Byte pro Zeichen (Datentyp *char*), bei dem das Ende des Textes mit dem Wert 0 gekennzeichnet ist. *QString* enthält Unicode-Text, also mit zwei Byte pro Zeichen (Datentyp *QChar*). Die Länge des Textes wird dabei extra gespeichert und nicht durch ein besonderes Zeichen markiert.

Alle Texte, die auf dem Bildschirm erscheinen sollen, sollten möglichst in einer Variablen der Klasse *QString* gespeichert werden, damit alle internationalen Zeichensätze unterstützt werden können. Das gilt natürlich insbesondere für Texte, die in die eingestellte Landessprache übersetzt werden sollen (siehe Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*). Alle Texte, die nur intern verwendet werden und keinen Gebrauch von ausländischen Zeichensätzen machen – zum Beispiel englische Texte, die nicht übersetzt werden –, können in einer *QCString*-Variablen gespeichert werden, da diese weniger Speicher verbraucht.

Die Qt-Bibliothek enthält alle nötigen Zuweisungsoperatoren, Cast-Operatoren und Copy-Konstruktoren, um Daten der Typen *const char**, *QCString* und *QString* ineinander umzuwandeln. Das geschieht meist automatisch, ohne dass sich der

Programmierer darüber Gedanken machen müsste. Man sollte natürlich darauf achten, dass möglichst wenig Konvertierungen vorgenommen werden, da eine Umwandlung immer Zeit benötigt, und dass beim Konvertieren von *QString* nach *QString* sowie von *QString* nach *const char** Informationen verloren gehen können.

Die folgende Zeile legt eine Variable vom Typ *QString* an und initialisiert sie mit dem Text »KDE- und Qt-Programmierung«:

```
QString text ("KDE- und Qt-Programmierung");
```

Die Klassen *QString* und *QString* verfügen außerdem über viele Methoden, um den Text zu manipulieren. Der *+*-Operator und der *+=*-Operator sind so überladen, dass man zwei *QString*- (bzw. *QString*-)Objekte aneinander hängen kann. Ebenso kann man mit den Methoden *append* und *prepend* einen anderen String hinten bzw. vorn an den Text anhängen lassen. *insert* fügt einen Text an einer bestimmten Stelle ein. Mit *remove* kann man einen Teil des Textes entfernen, mit *replace* kann man ihn durch einen anderen Text ersetzen.

Viele andere Methoden erzeugen ein neues *QString*-Objekt, ohne das alte zu verändern. Die Methode *lower* wandelt den Text beispielsweise Zeichen für Zeichen in Kleinbuchstaben um, *upper* entsprechend in Großbuchstaben. *left* und *right* liefern einen Teilstring am linken bzw. rechten Rand zurück, *mid* einen Teilstring aus der Mitte des Textes. Die Methode *stripWhiteSpace* liefert einen neuen String zurück, bei dem die Zeichen Leerschritt, Tabulator, Zeilenvorschub und Wagenrücklauf (die so genannten *Whitespace-Zeichen*) am Anfang und Ende des Textes entfernt worden sind. Das ist besonders praktisch beim Einlesen von Texten aus einer Datei. Die Methode *simplifyWhiteSpace* liefert ebenfalls eine Kopie des Strings. Sie entfernt aus dieser ebenfalls die *Whitespace-Zeichen* am Anfang und Ende des Textes und ersetzt zusätzlich mehrere aufeinander folgende Zeichen dieser Gruppe im Inneren des Textes durch einen einzigen Leerschritt. So können Sie zum Beispiel eingegebene Texte in ein einheitliches Format bringen.

Auch die Methoden zum Suchen und Ersetzen innerhalb der Texte sind sehr mächtig. Man kann nach einem einzelnen Zeichen suchen, nach einem Teilstring oder nach einem regulären Ausdruck. Man kann zusätzlich festlegen, ob man zwischen Groß- und Kleinschreibung unterscheiden möchte oder nicht. Ein regulärer Ausdruck wird in einem Objekt der Klasse *QRegExp* gespeichert. Von dieser Klasse benötigt man meist nur den Konstruktor, der drei Parameter erfordert. Der erste Parameter bestimmt das Suchmuster (als String), nach dem gesucht werden soll, der zweite Parameter (mit Default-Wert *true*) legt fest, ob zwischen Groß- und Kleinschreibung unterschieden wird (*true*) oder nicht (*false*). Der dritte Parameter (Default-Wert *false*) legt fest, ob es sich um einen vollständigen regulären Ausdruck handelt (*false*) oder ob nur die Wildcard-Zeichen *?* und ***

interpretiert werden sollen (*true*). In einem vollständigen regulären Ausdruck haben einige Sonderzeichen eine spezielle Bedeutung. Sie sind in Tabelle 4.9 aufgelistet.

Sonderzeichen	Beispiel	Bedeutung
<code>^</code>	<code>^ABC</code>	Der Ausdruck wird nur gematcht, wenn er am Anfang steht.
<code>\$</code>	<code>ABC\$</code>	Der Ausdruck wird nur gematcht, wenn er am Ende steht.
<code>[]</code>	<code>[0-9A-Za-z]</code>	Ein beliebiges Zeichen aus der Menge zwischen den Klammern wird gematcht (hier alle Ziffern, Groß- und Kleinbuchstaben).
<code>*</code>	<code>X*</code>	Der vorhergehende Ausdruck wird beliebig oft (auch keinmal) gematcht.
<code>+</code>	<code>X+</code>	Der vorhergehende Ausdruck wird beliebig oft, aber mindestens einmal gematcht.
<code>?</code>	<code>X?</code>	Der vorhergehende Ausdruck ist optional.

Tabelle 4-9 Sonderzeichen für reguläre Ausdrücke

Die Methoden in *QString* (und *QString*) zum Suchen heißen *find* (sucht ab einer bestimmten Stelle, standardmäßig vom Anfang des Strings), *findRev* (sucht rückwärts ab einer bestimmten Stelle, standardmäßig vom Ende des Strings) und *contains* (zählt die Anzahl der Vorkommen des Teilstrings). Das Suchen und Ersetzen geschieht mit der Methode *replace*. (Die oben beschriebene Methode *replace* zum Ersetzen eines Teilstrings durch einen anderen unterscheidet sich von dieser Methode nur in der Anzahl und dem Typ der Parameter.)

Die Methode *setNum* setzt den Text eines *QString*- oder *QString*-Objekts auf den Text, der die übergebene Zahl darstellt. Die folgende Zeile bewirkt nun, dass *s* den Text »-264« zugewiesen bekommt.

```
QString s;
s.setNum (-264);
```

Das kann auch benutzt werden, um einen Text zu erzeugen, der Zahlenwerte aus Variablen enthält. Die folgende Zeile erzeugt den String »Die obere Ecke liegt bei (10/30).«

```
QPoint p (10,30);
QString s = "Die obere Ecke liegt bei (" +
    QString().setNum (p.x()) +
    "/" +
    QString().setNum (p.y()) +
    ").";
```

Beachten Sie, dass der `++`-Operator so überladen wurde, dass er auch Elemente vom Typ `char*` und `QString` aneinander hängen kann. Dabei muss entweder das erste oder das zweite Argument des Operators vom Typ `QString` (oder `QString`) sein. Eine Addition von zwei Argumenten vom Typ `char*` ist nicht möglich.

Um nicht so viele einzelne Strings aneinander hängen zu müssen, gibt es in `QString` und `QString` die Methode `sprintf`, die genauso arbeitet wie die globale Funktion `sprintf` in `stdio.h` bzw. `stdlib.h`. Sie arbeitet aber natürlich auf einem `QString`- bzw. `QString`-Objekt. Das Ergebnis von oben kann man also auch mit folgender Zeile erreichen:

```
QString s = QString().
    sprintf ("Die obere Ecke liegt bei (%d/%d).") ,
    p.x(), p.y());
```

Beachten Sie, dass die angegebenen Platzhalter im Text mit den Datentypen der Parameter übereinstimmen müssen. Sonst kann es zu undefinierten Ergebnissen und im schlimmsten Fall zum Programmabbruch kommen. Beachten Sie außerdem, dass `QString` einen Pufferüberlauf erzeugen kann, wenn der resultierende Text länger als 256 Zeichen (und gleichzeitig länger als die bisherige Länge des Textes in `QString`) ist. `QString` hat diese Einschränkung nicht.

`QString` (aber nicht `QString`) bietet noch eine weitere Methode, `arg`, um Platzhalter durch Werte ersetzen zu lassen. Diese Methode wird hauptsächlich bei der Übersetzung in andere Sprachen benutzt. Genauer dazu erfahren Sie in Kapitel 4.9.3, *Texte mit Platzhaltern übersetzen*.

4.7.5 Flexibler Datentyp – `QVariant`

In manchen Fällen möchte man den Datentyp einer Variablen oder eines Parameters nicht festlegen, sondern dort beliebige Datentypen abspeichern bzw. übergeben. In C und C++ gibt es dazu zum einen einen Zeiger auf den Datentyp `void`, zum anderen die Konstruktion der *union*. Beide Konzepte haben aber den entscheidenden Nachteil, dass es keine Möglichkeit gibt, den Datentyp des abgespeicherten Elements nachträglich wieder zu bestimmen. Man muss sich also darauf verlassen, dass auch tatsächlich enthalten ist, was erwartet wird.

Speziell für die Realisierung der Properties in `QObject` wird ein Konzept benötigt: Man muss über eine einzige Methode `QObject::setProperty` verschiedenste Datentypen übergeben können (siehe auch Kapitel 3.1.4, *Informationen über die Klassenstruktur*). Daher wurde eine neue Klasse namens `QVariant` eingeführt.

In einer Variablen oder einem Parameter vom Typ `QVariant` kann man zur Zeit Daten der folgenden Typen abspeichern: `int`, `unsigned int`, `double`, `bool`, `QString`, `QString`, `QStringList`, `QFont`, `QPixmap`, `QBitmap`, `QImage`, `QBrush`, `QPoint`, `QRect`,

QSize, *QColor*, *QPalette*, *QColorGroup*, *QIconSet*, *QPointArray*, *QRegion* und *QCursor*. Diese Liste kann in Zukunft noch wachsen.

Man erzeugt ein *QVariant*-Objekt mit den gewünschten Daten, indem man dem Konstruktor diese Daten übergibt. Das folgende Listing erzeugt verschiedene *QVariant*-Objekte. Beachten Sie, dass zur Erzeugung eines *QVariant*-Objekts mit einem *bool*-Inhalt ein zusätzlicher Dummy-Parameter (z.B. die Zahl 0) übergeben werden muss, da einige Compiler nicht zwischen dem Datentyp *bool* und dem Datentyp *int* unterscheiden können.

```
QVariant v1 (42);           // Typ int
QVariant v2 (QPoint (40, 70)); // Typ QPoint
QVariant v3 (true, 0);      // Typ bool
QVariant v4 ("Hallo");      // Typ QString
QVariant v5 (QString ("Hallo")); // Typ QString
QVariant v6 (BarIcon ("filenew")); // Typ QPixmap
```

Mit Hilfe der Methode *type* kann man den aktuell gespeicherten Datentyp eines *QVariant*-Objekts ermitteln. Der Rückgabewert ist ein Aufzählungstyp, der alle möglichen Datentypen enthält. Mit der statischen Methode *QVariant::typeName* kann man diesen Aufzählungstyp in einen Text umwandeln lassen. Alternativ kann man mit der Methode *typeName* auch direkt die Bezeichnung des gespeicherten Datentyps erfahren. Hier folgt als Beispiel eine Funktion, die den Datentyp eines *QVariant*-Objekts auf dem Bildschirm ausgibt:

```
void printType (const QVariant &v)
{
    qDebug ("Variant Type is %s", v.typeName());
}

int main ()
{
    // Diese Aufrufe erzeugen automatisch temporäre
    // QVariant-Objekte
    printType (42);           // Typ int
    printType (QPoint (40, 70)); // Typ QPoint
    printType ("Hallo");      // Typ QString
    printType (QString ("Hallo")); // Typ QString

    // Für bool müssen wir selbst das Objekt anlegen
    printType (QVariant (true, 0)); // Typ bool
}
```

Um die Daten aus einer *QVariant*-Variable oder einem solchen Parameter wieder auszulesen, besitzt *QVariant* die Methoden *asT* und *toT*, wobei für *T* die unterstützten Datentypen einzusetzen sind. Die Methode *toT* erzeugt dazu ein neues Objekt der Klasse *T* und kopiert die Daten aus dem *QVariant*-Objekt in dieses

Objekt. Dieses Objekt wird von der Methode zurückgegeben. Falls im *QVariant*-Objekt Daten von einem anderen Typ abgespeichert waren, die nicht in den Typ *T* umgewandelt werden können, bleibt das erzeugte Objekt uninitialisiert, enthält also die Default-Werte des jeweiligen Typs. Im Gegensatz dazu verwandeln die *asT*-Methoden die Daten im *QVariant*-Objekt in den Datentyp *T* und geben dann eine Referenz auf diesen Datentyp zurück. Lagen die Daten ohnehin schon im Format *T* vor, so ändert sich am *QVariant*-Objekt nichts. Könnte die Umwandlung nicht durchgeführt werden, so hat das *QVariant*-Objekt anschließend den Datentyp *T*, aber uninitialisierte Daten. Hier sehen Sie ein paar Beispiele für den Umgang mit diesen Methoden:

```
QVariant v1 (QPoint (40, 70));
int x = v1.asPoint().x(); // keine Umwandlung

QVariant v2 (42);
int a = v2.toInt ();      // v2 bleibt int
double b = v2.toDouble (); // v2 bleibt int
double c = v2.asDouble (); // v2 wird double
```

Welche Datentypen ineinander umgewandelt werden können, ist in der Online-Dokumentation zur Klasse *QVariant* aufgeführt.

QVariant bietet zusätzlich die Möglichkeit, auch große Datenmengen und komplexe Strukturen abzuspeichern. Zusätzlich zu den oben angegebenen Typen können Sie in einem *QVariant*-Objekt auch eine *QValueList* <*QVariant*> oder eine *QMap* <*QString*, *QVariant*> ablegen. Mit der *QValueList* können Sie so eine Menge von Werten in einem *QVariant*-Objekt ablegen, wobei jeder Wert vom Typ *QVariant* ist. Die einzelnen Werte der Liste brauchen also nicht vom gleichen Typ zu sein, sie können sogar selbst wieder eine *QValueList* oder eine *QMap* sein. Hier ein Beispiel, wie eine solche Liste angelegt werden kann:

```
// eine Hilfsliste mit Primzahlen
QValueList<QVariant> primliste;
primliste << 2 << 3 << 5 << 7 << 11;

// zunächst leere Liste
QValueList<QVariant> list;
// Füllen mit verschiedenen Typen
list << 42 << QPoint (40, 70) << "Hallo" << primliste;

// Ablegen in einer einzigen Variablen
QVariant v (list);

// Datentyp von v ist "List"
printType (v);

// Ausgeben der Datentypen der einzelnen Elemente
```

```
// Ergebnis ist "int", "Point", "CString", "List"
QValueList<QVariant>::Iterator it;
for (it = v.listBegin(); it != v.listEnd(); ++it)
    printType (*it);
```

v enthält am Ende dieser Anweisungen eine Liste von vier Elementen, die der Reihe nach den Typ *int*, *QPoint*, *QString* und *QValueList<QVariant>* haben. Das letzte Element enthält selbst wiederum eine Liste von fünf *QVariant*-Objekten, hier alle vom Typ *int*.

4.8 Der Unicode-Standard

Der vom Unicode-Konsortium festgelegte Standard behebt eine Reihe von Problemen, die bei der Darstellung von Sonderzeichen aus verschiedenen Sprachen der Welt auftreten. Mit der weiteren Verbreitung von Unicode löst dieses Format mehr und mehr den ASCII-Standard und die lokalen Zeichensätze (z.B. Latin-1) ab.

Die Klasse *QString*, die in der Qt-Bibliothek an vielen Stellen verwendet wird, speichert Strings in diesem Unicode-Format ab. *QString* wird überall dort benutzt, wo Texte auf dem Bildschirm angezeigt werden sollen. Insbesondere die Internationalisierung wäre ohne Nutzung dieses Standards kaum möglich (siehe Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*).

Die genaue Definition des Unicode-Standards ist im Buch »The Unicode Standard, Version 3.0« aus dem Addison-Wesley Verlag (ISBN 0-201-61633-5) festgelegt. Die jeweils aktuellste Definition und weitere Informationen finden Sie auf der Homepage <http://www.unicode.org/>.

Wenn Sie keine speziellen Ansprüche an die Darstellung von Sonderzeichen stellen, können Sie dieses Kapitel überspringen. In vielen Fällen erledigt die Klasse *QString* den Umgang mit Unicode-Texten für Sie unsichtbar im Hintergrund.

4.8.1 Unicode-Zeichen

Jeder darzustellende oder zu verarbeitende Text besteht aus einer Reihe von Zeichen. Im Unicode-Standard ist jedem Zeichen der gängigen Schriftsprachen der Welt eine eindeutige Zahl zugewiesen. Jedes Zeichen belegt dabei genau 16 Bit Speicherplatz. Somit sind 65 536 verschiedene Zeichen möglich – ausreichend, um zum Beispiel neben den lateinischen Schriftzeichen, Ziffern und Satzzeichen auch kyrillische, griechische, japanische oder chinesische Schriftzeichen zu repräsentieren. Hinzu kommen noch Währungssymbole, mathematische Zeichen, Piktogramme (Dingbats) sowie Spezialzeichen, die für Zeilenumbruch und Worttrennung zuständig sind. Im Gegensatz dazu belegt ein Zeichen des Latin1-

Zeichensatzes, der eine Erweiterung des ASCII-Zeichensatzes darstellt und in Europa sehr verbreitet ist, nur 8 Bit. Daher sind in ihm nur 256 verschiedene Zeichen möglich – zu wenig, um allein alle europäischen Sprachen abzudecken.

Im Unicode-Standard sind für jedes Zeichen unter anderem folgende Eigenschaften festgelegt:

- die Zahl, durch die dieses Zeichen repräsentiert wird (üblicherweise beschrieben durch eine vierstellige Hexadezimalzahl, zum Beispiel 0x0051 für das Zeichen »Q«)
- den (eindeutigen) Namen des Zeichens (ein ASCII-String, zum Beispiel »LATIN CAPITAL LETTER Q« für das Zeichen »Q«)
- die Kategorie, der dieses Zeichen entspricht (festgelegt durch einen zweibuchstabigen Code, zum Beispiel Kategorie »Lu« = »letter upcase« für das Zeichen »Q«; siehe auch Tabelle 4.10)
- die natürliche Schreibrichtung (zum Beispiel »L« = »left-to-right« für das Zeichen »Q«)
- die Unicode-Werte für den zugehörigen Großbuchstaben oder Kleinbuchstaben, falls vorhanden (zum Beispiel 0x0071 als Kleinbuchstabe »q« für das Zeichen »Q«)

Nicht festgelegt ist dagegen, wie die Darstellung eines Zeichens auf dem Bildschirm auszusehen hat.

Kürzel	QChar::Category	Bedeutung	Beispiele
Lu	Letter_Uppercase	Großbuchstabe	A, B, C, Ä, Ö, ?
Ll	Letter_Lowercase	Kleinbuchstabe	a, b, c, ä, ö, p
Lt	Letter_Titlecase	Überschriften-Großbuchstabe (selten)	Dz, Lj
Lm	Letter_Modifier	Ergänzung vom Buchstaben (selten)	
Lo	Letter_Other	Buchstaben ohne Groß-/Kleinschreibung	japanische Schriftzeichen
Mn	Mark_NonSpacing	Markierungszeichen (siehe Kapitel 4.8.2, <i>Zusammengesetzte Unicode-Zeichen</i>)	Ä, å, Ô, ú, Ç, ñ
Mc	Mark_SpacingCombining	selten	
Me	Mark_Enclosing	selten	
Nd	Number_Digit	Ziffer	0, 1, 2, ...

Tabelle 4-10 Kategorien der Unicode-Zeichen

Kürzel	QChar::Category	Bedeutung	Beispiele
Nl	Number_Letter	römische Zahl	IX, viii, ...
No	Number_Other	andere Zahlen (Brüche, Potenzen, ...)	¼, ?, ³
Zs	Separator_Space	Leerschritt (verschiedene Breiten und Zeilenumbruch-Eigenschaften)	Space, Nonbreaking-Space, em-Space, ...
Zl	Separator_Line	Leerraum zwischen Zeilen	nur 0x2028
Zp	Separator_Paragraph	Leerraum zwischen Absätzen	nur 0x2029
Sm	Symbol_Math	mathematische Symbole	+, <, =, ¬, ±
Sc	Symbol_Currency	Währungssymbole	\$, ¢, £, ¥, _
Sk	Symbol_Modifier	wie Mark_NonSpacing (Mn), aber als eigene Zeichen	^, ´, `
So	Symbol_Other	andere Sonderzeichen	©, ?, TM
Pc	Punctuation_Connector	verbindendes Satzzeichen	_
Pd	Punctuation_Dask	waagerechte Linien (Minus, Trennstrich, Gedankenstrich, ...)	-, -, -
Ps	Punctuation_Open	öffnendes Satzzeichen	(, [, {
Pe	Punctuation_Close	schließendes Satzzeichen	),], }
Pi	Punctuation_Initial-Quote	öffnende Anführungszeichen	‘, », », <, >
Pf	Punctuation_Final-Quote	schließende Anführungszeichen	’, », >, »
Po	Punctuation_Other	andere Satzzeichen	", !, ?, ;
Cc	Other_Control	Kontrollzeichen	Tab, CR, LF, ...
Cf	Other_Format	spezielle Formatangaben (Schriftrichtung, ...)	Left-To-Right, Right-To-Left
Cs	Other_Surrogate	Stellvertreter-Bereiche	0xD000 bis 0xDFFF
Co	Other_PrivateUse	benutzerreservierte Zeichen	0xE000 bis 0xF8FF
Cn	Other_NotAssigned	(noch) undefinierte Zeichen	

Tabelle 4-10 Kategorien der Unicode-Zeichen (Forts.)

Diese Informationen zu allen Zeichen sind in der Datei *UnicodeData.txt* in maschinenlesbarer Form enthalten. Diese Datei finden Sie zum Beispiel auf der CD, die diesem Buch beiliegt oder in der aktuellsten Version auf der WWW-Seite des Unicode-Konsortiums (<http://www.unicode.org/>).

Die Zahlenbereiche sind in Blöcke unterschiedlicher Größe unterteilt, die so genannten Skripts. Aus Kompatibilitätsgründen sind die Unicode-Zeichen 0x0000 bis 0x007F identisch mit den Zeichen des 7-Bit-ASCII-Zeichensatzes, die

Unicode-Zeichen 0x0080 bis 0x00FF identisch mit den entsprechenden Zeichen des (meistverwendeten) Latin1-Zeichensatzes. Der Latin1-Zeichensatz enthält unter anderem auch die wichtigsten europäischen Sonderzeichen, wie zum Beispiel die deutschen Umlaute und die französischen Vokale mit Akzenten (siehe auch Kapitel 4.8.2, *Zusammengesetzte Unicode-Zeichen*). Eine Übersicht über die Unicode-Bereiche ist in Tabelle 4.11 aufgelistet.

Bereich	Inhalt
0x0000 – 0x007F	ASCII-Zeichensatz
0x0080 – 0x00FF	Latin1-Erweiterung
0x0100 – 0x1FFF	weitere alphabetische Zeichensätze, zum Beispiel das Internationale Phonetische Alphabet (0x0250 bis 0x02AF), Griechisch (0x0370 bis 0x03FF), Kyrillisch (0x0400 bis 0x04FF), Hebräisch (0x0590 bis 0x05FF), Arabisch (0x0600 bis 0x06FF)
0x2000 – 0x28FF	Symbole und Satzzeichen, zum Beispiel Währungszeichen (0x20A0 bis 0x20CF; _ ist 0x20AC), Pfeile (0x2190 bis 0x21FF), mathematische Symbole (0x2200 bis 0x22FF), Rahmenelemente (0x2500 bis 0x257F), Schachfiguren (0x2645 bis 0x265F), Dingbats (0x2700 bis 0x27BF), Braille-Blindenschrift (0x2800 bis 0x28FF) und vieles andere mehr
0x2E80 – 0x9FFF	Bildschriften (<i>Ideographs</i>), bestehend aus den vereinheitlichten CJK-Symbolen (Han-Zeichen aus China, Japan, Korea, Vietnam und Taiwan, 0x4E00 bis 0x9FFF) sowie verschiedenen Sonderzeichen, Satzzeichen, Alternativzeichen und Dialekten
0xA000 – 0xABFF	Yi-Silbenschrift
0xAC00 – 0xD7FF	Hangul-Silbenschrift
0xD800 – 0xDFFF	Stellvertreterbereiche
0xE000 – 0xF8FF	privater Bereich, bleibt undefiniert
0xF900 – 0xFFFF	verschiedene Zusätze zu den Zeichensätzen / Alternativen

Tabelle 4-11 Übersicht über die Bereiche des Unicode-Zeichensatzes

In Qt wird ein Unicode-Zeichen in einer Instanz der Klasse *QChar* gespeichert. Dazu können Sie dem Konstruktor von *QChar* die Unicode-Zahl des gewünschten Zeichens im Datentyp *unsigned short* übergeben. Auch für ASCII- oder Latin1-Zeichen vom Datentyp *char* oder *unsigned char* gibt es die entsprechenden Konstrukturen. Dadurch findet bei Bedarf auch eine automatische Typumwandlung von *char* nach *QChar* statt.

QChar bietet eine Vielzahl von Methoden, um Informationen über das gespeicherte Zeichen zu bekommen. Die Methode *unicode* liefert die Unicode-Zahl des Zeichens zurück. Die Methode *latin1* gibt für alle Unicode-Zeichen aus dem Bereich 0x0000 bis 0x00FF den Latin1-Code zurück (also das niederwertige Byte); für alle anderen Zeichen, die nicht im Latin1-Zeichensatz enthalten sind, den

Wert 0. Mit der Methode *category* erhält man die Kategorie des Zeichens als Aufzählungstyp *QChar::Category*. Die möglichen Werte dieses Aufzählungstyps finden Sie in Tabelle 4.10 als zweite Spalte. Eine Reihe weiterer Methoden testet – analog zu den Funktionen aus der C-Bibliothek *ctype.h* – ob das Zeichen zu einer speziellen Kategorie gehört, zum Beispiel *isDigit* (Ziffer), *isLetter* (Buchstabe), *isSpace* (Abstandszeichen: Leerschritt, Tabulator, Wagenrücklauf, Zeilenvorschub, Seitenvorschub), *isPrint* (druckbares Zeichen, im Gegensatz zu Steuerzeichen).

Für zusammengesetzte Zeichen kann man mit der Methode *decomposition* einen String der Einzel-Zeichen erhalten (siehe auch Kapitel 4.8.2, *Zusammengesetzte Unicode-Zeichen*).

Die Methoden *upper* und *lower* liefern zu einem Unicode-Zeichen den zugehörigen Groß- bzw. Kleinbuchstaben zurück. Für alle Zeichen, zu denen es keinen entsprechenden Buchstaben gibt (also alle Nicht-Buchstaben) wird das Zeichen selbst zurückgegeben. Eine Zeichenkette kann somit in Großbuchstaben umgewandelt werden, indem man für jedes Zeichen nacheinander mit *upper* das zugeordnete Zeichen ermittelt und diese aneinander hängt (siehe auch Kapitel 4.8.3, *Unicode-Zeichenketten*).

Beachten Sie aber, dass die Methoden *upper* und *lower* immer nur ein einziges, eindeutiges Unicode-Zeichen zurückgeben. Einige Zeichen benötigen bei der Umwandlung aber mehr Aufwand. So wird das deutsche »ß« (Unicode 0x00DF) bei der Umwandlung in Großbuchstaben zu zwei Buchstaben: »SS«. Vom griechischen Großbuchstaben Sigma (Σ , Unicode 0x03A3) gibt es zwei Kleinbuchstaben, abhängig davon, ob es im Wortinneren (σ , Unicode 0x03C3) oder am Wortende (ς , Unicode 0x03C2) benutzt wird. Diese und einige weitere Spezialfälle sind im gesonderten Unicode-Dokument *SpecialCasing.txt* festgelegt, Qt beachtet diese aber aus Effizienzgründen nicht.

Beispiele für die Nutzung der Klasse *QChar* finden Sie in Kapitel 4.8.3, *Unicode-Zeichenketten*.

4.8.2 Zusammengesetzte Unicode-Zeichen

Insbesondere in den europäischen Sprachen werden Zeichen um Markierungen (Marks) erweitert. Beispiele dafür sind unter anderem die deutschen Umlaute »Ä, ä, Ö, ö, Ü, ü«, bei denen die Vokale um zwei Punkte (*Trema*) erweitert werden, die französischen akzentuierten Vokale wie »á, à, â«, das spanische n mit Tilde »ñ« und viele andere mehr. Diese zusammengesetzten Zeichen (*composed characters*) werden in Unicode durch das Basiszeichen, gefolgt von einem oder mehreren Markierungszeichen repräsentiert. Das Basiszeichen belegt dann den benötigten Platz in der Darstellung, und die Markierungszeichen ergänzen das Basiszeichen um die entsprechende Erweiterung. Der Buchstabe »ü« wird zum Beispiel mit dem Basiszeichen »u« repräsentiert (Unicode 0x0075), gefolgt vom Markierungszeichen für zwei aufgesetzte Punkte (COMBINING DIAERESIS, Unicode 0x0308). Die wichtigsten Markierungszeichen sind in Tabelle 4.12 aufgelistet.

Code	Bezeichnung	Beispiele
0x0300	COMBINING GRAVE ACCENT	À, à, Ê, ê, Ì, ì, Ò, ò, Ù, ù
0x0301	COMBINING ACUTE ACCENT	Á, á, É, é, Í, í, Ó, ó, Ú, ú
0x0302	COMBINING CIRCUMFLEX ACCENT	Â, â, Ê, ê, Î, î, Ô, ô, Û, û
0x0303	COMBINING TILDE	Ã, ã, Ì, ì, Ñ, ñ, Õ, õ
0x0308	COMBINING DIAERESIS	Ä, ä, Ö, ö, Ü, ü, ë
0x030A	COMBINING RING ABOVE	Å, å
0x0327	COMBINING CEDILLA	Ç, ç

Tabelle 4-12 Liste der wichtigsten Markierungszeichen

Prinzipiell können alle Zeichen mit allen Markierungszeichen erweitert werden, auch wenn die entstehenden Zeichen in der Schrift eigentlich nicht vorkommen. Wird ein Zeichen um mehrere Markierungen erweitert, ist die Reihenfolge der Markierungszeichen unwichtig, sofern sie nicht gegenseitig ihre Position beeinflussen.

Viele der zusammengesetzten Zeichen tauchen auch als fertige Zeichen im Latin1-Zeichensatz auf und werden deshalb aus Kompatibilitätsgründen auch im Unicode-Zeichensatz als einzelne Zeichen geführt (*precomposed characters*). Mit der Methode *decomposition* der Klasse *QChar* können Sie zu einem Zeichen die Folge aus dem Basiszeichen und den Markierungszeichen ermitteln. Für die umgekehrte Umwandlungsrichtung besitzt die Klasse *QString* die (noch experimentelle) Methode *compose*. Die Normalform eines Unicode-Textes ist das Format, in dem alle Zeichen maximal nach Basiszeichen und Markierungszeichen aufgespalten sind. Beachten Sie aber, dass auf vielen Systemen die Zeichen des Latin1-Zeichensatzes ohne Probleme angezeigt werden können, während die Kombination aus Basiszeichen und Markierungszeichen nicht unterstützt wird.

4.8.3 Unicode-Zeichenketten

Ein Textstück – ein so genannter String – ist eine Folge von Unicode-Zeichen. Die Reihenfolge ist dabei immer die logische Reihenfolge, also die Reihenfolge, in der die Zeichen auch gesprochen werden, unabhängig davon, ob die Schrift von links nach rechts oder von rechts nach links (oder wie auch immer) geschrieben wird.

Für die Speicherung von Unicode-Zeichenketten ist in Qt die Klasse *QString* zuständig. Sie enthält ein Array von *QChar*-Objekten. Beachten Sie, dass die *QChar*-Folge nicht wie *char*-Strings aus C und C++ durch ein spezielles Null-Zeichen angeschlossen werden. Stattdessen speichert jede *QString*-Instanz zusätzlich die Länge des gespeicherten Textes. Mit der Methode *length* können Sie diese Länge ermitteln.

String-Konstanten

Sehr oft benötigt man String-Konstanten, zum Beispiel um Buttons zu beschriften oder Meldungen in Fenstern anzuzeigen. Da aber nahezu alle aktuellen Editoren, mit denen man den Quelltext seines Programms schreibt, nur ASCII-Dateien erzeugen, und auch die gängigen C++-Compiler (im Gegensatz zu JAVA-Compilern) nicht mit Unicode-Dateien zurechtkommen, gibt es keine Möglichkeit, Unicode-Zeichenketten direkt im Quelltext einzugeben. Die Klasse *QString* bietet daher eine große Anzahl an Konstruktoren und überladenen Zuweisungsoperatoren, um *QString*-Instanzen aus ASCII-Strings oder *QChar*-Objekten zu bilden. Durch automatische Typumwandlung ist es zunächst einmal möglich, eine ASCII-String-Konstante an allen Stellen anzugeben, an denen ein *QString*-Objekt gefragt ist. Wie schon oft in diesem Buch gezeigt, kann zum Beispiel der Text eines Buttons direkt im Konstruktor angegeben werden:

```
QPushButton *button = new QPushButton ("Übernehmen",  
                                         topwidget);
```

Obwohl der erste Parameter des *QPushButton*-Konstruktors eigentlich ein *QString*-Objekt erwartet, wird die ASCII-String-Konstante »ÜBERNEHMEN« (Datentyp *const char **) automatisch in eine Unicode-Zeichenkette umgewandelt. Dabei sind alle Zeichen aus dem Latin1-Zeichensatz erlaubt, also wie hier auch deutsche Umlaute. Ebenso können Sie *QString*-Objekte direkt mit ASCII-Strings belegen oder mit ihnen verknüpfen:

```
// Initialisierung von s1 mit Text  
QString s1 ("Der erste String");  
  
// Ersetzen des alten Texts durch einen neuen  
s1 = "Der zweite String";  
  
// Addieren von zwei Strings, hier Unicode und ASCII  
// gemischt  
s1 += " wird verlängert";
```

Solange Sie also nur Zeichen aus dem Latin1-Zeichensatz verwenden, merken Sie kaum, dass Sie intern mit Unicode-Zeichenketten arbeiten. Was kann man aber tun, wenn man Zeichen einfügen möchte, die nicht dem Latin1-Zeichensatz entspringen, die also einen Unicode-Wert zwischen 0x0100 und 0xFFFF besitzen? Handelt es sich nur um einzelne Zeichen, können Sie mit Hilfe des *+*-Operators zusammen mit *QChar*-Objekten die Strings zusammensetzen. So können Sie zum Beispiel das Euro-Symbol in einen String einbauen:

```
QString Monatsgehalt = QChar (0x20AC) +  
                        QString (" 3.244,23");
```

Beachten Sie, dass die Addition eines *QChar*-Objekts mit einem ASCII-String vom Typ *char** nicht definiert ist. Der zweite Teil des Strings muss daher vor der Addition in ein *QString*-Objekt umgewandelt werden. Bei mehreren Sonderzeichen im String können Sie auch mit einem *QString*-Objekt beginnen und anschließend dort *QChar*-Zeichen und *char**-Strings addieren:

```
QString s = QString()+ QChar (0x03B1)    // kleines Alpha
                + " und "
                + QChar (0x03C9)    // kleines Omega
                + " sind erster und letzter Buchstabe"
                + " im griechischen Alphabet";
```

Alternativ kann man die Sonderzeichen auch im Format UTF-8 in den String einflechten und dann zur Initialisierung die Methode *fromUtf8* benutzen. Eine genaue Beschreibung des UTF-8-Formats finden Sie in Kapitel 4.8.5, *Speichern und Laden von Unicode-Texten*. Das gleiche Beispiel von oben sieht dann so aus:

```
QString s;
s.fromUtf8 ("\xCE\xB1 und \xCF\x89 sind erster und "
            "letzter Buchstabe im griechischen Alphabet");
```

Jedes Zeichen, das nicht aus dem ASCII-Bereich stammt, wird dabei durch zwei oder drei Zeichen aus dem Bereich 0x80 bis 0xFF dargestellt. Diese Zeichen kann man durch Angabe der hexadezimalen Werte wie gezeigt direkt in den String einflechten. (Beachten Sie, dass die Umsetzung der hexadezimalen Werte in einzelne Zeichen je 8 Bit bereits vom Compiler bzw. vom Prozessor vorgenommen wird. Die Umwandlung dieses UTF-8-Strings in die interne 16-Bit-Repräsentation nimmt dann die Methode *fromUtf8* vor.)

Falls Sie diese Initialisierung übrigens unbedingt in einer einzigen Anweisung schreiben wollen, können Sie sie mit Hilfe eines temporären *QString*-Objekts auch so formulieren:

```
QString s = QString().fromUtf8 ("\xCE\xB1 und \xCF\x89 "
                                "sind erster und letzter Buchstabe im "
                                "griechischen Alphabet");
```

Besteht eine String-Konstante fast ausschließlich aus Sonderzeichen, ist der beschriebene Weg umständlich und ineffizient. In diesem Fall ist es günstiger, die Werte der Unicode-Zeichen in einem *ushort*-Array abzulegen und damit das *QString*-Objekt mit Hilfe der Methode *setUnicodeCodes* zu initialisieren. Ein kurzer Spruch von Konfuzius in der Originalschrift könnte beispielsweise so angelegt werden:

```
ushort k_data [] = {0x4F34, 0x6AA3, 0x8710,
                   0x8700, 0x6F8E, 0x53F2};
QString konfuzius;
konfuzius.setUnicodeCodes (k_data, 6); // Daten und Länge
```

(Die hier angegebenen Zahlenwerte sind völlig aus der Luft gegriffen. Man möge mir verzeihen, dass ich der chinesischen Sprache leider nicht mächtig bin.)

In vielen Fällen wird man Unicode-Texte auch einfach aus einer Datei einlesen. Dort kann er sowohl in den Formaten UTF-8 oder UTF-16 als auch in einer lokalen Kodierung abgelegt sein. Am einfachsten ist es natürlich, wenn man den Text vorher mit Hilfe eines *QTextStream*-Objekts aus einem *QString*-Objekt in die Datei geschrieben hat. (Siehe auch Kapitel 4.8.5, *Speichern und Laden von Unicode-Texten*.) Dann kann man auf dem umgekehrten Weg den Text wieder auslesen. Auf diese Art arbeitet zum Beispiel die Übersetzungstabelle bei der Internationalisierung (siehe auch Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*). Die übersetzten Texte, die je nach Sprache sehr viele Nicht-ASCII-Zeichen enthalten können, werden aus der Übersetzungsdatei ausgelesen und in *QString*-Objekten abgelegt.

Zugriff auf einzelne Zeichen im String

Um die einzelnen *QChar*-Zeichen des Strings auszulesen oder zu ändern, können Sie mit dem überladenen *operator[]* auf das *QString*-Objekt zugreifen. Dabei wird automatisch geprüft, ob der angegebene Index innerhalb des Strings liegt. Falls nicht, wird der String automatisch auf die entsprechende Länge erweitert und an den neuen Positionen mit dem Unicode-Zeichen 0x0000 aufgefüllt.

Eine Funktion, die in einem *QString*-Text alle Buchstaben in Großbuchstaben verwandelt, kann damit beispielsweise folgendermaßen aussehen:

```
void convertToUppcase (QString &s)
{
    for (int i = 0; i < s.length(); i++)
        // nutzt die Methode QChar::upper()
        s [i] = s [i].upper();
}
```

Sie können aber auch direkt die Methode *upper* von *QString* benutzen. Diese Methode ändert allerdings nicht den Original-String, sondern gibt eine umgewandelte Kopie zurück. Eine Umwandlung kann damit folgendermaßen aussehen:

```
s = s.upper(); // s ist ein QString-Objekt
```

Wenn Sie eine effizientere Möglichkeit brauchen, um einzelne Zeichen des Strings auszulesen, ohne sie verändern zu müssen, können Sie mit der Methode *unicode* der Klasse *QString* einen (konstanten) Zeiger auf das erste *QChar*-Objekt des Textes ermitteln. Diesen können Sie dann – wie in C und C++ üblich – wie ein Array benutzen. Auf die Grenzen des Arrays müssen Sie hier allerdings selbst achten. Ein schreibender Zugriff ist nicht möglich, da es sich um einen konstan-

ten Zeiger handelt. Der Zeiger bleibt übrigens nur so lange gültig, bis das *QString*-Objekt verändert oder gelöscht wird. Anschließend sollten Sie den Zeiger also nicht mehr benutzen.

Die folgende Funktion benutzt die Methode *unicode*. Sie zählt in einem *QString*-Objekt die Anzahl der Groß- und Kleinbuchstaben sowie die Ziffern, Satzzeichen und Leerzeichen und gibt die Werte aus:

```
void countCharacters (QString s)
{
    int letters=0, lowercase=0, uppercase=0, spaces=0,
        digits=0, punctuation=0, others=0;
    // Zeiger auf das erste Zeichen
    const QChar f = s.unicode();
    for (int i = 0; i < s.length(); i++)
    {
        if (f->isLetter())
        {
            letters++;
            if (f->category() == QChar::Letter_Uppercase)
                uppercase++;
            if (f->category() == QChar::Letter_Lowercase)
                lowercase++;
        }
        else if (f->isSpace())
            spaces++;
        else if (f->isDigit())
            digit++;
        else if (f->isPunct())
            punctuation++;
        else
            others++;
        f++; // bewegt bei jedem Schritt
            // den Zeiger ein Zeichen weiter
    }
    qDebug ("The String contains %d Letters (%d upper, "
        "%d lower), %d digits, %d punctuation marks, "
        "%d spaces and %d other characters.",
        letters, uppercase, lowercase, digits,
        punctuations, spaces, others);
}
```

Die Klasse *QString* bietet Ihnen übrigens noch viele weitere, zum Teil sehr mächtige Methoden an, um Zeichenketten zu analysieren oder zu manipulieren. Neben dem Verketteten von Texten mit dem *++*-Operator kann man Teile des Textes heraustrennen, nach Teil-Strings suchen oder den Text an bestimmten Zeichen in Teile aufspalten. Einige dieser Methoden sind in Kapitel 4.7.4, *Die String-Klassen – QString und QString*, näher erläutert. Eine vollständige Liste finden Sie in der Online-Dokumentation von Qt zur Klasse *QString*.

4.8.4 Umwandlung zwischen Unicode und lokalen Zeichensätzen

Solange Sie in Ihrem Programm zum Speichern und Bearbeiten von Texten die Klasse *QString* benutzen, können Sie alle Zeichen des Unicode-Standards benutzen. Geht es aber darum, Textdateien mit acht Bit pro Zeichen zu lesen oder zu schreiben, die auch von anderen Programmen benutzt werden sollen, so müssen Sie unter Umständen die Kodierung eines lokalen Zeichensatzes berücksichtigen. So werden zum Beispiel E-Mails nur selten direkt im Unicode-Zeichensatz verschickt. Der benutzte Zeichensatz wird meist im Kopf der E-Mail angegeben. Um den Inhalt der E-Mail korrekt auf dem Bildschirm darzustellen, muss der Byte-Strom der E-Mail in Unicode umgewandelt werden. Diese Umwandlung ist natürlich vom lokalen Zeichensatz der E-Mail abhängig. Und auch für den umgekehrten Weg gilt: Soll eine E-Mail verschickt werden und kann der Empfänger nur E-Mails mit einem bestimmten lokalen Zeichensatz lesen, so muss der Unicode-Text in den lokalen Zeichensatz umgewandelt werden. Dabei kann es natürlich unter Umständen passieren, dass einige Zeichen des Textes im lokalen Zeichensatz nicht enthalten sind.

Zeichensatz	MIB-Nr.	Beschreibung	Escape-Sequenzen
US-ASCII	3	7-Bit-ASCII-Zeichensatz	nein
ISO-8859-1	4	Latin1-Zeichensatz (West-Europa)	nein
ISO-8859-5	8	Kyrillisch	nein
ISO-8859-7	10	Griechisch	nein
ISO-8859-8	11	Hebräisch	nein
EUC-JP	18	Japanisch	ja
EUC-KR	38	Koreanisch	ja
UTF-8	106	Unicode, 1 bis 3 Byte pro Zeichen	ja
ISO-10646-UCS-2	1000	Unicode, 2 Byte pro Zeichen	ja
Big5	2026	Chinesisch	ja
KOI8-R	2084	lateinisch-kyrillischer Zeichensatz (Ost-Europa)	nein

Tabelle 4-13 Häufig verwendete lokale Zeichensätze

Ein lokaler Zeichensatz ist eine Zuordnung einer Zahl zwischen 0 und 255 zu einem Zeichen. Texte in einem lokalen Zeichensatz werden daher als Byte-Strom repräsentiert. Umfasst der Zeichensatz mehr als 256 Zeichen, so sind oft Escape-Sequenzen definiert: Eine bestimmte Kombination mehrerer Bytes wird dann als ein Zeichen interpretiert. Viele häufig benutzte Zeichensätze sind in der internationalen Norm ISO 8859 festgelegt. Außerdem führt die IANA (Internet Assigned Numbers Authority, <http://www.iana.org/>) eine Sammlung von Zeichensatzdefinitionen. Sie weist jedem Zeichensatz eine eindeutige Nummer zu, die so genannte

MIB-Nummer. Eine Liste der enthaltenen Zeichensätze ist auch auf der CD zum Buch enthalten, die aktuellste Version der Definitionsdatei können Sie unter <ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets> herunterladen.

Eine Liste der häufigsten Zeichensätze finden Sie in Tabelle 4.13. Alle Zeichensätze, die keine Escape-Sequenzen benutzen, enthalten maximal 256 Zeichen.

Wie Tabelle 4.13 zeigt, gibt es auch Kodierungen, um Unicode-Texte ohne Informationsverlust abzuspeichern. Hierbei handelt es sich also nicht um »lokale« Zeichensätze im eigentlichen Sinne. Näheres zu diesen Kodierungen finden Sie in Kapitel 4.8.5, *Speichern und Laden von Unicode-Texten*.

Die Umwandlung von Unicode in einen lokalen Zeichensatz und umgekehrt kann in Qt mit Hilfe der Klasse *QTextCodec* realisiert werden. Für die häufigsten lokalen Zeichensätze – auch alle in Tabelle 4.13 angegebenen – sind passende Umwandlungsroutinen bereits definiert. Eine Umwandlungsroutine für einen lokalen Zeichensatz wird dabei durch ein Objekt der Klasse *QTextCodec* oder einer Unterklasse realisiert. Qt verwaltet eine globale Liste aller vorhandenen Objekte. Sie wird beim Erzeugen des *QApplication*-Objekts automatisch angelegt. Mit einigen statischen Methoden der Klasse *QTextCodec* kann man einen Zeiger auf ein spezielles Objekt bekommen. Diese Methoden sind im Einzelnen:

- *codecForIndex (int i)*

Mit dieser Methode kann man die *QTextCodec*-Objekte der globalen Liste der Reihe nach durchgehen. Dazu übergibt man im Parameter *i* den gewünschten Index innerhalb der Liste. Ein Wert von 0 bezeichnet dabei das zuletzt erzeugte Objekt. Ist *i* größer oder gleich der Anzahl der vorhandenen Objekte, so liefert die Methode einen NULL-Zeiger zurück. Das folgende Code-Stück füllt beispielsweise ein *QListBox*-Fenster mit den Namen aller unterstützten lokalen Zeichensätze, so dass der Anwender einen der Zeichensätze auswählen kann:

```
QListBox *auswahl = new QListBox (parent);
int i = 0;
QTextCodec *tc = 0;
while ((tc = QTextCodec::codecForIndex (i)) != 0)
{
    auswahl->insertItem (tc->name());
    i++;    // nächsten Zeichensatz auswählen
}
```

Das Signal *QListBox::selected (int)* liefert nun den ausgewählten Index zurück, und mit *codecForIndex (int)* kann man wiederum das zugehörige *QTextCodec*-Objekt in Erfahrung bringen.

- *codecForMib (int mib)*

Diese Methode sucht in der globalen Liste das *QTextCodec*-Objekt, das für die Behandlung des lokalen Zeichensatzes mit der MIB-Nummer *mib* zuständig ist. Gibt es in der Liste keinen Eintrag für diese MIB-Nummer, so wird der NULL-Zeiger zurückgegeben. Da die MIB-Nummer eindeutig ist, ist dieses Verfahren das sicherste. Allerdings ist oftmals die MIB-Nummer eines Byte-Stroms unbekannt, so dass dieser Weg nicht weiterhilft.

- *codecForName (const char *name, int accuracy = 0)*

Diese Methode sucht in der globalen Liste das *QTextCodec*-Objekt, das für den Zeichensatz mit dem übergebenen Namen *name* zuständig ist. Gibt es in der Liste keinen Eintrag für diesen Zeichensatznamen, so wird der NULL-Zeiger zurückgegeben.

Das Ergebnis dieser Methode ist nicht immer absolut eindeutig, da es für viele lokale Zeichensätze eine ganze Reihe von Alias-Namen gibt, die in der Praxis auch oft benutzt werden. Daher können Sie mit dem zweiten Parameter *accuracy* angeben, wie exakt der übergebene Zeichensatzname mit dem Zeichensatznamen des gesuchten *QTextCodec*-Objekts übereinstimmen soll. Der Default-Wert 0 ist meist am besten geeignet. Bei diesem Wert reicht es aus, wenn die Namen in den relevanten Teilen übereinstimmen, also in der Buchstabenkombination und den Zahlen. Groß- und Kleinschreibung sowie Leerzeichen und andere Trennzeichen werden beim Vergleich nicht berücksichtigt. Gibt man als *accuracy* die Länge von *name* an, so muss das gesuchte Objekt in seinem Zeichensatznamen exakt übereinstimmen.

Aufgrund der Uneindeutigkeit sollte – falls möglich – die Methode *codecForMib* vorgezogen werden. Oft wird aber nur der Name eines lokalen Zeichensatzes übermittelt, aber nicht seine MIB-Nummer, so dass nur diese Möglichkeit bleibt.

- *codecForLocale ()*

Diese Methode liefert das *QTextCodec*-Objekt aus der globalen Liste zurück, das für die Sprache geeignet ist, die im System eingestellt ist. Um diese Sprache zu ermitteln, wird auf Unix-Systemen der Wert der Umgebungsvariablen *\$LANG* benutzt. Hat diese beispielsweise den Inhalt »german«, »de« oder »DE_de« (als Kennzeichen für die Sprache »Deutsch«), so wird das *QTextCodec*-Objekt für Latin-1 (MIB-Nummer 4) zurückgeliefert. Unter Microsoft Windows wird die Systemeinstellung der eingestellten Sprache benutzt.

Das von dieser Methode zurückgelieferte *QTextCodec*-Objekt können Sie immer dann benutzen, wenn Sie Texte auf diesem Rechner abspeichern oder einlesen wollen, da diese mit großer Wahrscheinlichkeit in genau diesem lokalen Zeichensatz abgelegt sind.

Unter X11 wird dieser lokale Zeichensatz auch benutzt, um die Eingaben von der Tastatur und die Ausgaben auf den Bildschirm richtig zu interpretieren, da hier eine Unicode-Unterstützung nur selten vorhanden ist. Unter Microsoft Windows ist das nicht nötig, da die neueren Versionen (Windows 2000, Windows NT 4.0) bereits vollständig auch intern mit Unicode arbeiten, und die älteren Versionen (Windows 95/98) eigene Umwandlungsroutinen bereitstellen.

- *codecForContent* (*const char *chars, int len*)

Diese Methode versucht zu ermitteln, zu welchem lokalen Zeichensatz Textdaten gehören, und gibt das passende *QTextCodec*-Objekt zurück. Dazu übergeben Sie die Textdaten (kodiert im lokalen Zeichensatz) im Parameter *chars* und die Länge der Daten (in Bytes) im Parameter *len*. Qt sucht nun nach einem lokalen Zeichensatz, in dem keines der übergebenen Zeichen ein ungültiger Code ist. Da jedoch viele lokale Zeichensätze fast alle Werte des Bereichs 0 bis 255 als gültiges Zeichen interpretieren, ist das Ergebnis dieser Methode nicht sehr verlässlich. Nur wenn Sie keine Informationen über den benutzten Zeichensatz haben, können Sie als letzten Ausweg diese Methode benutzen, sonst sind *codecForMib* und *codecForName* in jedem Fall vorzuziehen.

Nachdem Sie mit einer der oben aufgeführten Methoden das (hoffentlich) richtige *QTextCodec*-Objekt ermittelt haben, geht es nun daran, die Umwandlung eines Textes vorzunehmen. Das *QTextCodec*-Objekt bietet dazu Methoden für beide Umwandlungsrichtungen. Die Umwandlung von Unicode in den lokalen Zeichensatz bezeichnet man dabei als *Kodierung* (*encoding*), die umgekehrte Richtung vom lokalen Zeichensatz in einen Unicode-Text als *Dekodierung* (*decoding*).

Haben Sie den kompletten Text in einem einzigen Datenblock abgespeichert – als *QString*-Objekt für Unicode-Text oder als *char*-Array bzw. vom Typ *QByteArray* oder *QCString* für den lokalen Zeichensatz –, so können Sie ganz einfach die Methoden *fromUnicode* bzw. *toUnicode* des *QTextCodec*-Objekts für die entsprechende Umwandlung benutzen. Wenn Sie beispielsweise den Inhalt einer E-Mail darstellen wollen – kodiert in einem lokalen Zeichensatz, der im Header der E-Mail festgelegt ist –, so können Sie folgendes Code-Fragment benutzen:

```
// Name des lokalen Zeichensatzes
char *cs = getCharacterSet ();
// Inhalt der E-Mail
QCString content = getContent ();

// unwandeln
QTextCodec *tc = QTextCodec::codecForName (cs);
QString unicodeContent = tc->toUnicode (content);
```



```
// anzeigen
QMultiLineEdit *edit = new QMultiLineEdit (parent);
edit->setText (unicodeContent);
```

Vorsicht ist bei der Kodierung geboten: Treten im Unicode-Text Zeichen auf, die nicht im lokalen Zeichensatz vorhanden sind, so ist das Ergebnis der Kodierung an diesen Stellen undefiniert. Die in Qt vordefinierten *QTextCodec*-Objekte setzen dort das Fragezeichen ein. Um zu testen, ob alle vorkommenden Zeichen auch vom lokalen Zeichensatz unterstützt werden, können Sie die Methode *QTextCodec::canEncode* benutzen. Sie liefert *true* zurück, falls das übergebene *QString*-Objekt problemlos in den lokalen Zeichensatz umgewandelt werden kann.

Etwas komplizierter wird es, wenn die Kodierung oder die Dekodierung stückchenweise erfolgen soll, zum Beispiel weil die Daten nur in kleinen Brocken über eine Socketverbindung eintreffen oder weil die Datenmenge zu groß ist, um in einem Stück verarbeitet zu werden. Enthält der benutzte lokale Zeichensatz Escape-Sequenzen, so kann es passieren, dass ein Stück der Daten mit dem Beginn einer Escape-Sequenz endet, und der Rest der Sequenz erst am Anfang des nächsten Datenblocks folgt. Würde jedes einzelne Stück mit der Methode *decode* bearbeitet, so würde das hintere Datenstück falsch dekodiert, da der Anfang der Escape-Sequenz nicht beachtet wird. Daher muss der Status der Dekodierung abgespeichert werden, damit er für den nächsten Datenblock erhalten bleibt.

Speziell für diesen Zweck stellt Qt die Hilfsklassen *QTextEncoder* und *QTextDecoder* zur Verfügung. Die Methode *QTextCodec::makeDecoder* liefert einen Zeiger auf ein neu erzeugtes *QTextDecoder*-Objekt für diesen lokalen Zeichensatz. Dieses Objekt besitzt die Methode *toUnicode*, die genauso arbeitet wie bei der Klasse *QTextCodec*. Zusätzlich wird aber noch der Status abgespeichert, so dass der Anfang einer Escape-Sequenz beim nächsten Aufruf von *toUnicode* berücksichtigt wird. Nach getaner Arbeit muss das *QTextDecoder*-Objekt mit *delete* wieder freigegeben werden. Ganz analog stellt *QTextCodec::makeEncoder* ein neu erzeugtes *QTextEncoder*-Objekt zur Verfügung, dessen Methode *fromUnicode* benutzt werden kann, um einen Unicode-String stückweise in den lokalen Zeichensatz umzuwandeln.

Als Beispiel für die Anwendung folgt hier das Listing eines Programms, das den Inhalt einer Textdatei, die im lokalen Zeichensatz KOI8-R vorliegt, stückweise einliest, dekodiert und jedes Stück sofort auf dem Bildschirm darstellt. Nach jedem Stück wird die Methode *QApplication::processEvents* aufgerufen, um ein Blockieren des Programms zu vermeiden.

```

// Anzeige-Widget
QMultiLineEdit *edit = new QMultiLineEdit (parent);
edit->setReadOnly (true);

// Dekoder-Objekt
QTextCodec *tc = QTextCodec::codecForName ("KOI8-R");
QTextDecoder *decoder = tc->makeDecoder();

QFile f ("textfile.txt");
f.open (IO_ReadOnly);
char data [1024];    // Größe eines Blocks: 1 kByte
while (!f.atEnd())
{
    // Block einlesen
    int len = f.readBlock (data, 1024);
    // dekodieren
    QString newText = decoder->toUnicode (data, len);
    // ins Fenster einfügen
    edit->insert (newText);
    // anstehende Events abarbeiten
    qApp->processEvents();
}

delete decoder;    // Dekoder-Objekt wieder freigeben

```

4.8.5 Speichern und Laden von Unicode-Texten

Die interne Speicherung von Unicode-Texten erledigt die Klasse *QString* zu unserer vollsten Zufriedenheit. Will man allerdings Textdokumente in einer Datei abspeichern, so muss man sich zunächst für ein Format entscheiden. Die meisten existierenden Textdateien benutzen das Latin-1-Format, können also nur die westeuropäischen Sonderzeichen abspeichern.

Man kann stattdessen die Daten auch im reinen Unicode-Format (so genanntes UTF-16) abspeichern, so dass jedes Zeichen in der Datei zwei Byte belegt. Auch von diesem Format gibt es zwei Varianten: In *UnicodeNetworkOrder* wird das höherwertige Byte eines Zeichens zuerst gespeichert, in *UnicodeReverse* das niederwertige. Um diese Formate automatisch unterscheiden zu können, wird in solchen Dateien als erstes Zeichen das Spezialzeichen FEFFh abgelegt (also höherwertiges Byte 254, niederwertiges Byte 255). Beim Einlesen wird daran automatisch erkannt, in welcher Reihenfolge die Bytes gespeichert sind.

Ein noch besseres Format ist das so genannte UTF-8-Format. Hierbei wird jedes Unicode-Zeichen durch ein, zwei oder drei Byte dargestellt. Tabelle 4.14 zeigt die Kodierung der Zeichen. Die linke Spalte gibt dabei das Bitmuster des Unicode-Zeichens an, die rechte das Bitmuster der Kodierung durch UTF-8.

Unicode-Zeichen	UTF-8
00000000 0xxxxxxx	0xxxxxxx
00000xxx xxxxxxxx	110xxxxx 10xxxxxx
xxxxxxxx xxxxxxxx	1110xxxx 10xxxxxx 10xxxxxx

Tabelle 4-14 Kodierung der Unicode-Zeichen in UTF-8

Der entscheidende Vorteil von UTF-8 ist, dass reiner ASCII-Text (7 Bit) unverändert und ohne höheren Speicherplatzbedarf abgespeichert wird. Texte, die zum größten Teil aus ASCII-Zeichen bestehen (also auch europäische Texte, die ja nur wenige Sonderzeichen verwenden), werden also auch sehr kompakt gespeichert. Eine solche Datei kann man auch in einem »normalen« Latin-1-Texteditor betrachten, da nur die Sonderzeichen falsch dargestellt werden. Außerdem ist das Format so angelegt, dass man auch mitten in eine UTF-8-Datei springen und dennoch problemlos den Anfang des nächsten Zeichens erkennen kann.

Um eine Textdatei einzulesen oder abzuspeichern, benutzen Sie am besten die Klasse *QTextStream*. Diese wird in Kapitel 4.18.4, *Stream-basierte Ein-/Ausgabe*, noch genauer beschrieben. Nach dem Anlegen des Stream-Objekts können Sie mit *setEncoding* wählen, welches Format benutzt werden soll. Per Default wird die lokale Kodierung benutzt, also für Deutschland in der Regel der Latin-1-Zeichensatz. Mit der folgenden Zeile ändern Sie das Format beispielsweise in UTF-8:

```
QTextStream stream (file);
stream.setEncoding (QTextStream::UnicodeUTF8);
// anschließend wird gemäß UTF-8 gelesen und
// geschrieben.
```

Auch KDE setzt inzwischen verstärkt auf UTF-8 als Format für seine Konfigurations- und Übersetzungsdateien (siehe Kapitel 2.3, *Das erste KDE-Programm*). Leider gibt es noch nicht viele Texteditoren, die das UTF-8-Format auch lesen und schreiben können. Das wird sich in naher Zukunft hoffentlich ändern. Vorübergehend können Sie den einfachen Texteditor verwenden, den wir in Kapitel 3.5.6, *Anwendung für ein Dokument*, entwickelt haben.

4.8.6 Darstellung von Unicode-Zeichen auf dem Bildschirm

Wie wir in Kapitel 4.8.3, *Unicode-Zeichenketten*, gesehen haben, ist die Speicherung und Verarbeitung von Unicode-Texten mit Hilfe der Klasse *QString* sehr einfach und für den Programmierer fast transparent. An allen Stellen, an denen Texte auf dem Bildschirm dargestellt werden können, benutzt Qt diese Klasse *QString*, zum Beispiel als Beschriftung in einem *QLabel*-Objekt oder als einzugebender Text in einem *QLineEdit*-Feld.

Problematisch wird es allerdings, wenn auf dem System kein Zeichensatz installiert ist, der die Unicode-Zeichen enthält, die benötigt werden. Schließlich definiert keine Font-Datei alle ca. 40.000 möglichen Zeichen. Insbesondere unter Linux ist die Unterstützung von Unicode durch den X-Server eher rudimentär.

Alle Unicode-Zeichen, die nicht dargestellt werden können, werden auf dem Bildschirm durch ein Fragezeichen ersetzt.

Wenn Sie wissen wollen, ob ein spezielles Zeichen dargestellt werden kann, können Sie `QFontMetrics::inFont` benutzen. Das folgende Beispiel fragt ab, ob das Euro-Zeichen `_` dargestellt werden kann, und ersetzt es gegebenenfalls durch die Abkürzung `EUR`:

```
QFontMetrics metr (font());
QString waehrungssymbol;
if (metr.inFont (QChar (0x20AC)))
    waehrungssymbol = QChar (0x20AC);
else
    waehrungssymbol = "EUR";
```

Allgemeine Informationen über die vorhandenen Zeichensätze können Sie mit der Klasse `QFontDatabase` erhalten.

Ist Ihr Programm auf bestimmte exotische Zeichen angewiesen, so helfen Ihnen die installierten Systemfonts in der Regel nicht weiter, denn auf einem anderen Rechner könnten die benötigten Zeichensätze nicht vorhanden sein. Ein Ausweg wäre in diesem Fall die Klasse `FTFont`. Diese Klasse war ursprünglich als Beispielprogramm für dieses Buch gedacht, hat sich nun aber zu einer eigenständigen Klasse entwickelt. Sie ist nicht Bestandteil der Qt- oder KDE-Bibliothek. Diese Klasse benutzt TrueType-Fonts, die sie direkt aus einer beliebigen Datei liest. Es ist daher nicht nötig, dass dieser Font auch installiert ist. Wenn Sie also bestimmte Sonderzeichen unbedingt benötigen, so benutzen Sie eine TrueType-Font-Datei, die Sie zusammen mit Ihrem Programm verschicken.

`FTFont` bietet außerdem noch eine Reihe weiterer Vorteile: Sie können ein Antialiasing auf die Zeichen anwenden, so dass insbesondere bei kleinen Schriftgrößen die Kanten sehr glatt und nicht treppenartig wirken. Außerdem können Sie sich eine Schrift auch als Polygonzug zurückgeben lassen. Auf diese Weise können Sie nachträglich noch beliebige Verformungen und Füllmuster benutzen. Ein großer Nachteil der Klasse ist allerdings, dass es sich nicht um eine Unterklasse von `QFont` handelt. Damit kann sie nicht in den bereits fertigen Klassen der Qt- oder KDE-Bibliothek benutzt werden. Sie können Sie nur in eigenen Zeichenbefehlen mit `QPainter` verwenden oder die Schrift in ein `QImage`-Objekt zeichnen lassen. Weitere Informationen zur Klasse `FTFont` und die Möglichkeit zum Download finden Sie im Internet unter <http://www.ksource.org/>.

4.9 Mehrsprachige Anwendungen und Internationalisierung

Um dem Benutzer ein intuitives Programm zu bieten, ist es wichtig, alle sichtbaren Texte des Programms in seiner Muttersprache darzustellen. Ein Programmierer, der etwas auf sich hält, wird sich nicht einfach darauf berufen, dass der Benutzer Englisch sprechen kann. Dazu gibt es sowohl im KDE-System als auch im Qt-System ähnliche, aber nicht kompatible Konzepte. Für reine Qt-Programme kommt nur die Lösung in Frage, die in Kapitel 4.9.2, *Qt-Übersetzungen – Die Methode QObject::tr*, beschrieben wird, für KDE-Programme empfiehlt sich dagegen auf jeden Fall die Lösung in Kapitel 4.9.1, *KDE-Übersetzungen – Die Funktion i18n*.

4.9.1 KDE-Übersetzungen – Die Funktion *i18n*

Auch KDE enthält Methoden zur Umwandlung von Texten in die eingestellte Sprache. Das zentrale Objekt ist dabei die Klasse *KLocale*, die diese Umwandlungen vornimmt. Allerdings benötigt man sie nur selten direkt.

Die Vorgehensweise ist recht einfach: Jeden sichtbaren Text (Labels, Buttonbeschriftungen usw.) in Ihrem Programmcode müssen Sie an die Funktion *i18n* übergeben, die den übersetzten Text zurückliefert. *i18n* ist die Abkürzung für *internationalization*, wobei 18 für die Anzahl der ausgelassenen Buchstaben steht.

Lautete die Zeile zur Erzeugung eines Buttons vorher zum Beispiel

```
myButton = new QPushButton ("Cancel", this);
```

so schreiben Sie nun:

```
myButton = new QPushButton (i18n ("Cancel"), this);
```

Die Funktion *i18n* benutzt nun das in *KGlobal* enthaltene *KLocale*-Objekt und ruft die Methode *translate* in diesem Objekt auf.

Wie übersetzt nun aber das Programm die Strings in eine andere Sprache? Künstliche Intelligenz ist hier nicht im Spiel, sondern einfach eine Tabelle mit den Originalstrings und den zugehörigen Übersetzungen. Im Folgenden werden noch einmal kurz die Schritte beschrieben, die zum Erstellen einer solchen Tabelle nötig sind. Die Vorgehensweise wurde bereits ausführlich in Kapitel 2.3.3, *Landsprache: Deutsch*, erläutert.

Aus dem fertigen Programmcode müssen Sie mit dem GNU-Tool *xgettext* alle Strings herausfiltern, die hinter dem Ausdruck *i18n* stehen. Das kann für alle Quellcode-Dateien gleichzeitig geschehen. Diese werden dann in einer *po*-Datei

in der Form von Paaren abgelegt. Die oben angegebene Zeile erzeugt zum Beispiel folgenden Text in der erzeugten *po*-Datei:

```
msgid "Cancel"  
msgstr ""
```

Hinter dem Wort *msgstr* muss nun die Übersetzung in die gewünschte Landessprache eingefügt werden. Dazu kopieren Sie die erzeugte Datei für jede Landessprache, für die Sie eine Übersetzung erstellen wollen, in eine Datei, die den entsprechenden Landeskürzelnamen trägt (z.B. *de.po*), und tragen dort die Übersetzungen zu den Texten ein. In unserem Fall würde die Datei *de.po* dann so aussehen:

```
msgid "Cancel"  
msgstr "Abbrechen"
```

Achten Sie darauf, dass Sie die Datei im UTF-8-Format abspeichern, wenn Sie Zeichen verwenden, die nicht im ASCII-Zeichensatz sind (beispielsweise die deutschen Umlaute). Dazu können Sie beispielsweise unseren Editor aus Kapitel 3.5.6, *Applikation für ein Dokument*, benutzen, oder einen beliebigen anderen Editor, der dieses Format unterstützt. Oder Sie benutzen das Hilfsprogramm *KBabel*, das im SDK-Paket zu finden ist.

Nachdem Sie so alle Textstücke übersetzt haben, wandeln Sie die Datei in eine *gmo*-Datei mit dem Namen Ihres Programms um (zum Beispiel *myprogram.gmo*). Sie hat eine kompaktere Form und kann leichter eingelesen werden. Diese Datei müssen Sie in das KDE-Verzeichnis der entsprechenden Landeskennung kopieren.

Bei der Erzeugung des *KApplikation*-Objekts in Ihrem Programm wird nun aus dem Programmnamen und der eingestellten Landeskennung die richtige Datei ermittelt und geladen. Der Aufruf der Funktion *i18n* übernimmt dann die Umsetzung des Originaltextes (in Englisch) in die eingestellte Sprache (hier Deutsch).

Es ist zwar theoretisch möglich, die Texte im Programmcode in jeder Sprache zu schreiben, es hat sich jedoch durchgesetzt, diese Texte in Englisch zu schreiben und für alle anderen Sprachen Übersetzungen vom Englischen in die jeweilige Sprache zu benutzen. Da Englisch die am weitesten verbreitete Sprache ist – nahezu jeder Computeranwender kann zumindest ein wenig Englisch –, kann das Programm so meist auch bedient werden, wenn aus irgendeinem Grund die jeweilige Landessprache nicht verfügbar ist. Außerdem finden sich eher Übersetzer vom Englischen ins beispielsweise Spanische als von Deutsch ins Spanische.

Die Funktion *i18n* hat noch eine andere Form, in der sie zwei Parameter erhält: Als ersten Parameter enthält sie einen Identifikationsstring, anhand dessen die Übersetzung erzeugt wird, und als zweiten einen Text, der benutzt werden soll, wenn keine Übersetzung gefunden wird. Diese Form kann immer dann einge-

setzt werden, wenn der zu übersetzende Text in der Originalsprache mehrdeutig ist. Der Identifikationsstring ist dann eine eindeutige Beschreibung, so dass der Übersetzer genau weiß, was gemeint ist. Soll beispielsweise das englische Wort »open« übersetzt werden, dann kann es sowohl die Anweisung »öffnen« bedeuten, als auch den Zustand »offen«. Eindeutigkeit kann also man folgendermaßen schaffen:

```
QString text1 = i18n ("open (status)", "open");  
QString text2 = i18n ("open (command)", "open");
```

In der Übersetzungsdatei tauchen nun nur die Identifikationsstrings auf:

```
msgid = "open (status)"  
msgstr = "offen"  
  
msgid = "open (command)"  
msgstr = "öffnen"
```

In der deutschen Übersetzung erhalten wir also in den beiden Textvariablen die Texte »offen« und »öffnen«, im Englischen dagegen in beiden Variablen den Text »open«.

An einigen Stellen im Programm kann man die Funktion *i18n* für die Übersetzung nicht benutzen. Die beiden häufigsten Gründe dafür sind:

- Sie legen den Text an einer Stelle an, an der Sie das *KApplication*-Objekt noch nicht angelegt haben (z.B. für den Konstruktor der Klasse *KAboutData*).
- Sie legen ein statisches Array mit String-Konstanten an.

In beiden Fällen müssen die Übersetzungen mit *i18n* später vorgenommen werden. Wenn allerdings die String-Konstante nicht mehr hinter dem Schlüsselwort *i18n* steht, wird sie von *xgettext* auch nicht mehr gefunden, fehlt also in der Übersetzungstabelle. Um sie nicht von Hand einfügen zu müssen, können wir auch das Hilfsmakro *I18N_NOOP* benutzen. Dieses Makro macht nichts: Es ersetzt nur sich selbst mit dem Parameter, der in Klammern folgt (also unserer Textkonstanten). *xgettext* erkennt aber auch *I18N_NOOP* als Schlüsselwort und speichert entsprechend den String in der Übersetzungstabelle.

Hier ein Beispiel für die Übersetzung von Fehlermeldungen:

```
void MyMainWindow::printErrorMessage (int i)  
{  
    static char *messages [] =  
    {  
        I18N_NOOP ("Could not open file"),  
        I18N_NOOP ("Read Error"),  
        I18N_NOOP ("Write Error"),  
        I18N_NOOP ("Timeout"),
```

```

        I18N_NOOP ("Unknown File Type")
    }
    statusBar()->message (i18n (messages [i]));
}

```

Sie dürfen natürlich nicht vergessen, bei der Ausgabe dann *i18n* zu verwenden, da *I18N_NOOP* keine Übersetzung durchführt.

4.9.2 Qt-Übersetzungen – Die Methode *QObject::tr*

Auch Qt bietet die Möglichkeit, Texte mit Hilfe einer Übersetzungsdatei in eine andere Landessprache umsetzen zu lassen. Dazu benutzt man die statische Methode *tr* (Abkürzung für *translate*) in der Klasse *QObject*.

Die Methode *tr* nimmt einen Text vom Typ *char** entgegen und liefert die Übersetzung in einem *QString*-Objekt zurück. Somit kann sie auch in Unicode übersetzen. Gerade bei Übersetzungen in andere Sprachen mit anderen Zeichensätzen kann das sehr wichtig sein. Die Anwendung dieser Methode im eigenen Programm funktioniert wie beim Makro *i18n*:

```
myButton = new QPushButton (tr ("Cancel"), this);
```

Beachten Sie, dass die Methode *tr* eine statische Methode der Klasse *QObject* ist. Außerhalb einer Methode einer Klasse, die von *QObject* abgeleitet ist, müssen Sie somit folgende Zeile benutzen:

```
myButton = new QPushButton
(QObject::tr ("Cancel"), widget);
```

Die Firma Trolltech stellt eigene Programme, *findtr*, *msg2qm* und *mergetr*, zur Verfügung, die den Quellcode nach dem String *tr* durchsuchen und die übersetzten Dateien in eine lesbare Form umwandeln. Um eine solche Datei zur Übersetzung zu benutzen, laden Sie sie mit der Methode *load* in ein Objekt der Klasse *QTranslator*. Dieses *QTranslator*-Objekt müssen Sie noch mit *QApplication::installTranslator* registrieren lassen. Ab dem Moment stehen die Übersetzungen zur Verfügung.

Genauere Informationen finden Sie in der Online-Referenz der Qt-Bibliothek.

4.9.3 Texte mit Platzhaltern übersetzen

Die Übersetzung von Texten, in die nachträglich Zahlen oder andere Texte eingefügt werden sollen, ist nicht ganz unproblematisch. Nehmen wir zum Beispiel den Text aus Kapitel 4.7.4, *Die String-Klassen – QString und QString*:


```
QString s = "Die obere Ecke liegt bei (" +  
    QString().setNum (p.x()) +  
    "/" +  
    QString().setNum (p.y()) +  
    ").";
```

Um diesen Text übersetzbar an eine Landessprache anpassen zu lassen, nehmen wir als zugrunde liegende Sprache Englisch und fügen für alle Textkonstanten die Funktion *i18n* ein:

```
QString s = i18n ("The upper left corner is at (" +  
    QString().setNum (p.x()) +  
    i18n ("/") +  
    QString().setNum (p.y()) +  
    i18n (").");
```

Die Übersetzungsdatei sähe dann so aus:

```
msgid "The upper left corner is at ("  
msgstr "Die obere linke Ecke liegt bei ("  
msgid "/"  
msgstr "/"  
msgid ")."  
msgstr ")."
```

Sie enthält hierbei zwei kleine Ausdrücke, die nicht übersetzt werden müssen. Auch für einen Übersetzer ist es nicht leicht herauszufinden, wie dieser kurze Text übersetzt werden muss. Einfacher wäre es, einen einzigen String mit Platzhaltern für die einzufügenden Zahlen anzugeben, der dann nur eine Übersetzung benötigt. Das Problem kann zum Beispiel so gelöst werden:

```
QString s = QString().  
    sprintf (i18n ("The upper left corner is at (%d/%d)."),  
        p.x(), p.y());
```

Die Übersetzungsdatei sieht nun so aus:

```
msgid "The upper left corner is at (%d/%d)."  
msgstr "Die obere linke Ecke liegt bei (%d/%d)."
```

Noch besser ist hier aber die Methode *arg* der Klasse *QString*, mit der ebenfalls Platzhalter durch Zahlen oder Text ersetzt werden können. Diese Methode hat gegenüber der Anwendung von *QString::sprintf* zwei große Vorteile: Erstens kann es keine schwerwiegenden Probleme geben, wenn die Platzhalter und die aufgeführten Argumente nicht in Typ oder Anzahl übereinstimmen. (Eine fehlerhafte Übersetzung, in der mehr Platzhalter vorkommen als im Originalstring, führt bei *sprintf* fast immer zum Programmabsturz.) Zweitens kann die Reihenfolge der Argumente innerhalb des Textes in der Übersetzung eine andere sein als in der Originalsprache.

Die Methode *arg* wird auf ein *QString*-Objekt angewandt. Sie ersetzt dabei ein Vorkommen eines Prozentzeichens, dem eine einstellige Ziffer folgt, durch das Argument, das der Methode mitgegeben wird. Sind mehrere solcher Platzhalter vorhanden, wird der Platzhalter mit der kleinsten Ziffer ersetzt. Der Rückgabewert der Methode ist wiederum das *QString*-Objekt, so dass man mehrere Aufrufe von *arg* direkt hintereinander anwenden kann. Dabei werden die Platzhalter in aufsteigender Ziffernreihenfolge ersetzt.

Auf unser Beispiel angewandt, kann man folgenden Code benutzen:

```
QString s =
    QString (i18n ("The upper left corner is at (%1/%2)."))
    .arg (p.x())
    .arg (p.y());
```

Hier wird der Platzhalter *%1* durch *p.x()* ersetzt, und *%2* wird durch *p.y()* ersetzt. Die Methode *arg* ist dabei so überladen, dass man ihr Parameter von allen gängigen Datentypen übergeben kann, z.B. *double*, *int*, *char*, *char**, *QString* usw.

Besonders interessant ist die Anwendung der Methode *arg*, wenn sich bei der Übersetzung die Reihenfolge der Argumente verändert. Wenn zum Beispiel das aktuelle Tagesdatum ausgegeben werden soll, steht im Englischen der Monat vor dem Tag, im Deutschen der Tag vor dem Monat. Eine Lösung für dieses Problem sieht zum Beispiel so aus:

```
QDate now = QDate::currentDate();
QString s = QString (i18n ("Today is %1/%2/%3"))
    .arg (now.month())
    .arg (now.day())
    .arg (now.year());
```

In der Übersetzungsdatei steht nun Folgendes:

```
msgid "Today is %1/%2/%3"
msgstr "Heute ist %2.%1.%3"
```

Allein durch die andere Reihenfolge der Platzhalter befindet sich in der deutschsprachigen Ausgabe nun der Tag vor dem Monat. Speziell für das Datumsformat besitzt KDE in der Klasse *KLocale* Methoden zur formatierten Ausgabe, die Sie in diesem Fall vorziehen sollten (siehe Kapitel 4.9.4, *Zahlen-, Währungs- und Datumsformate*).

Beachten Sie aber auch die Einschränkungen der Methode *arg*: Sie können maximal zehn Argumente ersetzen, und Ihre Texte dürfen an keiner Stelle ein Prozentzeichen gefolgt von einer Ziffer enthalten, die nicht ersetzt werden sollen. Diese Einschränkungen sind aber nicht sehr gravierend und machen sich in der Praxis kaum bemerkbar.

4.9.4 Zahlen-, Währungs- und Datumsformate

Die Klasse *KLocale* in KDE bietet neben der Übersetzung in eine Landessprache auch andere Formatierungsmöglichkeiten für Zahlen, Geldbeträge und Datums- und Zeitangaben, angepasst an die Einstellungen, die der Anwender im KDE-System eingestellt hat. Die Einstellungen werden im KDE-Control-Center im Abschnitt PERSONALIZATION / COUNTRY & LANGUAGE vorgenommen und gelten für alle KDE-Programme.

Hier ein paar einfache Beispiele, wie Sie Daten formatieren können:

```
#include <kapp.h>
#include <kglobal.h>
#include <klocale.h>
#include <qlabel.h>
#include <qvbox.h>
#include <qdatetime.h>

int main (int argc, char **argv)
{
    KApplication app (argc, argv, "localisation");

    QVBox *box = new QVBox ();

    QDateTime now = QDateTime::currentDateTime();
    QString zeitstr = KGlobal::locale()->
        formatDateTime (now, false, true);
    new QLabel ("Es ist nun " + zeitstr , box);

    double zahl = 14852.87;
    QString zahlstr = KGlobal::locale()->formatNumber (zahl, 3);
    new QLabel ("Zahl: " + zahlstr, box);

    double geld = 86000.0 * 1.16;
    QString geldstr = KGlobal::locale()->formatMoney (geld);
    new QLabel ("Geld: " + geldstr, box);

    box->show();
    app.setMainWidget (box);
    return app.exec();
}
```

Beachten Sie aber auf jeden Fall, dass die Ausgabe eines Geldbetrags das Kürzel der eingestellten Landeswährung voranstellt, aber natürlich keine Umrechnungen vornimmt. Sie selbst müssen dafür sorgen, dass der Zahlenwert zur eingestellten Währung passt, denn 100 US-Dollar sind nicht das gleiche wie 100 Deutsche Mark oder wie 100 Euro.

Auch für die umgekehrte Umwandlung von einem String in ein Datum, eine Zeitangabe oder eine Zahl besitzt *KLocale* entsprechende Methoden: *readTime*, *readDate*, *readNumber* und *readMoney*. Falls bei der Konvertierung ein Fehler auftrat, wird dieses über einen zusätzlichen Parameter zurückgemeldet bzw. wird ein ungültiges *QDate*- oder *QTime*-Objekt zurückgegeben. Genauere Informationen hierzu finden Sie in der Online-Dokumentation zur Klasse *KLocale*.

4.10 Konfigurationsdateien

Als Vorgabe für alle KDE-Programme gilt, dass alle Einstellungen innerhalb der Applikation gespeichert werden sollen, so dass sie beim nächsten Start des Programms wiederhergestellt werden können. Alle Einstellungen sollten dabei über einen Bildschirmdialog vorgenommen werden können. Kein Anwender sollte gezwungen sein, Textdateien von Hand mit einem Editor zu editieren.

Um den Programmierern hier eine Hilfestellung zu geben, sind in der KDE-Bibliothek bereits Klassen zum Einlesen, Interpretieren, Ändern und Schreiben von Konfigurationsdateien enthalten. Diese Klassen generieren Konfigurationsdateien in einer festen Struktur. Diese Konfigurationsdateien sind ASCII-Dateien, die lesbar strukturiert sind und somit auch – zum Beispiel in Notfällen, wenn aufgrund einer falschen Einstellung das Programm nicht mehr gestartet werden kann – von unerfahrenen Anwendern mit einem einfachen Texteditor verändert und korrigiert werden können.

4.10.1 Struktur der Konfigurationsdateien

Hier sehen Sie ein Beispiel für eine Konfigurationsdatei. Das Format der KDE-Konfigurationsdateien ähnelt dem Format, wie es beispielsweise die Datei *win.ini* unter Microsoft Windows hat. Dies ist der Inhalt der Datei *kfmrc*, also die Einstellungen des Datei-Managers von KDE:

```
# KDE Config File
[Paths]
Trash=$HOME/Desktop/Trash/
Desktop=$HOME/Desktop/
Templates=$HOME/Desktop/Templates/
Autostart=$HOME/Desktop/Autostart/
[Icons]
Batch=terminal.xpm
Default=unknown.xpm
Folder=folder.xpm
Executable=exec.xpm
[Terminal]
Terminal=kvt
[KFM HTML Defaults]
```

```
StandardFont=Helvetica  
FixedFont=Courier  
[KFM Root Icons]  
Style=0
```

Zeilen, die mit einem `#` beginnen, sind Kommentare und haben keine Auswirkungen. Kommentarzeilen und Leerzeilen werden beim Einlesen ignoriert. Die erste Zeile einer Konfigurationsdatei ist immer eine Kommentarzeile mit dem Inhalt `# KDE Config File`. An diesem Kommentar kann zum Beispiel *Konqueror* Konfigurationsdateien eindeutig erkennen. Zum Einlesen der Datei ist diese Zeile aber nicht relevant.

Der Inhalt der Konfigurationsdatei ist in Gruppen unterteilt. Der Beginn einer Gruppe wird durch einen Gruppennamen in eckigen Klammern bezeichnet (hier zum Beispiel `[Paths]`, `[Icons]` usw.). Anschließend werden Paare von Schlüsseln und zugehörigen Werten gebildet. Ein Schlüssel ist dabei ein String – in der Regel ein einzelnes Wort. Er kann aber theoretisch alle Zeichen außer dem Gleichheitszeichen, den eckigen Klammern und Leerschritten enthalten. Ihm folgt ein Gleichheitszeichen, und daran schließt sich der Wert – ein beliebiger Text-String – an (hier zum Beispiel in der Gruppe `[Icons]` das Schlüssel/Wert-Paar mit dem Schlüssel `Batch` und dem Wert `terminal.xpm`). Außerdem kann ein Schlüssel noch eine länderspezifische Kennung in eckigen Klammern erhalten. Dieser Eintrag ist wird nur dann benutzt, wenn dieses Land in KDE eingestellt ist. Eine Konfigurationsdatei könnte zum Beispiel folgende Schlüssel/Wert-Paare enthalten:

```
Agree=Yes  
Agree[de]=Ja  
Agree[it]=Si  
Agree[fr]=Oui
```

Ist die Ländereinstellung Deutschland (*de*), wird zum Schlüssel `Agree` der Wert *Ja* gewählt. Ist die Ländereinstellung dagegen Italien (*it*), gehört der Wert *Si* zum Schlüssel `Agree`. Entspricht die aktuelle Ländereinstellung keinem der Schlüssel, wird der Default-Wert `Yes` benutzt. Für computergenerierte Konfigurationsdateien hat das meist keine Bedeutung. Diese Konfigurationsdateien werden jedoch an vielen anderen Stellen im KDE-System genutzt, zum Beispiel in den *desktop*-Dateien, die zu einer Applikation spezifische Angaben machen, zum Beispiel eine kurze Beschreibung geben. In diesen Dateien wird die Beschreibung auf diese Weise in die verschiedenen Sprachen übersetzt, die KDE unterstützt. Als generelle Regel gilt, dass alle Gruppennamen und alle Schlüssel englisch sein sollten.

4.10.2 Zugriff auf die Daten – *KConfigBase*

Der Inhalt von Konfigurationsdateien kann eingelesen und in einem Objekt einer von *KConfigBase* abgeleiteten Klasse abgespeichert werden. (*KConfigBase* ist eine abstrakte Klasse. Zwei oft benutzte abgeleitete Klassen sind *KConfig* und

KSimpleConfig. Sie werden weiter unten beschrieben.) Mit diesem Objekt kann man sehr effizient auf die gespeicherten Daten zugreifen. Sie werden in einer zweiphasigen Hash-Tabelle gespeichert (siehe auch Kapitel 4.7.2, *Container-Klassen*, Abschnitt *Die Zuordnungstabelle – QMap und QDict*). Zunächst wählt man die entsprechende Gruppe mit der Methode *setGroup* aus. Anschließend kann man den Wert zu einem speziellen Schlüssel in dieser Gruppe mit der Methode *readEntry* auslesen. Mit *writeEntry* kann man ein Schlüssel/Wert-Paar in die Konfigurationsdaten eintragen. Falls der Schlüssel in der ausgewählten Gruppe schon vorhanden ist, wird nur sein Wert ausgetauscht. Als einfaches Beispiel dient uns folgendes Programm, das den Eintrag *Batch* aus der Gruppe *Icons* in der oben gezeigten Datei *kfmrc* ausliest und anschließend ändert. Wir benutzen hier ein Objekt der Klasse *KSimpleConfig*, die von *KConfigBase* abgeleitet ist:

```
// Erzeugen eines Konfigurationsobjekts und Laden
// der Konfigurationsdatei kfmrc
KSimpleConfig conf ("kfmrc");
// Gruppe Icons wählen
conf.setGroup ("Icons");
// Wert zum Schlüssel Batch auslesen. Falls Batch nicht
// gefunden wird, wird der Default-Wert default.xpm
// zurückgeliefert.
QString value = conf.readEntry ("Batch", "default.xpm");
// Wert zum Schlüssel Batch in kvt.xpm ändern
conf.writeEntry ("Batch", "kvt.xpm");
```

Beim Destruktor der Klasse *KSimpleConfig* werden automatisch alle geänderten Daten in die Konfigurationsdatei zurückgeschrieben, so dass die Änderungen dauerhaft werden.

Die Klasse *KConfigBase* definiert eine ganze Reihe von Methoden zum Lesen und Schreiben von Daten verschiedenen Typs. Neben dem Einlesen eines Strings mit *readEntry* kann man auch Daten vom Typ *bool*, *int*, *unsigned int*, *long*, *unsigned long*, *double*, *QStrList*, *QColor*, *QPoint*, *QSize*, *QRect* und *QFont* auslesen. Die entsprechenden Methoden dazu heißen *readBoolEntry*, *readNumEntry*, *readUnsignedNumEntry*, *readLongNumEntry*, *readUnsignedLongNumEntry*, *readDoubleNumEntry*, *readListEntry*, *readColorEntry*, *readPointEntry*, *readSizeEntry*, *readRectEntry* bzw. *readFontEntry*. Alle Methoden, außer *readListEntry*, erhalten als ersten Parameter den Schlüssel und als zweiten Parameter ein Objekt des entsprechenden Typs als Default-Wert, der benutzt werden soll, wenn der Schlüssel nicht in der Konfigurationsdatei vorhanden ist. Der Rückgabewert der Methode ist ein Objekt des jeweiligen Typs. Bei *readListEntry* gibt der erste Parameter ebenfalls den Schlüssel an. Der zweite Parameter ist eine Referenz auf ein Objekt der Klasse *QStrList*, in der das Ergebnis abgelegt wird. Der dritte Parameter legt fest, mit welchem Zeichen die einzelnen Teilstrings im Eintrag voneinander getrennt sind (Default-Wert ist hier das Komma). Einen Default-Wert, der benutzt werden soll, wenn

kein Eintrag zum Schlüssel vorliegt, gibt es bei dieser Methode nicht. Der Rückgabewert der Methode ist die Anzahl der erkannten Teil-Strings, die im zweiten Parameter zurückgegeben werden.

Um Einträge der entsprechenden Datentypen zu schreiben, gibt es die Methode *writeEntry*, die für jeden unterstützten Datentyp überladen ist. Meist sind nur die ersten beiden Parameter dieser Methode von Belang: Der erste Parameter legt den Schlüssel fest, der zweite Parameter den Wert. Erlaubt sind für den Wert wiederum alle oben aufgelisteten Datentypen. Es folgen noch drei weitere Parameter vom Typ *bool*. Der erste legt fest, ob diese Änderung des Eintrags auch in die Konfigurationsdatei übernommen werden soll. Der Default-Wert ist *true*. Wählt man hier *false*, so wird die Änderung zwar intern im Konfigurationsobjekt abgespeichert, so dass weitere Anfragen den neuen Wert ermitteln; beim Beenden wird der neue Wert aber nicht in der Konfigurationsdatei gespeichert, geht also verloren. Der zweite *bool*-Parameter legt fest, ob die Änderung in der globalen oder der lokalen Konfigurationsdatei geschrieben werden soll. Das ist nur für die Klasse *KConfig* sinnvoll (siehe Kapitel 4.10.3, *Standardkonfigurationsdateien*), die auf zwei Konfigurationsdateien gleichzeitig arbeitet. Für *KSimpleConfig* macht es keinen Unterschied. Der Default-Wert *false* legt hier fest, dass in die lokale Datei geschrieben wird. Der dritte *bool*-Parameter gibt schließlich an, ob beim Schreiben der Datei das Landeskürzel in eckigen Klammern an den Schlüssel angehängt werden soll. Der Default-Wert ist hier *false*. In der Regel sind die Default-Werte dieser drei Parameter korrekt, so dass man nur die ersten beiden Parameter angeben muss.

Eine Ausnahme bildet wieder die Methode *writeEntry* für den Datentyp *QStrList*. Hier kann man im dritten Parameter noch angeben, mit welchem Zeichen die einzelnen Strings der Liste in der Konfigurationsdatei getrennt werden sollen. Der Default-Wert ist hier (wie bei *readListEntry*) das Komma als Trennzeichen.

In den Konfigurationsdaten werden alle Daten immer als Strings abgelegt. Achten Sie also darauf, dass Sie einen Eintrag mit dem gleichen Datentyp lesen, mit dem Sie ihn geschrieben haben. Ansonsten können die Daten undefiniert sein.

4.10.3 Standardkonfigurationsdateien

Die Klasse *KApplication* öffnet im Konstruktor automatisch zwei Konfigurationsdateien. Der Name der Datei wird dabei aus dem Namen des Programms gebildet (also dem Dateinamen der ausführbaren Datei, der im Konstruktor von *KApplication* bzw. beim Aufruf von *KCmdLineArgs::init* angegeben wurde), an den die beiden Buchstaben *rc* angehängt werden. (Daher heißt die Konfigurationsdatei des *kfm* auch *kfmrc*.) Die eine Konfigurationsdatei steht dabei im Verzeichnis *\$KDEDIR/share/config/*, die andere im Verzeichnis *\$HOME/.kde/share/config/*. Die erste ist die globale Konfigurationsdatei. Alle Anwender benutzen diese Datei. In ihr kann man allgemeine Einstellungen vornehmen, die als Default für alle

neuen Anwender gelten sollen. Diese Datei wird auch oft bereits mit der Applikation mitgeliefert. Die Einstellungen in dieser Datei werden meist nicht vom Anwender selbst vorgenommen, sondern vom Administrator. Die zweite Datei ist die lokale Konfigurationsdatei. Sie liegt unterhalb des Heimatverzeichnisses des Anwenders und ist daher für jeden Anwender verschieden. Hier kann der Anwender seine Einstellungen speichern. In der Regel werden die Einstellungen in einem Dialogfenster in der Applikation selbst vorgenommen. Die Applikation schreibt die geänderten Einträge in die lokale Datei zurück.

Beide Dateien dürfen gleiche Gruppen und gleiche Schlüssel enthalten. Die lokale Konfigurationsdatei hat dabei Vorrang vor der globalen. Der Anwender kann auf diese Weise die Einstellungen an seine Bedürfnisse anpassen und dabei die Standardeinstellungen aus der globalen Konfigurationsdatei überdecken.

Auf diese beiden Konfigurationsdateien können Sie über ein Objekt der Klasse *KConfig* zugreifen, dessen Adresse Ihnen die Klasse *KGlobal* in ihrer statischen Methode *config* liefert. Die beiden folgenden Programmfragmente sollen demonstrieren, wie eine Einstellung (hier der Zahlenwert zum Schlüssel *Timeout* in der Gruppe *Internet*) aus der Konfigurationsdatei gelesen bzw. in ihr geändert werden kann:

```
// Auslesen des Werts
KGlobal::config()->setGroup ("Internet");
int value = KGlobal::config()->readNumEntry ("Timeout", 5);

// Ändern des Werts
KGlobal::config()->setGroup ("Internet");
KGlobal::config()->writeEntry ("Timeout", value);
```

Beachten Sie, dass in der Klasse *KConfig* nur Einträge und Gruppen geändert oder hinzugefügt werden können. Sie können aber keine Gruppe oder einen Eintrag löschen! Der Grund dafür besteht darin, dass nicht eindeutig ist, aus welcher der beiden Konfigurationsdateien der Eintrag gelöscht werden soll. Wenn Sie Einträge auch löschen wollen, müssen Sie ein Objekt der Klasse *KSimpleConfig* anlegen (siehe Kapitel 4.10.4, *Eigene Konfigurationsdateien*).

Wenn Sie die Einstellungen in Ihrem Programm mit einem Dialogfenster vornehmen lassen (siehe Kapitel 3.8.3, *Modale Dialoge*), so können Sie am besten zwei Methoden schreiben: Eine Methode, zum Beispiel mit Namen *getConfig*, liest die Einstellungen aus der Konfigurationsdatei aus und setzt die Werte in das Dialogfenster ein. Die andere Methode, *setConfig*, entnimmt die eingestellten Werte aus dem Dialogfenster und schreibt sie in die Konfigurationsdatei. Beim Erstellen des Dialogfensters rufen Sie einmal *getConfig* auf, um die Werte zu setzen. Die drei Buttons am unteren Ende des Dialogfensters bewirken Folgendes: Ein Klick auf den Button OK ruft *setConfig* auf und versteckt das Fenster mit *hide*. Ein Klick auf ABBRECHEN ruft *getConfig* auf (wodurch die alten Werte wiederherge-

stellt werden) und versteckt ebenfalls das Fenster mit *hide*. Ein Klick auf RÜCKSETZEN ruft nur *getConfig* auf, um so die alten Werte wieder in das Dialogfenster einzutragen. Wenn Sie die Methoden *setConfig* und *getConfig* als parameterlose Slots realisieren, können Sie die Buttons ganz einfach mit diesen Slots verbinden und benötigen keine zusätzlichen Slots. (Auch *QWidget::hide* ist ein Slot.)

Neben dem Standardobjekt, das Sie mit *KGlobal::config* erhalten, legt *KApplication* ein weiteres Konfigurationsobjekt an, wenn die Daten beim Session-Management gesichert werden müssen (siehe Kapitel 4.16, *Session-Management*). Dieses Objekt liefert Ihnen die Methode *sessionConfig* der Klasse *KApplication*.

4.10.4 Eigene Konfigurationsdateien

Wenn Sie die Klassen der Konfigurationsdateien selbst nutzen wollen, um eigene Daten einzulesen oder abzuspeichern, die nicht unbedingt in die Standardkonfigurationsdateien gehören, können Sie auch eigene Objekte erzeugen. So können Sie zum Beispiel die Dokumente, die Sie in Ihrer Applikation bearbeiten wollen, in Dateien im KDE-Konfigurationsformat abspeichern. Dazu benutzen Sie am besten die Klasse *KSimpleConfig*. Diese Klasse arbeitet nur auf einer einzelnen Konfigurationsdatei, deren Namen Sie im Konstruktor angeben können. Existiert noch keine Datei mit diesem Namen, so wird sie beim Abspeichern der Konfigurationsdaten angelegt.

In Ihrem *KSimpleConfig*-Objekt können Sie mit *deleteEntry* ein Schlüssel/Wert-Paar löschen, mit *deleteGroup* sogar eine ganze Gruppe mit allen Einträgen.

Wenn Sie eine Liste von komplexeren Daten in der Konfigurationsdatei abspeichern wollen, können Sie so vorgehen, dass Sie eine aufsteigende Zahlenfolge an einen Schlüsselnamen anhängen und unter diesen Schlüsseln die einzelnen Datenelemente speichern. Beispielsweise könnte eine Adressdatenbank auf folgende Weise in einer Konfigurationsdatei gespeichert sein:

```
[Adressbuch]
Anzahl=35
Person1=Müller,Peter,Hafenstraße 16,Hamburg
Person2=Meier,Hans,Pariser Straße 23, Kaiserslautern
Person3=Schmidt,Helmut,Kanzlerweg 1,München
...
```

Der Eintrag zum Schlüssel *Anzahl* gibt dabei an, wie viele Einträge im Adressbuch vorhanden sind. Alternativ kann man den Eintrag *Anzahl* auch weglassen, alle möglichen Schlüsselbezeichnungen durchlaufen und dabei mit der Methode *hasKey* ermitteln, ob ein Schlüssel mit dieser Bezeichnung enthalten ist.

Mit der Methode *groupList* kann man eine Liste aller Gruppennamen (Typ *QStringList*) erhalten. Um an alle Einträge in einer Gruppe zu gelangen, können

Sie die Methode *entryMap* benutzen. Sie liefert ein *QMap*-Objekt – also eine Zuordnungstabelle – mit Schlüsseltyp und Datentyp *QString*. Sie können nun zum Beispiel alle Einträge der Gruppe durchlaufen, ohne die Schlüsselbezeichnungen zu kennen, indem Sie einen Iterator benutzen (siehe Kapitel 4.7.3, *Iterator-Objekte*).

Theoretisch können Sie in einem Eintrag einen beliebig langen String als Wert benutzen. Ab einer Eintragslänge von mehr als 500 Zeichen wird die Konfigurationsdatei aber unübersichtlich und ist nur noch schwer von Hand editierbar. Sollen mehr als 100 kByte Daten im String gespeichert werden, wird außerdem die Performance extrem schlecht. Es ist also ungünstig, einen großen Datenblock – als ASCII-String kodiert – in einer Konfigurationsdatei zu speichern. Speichern Sie stattdessen die Daten in einer Datei ab, und legen Sie den Namen der Datei in einer Konfigurationsdatei ab. Die temporäre Datei sollte möglichst nicht im Verzeichnis */tmp* abgelegt werden, da je nach Systemeinstellung dieses Verzeichnis in regelmäßigen Abständen gelöscht wird.

4.10.5 Binäre Konfigurationsdateien mit *UConfig*

Wenn Sie Binärdaten – beispielsweise Icons oder Messwertdaten – in einer Konfigurationsdatei speichern wollen, so ist *KSimpleConfig* nicht besonders gut geeignet. Der Vorteil entfällt, dass die Datei auch von Hand betrachtet und editiert werden kann, und die schlechte Performance bei großen Datenmengen sowie der große Speicherplatzbedarf machen ein Arbeiten oft unmöglich.

Eine mögliche Alternative ist die Klasse *UConfig*, die von Kir Kostuchenko entwickelt wurde und sich zur Zeit noch in der Entstehung und Erweiterung befindet. Sie gehört (noch) nicht zur KDE-Bibliothek. Sie setzt nur die Qt-Bibliothek, nicht aber die KDE-Bibliotheken voraus, ist somit auch unter Microsoft Windows einsetzbar (eventuell mit Anpassungen). Sie erhalten die Quellen sowie die Dokumentation zu dieser Klasse zur Zeit unter <http://uconfig.sourceforge.net/>.

Hier eine Liste der Möglichkeiten, die *UConfig* bietet:

- Die Daten können in einer hierarchischen Struktur – ähnlich eines Verzeichnisbaums – abgelegt werden.
- Als Suchschlüssel sind Zahlen vom Typ *int* sowie Texte vom Typ *QString* möglich.
- Die Suche nach einem Eintrag ist sehr effizient programmiert. Auch bei extrem großen Datenmengen (auch mehr als 1 MByte, theoretisch fast nicht beschränkt) ist der Performance sehr gut. Die Konfigurationsdatei wird nicht in den Speicher geladen, sondern es werden immer nur die benötigten Teile geladen. Eine Caching-Strategie sorgt für schnelle Zugriffe.
- Abgespeichert werden können alle Daten, die in einer *QVariant*-Variable abgelegt werden können (siehe Kapitel 4.7.5, *Flexibler Datentyp – QVariant*).

- Die Daten werden in der Konfigurationsdatei binär gespeichert. Sie ist zwar nicht mehr mit einem Texteditor lesbar, aber dafür sehr kompakt gespeichert.
- Mit einem Hilfsprogramm mit grafischer Oberfläche lassen sich die Konfigurationsdateien anschauen und auch verändern.

4.11 Online-Hilfe

Ein gutes Programm zeichnet sich durch eine intuitive Bedienbarkeit aus. Oft ist es aber nicht möglich, den gesamten Funktionsumfang einer Applikation auf einen Blick darzustellen. In diesem Fall ist es unumgänglich, dass das Programm Hilfedokumente zur Verfügung stellt, in denen sich der Anwender genauer über die Besonderheiten des Programms informieren kann. Man unterscheidet bei KDE-Programmen dabei drei Stufen der Hilfestellung:

- Tooltips
- What's-This-Hilfe
- Online-Handbuch

Kleine Hilfswörter, die in einem Fenster erscheinen, sobald man mit der Maus etwas länger über einem Bedienelement verweilt, heißen *Tooltips* (auch *Balloon-Help* genannt, weil diese Fenster in einigen Betriebssystemen wie Sprechblasen oder Ballons aussehen). Sie sind bei Bedienelementen sinnvoll, denen man ihre Bedeutung nicht sofort ansieht oder die mehrdeutig sein könnten. Ganz besonders wichtig sind Tooltips für die Werkzeugleisten. Da Icons nicht immer ganz eindeutig ausdrücken können, welche Funktion sich hinter einem Button verbirgt, muss der Anwender eine einfache Möglichkeit haben, sich schnell zu informieren, welchen Zweck eine Schaltfläche hat.

Die nächste Stufe der Hilfestellung ist die Anzeige eines etwas umfangreicheren Hilfefensters zu Bedienelementen, die so genannte *What's-This-Hilfe*. Sie funktioniert im Prinzip wie ein Tooltip, allerdings öffnet sich das Fenster nicht automatisch, sobald sich die Maus über dem Element befindet. Da das Fenster einen mehrzeiligen Text umfasst, wäre das sehr störend. Stattdessen muss der Anwender zunächst einen Button – meist in der Werkzeugleiste – anklicken. Für das Bedienelement, das er als Nächstes anklickt, erscheint dann das What's-This-Fenster. Ein weiterer Klick lässt es wieder verschwinden.

Die letzte Stufe der Hilfestellung ist das *Online-Handbuch*, das bei KDE-Programmen in Form von HTML-Dateien vorliegt. Wenn man in der Menüleiste auf den Punkt HILFE klickt, findet man den Menüeintrag INHALT. Mit diesem Eintrag öffnet sich ein neues Fenster, in dem das Handbuch angezeigt wird. In der Regel besteht das Handbuch aus mehreren Seiten, die – mit einem Inhaltsverzeichnis versehen – jeweils einzelne Aspekte und Funktionen der Applikation erläutern.

4.11.1 Tooltips

Tooltips werden in Qt mit Hilfe der Klasse *QToolTip* realisiert. Es ist sehr einfach, ein Widget mit einem Tooltip-Text auszustatten. Um beispielsweise das GUI-Element *myWidget* mit einem Tooltip-Text »Macht nix« zu versehen, benutzen Sie einfach die statische Methode *QToolTip::add*:

```
QToolTip::add (myWidget, "Macht nix");
```

Sobald nun die Maus über dem Widget verweilt, erscheint nach kurzer Zeit ein kleines Fenster mit dem Text. Sobald die Maus bewegt wird, verschwindet es wieder.

Als Tooltips sollten Sie ein Wort, maximal ein paar Wörter benutzen. Es ist zwar auch möglich, mehrzeiligen Text zu benutzen, indem Sie einfach Zeilenumbrüche mit `\n` in den Text einfügen. Ein Tooltip sollte aber sehr kurz sein, damit er nicht störend wirkt. Die Buttons der Werkzeugleiste sollten auf jeden Fall mit Tooltips ausgestattet werden. Der Text zu einem Button sollte dabei genau dem Namen des Befehls im Menü entsprechen, den der Button ausführt. Beim Einfügen eines Buttons in die Werkzeugleiste mit *KToolBar::insertButton* können Sie als letzten Parameter direkt den Tooltip-Text angeben (siehe auch Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*). Der Text sollte natürlich ebenso wie die Bezeichnungen der anderen Menüeinträge an die Landessprache angepasst sein.

Überflüssig und daher eher störend wirkt ein Tooltip-Fenster an GUI-Elementen, deren Bedeutung ohnehin schon durch eine Bezeichnung beschrieben ist. So ist zum Beispiel ein Button mit Text selbsterklärend. Der Tooltip-Text könnte kaum informativer sein, ohne zu lang zu werden. Ebenso benötigen Eingabezeilen, die mit einem Label beschriftet sind, keinen Tooltip. Auch für die Tooltip-Hilfe gilt also, dass man es nicht übertreiben sollte.

Sobald ein Widget mit Tooltip-Hilfe gelöscht wird, wird auch der dazugehörige Tooltip-Text gelöscht. Sie brauchen sich also nicht um die Freigabe zu kümmern. Wenn Sie dennoch ein Widget wieder von der Tooltip-Hilfe lösen wollen, benutzen Sie einfach die statische Funktion *QToolTip::remove*, der Sie als Parameter einen Zeiger auf das Widget übergeben.

Zusätzlich zu dem Text im Tooltip-Fenster kann man einen – meist etwas längeren – Erklärungstext zum Element per Signal verschicken. Dazu muss man ein Objekt der Klasse *QToolTipGroup* erzeugen. In der Methode *QToolTip::add* können Sie nun als dritten und vierten Parameter das *QToolTipGroup*-Element und den längeren Erklärungstext angeben. Sobald die Maus in das Widget kommt, wird der längere Text über das Signal *showTip* des *QToolTipGroup*-Objekts verschickt. Nach einer kurzen Zeit öffnet sich dann das Tooltip-Fenster mit dem kürzeren Text. Es hat sich durchgesetzt, den längeren Erklärungstext in der Statuszeile

anzeigen zu lassen. Um beispielsweise die Buttons der Werkzeugleiste mit der doppelten Erklärung auszustatten, können Sie folgendes Programmfragment (innerhalb des Konstruktors einer Hauptfensterklasse *MyMainWindow*) benutzen:

```
QToolTipGroup *ttgroup = new QToolTipGroup (this);
connect (ttgroup, SIGNAL (showTip (const QString &)),
        this, SLOT (statusTip (const QString &)));
connect (ttgroup, SIGNAL (remove ()),
        statusBar(), SLOT (clear ()));
toolBar()->insertButton ();
```

In der Klasse *MyMainWindow* muss dazu noch der Slot *statusTip* definiert werden:

```
void MyMainWindow::statusTip (const QString &tip)
{
    statusBar()->message (tip);
}
```

4.11.2 What's-This-Hilfe

Für die What's-This-Hilfe ist die Klasse *QWhatsThis* zuständig. Sie hat nur statische Methoden, mit denen man den Text zuordnen kann. Um beispielsweise das Widget *myWidget* mit einer What's-This-Hilfe auszustatten, benutzen Sie folgenden Methodenaufruf:

```
QWhatsThis::add (myWidget,
    i18n ("Here is a very long explanation for a widget \n"
        "that does nothing. You cannot enter a text,\n"
        "cannot click into the widget, get no context \n"
        "help... "));
```

Auch hier kann man eine What's-This-Erklärung wieder entfernen, indem man die statische Methode *remove* benutzt und ihr einen Zeiger auf das Widget als Parameter übergibt.

Besonders häufig wird diese Hilfeart für die Befehle von Menü- und Werkzeugleiste benutzt. Aus diesem Grund besitzen die Klassen *QAction* und *KAction* die Methode *setWhatsThis*, mit der Sie diesen Aktionen einen What's-This-Hilfstext zuordnen können (siehe Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*).

Einen Button mit einem Icon, das diese What's-This-Hilfe symbolisiert, können Sie mit der statischen Methode *QWhatsThis::whatsThisButton* erzeugen. Als Parameter müssen Sie das Vater-Widget angeben, dem der Button zugeordnet sein soll. Der Button ist vom Typ *QToolButton*. Wenn Sie dagegen einen eigenen Button oder einen Menüpunkt benutzen wollen, können Sie zum Start des What's-This-Modus die statische Methode *QWhatsThis::enterWhatsThisMode* aufrufen.

Sobald dieser Modus aktiviert ist, erscheint die What's-This-Hilfe für das nächste Fenster, auf das der Anwender klickt. (Die Methode *enterWhatsThisMode* ist leider keine Slot-Methode, da Slots nicht statisch sein können.)

4.11.3 Anwenderhandbuch

Die ausführlichste Hilfe findet der Anwender natürlich im Online-Handbuch. KDE benutzt für die Speicherung das HTML-Format. Dieses Format, das sich durch das World Wide Web sehr stark verbreitet hat, bietet den Vorteil, dass man damit sehr ansprechende Hilfedokumente erstellen kann. Man kann Grafiken einbinden, Verweise zu anderen Seiten des Hilfedokuments und zu Seiten im WWW einfügen, den Text in verschiedenen Zeichensätzen und Schriftgrößen anzeigen lassen und Tabellen erstellen. Um den Zeilenumbruch kümmert sich das anzeigende Programm. Die Anzeige der HTML-Dokumente im Hilfesystem von KDE übernimmt das Programm *kdehelp*. Es ist ein vollständiger Browser, der mit Hilfe des *kfm* sogar auf Seiten im Internet zugreifen kann. Die Hilfedokumente können aber auch mit jedem anderen Browser angezeigt werden.

Es würde den Rahmen dieses Buches sprengen, wenn wir hier die Sprache HTML erklären wollten. Es gibt im Internet aber genügend gute Anleitungen, die sehr detailliert beschreiben, wie man eigene HTML-Seiten erstellt. Hier sind jedoch einige Tipps, wie die Seiten aufgebaut sein sollten:

Die Startseite (meist eine Datei mit dem Namen *index.html*) sollte ein Inhaltsverzeichnis enthalten, das auf die verschiedenen Seiten der Online-Hilfe verweist. Jedes Thema sollte auf einer eigenen Seite erklärt sein. Querverweise zwischen den Seiten können sehr hilfreich sein. Es sollte mindestens eine Seite vorhanden sein, die angibt, wer der Autor des Programms ist und wie man ihn erreichen kann, sowie eine Seite, die über das Copyright des Programms informiert. Es empfiehlt sich, eine eigene Seite anzulegen, in der die Befehle der Menüleiste aufgelistet und erklärt sind. Ist die Menüleiste sehr umfangreich, sollte man diese Seite in sinnvolle kleinere Seiten aufspalten. Eine Seite, auf der die bisherigen Versionen des Programms und ihre Unterschiede kurz erläutert werden, sowie eine Seite, die einen Ausblick auf die geplanten Veränderungen bietet, sind ebenfalls sinnvoll. Beachten Sie beim Schreiben der Texte, dass die Online-Hilfe in der Regel von Anwendern gelesen wird, die selbst nicht viel mit Programmierung zu tun haben. Vermeiden Sie also Programmierer-Jargon. Machen Sie deutlich, welche Teile und Funktionen Ihres Programms besonders wichtig sind, und welche Funktionen eher für ganz spezielle Probleme genutzt werden können.

Viele KDE-Programmierer sind dazu übergegangen, das Online-Handbuch nicht mehr direkt in HTML zu schreiben. Sie benutzen stattdessen die Sprache SGML. Sie entspricht eher einer Buchbeschreibungssprache, in der ein Dokument in verschiedene Kapitel und Absätze unterteilt wird. Die Formatierung wird hier nicht direkt festgelegt, sondern es werden nur Attribute vergeben. SGML hat zwei Vor-

teile: Zum einen wird das ganze Online-Handbuch in einer Datei abgelegt. Man spart so das lästige Wechseln zwischen verschiedenen Dateien, wenn man das Handbuch bearbeitet. Zum anderen kann SGML auch mit automatischen Tools in HTML (in mehrere Dateien, von denen jede Datei ein Kapitel enthält), in LaTeX (zum Ausdrucken auf Papier) und einige andere Formate umgewandelt werden. Der Nachteil von SGML ist, dass man nicht so einfach alle Möglichkeiten von HTML ausschöpfen kann. Auch gibt es bereits sehr mächtige Editoren für HTML-Dateien, während SGML meist noch mit einem normalen Texteditor getippt wird.

Im KDE-Paket *kdesdk* sind einige Dateien für die Umwandlung von SGML- in HTML-Dokumente vorhanden, die ein einheitliches Format der erzeugten HTML-Dateien ermöglichen. Zusammen mit dem Tool *sgml2html* und diesen Dateien erzeugt man einheitliche Dokumente für seine Programme.

Es ist sehr einfach, das Online-Handbuch aufzurufen: Die Klasse *KApplication* enthält die Methode *invokeHTMLHelp*. Ein Aufruf dieser Methode startet automatisch das Programm *kdehelp*, das die Datei *index.html* aus dem Verzeichnis *\$KDEDIR/share/doc/HTML/<Länderkürzel>/<Programmname>/* anzeigt. Sie können im ersten Parameter der Methode *getHelp* auch den Namen einer anderen Datei angeben.

Mit der Methode *KMainWindow::helpMenu* können Sie ein Popup-Menü für den Menüpunkt HILFE erzeugen lassen. Es enthält den Menüpunkt INHALT, der das Online-Handbuch startet. Sie können diesem Popup-Menü noch eigene Menüpunkte (in der Regel an den Positionen hinter INHALT) hinzufügen, wenn Sie spezielle Unterthemen direkt aufrufen lassen wollen. Das kann beispielsweise folgendermaßen geschehen:

```
const char *aboutText = i18n ("...");
QPopupMenu *helpMenu = helpMenu(true, aboutText);
helpMenu->insertItem ("Tastaturbelegung", this,
                     SLOT (helpTastatur ()), 0, 0, 1);
helpMenu->insertItem ("Copyright", this
                     SLOT (helpCopyright ()), 0, 0, 2);
```

Die Slot-Methode *helpTastatur* (und analog *helpCopyright*) kann dann beispielsweise so aussehen:

```
void MyMainWindow::helpTastatur ()
{
    kapp->invokeHTMLHelp ("tastatur.html");
}
```

4.12 Timer-Programmierung, Datum und Uhrzeit

Viele Applikationen benötigen eine Zeitbasis. Die Aufgabenstellungen lassen sich dabei grob in vier Bereiche unterteilen. Zu jedem der Bereiche sind im Folgenden einige typische Anwendungsbeispiele angegeben.

1. Uhrzeit und Datum aus der internen Rechneruhr auslesen (Bildschirmuhr, Kalenderprogramm, Datum/Uhrzeit-Angabe in Log-Dateien)
2. Zeitspannen messen (Programmlaufzeiten oder Antwortzeiten messen)
3. Nach einer vorgegebenen Zeitspanne eine Aktion ausführen (Timeouts, Weckrufe)
4. In regelmäßigen Abständen eine Aktion ausführen (Autorepeat-Funktion bei Buttons und anderen Widgets, Echtzeit-Simulation, regelmäßiges Polling)

Für die ersten beiden Aufgabengebiete bietet Qt die Klassen *QDate*, *QTime* und *QDateTime* an, mit denen man Zeitpunkte festlegen, die aktuelle Uhrzeit aus der internen Rechneruhr auslesen und die Zeitspanne zwischen zwei Zeitpunkten bestimmen kann.

Die letzten beiden Aufgabenstellungen werden in Qt über *TimerEvents* – oder komfortabler mit der Klasse *QTimer* – gelöst, mit denen man Aktionen nach vorgegebenen Zeitspannen einmalig oder regelmäßig ausführen kann.

4.12.1 Die Klassen *QDate*, *QTime* und *QDateTime*

Ein Objekt der Klasse *QDate* stellt die Datenstruktur zur Verfügung, um ein Datum zu repräsentieren, und bietet Methoden zur Manipulation des Datums. Entsprechend kann ein Objekt der Klasse *QTime* eine Uhrzeit mit einer Genauigkeit von Millisekunden darstellen. Ein Objekt der Klasse *QDateTime* enthält ein Objekt von *QDate* und ein Objekt von *QTime* und kann somit einen Zeitpunkt eindeutig festlegen.

Alle drei Klassen, *QDate*, *QTime* und *QDateTime*, sind in der Header-Datei *qdatetime.h* deklariert.

Mit Hilfe der Konstruktoren

- *QDate* (*int y*, *int m*, *int d*)
- *QTime* (*int h*, *int m*, *int s=0*, *int ms=0*)
- *QDateTime* (*const QDate &*, *const QTime &*)

kann man die Datenstrukturen auf ein bestimmtes Datum, eine Uhrzeit oder einen Zeitpunkt (aus Datum und Uhrzeit) initialisieren. Die gültigen Wertebereiche für die Parameter sowie die einzelnen Methoden zum Auslesen und zur Manipulation der Werte lesen Sie bitte im Referenzteil der Klassen *QDate*, *QTime* und *QDateTime* nach.

Um das aktuelle Datum, die aktuelle Uhrzeit oder beides aus der internen Rechneruhr auszulesen, können Sie eine der statischen Methoden

- `QDate::currentDate ()`
- `QTime::currentTime ()`
- `QDateTime::currentDateTime ()`

benutzen. Diese Methoden liefern ein Objekt der Klasse `QDate`, `QTime` bzw. `QDateTime` zurück, das den aktuellen Wert enthält. Beachten Sie dabei, dass die zeitliche Auflösung von der Auflösung der internen Rechneruhr abhängt und nicht unbedingt die exakten Millisekunden wiedergibt (siehe auch Übungsaufgabe 4.7).

Um beispielsweise in einer Fehlermeldung das Datum und die aktuelle Uhrzeit einzufügen, können Sie folgende Zeile benutzen:

```
fprintf (stderr, "[%s] Verbindung unterbrochen\n",
        QDateTime::currentDateTime().toString().data());
```

Die Methode `QDateTime::toString()` erzeugt allerdings eine Ausgabe, die nicht an die Landessprache angepasst ist (siehe Kapitel 4.9.4, *Zahlen-, Währungs- und Datumsformate*).

Wenn Sie Datum und Uhrzeit des aktuellen Zeitpunkts bestimmen wollen, benutzen Sie `QDateTime::currentDateTime` statt zwei getrennter Aufrufe von `currentDate` und `currentTime`. Sollte nämlich zwischen den beiden Aufrufen das Datum umspringen, passen die Werte nicht mehr zueinander. In `currentDateTime` wird dieser Sonderfall abgefangen.

Die drei Klassen `QDate`, `QTime` und `QDateTime` können außerdem benutzt werden, um Zeitspannen zwischen zwei Zeitpunkten zu berechnen. Die Klassen besitzen dazu folgende Methoden:

- `int QDate::daysTo (const QDate &)`
- `int QDateTime::daysTo (const QDateTime &)`
- `int QTime::secsTo (const QTime &)`
- `int QDateTime::secsTo (const QDateTime &)`
- `int QTime::msecsTo (const QTime &)`

Diese Methoden ergeben die Anzahl der Tage, Sekunden bzw. Millisekunden von dem Zeitpunkt an, den das Objekt angibt, bis zu dem Zeitpunkt, der durch das Argument angegeben ist. Um beispielsweise die Tage von heute bis Weihnachten zu berechnen, können Sie folgenden Code benutzen:

```
QDate heute = QDate::currentDate();
QDate weihnachten (25, 12, heute.year ());
printf ("Noch %d Tage bis Weihnachten!\n",
        heute.daysTo (weihnachten));
```

Oder, wenn Sie es kürzer (aber auch unübersichtlicher) formulieren möchten:

```
printf ("Noch %d Tage bis Weihnachten!\n",
        QDate::currentDate().daysTo (
        QDate (25, 12, QDate::currentDate().year())));
```

Wenn der Zeitpunkt im Argument der Funktion vor dem Zeitpunkt des Objekts liegt, ist das Ergebnis negativ. Wenn Sie also einen der beiden Codeausschnitte an Silvester ausführen, teilt Ihnen das Programm mit, dass Sie Weihnachten um sechs Tage verpasst haben:

```
Noch -6 Tage bis Weihnachten!
```

Da die Rückgabewerte der Methoden vom Typ *int* sind, und *int* auf den meisten Rechnerarchitekturen als vorzeichenbehaftete 32-Bit-Zahl implementiert wird, sollten Sie davon ausgehen, dass nur Werte zwischen -2.147.483.648 und 2.147.483.647 (-2^{31} bis $2^{31} - 1$) korrekt ohne Überlauf dargestellt werden. Daraus ergibt sich, dass die Methode *QDateTime::secsTo* nur für eine Zeitspanne, die kürzer als 68 Jahre ist, korrekte Ergebnisse liefert. Die anderen vier Methoden erreichen diese Grenze nicht.

Beachten Sie auch, dass zur Berechnung bei *QDateTime::daysTo* ausschließlich das Datum herangezogen wird und die Uhrzeit nicht in die Berechnung eingeht. So ergibt zum Beispiel ein Aufruf für den Zeitraum vom 1. Januar, 6.00 Uhr, bis zum 2. Januar, 18:00 Uhr, den Wert von einem Tag, für den Zeitraum zwischen dem 1. Januar, 18.00 Uhr, und dem 3. Januar, 6:00 Uhr, dagegen einen Wert von zwei Tagen, obwohl in beiden Fällen der Zeitraum exakt 36 Stunden umfasst. Abbildung 4.53 verdeutlicht dieses Problem noch einmal.

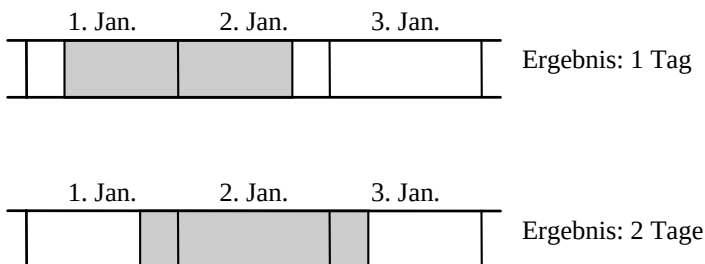


Abbildung 4-53 *QDateTime::daysTo* berücksichtigt nur das Datum.

Wenn Sie kurze Zeitspannen messen wollen – zum Beispiel die Laufzeit eines Programmfragments –, können Sie folgendes Code-Fragment benutzen:

```
QTime time = QTime::currentTime ();
/*
    Hier steht der Teil Ihres Programms,
    von dem Sie die Laufzeit messen wollen.
*/
printf ("Die Routine benötigte %d Millisekunden.\n",
        time.msecsTo (QTime::currentTime ()));
```

Um genau diese Anwendung weiter zu vereinfachen, enthält die Klasse *QTime* die Methoden *start* und *elapsed*. *start* setzt die Uhrzeit des Objekts auf die aktuelle Uhrzeit, und *elapsed* berechnet den Unterschied von der gespeicherten Uhrzeit zur aktuellen Uhrzeit. Das obige Programmstück kann also auch so aussehen:

```
QTime time;
time.start ();
/*
    Hier steht der Teil Ihres Programms,
    von dem Sie die Laufzeit messen wollen.
*/
printf ("Die Routine benötigte %d Millisekunden.\n",
        time.elapsed ());
```

Weiterhin gibt es noch die Methode *restart*, die die verstrichene Zeit in Millisekunden zurückgibt und das *QTime*-Objekt wieder auf die aktuelle Zeit zurücksetzt. Dabei wird nur einmal die aktuelle Uhrzeit ermittelt.

Die Zeitspanne, die Sie mit einem *QTime*-Objekt messen können, ist auf 24 Stunden beschränkt. Danach wird die Messung wieder bei 0 gestartet. Alternativ können Sie ein *QDateTime*-Objekt benutzen, haben dann allerdings nur eine Zeitauflösung von einer Sekunde.

4.12.2 Nutzung von Timer-Events

Wenn Ihr Programm in regelmäßigen Abständen Aktionen ausführen soll oder eine Aktion erst nach einer genau definierten Verzögerung ausgeführt werden soll, benötigt Ihr Programm einen anders gearteten Mechanismus. Zwar lassen sich diese Aufgaben theoretisch auch mit der Klasse *QTime* lösen, aber nur, wenn man gravierende Nachteile in Kauf nimmt. Nehmen Sie beispielsweise an, Ihr Programm soll zehn Sekunden lang pausieren und erst danach eine Aktion ausführen. Eine Lösung für dieses Problem könnte etwa so aussehen:

```
// Schlechte Lösung, NICHT empfehlenswert
QTime time;
time.start ();
```

```
// Leere Schleife, bis 10.000 Millisekunden  
// verstrichen sind  
while (time.elapsed () < 10000);  
aktion ();
```

Dieses Programm wartet zwar tatsächlich zehn Sekunden, bevor die Funktion *aktion* aufgerufen wird, diese Zeit verbringt es aber mit Busy-waiting, d.h. die CPU wird die ganze Zeit voll in Beschlag genommen, obwohl keine sinnvolle Aktion oder Berechnung stattfindet. Ein zweiter gravierender Nachteil, der gerade bei grafischen Benutzerschnittstellen von enormer Bedeutung ist, besteht darin, dass in dieser Zeit das Programm nicht auf Benutzerinteraktionen reagiert. Die von der Maus und der Tastatur eintreffenden Events werden nicht verarbeitet und warten in der Event-Queue; das Programm »reagiert nicht mehr« und das für volle zehn Sekunden.

Eine Lösung, die zumindest den ersten Nachteil vermeidet, ist der Aufruf der Betriebssystemfunktion *sleep (unsigned int seconds)*, mit der der Prozess für eine definierte Zeit die CPU freigibt.

```
// Besser, aber noch immer blockierend  
sleep (10);  
aktion ();
```

Der Nachteil, dass das Programm in dieser Zeit nicht auf Tastatur- und Mauseingaben reagiert, bleibt aber bestehen. Weiterhin kann *sleep* durch ein Unix-Signal unterbrochen werden. Das muss noch abgefangen und entsprechend behandelt werden.

Eine dritte Lösung wäre der Einsatz eines Signal-Handlers, der das Unix-Signal *SIGALARM* abfängt, in Kombination mit der Funktion *alarm*. Das Programm läuft dann normal weiter, und erst nach Ablauf der Zeit wird es durch das asynchrone Unix-Signal unterbrochen. Das Programm bleibt daher bedienbar und verbraucht auch nicht unnötig CPU-Zeit. Der Nachteil liegt jedoch zum einen darin, dass innerhalb des asynchron aufgerufenen Signal-Handlers nur bestimmte Funktionen benutzt werden dürfen, um Konsistenz zu wahren. Zum anderen ist die Anzahl der gleichzeitig aktivierbaren Alarm-Aufträge begrenzt (in Linux auf einen einzigen). Außerdem ist die Programmierung von Unix-Signal-Handlern kompliziert und plattformabhängig.

Qt bietet zum Glück eine sehr gute Lösung für diese Aufgaben an, die diese Probleme umgeht. Es verwaltet intern eine Timer-Liste, und sobald ein Timer abgelaufen ist, wird ein (synchroner) Timer-Event an das zugehörige *QObject* geschickt. Dieser Timer-Event erfolgt immer synchron, unterbricht also nicht laufende Befehle. Timer-Events werden nur in der Haupt-Event-Schleife erkannt und bearbeitet. Inkonsistenzen sind damit ausgeschlossen. Als Reaktion auf

einen Timer-Event dürfen Sie daher alle Aktionen ausführen, zum Beispiel auch auf den Bildschirm zugreifen, auf Dateien zugreifen und sogar neue Timer setzen oder löschen.

Beachten Sie, dass die Klasse *QTimer* meist einfacher und komfortabler zu benutzen ist als die Timer-Events von *QObject*. Sie können den Rest dieses Teilkapitels überspringen und direkt bei Kapitel 4.12.3, *Die Klasse QTimer*, fortfahren.

Da der Timer-Event nur in der Haupt-Event-Schleife bearbeitet wird, kann nicht garantiert werden, dass der Timer auch wirklich zu dem Zeitpunkt die Aktion auslöst, zu dem seine Zeit abläuft. Ist das Programm gerade in dem Moment mit einer längeren Berechnung beschäftigt, so verzögert sich die Ausführung der Aktion, bis die Berechnung beendet ist und die Kontrolle wieder an die Haupt-Event-Schleife zurückgegeben wird.

Qt stellt Timer-Events in der Klasse *QObject* zur Verfügung. Diese Klasse enthält folgende Methoden:

- *int QObject::startTimer (int interval)*
- *void QObject::killTimer (int id)*
- *void QObject::killTimers ()*

Mit der Methode *startTimer* wird ein neuer Timer für dieses Objekt gestartet, der in regelmäßigen Abständen von *interval* Millisekunden einen Event vom Typ *timerEvent* an das Objekt schickt. Um beim Einsatz mehrerer Timer in einem Objekt die einzelnen Timer voneinander unterscheiden zu können, wird jedem Timer eine eindeutige Nummer zugeordnet. Diese Nummer liefert die Methode *startTimer* als Rückgabewert.

Ist der Timer abgelaufen, so wird ein Event vom Typ *TimerEvent* an das Objekt gesendet. Am einfachsten können Sie diesen Event nutzen, indem Sie eine eigene Klasse von *QObject* ableiten und die virtuelle Methode *timerEvent* überschreiben. Diese Methode hat als Parameter ein Objekt der Klasse *QTimerEvent*, mit der Sie über die Methode *QTimerEvent::timerId* die eindeutige Nummer des Timers herausfinden können, der abgelaufen ist.

Will man einen bestimmten Timer wieder löschen, kann man die Methode *killTimer* benutzen. Will man alle Timer löschen, leistet *killTimers* gute Dienste. Aktive Timer werden auch automatisch beim Löschen des Objekts mit gelöscht.

Wenn man im Objekt nur einen Timer benötigt, braucht man die Identifikationsnummer nicht zu speichern, denn die Methode *timerEvent* kann ja nur von dem einzigen Timer ausgelöst worden sein. Wenn Sie diesen Timer dann löschen wollen, benutzen Sie einfach *killTimers*.

Hier sehen Sie zunächst ein einfaches Beispiel eines regelmäßigen Events. Eine Instanz der folgenden Klasse simuliert eine »tickende« Uhr und gibt im Sekundenrhythmus abwechselnd die Wörter »tick« und »tack« aus.

```
class ticktack : public QObject {
    Q_OBJECT
public:
    ticktack (QObject *parent=0, const char *name=0);
    ~ticktack () {}
protected:
    void timerEvent (QTimerEvent *ev);
private:
    bool tick_next;
};

ticktack::ticktack (QObject *parent, const char *name) :
    QObject (parent, name)
{
    tick_next = true; // Die Ausgabe beginnt mit "tick".
    startTimer (1000); // Einen Timer mit Intervall
                      // von einer Sekunde starten
}

void ticktack::timerEvent (QTimerEvent *ev) {
    // Ausgabe des Geräusches
    printf ("%s\n", tick_next ? "tick" : "tack");
    // Beim nächsten Mal das andere
    tick_next = !tick_next;
}
```

Im Konstruktor wird zunächst die Variable *tick_next* initialisiert, die angibt, ob als nächstes »tick« oder »tack« ausgegeben wird. Danach wird ein Timer mit einem Zeitintervall von 1000 Millisekunden gestartet. Alle 1000 Millisekunden wird nun die Methode *timerEvent* aufgerufen, und in ihr wird die Ausgabe vorgenommen.

Ein Beispiel für den Einsatz von mehr als einem Timer in einem Objekt wird in Übungsaufgabe 4.8 besprochen.

Für einige Anwendungen lässt sich ein Timer mit dem Zeitintervall 0 gut einsetzen. Ein solcher Timer schickt jedes Mal einen Event, Wenn die Applikation in die Event-Schleife eintritt. Das kann zum Beispiel benutzt werden, um Berechnungen durchführen zu lassen und gleichzeitig Events zu verarbeiten. Eine ausführlichere Besprechung dieser Technik finden Sie in Kapitel 4.11, *Blockierungsfreie Programme*.

Es gibt noch ein anderes Problem, zu dessen Lösung der Null-Intervall-Timer gut eingesetzt werden kann: Oftmals beginnt ein Objekt seine Arbeit unmittelbar nach der Erzeugung mit dem Konstruktor. Wenn es dann allerdings bereits

Signale sendet, laufen diese meist ins Leere, weil noch keine Slots mit den Signalen verbunden werden konnten. Die Arbeit innerhalb des Objekts sollte also erst beginnen, nachdem man dem Hauptprogramm Gelegenheit gegeben hat, Signal/Slot-Verbindungen aufzubauen. Dazu kann man das Objekt zum Beispiel mit einer Start-Methode versehen, die aufgerufen wird, nachdem alle Verbindungen hergestellt wurden. Eleganter geht es jedoch mit einem Null-Intervall-Timer, der aktiviert wird, sobald die Haupt-Event-Schleife zum erstenmal aufgerufen wird. In diesem Fall entfällt der Aufruf einer Start-Methode von Hand.

In einem Programm kann dieses Verhalten zum Beispiel so implementiert werden:

```
class MyObject : public QObject {
    Q_OBJECT
public:
    MyObject (QObject *parent, const char *name);
    ~MyObject () {}
signals:
    void aktion1 ();
protected:
    void timerEvent (QTimerEvent *ev);
}

MyObject::MyObject (QObject *parent, const char *name) :
    QObject (parent, name)
{
    // Initialisierungen
    startTimer (0);    // Null-Timer starten
}

void MyObject::timerEvent (QTimerEvent *)
{
    killTimers ();    // Timer nur einmal ausführen

    // Berechnungen durchführen...

    emit aktion1 ();    // Signal schicken, ist jetzt
                        // mit einem Slot verbunden
}

int main () {

    ...
    // Objekt wird erzeugt, Berechnung aber noch
    // zurückgestellt
    MyObject *obj = new MyObject ();

    // Signal wird verbunden
```

```
connect (obj, SIGNAL (aktion1 ()),
        receiverObj, SLOT (slotXY ()));
...

// Eintritt in die Event-Schleife, Timer mit
// Intervall 0 wird unmittelbar ausgeführt
return a.exec ();
}
```

4.12.3 Die Klasse *QTimer*

Die Arbeit mit dem Timer-Event besitzt zwei unschöne Eigenschaften: Man muss eine Klasse ableiten, um die *timerEvent*-Methode überschreiben zu können, und wenn man mit mehreren Timern arbeitet, muss man selbst unterscheiden, welcher der Timer aktiv wurde. Um diese beiden Nachteile zu vermeiden und die Arbeit mit Timern komfortabler zu machen, besitzt Qt eine Klasse *QTimer*. Sie ist direkt von der Klasse *QObject* abgeleitet und benutzt intern genau den Mechanismus des Timer-Events, kapselt diesen jedoch, so dass er dem Programmierer verborgen bleibt.

Um einen Timer zu starten, muss man zunächst ein Objekt vom Typ *QTimer* anlegen. Anschließend ruft man die Methode *QTimer::start (int msec, bool singleshot)* auf. Im ersten Parameter, *msecs*, kann man die Intervalllänge angeben. Hat der zweite Parameter, *singleshot*, den Wert *false* (Default-Einstellung), handelt es sich um einen regelmäßigen Timer. Ist er dagegen *true*, läuft der Timer nur einmal ab und wird danach wieder gestoppt.

Jedes Mal, wenn der Timer abgelaufen ist, sendet er das Signal *timeout* (ohne Parameter). Um den Timer also zu benutzen, müssen Sie dieses Signal mit einem Slot verbinden, der die gewünschte Aktion ausführt.

Mit der Methode *stop* können Sie den Timer abbrechen; mit der Methode *isActive* können Sie testen, ob der Timer gerade läuft. Mit der Methode *changeInterval* können Sie den Timer auf ein anderes Intervall einstellen. Sollte der Timer bereits aktiv sein, so wird er abgebrochen und mit dem neuen Intervall neu gestartet. Ansonsten wird ein neuer wiederholender Timer mit dem angegebenen Intervall angelegt und gestartet.

Beispiele für den Einsatz der *QTimer*-Klasse werden in den Übungsaufgaben 4.7 bis 4.10 besprochen.

Die Klasse *QTimer* enthält außerdem die statische Methode *singleShot (int msecs, QObject *receiver, const char *member)*. Mit dieser Methode kann man einen einmaligen Timer starten, ohne ein Objekt vom Typ *QTimer* erzeugen zu müssen. Dazu gibt man der Methode als ersten Parameter, *msecs*, das Zeitintervall in Millisekunden an und als zweiten und dritten Parameter, *receiver* und *member*, ein Objekt und einen dazugehörigen, parameterlosen Slot, der bei Ablauf des Timers aufgerufen werden soll.

Auch die Klasse *QTimer* lässt sich zur Bildung eines Null-Intervall-Timers einsetzen (siehe auch Übungsaufgabe 4.9).

4.12.4 Einschränkungen von Timern

Die Timer von Qt, die in Kapitel 4.10.2, *Die Klasse QTimer*, beschrieben wurden, arbeiten sehr genau und sind leicht und universell einsetzbar. Dennoch ergeben sich ein paar Einschränkungen aufgrund der Tatsache, dass es sich um synchrone Timer handelt. Wenn sich Ihr Programm in einer aufwendigen Berechnung befindet, werden währenddessen die Timer nicht geprüft. Erst wenn das Programm nach dem Beenden der Berechnung in die Event-Schleife zurückkehrt, werden die Timer getestet und die entsprechenden Events verschickt. Timer-Events werden dabei jedoch nicht angesammelt: Auch wenn ein Timer mehr als eine Aktivierung »verschlafen« hat, wird dennoch nur ein Event verschickt, und die nächste Aktivierung des Timers geschieht erst zum Zeitpunkt (*aktuelle Zeit + Timer-Intervall*). Sie können sich also nicht unbedingt darauf verlassen, dass beispielsweise ein 10-ms-Timer in einer Sekunde auch genau 100 Events erzeugt. Falls das Programm mit deren Abarbeitung nicht schnell genug nachkommt, können unter Umständen einige Events verschluckt werden. Falls die Anzahl der Timer-Events auf jeden Fall stimmen muss, können Sie die Klasse benutzen, die in Übungsaufgabe 4.10 entwickelt wird.

Da das Intervall eines Timers durch ein Argument vom Typ *int* angegeben wird, ist $2^{31} - 1$ ms, also 24,8 Tage, das längste mögliche Intervall. Wenn Sie längere Intervalle benötigen, können Sie einen kürzeren Timer starten und mit einem Zähler die Anzahl der Events mitzählen. Siehe dazu auch Übungsaufgabe 4.6.

Durch die momentane Realisierung der Timer in Linux kann es auch zu Problemen kommen, wenn Sie Timer mit sehr langen Intervallen (länger als ein paar Stunden) starten und Ihr Programm ansonsten keine Events zu verarbeiten hat. Das hängt mit der Erkennung der Mitternachtsgrenze zusammen, soll hier aber nicht weiter ausgeführt werden. Sie können dem leicht abhelfen, indem Sie einen »Leer«-Timer mit kurzem Intervall starten (z.B. eine Minute), der keine Aktion ausführt. Sie müssen dazu nicht für jedes Objekt einen solchen Leer-Timer starten. Es reicht, wenn Ihr Programm insgesamt einen solchen Timer enthält. Dieses Problem wird vielleicht in einer der nächsten Qt-Versionen behoben sein.

4.12.5 Übungsaufgaben

Übung 4.7

Entwerfen Sie eine Klasse *AlarmTimer*, deren Objekte im Konstruktor einen Zeitpunkt vom Typ *QDateTime* mitgeteilt bekommen und die zu diesem Zeitpunkt ein Signal, *alarm*, aussenden. Beachten Sie dabei folgende Punkte:

- Die Dauer des Zeitraums sollte beliebig lang sein können.
- Die volle Genauigkeit des Endzeitpunkts (Millisekunden) sollte ausgenutzt werden.
- Die benötigte CPU-Zeit sollte möglichst gering sein.

Überlegen Sie auch, was geschehen sollte, wenn beim Aufruf des Konstruktors der Endzeitpunkt bereits vorbei ist.

Übung 4.8

Schreiben Sie ein Programm, das die minimale Auflösung der Timer-Events ermittelt und ausgibt, das also ermittelt, ob Qt zum Beispiel den Unterschied zwischen einem 30-ms-Timer und einem 31-ms-Timer darstellen kann.

Übung 4.9

Schreiben Sie ein Programm, das die Zahl der verbleibenden Sekunden bis zum Jahr 2100 in einem Fenster ausgibt und regelmäßig aktualisiert. Versuchen Sie dabei, möglichst viele der folgenden Punkte zu berücksichtigen:

- Die verbleibende Zeit sollte auf volle Sekunden aufgerundet werden, d.h. die Anzeige sollte genau am 1. Januar 2100 um Mitternacht auf 0 springen.
- Die Aktualisierung sollte möglichst exakt am Beginn einer neuen Sekunde geschehen.
- Die benötigte Rechenzeit sollte gering sein.

Übung 4.10

Entwerfen Sie eine Widget-Klasse *AutoRepeatButton*, die eine Schaltfläche mit automatischer Wiederholung erzeugt. Wenn man mit der Maus auf die Schaltfläche klickt und die Maustaste gedrückt hält, soll das Signal *clicked* gesendet werden:

- unmittelbar nach dem Klicken einmal
- nach einer Sekunde alle 200 ms erneut
- nach einer weiteren Sekunde alle 50 ms erneut
- nach einer weiteren Sekunde so schnell wie möglich

Leiten Sie die Klasse *AutoRepeatButton* von der Klasse *QPushButton* ab, und überschreiben Sie die virtuellen Methoden *QMouseEvent* und *QMouseEvent*. (Auch *QPushButton* besitzt bereits die Möglichkeit, mit *setAutoRepeat* eine automatische Wiederholung zu aktivieren. In der hier entwickelten Klasse wird jedoch eine schrittweise ansteigende Wiederholungsrate implementiert.)

Übung 4.11

Entwerfen Sie eine Timer-Klasse, die ein »Verschlucken« von Timer-Aktivierungen verhindert. Wenn also Timer-Aktivierungen verpasst wurden, weil das Programm anderweitig beschäftigt war, sollen die Aktivierungen nachgeliefert werden.

Ein solcher Timer führt natürlich zu Problemen, wenn die Abarbeitung einer Timer-Aktivierung länger dauert als das Timer-Intervall, da sich dann immer mehr Aktivierungen ansammeln, die nicht mehr abgebaut werden können. Wie könnte man dieses Problem entschärfen?

4.13 Blockierungsfreie Programme

Wenn ein Programm eine zeitaufwendige Berechnung oder Ein-/Ausgabe-Operation vornimmt, ist während dieser Zeit die Haupt-Event-Schleife nicht aktiv. Die grafische Oberfläche der Applikation ist solange nicht bedienbar. Das wirkt sehr irritierend auf den Anwender, der oft nicht weiß, ob das Programm abgestürzt ist. Außerdem sollten langwierige Berechnungen immer abgebrochen werden können.

Um dem Benutzer zumindest den Hinweis zu geben, dass das Programm zur Zeit eine Berechnung ausführt und daher nicht bedienbar ist, kann man als Mauszeiger innerhalb der Applikationsfenster eine Uhr darstellen lassen. Das erreichen Sie mit folgender Zeile:

```
QApplication::setOverrideCursor (waitCursor);
```

Wenn sich der Mauszeiger nun innerhalb eines der Applikationsfenster befindet, wird er automatisch als Armbanduhr oder Sanduhr dargestellt (abhängig vom X-Server). Nach der blockierenden Aktion kann dann mit der Zeile

```
QApplication::restoreOverrideCursor ();
```

der Mauszeiger wieder seine normale Gestalt bekommen. Diese Zeile sollten Sie auf keinen Fall vergessen.

Diese Möglichkeit ist natürlich nur ein Hinweis für den Anwender, dass die Applikation zur Zeit beschäftigt ist und nicht bedient werden kann. In guten Programmen sollte das grundsätzlich vermieden werden. Sie sollten so aufgebaut sein, dass der Anwender immer ohne wesentliche Verzögerung das Programm beeinflussen kann.

Qt bietet eine ganze Reihe von Möglichkeiten, ein Programm interaktiv zu halten. Wie man die Abfrage von Events auch während einer längeren Berechnung sicherstellt, wird in Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*, beschrieben. Wie Sie verhindern können, dass Ein-/Ausgabe-Operationen Ihr

Programm blockieren, erfahren Sie in Kapitel 4.13.2, *Blockierungsfreie Ein- und Ausgabe*. Die Übertragung von Daten im Hintergrund wird in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*, beschrieben. Wie Sie mit Qt auch mehrere Prozesse oder Threads nutzen können und was Sie dabei beachten müssen, wird in Kapitel 4.13.4, *Mehrere Prozesse und Threads*, beschrieben.

4.13.1 Event-Verarbeitung während langer Berechnungen

Berechnungen werden in einem Programm mit grafischer Oberfläche fast immer durch eine Aktion des Anwenders gestartet. In diesem Fall erkennt die Haupt-Event-Schleife den Event von Maus oder Tastatur und sendet ihn an das betroffene Widget weiter – zum Beispiel ein *QPushButton*-Objekt. Dieses Widget interpretiert den Event und ruft zum Beispiel eine Signalmethode auf, die eine verbundene Slot-Methode aufruft. In dieser Slot-Methode findet dann beispielsweise eine Berechnung statt. Solange diese Berechnung aber nicht abgeschlossen ist, kehrt die Kontrolle nicht zur Haupt-Event-Schleife zurück. Während dieser Zeit reagiert das Programm nicht mehr auf Events von Tastatur oder Maus. Auch die Widgets werden nicht aktualisiert. Wird beispielsweise ein Fenster der Applikation plötzlich aufgedeckt, so wird der ehemals verdeckte Bereich nur mit der Hintergrundfarbe ausgefüllt, nicht aber neu gezeichnet. In einer guten Applikation sollte diese Blockierung nicht bemerkbar sein. Als Faustregel gilt, dass eine Berechnung die Haupt-Event-Schleife möglichst nicht länger als 200 ms unterbrechen sollte. Das ist für den Anwender dann kaum feststellbar. Eine Blockierung von bis zu zwei Sekunden wird vom Anwender oft noch toleriert, wenn sie nur selten vorkommt. Blockierungen bis zu 20 Sekunden sind im Notfall gerade noch erträglich. Länger sollte eine Blockierung aber auf keinen Fall dauern.

Als Beispiel für eine solche Situation werden wir hier ein Problem aus der theoretischen Informatik betrachten: ein modifiziertes Knapsack-Problem. Aus 30 reellen Zahlen sollen die Zahlen ausgewählt werden, deren Summe maximal, aber kleiner als 10 ist.

Eine Funktion, die die Lösung dieses Problems liefert, indem sie einen vollständigen Suchbaum durchsucht, könnte zum Beispiel so aussehen:

```
double loesung (double a[30])
{
    double maximum = 0.0; // aktuelles Maximum
    bool use [30]; // true, wenn eine Zahl benutzt wird
    int i;

    // zunächst alle Einträge in use löschen
    for (i = 0; i < 30; i++)
        use [i] = false;

    while (1) {
```

```

// Summe berechnen
double summe = 0.0;
for (i = 0; i < 30; i++)
    summe += a[i];

// evtl. neues Maximum setzen
if (summe <= 10.0 && summe > maximum)
    maximum = summe;

// nächste Kombination bilden
for (i = 0; i < 30; i++) {
    use[i] = !use[i];
    if (use[i])
        break;
}

// falls alle Kombinationen getestet, beenden
if (i == 30)
    return maximum;
}
}

```

Wir wollen hier nicht weiter auf diese Funktion eingehen. Da insgesamt 2^{30} , also 1.073.741.824 Kombinationen getestet werden müssen, benötigt diese Funktion auch auf schnellen Rechnern eine lange Zeit. (Selbst bei 1.000.000 getesteten Kombinationen pro Sekunde braucht diese Funktion noch 18 Minuten.) Würden wir diese Funktion in einer Applikation benutzen, wäre die grafische Benutzeroberfläche der Applikation also 18 Minuten lang nicht bedienbar.

Dieses Problem kann umgangen werden, wenn wir die Berechnung in kleine Teile zerlegen, zwischen denen jedes Mal die Kontrolle kurz an die Haupt-Event-Schleife zurückgegeben wird, damit diese auf neu eingetroffene Events reagieren kann. Dazu können wir zum Beispiel die Methode *processEvents* des *QApplication*-Objekts benutzen. Durch den Aufruf dieser Methode werden die Events abgearbeitet, die in der Event-Queue warten. Sobald die Event-Queue leer ist oder eine bestimmte Zeit überschritten ist (die im Parameter der Methode *processEvents* als *int*-Wert in Millisekunden angegeben werden kann), kehrt die Methode *processEvents* zum Aufrufer zurück. Sollte die Event-Queue bereits beim Aufruf leer sein, kehrt die Methode sofort zurück.

In unserem oberen Beispiel können wir beispielsweise nach jeder getesteten Kombination die Methode aufrufen. Die Funktion sähe dann zum Beispiel so aus:

```

double loesung (double a[30])
{
    double maximum = 0.0; // aktuelles Maximum
    bool use [30]; // true, wenn eine Zahl benutzt wird

```

```

int i;

// zunächst alle Einträge in use löschen
for (i = 0; i < 30; i++)
    use [i] = false;

while (1) {
    // Summe berechnen, Maximum setzen und
    // Kombination neu wählen
    ...

    // falls alle Kombinationen getestet, beenden
    if (i == 30)
        return maximum;

    // Events abarbeiten
    kapp->processEvents ();
}
}

```

Wenn wir wie in diesem Beispiel den Parameter der Methode *processEvents* frei lassen, kehrt die Methode spätestens nach 300 ms zurück.

Da der Aufruf von *processEvents* eine gewisse Zeit in Anspruch nimmt, auch wenn die Event-Queue leer ist (je nach Rechner zwischen 500 µs und 30 ms), ist in unserem Fall der Aufruf nach jeder getesteten Kombination ineffizient. Nur ein kleiner Teil der CPU-Zeit wird tatsächlich für die Berechnung verwendet, ein großer Teil entfällt auf die Abfrage der Event-Queue. Daher empfiehlt es sich hier, zum Beispiel einen Zähler mitlaufen zu lassen und die Methode *processEvents* nur jedes tausendste Mal auszuführen. Auch auf einem langsamen Rechner benötigt der Test von 1000 Kombinationen maximal ein paar Millisekunden. Der Code könnte zum Beispiel so aussehen:

```

double loesung (double a[30])
{
    ...
    int counter = 0;
    while (1) {
        ...
        // bei jedem tausendsten Mal Events abarbeiten
        if (++counter >= 1000) {
            kapp->processEvents ();
            counter = 0;
        }
    }
}

```

Beachten Sie, dass ein Aufruf des Slots `QApplication::quit` zunächst noch keine Wirkung hat, da sich das Programm nicht in der Haupt-Event-Schleife befindet. Selbst wenn der Anwender während der Berechnung den Menübefehl **BEENDEN** wählt, wird das Programm nicht sofort verlassen, sondern erst nach dem Ende der Berechnung. Man sollte dem Anwender immer die Möglichkeit geben, die Berechnung auch wieder abubrechen. Das kann zum Beispiel geschehen, indem eine globale Variable gesetzt wird, sobald der Anwender die Berechnung abbricht. Diese Variable wird innerhalb der Berechnungsschleife geprüft. Ist sie gesetzt, wird die Schleife beendet.

Beachten Sie auch, dass der Kontrollfluss des Programms hier immer wieder in die Berechnungsfunktion läuft. Wenn Sie eine weitere Berechnung starten lassen, die den gleichen Mechanismus benutzt, so wird zuerst die neu gestartete Berechnung ausgeführt. In ihr wird durch die regelmäßigen Aufrufe von `processEvents` die grafische Oberfläche nicht blockiert. Die zuerst gestartete Berechnung erhält jedoch die Kontrolle erst zurück, wenn die zweite Berechnung beendet ist.

Wenn das weitere Programm vom Ergebnis der Berechnung abhängt, ist es oft nicht sinnvoll, dass das Programm während der Berechnung bedient werden kann. Dennoch sollte man in regelmäßigen Abständen `processEvents` aufrufen, damit die Fenster des Programms neu gezeichnet werden können. Außerdem sollte man dem Anwender die Möglichkeit geben, die Berechnung abubrechen. Alle anderen Befehle und Bedienelemente sollten jedoch blockiert sein.

Am besten benutzt man hierzu die Klasse `QProgressDialog`. Es handelt sich hierbei um ein Dialogfenster, das die Eingabe in ein anderes Fenster blockiert (siehe Abbildung 4.54). Gleichzeitig stellt es einen Fortschrittsbalken zur Verfügung, in dem man den Fortschritt der Berechnung anzeigen lassen kann, sowie eine Schaltfläche **ABBRECHEN**, mit der der Anwender die Berechnung vorzeitig beenden kann. `QProgressDialog` enthält noch ein weiteres Feature: Das Dialogfenster wird nur geöffnet, wenn die geschätzte Zeit, die die Berechnung benötigt, mehr als drei Sekunden beträgt. So wird verhindert, dass das Dialogfenster nur kurz aufgeht, um direkt danach wieder geschlossen zu werden.

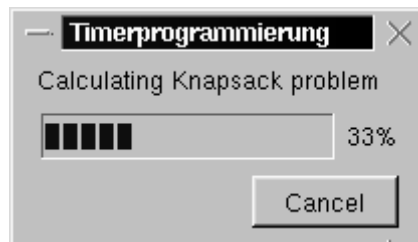


Abbildung 4-54 `QProgressDialog`

QProgressDialog ist von der Klasse *QSemimodal* abgeleitet, mit der man ein modales Dialogfenster erzeugen kann und bei der gleichzeitig der Kontrollfluss in der Berechnung verbleibt. In der Berechnung sollten Sie regelmäßig die Methode *setProgress* aufrufen. Sie aktualisiert den Fortschrittsbalken und ruft *QApplication::processEvents* auf. Ebenso sollten Sie mit der Methode *wasCanceled* regelmäßig prüfen, ob die Berechnung vom Anwender abgebrochen wurde. Für unsere Berechnung kann das Programm folgendermaßen aussehen:

```
double loesung (double a[30])
{
    ...
    int counter = 0, progress_steps = 0;
    QProgressDialog *prog = new QProgressDialog (
        i18n ("Calculation Knapsack problem"),
        i18n ("Cancel"),
        1048576, // insgesamt 1048576 Schritte
        0,      // kein Vater-Widget
        0,      // kein Name
        true);  // modaler Dialog
    prog->setProgress (0);
    while (1) {
        ...
        // bei jedem 1024. Mal Fortschrittsbalken
        // aktualisieren
        if (++counter >= 1024) {
            prog->setProgress (++progress_steps)
            counter = 0;
            if (prog->wasCanceled())
                return -1.0;    // Abbruch durch Anwender
        }
    }
    prog->setProgress (1048576);
    delete prog;
}
```

Es gibt noch einen prinzipiell anderen Ansatz, während einer Berechnung Events zu verarbeiten. Bei diesem Ansatz bleibt die Haupt-Event-Schleife weiterhin aktiv, während die Berechnung in kleine Stücke zerteilt wird, die über einen Timer in regelmäßigen Abständen von der Haupt-Event-Schleife aufgerufen werden. Dieser Ansatz ist meist schwieriger zu realisieren, da die Algorithmen der Berechnung so umgestellt werden müssen, dass bei jedem Aufruf der Berechnungsmethode nur ein Teil abgearbeitet wird. Der Vorteil dieses Ansatzes liegt allerdings auch auf der Hand: Die Berechnung tritt nun völlig in den Hintergrund. Die Applikation selbst bleibt davon völlig unberührt. Es ist kein Problem mehr, die Applikation durch Aufruf der Slot-Methode *QApplication::quit* zu beenden. Auch können mehrere Berechnungen gestartet werden, die sich die vorhandene CPU-Zeit gleichmäßig teilen.

Am besten implementiert man diesen Ansatz, indem man eine neue Klasse erzeugt, die man von der Klasse *QObject* ableitet. Die Erzeugung einer Instanz nimmt im Konstruktor der Klasse die nötigen Initialisierungen vor und startet einen regelmäßigen Timer mit einem *QTimer*-Objekt (siehe auch Kapitel 4.12.3, *Die Klasse QTimer*). Als Zeitintervall für den Timer kann man jeden beliebigen Wert benutzen. Je kürzer das Intervall ist, desto häufiger wird eine Teilberechnung ausgeführt und desto schneller ist die Berechnung beendet. Sinnvoll ist zum Beispiel auch ein Timer-Intervall von Null. In diesem Fall wird bei jedem Auslösen des Timers bereits der nächste Timer-Event in die Event-Queue eingetragen. Er wird hinter den anderen Events – zum Beispiel von Maus und Tastatur – eingeordnet, die in der Zwischenzeit eingetroffen sind. Nach der Rückkehr von der Teilberechnung werden dann zunächst die anderen Events abgearbeitet, bevor direkt danach die nächste Teilberechnung ausgeführt wird. Falls also keine anderen Events anliegen, wird die vollständige CPU-Zeit des Prozesses – bis auf einen kleinen Anteil für die Event-Verarbeitung – für die Berechnung genutzt.

Die Klasse, die die Berechnung ausführt, kann ihre Ergebnisse zum Beispiel über Signale nach außen melden. Da die ersten Ergebnisse erst aufgrund eines Timer-Events entstehen können, liegen die ersten Ergebnisse frühestens nach der Aktivierung der Haupt-Event-Schleife vor, so dass zu diesem Zeitpunkt die Signale bereits mit passenden Slot-Methoden verbunden sein können.

Für unser Knapsack-Problem könnte eine Lösung mit einem Timer-Event zum Beispiel folgendermaßen aussehen:

```
class Knapsack : public QObject
{
    Q_OBJECT
public:
    Knapsack (double a [30], QObject *parent = 0,
              const char *name = 0);
    ~Knapsack () {}

protected:
    void timerEvent (QTimerEvent *);

signals:
    void result (double max);

private:
    double values [30];
    double maximum; // aktuelles Maximum
    bool use [30]; // true, wenn eine Zahl benutzt wird
}

Knapsack::Knapsack (double a[30], QObject *parent,
                    const char *name)
```

```

    : QObject (parent, name)
{
    // kopiere Werte vom Array a in lokales Array values,
    // und initialisiere use
    for (int i = 0; i < 30; i++)
    {
        values [i] = a [i];
        use [i] = false;
    }
    maximum = 0.0;

    // Timer starten, Intervall 0 ms
    startTimer (0);
}

void Knapsack::timerEvent (QTimerEvent *)
{
    // eine Teilberechnung ausführen und sofort zurückkehren
    int i;
    double summe = 0.0;
    for (i = 0; i < 30; i++)
        summe += a [i];

    // evtl. neues Maximum setzen
    if (summe <= 10.0 && summe > maximum)
        maximum = summe;

    // nächste Kombination bilden
    for (i = 0; i < 30; i++) {
        use [i] = !use [i];
        if (use [i])
            break;
    }

    // Falls alle Kombinationen getestet,
    // Ergebnis per Signal aussenden und Timer
    // stoppen
    if (i == 30) {
        killTimers ();
        emit result (maximum);
    }
}

```

In unserem Fall ist die Aufteilung der Berechnung in kleine Teilberechnungen sehr einfach, da der aktuelle Zustand der Berechnung vollständig in der Variablen *use* gespeichert wird. Für andere Algorithmen kann es schwieriger sein, die Berechnung aufzuteilen. Oft kann man sich behelfen, indem man zusätzliche

Variablen einführt, die den Zustand der Berechnung speichern. Beim nächsten Aufruf wird dann anhand des Inhalts der Variablen entschieden, was als Nächstes berechnet werden muss.

Die Zeit, die eine Teilberechnung benötigt, darf natürlich nicht zu lang sein, da auch während der Abarbeitung einer Teilberechnung andere Events nicht abgearbeitet werden können. Eine Teilberechnung sollte daher auch auf langsamen Systemen möglichst maximal 500 ms beanspruchen, so dass die Applikation auch weiterhin ohne störende Verzögerungen bedienbar bleibt. Es ist nicht immer ganz einfach, diese Zeit vorher abzuschätzen. Daher ist es besser, im Zweifelsfall die Aufteilung in Einzelberechnungen lieber kleiner als nötig zu machen. Dadurch steigt zwar der Anteil des Verwaltungsaufwands an der genutzten CPU-Zeit, die Bedienung der Applikation ist aber in keiner Weise eingeschränkt.

4.13.2 Blockierungsfreie Ein- und Ausgabe

Ein schwierigeres Problem bei der Aufgabe, ein Programm interaktiv zu halten, sind die Aufrufe von Betriebssystemfunktionen. Viele Betriebssystemfunktionen – insbesondere beim Einlesen von Daten aus Pipes oder Sockets – wirken blockierend: Wenn zur Zeit keine Daten vorliegen, die eingelesen werden können, hält das Betriebssystem den Prozess so lange an, bis neue Daten vorliegen. In dieser Zeit wird dem Programm natürlich die Kontrolle entzogen. Es kann dann nicht mehr auf Events reagieren. Besonders schlimm wirken sich dann Fehlersituationen aus, in denen keine weiteren Daten mehr ankommen. Das Programm bleibt dann für immer blockiert und kann nicht mehr auf regulärem Weg beendet werden. Es ist daher für Programme mit grafischer Benutzeroberfläche besonders wichtig, eine solche Blockierung durch das Betriebssystem zu vermeiden.

Das Unix-Betriebssystem (und auch Linux) bietet die Möglichkeit, mit der C-Funktion *fnctl* einen Dateideskriptor in den Modus *nicht blockierend* zu versetzen. Ein lesender Funktionsaufruf auf diese Datei kehrt dann sofort zum Aufrufer zurück, wenn keine Daten vorhanden sind. Man könnte nun zum Beispiel mit einem Timer regelmäßig einen Lesevorgang »auf Verdacht« vornehmen, um zu testen, ob inzwischen neue Daten angekommen sind. Diese Methode verbraucht aber unnötig CPU-Zeit.

Qt bietet auch hier ein viel besseres Konzept mit der Klasse *QSocketNotifier* an. Diese Klasse ist zwar speziell für Sockets gedacht, kann aber – zumindest unter Linux, aber auch für die meisten anderen Unix-Systeme – ebenso auf Pipes und reguläre Dateien sowie die speziellen Dateideskriptoren *stdin*, *stdout* und *stderr* angewandt werden. Ein Objekt der Klasse *QSocketNotifier* kontrolliert ein Socket. Sockets können sowohl Daten empfangen als auch verschicken. Man kann im Konstruktor festlegen, ob das *QSocketNotifier*-Objekt darauf achten soll, ob neue Daten am Socket eingetroffen sind oder ob der Socket wieder bereit ist, neue Daten zu verschicken. Da Sockets gepuffert sind, sind sie fast immer bereit, Daten zu verschicken. Kritischer ist meist das Warten auf neue Daten von außen.

Wir wollen an dieser Stelle nicht genauer auf das Erzeugen von Sockets und das Verschicken von Daten über eine Socket-Verbindung eingehen. Wollen Sie ein bereits existierendes Programm, das Sockets nutzt, mit einer Qt-Oberfläche versehen, so können Sie *QSocketNotifier* benutzen. Nähere Informationen zu Sockets finden Sie zum Beispiel in dem Buch *Anwendungen entwickeln unter Linux* von Michael K. Johnson und Erik W. Troan aus dem Addison-Wesley Verlag (ISBN 3-8273-1449-6). Wollen Sie dagegen eigene Sockets aufbauen und nutzen, sollten Sie in Erwägung ziehen, die Socketklassen *QSocket*, *QSocketDevice* und *QServerSocket* von Qt zu nutzen. Sie machen den Umgang mit Sockets noch viel einfacher. Intern nutzen diese Klassen *QSocketNotifier*, um ein Blockieren des Programms in jedem Fall zu vermeiden. Auch die KDE-Bibliothek enthält eigene Socket-Klassen namens *KSocket* und *KServerSocket*. Detaillierte Informationen über diese Klassen mit einfachen Beispielen finden Sie in Kapitel 4.19.1, Socket-Verbindungen.

Gehen wir also im weiteren Verlauf dieses Abschnitts davon aus, dass ein Socket bereits erzeugt wurde. Der Deskriptor des Sockets liegt dabei als *int*-Zahl vor. Die Variable, die diese Zahl speichert, heißt in unseren Beispielen meist *sock*. (Statt eines Socket-Deskriptors kann man auch Dateideskriptoren oder Pipe-Deskriptoren verwenden. Zumindest unter Linux kann man diese Deskriptoren auch mit einem *QSocketNotifier*-Objekt kontrollieren lassen. Die Zahlen 0, 1 und 2 stehen zum Beispiel für die Dateideskriptoren von *stdin*, *stdout* und *stderr*.)

Intern werden *QSocketNotifier*-Objekte in einer Liste gespeichert und in der Haupt-Event-Schleife abgefragt. In der Haupt-Event-Schleife wird mit der Betriebssystemfunktion *select* so lange gewartet, bis einer der Sockets neue Daten zum Lesen bzw. Schreiben besitzt oder der nächste Timer abgelaufen ist. Die *select*-Funktion entzieht der Applikation so lange die CPU, bis eine der Bedingungen eingetreten ist. (Ist bereits beim Aufruf von *select* eine der Bedingungen erfüllt, kehrt die Kontrolle sofort an die Applikation zurück.) Da auch die Kommunikation mit dem X-Server über eine Socket-Verbindung stattfindet, werden die anderen Sockets, die per *QSocketNotifier* kontrolliert werden, gleichberechtigt mit dem Socket zum X-Server behandelt.

Um nun eine Socket-Verbindung mit einem *QSocketNotifier*-Objekt kontrollieren zu lassen, müssen wir zunächst den Socket erzeugen. Anschließend erzeugen wir ein *QSocketNotifier*-Objekt, das im Konstruktor den Deskriptor des Sockets erhält, sowie einen Parameter, mit dem man festlegt, ob der Socket auf eintreffende Daten oder auf die Möglichkeit des Versendens von Daten untersucht werden soll. (Es gibt auch einen Modus, mit dem man den Socket auf Ausnahmefehler hin untersuchen kann. Diese Ausnahmefehler sind aber nur für ganz spezielle Anwendungen gedacht. Ein Abbruch der Socket-Verbindung wird nicht über solche Ausnahmefehler mitgeteilt. Wir gehen später darauf ein, wie man so einen Verbindungsabbruch erkennen kann.) Da *QSocketNotifier* von *QObject* abgeleitet ist, kann man im Konstruktor weiterhin ein Vaterobjekt im Parameter *parent* und einen Namen im Parameter *name* festlegen.

Der häufigste Fall ist, dass man auf eintreffende Daten warten will. Ein einfaches Lesen vom Socket würde das Programm blockieren, wenn keine Daten vorhanden sind. Mit einem *QSocketNotifier*-Objekt können wir uns informieren lassen, wenn neue Daten auf dem Socket eintreffen. Wenn wir annehmen, dass der Socket-Deskriptor in der *int*-Variablen *sock* gespeichert ist, kann das Listing dazu zum Beispiel so aussehen:

```
QSocketNotifier *notify =  
    new QSocketNotifier (sock, QSocketNotifier::Read);  
connect (notify, SIGNAL (activated (int)),  
        this, SLOT (readSocketData (int)));
```

Nachdem das *QSocketNotifier*-Objekt erzeugt worden ist, wird sein Signal *activated* mit dem Slot *readSocketData* verbunden. Dieses Signal wird immer aktiviert, wenn die Testbedingung des *QSocketNotifier*-Objekts erfüllt wird. Sobald also neue Daten eintreffen, wird der Slot *readSocketData* aufgerufen. Dieser Slot wird nun von Ihnen selbst geschrieben. Er kann die Daten des Sockets auslesen und verarbeiten. Als Parameter hat das Signal *activated* einen *int*-Wert, in dem nochmals die Deskriptornummer des Sockets gespeichert wird. So kann ein Slot zum Beispiel sehr einfach die Daten von verschiedenen Sockets lesen. Wenn es nur einen Socket geben kann, der den Aufruf von *readSocketData* bewirkt haben kann, kann man diesen Parameter auch weglassen.

Beachten Sie, dass Sie im Slot *readSocketData* auch wirklich Daten vom Socket auslesen. Nach der Rückkehr in die Haupt-Event-Schleife wird sofort wieder getestet, ob Daten an diesem Socket anliegen. Wenn Sie also in *readSocketData* keine Daten vom Socket lesen, wird dieser Slot ständig aufgerufen.

Sie können natürlich das Signal *activated* vom *QSocketNotifier*-Objekt mit mehreren Slots verbinden. Da neue Daten aber nur einmal vom Socket gelesen werden können, macht das meist nicht viel Sinn.

Wird eine Socket-Verbindung unterbrochen, so wird ebenfalls das Signal *activated* aufgerufen (aber nur im Modus *QSocketNotifier::Read*). Ein Versuch, Daten vom Socket zu lesen, liefert allerdings 0 Byte zurück. Daran erkennen Sie einen Verbindungsabbruch. Sie sollten also in der Methode *readSocketData* diesen Sonderfall abfangen und eine entsprechende Reaktion einleiten. Wichtig ist es dabei auch, dass Sie das *QSocketNotifier*-Objekt löschen oder deaktivieren, da es sonst permanent das Signal *activated* aufruft. Die Slot-Methode *readSocketData* könnte zum Beispiel so aussehen:

```
void myObject::readSocketData (int sock)  
{  
    unsigned char daten;  
    int laenge;
```

```

// Lies ein Byte vom Socket in die Variable daten
laenge = read (sock, &daten, 1);

if (laenge <= 0)    // Verbindungsabbruch
{
    debug ("Socketverbindung unterbrochen!");
    delete notify; // Notifier-Objekt löschen!
    return;
}

// Daten verarbeiten
...
}

```

Das Schreiben in einen Socket geschieht in der Regel vom Betriebssystem gepuffert, d.h. die zu schreibenden Daten werden zunächst in einem Speicher zwischengelagert, bis die andere Seite der Socket-Verbindung die Daten liest. Allerdings ist auch dieser Speicher begrenzt. Daher sollte man auch beim Schreiben in einen Socket die Möglichkeiten von *QSocketNotifier* benutzen, um ein Blockieren der Applikation zu verhindern. Dazu schreibt man die Daten zunächst in einen eigenen Zwischenspeicher (zum Beispiel in ein Objekt der Klasse *QByteArray* oder *QBuffer*) und aktiviert ein *QSocketNotifier*-Objekt, das den Socket auf Beschreibbarkeit hin untersucht. Das Signal *activated* wird mit einem Slot verbunden, der Daten in den Socket schreibt. Konnten alle Daten geschrieben werden, wird das *QSocketNotifier*-Objekt wieder deaktiviert. Ansonsten bleibt es aktiv, um bei der nächsten Gelegenheit weitere Daten schreiben zu können. Damit Sie hierbei nicht jedes Mal ein neues *QSocketNotifier*-Objekt erzeugen müssen, um es anschließend wieder zu löschen, können Sie die Methode *QSocketNotifier::setEnabled (bool enable)* benutzen. Wenn Sie diese Methode mit dem Wert *false* aufrufen, wird das Objekt deaktiviert und ruft nicht mehr das Signal *activated* auf. Mit dem Wert *true* wird es wieder aktiviert.

Ein Beispiel für den Einsatz von *QSocketNotifier* beim Schreiben in einen Socket wird im folgenden Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QData-Pump*, anhand der selbst entwickelten Klasse *SocketSink* ausführlich besprochen.

Ein *QSocketNotifier*-Objekt kann einen Socket nur auf eintreffende Daten oder auf die Möglichkeit des Schreibens von Daten hin untersuchen. Wenn Sie beide Bedingungen testen wollen, müssen Sie zwei *QSocketNotifier*-Objekte benutzen. Wenn Sie in ein Socket nur schreiben, aber zusätzlich feststellen wollen, ob die Verbindung unterbrochen wurde, benötigen Sie ebenfalls zwei *QSocketNotifier*-Objekte: eines im Modus *QSocketNotifier::Read* und eines im Modus *QSocketNotifier::Write*.

4.13.3 Datenübertragung im Hintergrund mit *QDataPump*

Das Schreiben in eine Datei oder das Lesen aus einer Datei ist eine langsame Operation. Wenn große Datenmengen gelesen oder geschrieben werden müssen, ist für diese Zeit die grafische Benutzeroberfläche der Applikation blockiert. Mit Hilfe der in den letzten beiden Kapiteln, Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*, und Kapitel 4.13.2, *Blockierungsfreie Ein- und Ausgabe*, beschriebenen Konzepte kann man das Einlesen bzw. das Schreiben von Daten in kleine Teilblöcke aufspalten. In der Qt-Bibliothek befinden sich aber bereits einige abstrakte Klassen, die diese Übertragung von Daten im Hintergrund vornehmen. Diese Klassen sind nicht auf die Übertragung von Daten aus oder in eine Datei beschränkt, können aber sehr einfach auf diese angewendet werden.

Die drei Klassen, die die Datenübertragung in kleinen Blöcken im Hintergrund vornehmen, sind *QDataSource*, *QDataSink* und *QDataPump*.

QDataSource ist dabei eine abstrakte Klasse, die als Quelle von Daten dient. Das kann zum Beispiel eine Datei sein, aus der man die Daten in kleinen Blöcken aufliest. Die Daten können aber auch selbst erzeugt werden, indem man eine eigene Klasse von *QDataSource* ableitet. So könnte man zum Beispiel eine Klasse entwerfen, die eine Folge von Zahlen als Ergebnis einer Berechnung ausgibt (eine Folge von Primzahlen, Messwerten, Funktionswerten oder Ähnlichem). Von *QDataSource* gibt es die abgeleitete konkrete Klasse *QIODeviceSource*, mit der man als Quelle ein Objekt der Klasse *QIODevice* angeben kann, also zum Beispiel eine Datei.

QDataSink ist eine abstrakte Klasse, die Daten in Blöcken entgegennehmen kann und diese verarbeitet. Es kann sich hierbei zum Beispiel um eine Datei handeln, in die man Daten schreibt. Es kann aber zum Beispiel auch ein selbst geschriebenes Widget sein, das die Daten auf dem Bildschirm darstellt. So arbeitet zum Beispiel auch die Klasse *QMovie*, die ein bewegtes Bild aus einer Datei einliest. Dieses Einlesen geschieht dabei im Hintergrund, damit die Applikation nicht blockiert wird, wenn die Datei sehr groß ist. Von *QDataSink* gibt es noch keine abgeleitete konkrete Klasse. Hier können Sie also eine eigene Klasse entwerfen, die die Daten nach Ihren Wünschen verarbeitet.

QDataPump ist eine Klasse, mit der man ein *QDataSource*-Objekt mit einem *QDataSink*-Objekt verbinden kann. Sobald die Verbindung hergestellt wurde, wird automatisch mit der Datenübertragung in kleinen Blöcken begonnen. Diese Übertragung geschieht mit der Hilfe von Timer-Events mit einer Intervalllänge von 0 Millisekunden, so wie es in Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*, beschrieben wurde. Das *QDataPump*-Objekt fragt dabei regelmäßig bei dem *QDataSource*-Objekt nach, wie viele Daten es zur Zeit senden kann, und beim *QDataSink*-Objekt, wie viele Daten es empfangen kann. Anschließend überträgt es einen Block mit dem Minimum der beiden Werte.

Damit die Datenübertragung nicht zu lange dauert, sollten *QDataSource* und *QDataSink* Werte zurückliefern, die innerhalb von 250 ms geliefert bzw. verarbeitet werden können. So kann gewährleistet werden, dass spätestens nach 500 ms die Events wieder verarbeitet werden. Die von *QDataSource* abgeleitete Klasse *QIODeviceSource* benutzt standardmäßig Datenblöcke bis zu einer Größe von 4096 Byte. Ein solcher Block kann von der Festplatte problemlos in kurzer Zeit geliefert werden.

In zwei Beispielen wollen wir uns Anwendungen für dieses Konzept der Datenübertragung anschauen. Im ersten Beispiel wollen wir die Leerzeichen in einer Datei bestimmen. Damit auch bei extrem großen Dateien das Einlesen nicht die Applikation blockiert, lesen wir nur kleine Blöcke und analysieren sie. Am Ende der Datei senden wir ein Signal, das das Ergebnis zurückgibt und gleichzeitig als Zeichen dient, dass die Analyse beendet ist. Als Datenquelle benutzen wir ein *QIODeviceSource*-Objekt, als Datenziel benutzen wir ein Objekt einer selbst geschriebenen Klasse *SpaceCounter*, die von *QDataSink* abgeleitet wird. Wir überladen dazu die virtuellen Methoden *eof*, *readyToReceive* und *receive*. *eof* wird vom *QDataPump*-Objekt aufgerufen, sobald die Datenquelle keine Daten mehr liefert. In unserem Fall wird dann das Ergebnis mit dem Signal *result* gemeldet. *readyToReceive* soll die maximale Größe des Datenblocks liefern, der in einem Schritt verarbeitet werden soll. Wir benutzen hier einfach eine Konstante von 10.000, da unsere Klasse einen solchen Block von 10.000 Byte sehr leicht innerhalb von 250 ms abarbeiten kann. Die Methode *receive* erhält den Datenblock, der übertragen wird (als Zeiger auf *unsigned char* sowie die Länge des Blocks als *int*-Parameter), und zählt die darin vorhandenen Leerzeichen.

```
class SpaceCounter : public QDataSink, public QObject
{
    Q_OBJECT
public:
    SpaceCounter (QObject *parent=0, const char *name=0);
    ~SpaceCounter () {}

    int readyToReceive () {return 10000;}
    void eof ();
    void receive (const uchar *data, int length);

signals:
    void result (int spaces);
private:
    int counter;
}

SpaceCounter::SpaceCounter (QObject *parent,
                             const char *name)
    : QObject (parent, name)
```



```

{
    counter = 0;
}

void SpaceCounter::eof ()
{
    emit result (counter);
}

void SpaceCounter::receive (const uchar *data, int length)
{
    for (int i = 0; i < length; i++)
        if (data [i] == ' ')
            counter++;
}

```

Angewendet wird diese Klasse beispielsweise so:

```

QFile *f1;
SpaceCounter *counter;
QDataPump *pump;
f1 = new QFile ("datei.txt");
if (!f1->open())
    debug ("Could not open the file!");
else
{
    counter = new SpaceCounter();
    connect (counter, SIGNAL (result (int)),
            this, SLOT (counterReady (int)));
    pump = new QDataPump (file, counter);
}

```

In der Slot-Methode *counterReady* kann nun das Ergebnis der Zählung verarbeitet werden. Außerdem können hier die Objekte *f1*, *counter* und *pump* mit *delete* gelöscht werden.

Im zweiten Anwendungsbeispiel wollen wir eine von unserem Programm berechnete Zeichenfolge über eine Socket-Verbindung schicken. Die Zeichenfolge ist in unserem Fall insgesamt 1 MByte groß und besteht nur aus dem Buchstaben A. Sie wird von der Klasse *CharacterSource* erzeugt, die wir von *QDataSource* ableiten. Ein Block dieser Daten wird direkt auf Anfrage erzeugt. Das Verschicken eines Datenblocks über eine Socket-Verbindung geschieht ebenfalls blockweise mit der Klasse *SocketSink*, die wir von *QDataSink* ableiten. Da jedoch im Voraus nicht bestimmt werden kann, wie viele Daten noch in den Socket geschrieben werden können, wird ein Block zunächst zwischengespeichert. Erst wenn im Zwischenspeicher wieder Platz ist, kann ein neuer Block – maximal in der Größe des freien Platzes – entgegengenommen werden. Die Klasse *SocketSink*, die wir in diesem

Beispiel entwickeln, können Sie auch für eigene Programme benutzen. Hier sehen Sie zunächst die Klasse *CharacterSource*, die die Zeichenfolge erzeugt:

```
class CharacterSource : public QDataSource
{
public:
    CharacterSource () {rest = 1048576;} // 1 MByte
    ~CharacterSource () {}

    int readyToSend ();
    void rewind () {rest = 1048576;} // von vorn beginnen
    bool rewindable () {return true;}
    void sendTo (QDataSink *sink, int count);

private:
    int rest; // Anzahl der noch zu versendenden Bytes
}

int CharacterSource::readyToSend ()
{
    if (rest > 0)
        return rest;
    else
        return -1;
}

void CharacterSource::CharacterSource
(QDataSink *sink, int count)
{
    uchar *mem = new uchar [count]; // Block anlegen
    for (int i = 0; i < count; i++)
        mem [i] = 'A'; // mit Daten füllen
    rest -= count; // versendete Bytes
    sink->receive (mem, count); // versenden
    delete mem []; // Block löschen
}
```

Wir müssen die Methoden *readyToSend* und *sendTo* der Klasse *QDataSource* überschreiben. *readyToSend* muss dabei die Größe des zur Zeit versendbaren Blocks in Bytes zurückliefern. In unserem Fall kann dies maximal die Restlänge sein, die noch verschickt werden muss. Ist nichts mehr zu versenden, kann man den Wert -1 zurückliefern. So unterscheidet man zwischen dem Fall, dass nur im Moment nichts zu versenden ist (Wert 0) und dem Abschluss der Aktion. Liefert man -1 zurück, ruft das *QDataPump*-Objekt die Methode *eof* vom *QDataSink*-Objekt auf. Die Methode *sendTo* übernimmt das Verschicken der Daten. Dabei ruft man einfach die Methode *receive* des *QDataSink*-Objekts auf. Anschließend kann der Datenblock wieder gelöscht werden.

Wir haben auch die Methoden *rewindable* und *rewind* überladen. Mit diesen Methoden kann ein Programmierer testen, ob ein *QDataSource*-Objekt wieder an den Anfang gesetzt werden kann. Will ein Programmierer diese Möglichkeit nutzen, so sollte er zunächst testen, ob *rewindable* *true* zurückliefert. Dann kann er die Methode *enableRewind* mit dem Parameter *true* aufrufen. Diese Methode kann überladen werden, wenn spezielle Initialisierungen nötig sind, um die oben genannte Möglichkeit zu realisieren. Mit dem Aufruf von *rewind* kann der Programmierer dann das Zurücksetzen ausführen lassen. In unserem Fall wollen wir das Zurücksetzen ermöglichen und geben daher bei *rewindable* immer *true* zurück. *enableRewind* braucht nicht überladen zu werden, da keine speziellen Initialisierungen nötig sind. *rewind* legt in unserem Fall die Restlänge einfach wieder auf 1.048.576 Byte (1 MByte) fest.

Als Nächstes implementieren wir die Klasse *SocketSink*, die Datenblöcke entgegennimmt und sie in das Socket schreibt. Diese Klasse ist sehr allgemein gehalten. Sie können sie auch für eigene Projekte benutzen.

```
class SocketSink : public QDataSource, public QObject
{
    Q_OBJECT

public:
    SocketSink (int sock);
    ~SocketSink () {}

    void eof ();
    int readyToReceive () {return 1024 - length;}
    void receive (const uchar *data, int count);

signals:
    void done ();

private slots:
    void socketReady (int sock);

private:
    uchar buffer [1024]; // Zwischenspeicher
    int length;           // aktuell enthaltene Datenmenge
    bool ready;           // true, falls Übertragung fertig
    QSocketNotifier notify;
}

SocketSink::SocketSink (int sock) :
    notify (sock, QSocketNotifier::Write)
{
    length = 0; // Zwischenspeicher anfangs leer
    connect (notify, SIGNAL (activated (int)),
```

```
        this, SLOT (socketReady ());
    notify.setEnabled (false); // noch nichts zu senden
    ready = false;
}

void SocketSink::receive (const uchar *data, int count)
{
    // Zuerst Daten in den Zwischenspeicher kopieren
    for (int i = 0; i < count; i++)
        buffer [length + i] = data [i];
    length += count;

    // Socket auf Beschreibbarkeit untersuchen
    if (length > 0)
        notify.setEnabled (true);
}

void SocketSink::eof ()
{
    if (length == 0)
        emit done (); // Alle Daten gesendet, dann fertig
    else
        ready = true; // sonst merken
}

void SocketSink::socketReady (int sock)
{
    int sendCount;
    // Versuch, length Byte zu schreiben
    sendCount = write (sock, &buffer, length);
    if (sendCount >= 0)
    {
        // gesendete Zeichen aus dem Zwischenspeicher
        // entfernen
        for (int i = 0; i < length - sendCount; i++)
            buffer [i] = buffer [sendCount + i];
        length -= sendCount;

        // Falls Zwischenspeicher jetzt leer, Socket
        // deaktivieren.
        // Falls keine weiteren Daten, fertig.
        if (length == 0) {
            notify.setEnabled (false);
            if (ready)
                emit done ();
        }
    }
}
```

Die Verbindung der beiden Klassen *CharacterSource* und *SocketSink* geschieht wieder über ein *QDataPump*-Objekt:

```
CharacterSource *source;
SocketSink *sink;
QDataPump *pump;

source = new CharacterSource ();
sink = new SocketSink (socketNumber);
connect (sink, SIGNAL (done ()), this, SLOT (ready ()));
pump = new QDataPump (source, sink);
```

Nachdem die Daten übertragen worden sind, wird das Signal *done* aktiviert. Im Slot *ready* können nun die drei Objekte *source*, *sink* und *pump* mit *delete* wieder gelöscht werden.

4.13.4 Mehrere Prozesse und Threads

Eine weitere Möglichkeit, um Berechnungen im Hintergrund ablaufen zu lassen, ist der Einsatz von weiteren Prozessen oder Threads. Worauf Sie dabei beim Einsatz mit Qt und KDE achten müssen, beschreiben wir in diesem Unterkapitel.

Mehrere Prozesse

Prozesse sind unabhängige Programmstücke, wobei jeder Prozess einen eigenen Datenbereich besitzt. Das Betriebssystem regelt die Vergabe der Rechenzeit. Auf einem Ein-Prozessor-System wird jeder Prozess für eine kurze Zeit ausgeführt und danach unterbrochen. Anschließend wird der nächste Prozess gestartet.

Um einen weiteren Prozess zu starten, der unabhängig vom Hauptprozess der Applikation eine Berechnung durchführt, wird die Systemfunktion *fork* benutzt. Sie verdoppelt den laufenden Prozess. Beide nun vorhandenen Prozesse besitzen die gleichen Variableninhalte und werden an der Stelle nach der Ausführung der *fork*-Funktion fortgesetzt. Der Rückgabewert der *fork*-Funktion dient zur Unterscheidung, ob es sich um den Originalprozess oder den duplizierten Prozess handelt. Der Rückgabewert ist im Originalprozess die Prozessnummer des Duplikats, im Duplikatprozess ist der Rückgabewert 0. Eine einfache Anwendung kann zum Beispiel so strukturiert sein:

```
if (fork() == 0)
{
    // Kindprozess
    // Hier kann eine Berechnung stattfinden,
    // ohne dass der Vaterprozess gestört wird.
    ...
    exit (0); // Kindprozess wird beendet
}
```

```
// Vaterprozess  
// unabhängig vom Kindprozess  
...
```

Da die beiden Prozesse unabhängig voneinander sind, müssen die Ergebnisse der Berechnung im Kindprozess über einen umständlicheren Weg zum Vaterprozess gebracht werden, zum Beispiel über eine Pipe, eine Socket-Verbindung oder eine temporäre Datei. (Um eine Pipe zu erzeugen, rufen Sie vor der *fork*-Funktion die Funktion *pipe* auf. Der Kindprozess schreibt dann in der Regel in die Pipe, der Vaterprozess liest die Daten aus. Auch für eine Pipe können Sie die Klasse *QSocketNotifier* benutzen. Um eine Socket-Verbindung zwischen Vater- und Kindprozess zu erzeugen, können Sie die Systemfunktion *socketpair* benutzen.)

Problematisch bei der Verdopplung eines Prozesses ist die Socket-Verbindung zum X-Server. Diese Verbindung wird durch *fork* nicht verdoppelt. Beide Prozesse können nun also Zeichenbefehle zum X-Server absetzen und die Event-Queue des X-Servers auslesen. Das führt zwangsläufig zu Problemen, da im voraus nicht bestimmt werden kann, welcher Prozess welche Events verarbeiten soll. Ein weiteres Problem sind die Widget-Objekte: Obwohl ein Widget-Objekt durch *fork* verdoppelt wird, beziehen sich beide Widget-Objekte auf das gleiche Fenster auf dem Bildschirm. Ändert ein Prozess zum Beispiel die Fenstergröße, stimmt die Größe auf dem Bildschirm nun nicht mehr mit der internen Größe des anderen Prozesses überein. Die einzige sinnvolle Lösung ist, dass nur ein Prozess (in der Regel der Vaterprozess) im weiteren Verlauf auf den X-Server zugreift. Das bedeutet für den anderen Prozess, dass er keine Zeichenbefehle einsetzen darf, keine neuen Widgets erzeugen darf, auf die vorhandenen Widgets nicht zurückgreifen darf, nicht in die Haupt-Event-Schleife zurückkehren darf und auch die Methoden *QApplication::processOneEvent* und *QApplication::processEvents* nicht aufrufen darf. Der Kindprozess kann daher auch keine Qt-Timer oder die Klasse *QSocketNotifier* einsetzen.

Um die Verbindung zum X-Server unmittelbar vor dem *fork*-Befehl zu synchronisieren, sollten Sie die statische Methode *QApplication::XFlush* oder *QApplication::XSync* aufrufen.

Wichtig beim Einsatz von zusätzlichen Prozessen für Berechnungen ist also, strikt zwischen dem Hauptprozess, der die Darstellung auf dem Bildschirm übernimmt, und den berechnenden Nebenprozessen zu trennen, die sich aus der Darstellung vollständig heraushalten und ihre Ergebnisse an den Hauptprozess liefern. Die Kindprozesse sollten in jedem Fall mit *exit* beendet werden. Wenn Sie sie beenden würden, indem Sie die *main*-Funktion bis zum Ende ausführen ließen, würde das *QApplication*-Objekt und damit die Verbindung zum X-Server gelöscht.

Die Struktur eines KDE-Programms, das einen Kindprozess erzeugt, dort eine Berechnung ausführt und deren Ergebnis in einer unbenannten Pipe überträgt, kann beispielsweise so aussehen:

```
int pipefds [2]; // Dateideskriptoren der Pipes

// unbenannte Pipe erzeugen
pipe (pipefds);

// Verbindung zum X-Server synchronisieren
QApplication::XFlush ();

// Kindprozess erzeugen
if (fork () == 0)
{
    // Berechnung durchführen
    // Zugriff auf X-Server, Widget, Timer,
    // QSocketNotifier-Objekte oder Haupt-Event-Schleife
    // hier nicht erlaubt
    ...
    // Ergebnis der Berechnung in die Pipe schreiben
    write (pipefds [1], daten, laenge);

    // Kindprozess beenden
    exit (0);
}

// Vaterprozess
notify = new QSocketNotifier (pipefds [0],
                             QSocketNotifier::Read);
connect (notify, SIGNAL (activated (int)),
         this, SLOT (childResults (int)));

// ganz normal in die Event-Schleife zurückkehren
...
```

Wenn der Kindprozess seine Ergebnisse in die Pipe schreibt, sendet das *QSocketNotifier*-Objekt *notify* das Signal *activated* aus. Im angeschlossenen Slot *childResult* können nun die Ergebnisse der Pipe ausgelesen werden. Während der Berechnung im Kindprozess läuft der Vaterprozess ganz normal weiter und gewährleistet, dass die Benutzeroberfläche bedienbar bleibt.

Ausführen anderer Programme

KDE besitzt eine eigene Klasse, *KProcess*, die die Ausführung von anderen Programmen parallel zur eigenen Applikation erlaubt. Die Vorgehensweise ist dabei die für Unix-Systeme übliche Art, andere Programme parallel zu starten: Zuerst wird mit *fork* ein neuer Prozess erzeugt. Dieser Prozess wird dann durch den Auf-

ruf der Systemfunktion *execv* durch den Code eines anderen Programms aus einer ausführbaren Datei ersetzt. *KProcess* achtet dabei darauf, dass vor dem Aufruf von *fork* die Kommunikation mit dem X-Server synchronisiert ist. Außerdem bietet *KProcess* eine elegante Art, dem aufgerufenen Programm Kommandozeilenparameter mitzugeben. Das folgende Beispiel ruft den Befehl `ls -l -a` auf und liest die Ausgabe ein:

```
KProcess *proc = new KProcess ();
(*proc) << "ls" << "-l" << "-a"; // Befehl und Parameter
connect (proc, SIGNAL (processExited (KProcess*)),
        this, SLOT (programReady (KProcess*)));
connect (proc,
        SIGNAL (receivedStdout (KProcess*, char*, int)),
        this,
        SLOT (programOutput (KProcess*, char*, int)));
proc->start (KProcess::NotifyOnExit, KProcess::Stdout);
```

Mit dem `<<`-Operator werden nacheinander der Befehlsname und die einzelnen Kommandozeilenparameter angegeben. In unserem Fall werden weiterhin die beiden Signale *processExited* und *receivedStdout* mit Slots verbunden. Das erste Signal, *processExited*, wird aktiviert, sobald das Programm beendet ist. Der Exit-Status des Programms lässt sich mit der Methode *exitStatus* erfragen. Das zweite Signal, *receivedStdout* wird aktiviert, wenn das Programm eine Ausgabe auf den Stream *stdout* ausgibt. Im Signal wird neben dem *KProcess*-Objekt auch ein Zeiger auf die ausgegebenen Daten und die Länge der Daten angegeben. Für die Ausgaben auf *stderr* kann das Signal *receivedStderr* benutzt werden. Mit der Methode *writeStdin* kann man auch Daten auf den *stdin*-Stream des Programms leiten. Mit dem Signal *wroteStdin* kann man sich informieren lassen, wenn die vollständigen Daten, die auf den *stdin*-Stream geschrieben wurden, auch vom Programm gelesen wurden. Mit *closeStdin* kann man ein EOF-Zeichen an das Programm schicken.

Ausgeführt wird das Programm mit der Methode *start*. Diese Methode hat die beiden Parameter *runmode* und *comm*. Im ersten Parameter, *runmode*, kann man auswählen, wie die Verbindung zwischen dem eigenen Programm und dem von *KProcess* ausgeführten Programm ist. Wählt man als Wert *DontCare*, so sind beide Programme unabhängig. *processExited* wird nicht aktiviert, wenn das Programm endet. Der Wert *NotifyOnExit* ist der Default-Wert. Mit diesem Wert laufen beide Prozesse parallel nebeneinander. Allerdings wird der eigene Prozess informiert, wenn der andere Prozess beendet wird. Im Modus *Block* wird das eigene Programm angehalten, bis der gestartete Prozess beendet wird. Dieser Modus ist allerdings ungünstig, da er die Applikation blockiert und so die Events von Maus und Tastatur nicht verarbeitet werden.

Der zweite Parameter, *comm*, gibt an, welche der Standard-Streams des Prozesses vom eigenen Prozess benutzt werden können. Dazu muss man die Konstanten *Stdin*, *Stdout* und *Stderr* oder-verknüpfen.

Beachten Sie, dass *KProcess* bei der Verarbeitung der angegebenen Kommandozeilenparameter keine Shell-Umformungen vornimmt. Daher können nicht zwei Parameter in einem String – durch ein Leerzeichen getrennt – angegeben werden. Die folgende Zeile ruft das Programm *ls* mit nur einem Parameter auf, der ein Leerzeichen enthält:

```
(*proc) << "ls" << "-l -a";
```

Folgende Zeile hat ebenfalls nicht die erwartete Wirkung:

```
(*proc) << "ls" << "-l" << "-a" << "*.cpp";
```

In einer Shell wird der Parameter **.cpp* automatisch durch alle Dateinamen im aktuellen Verzeichnis ersetzt, die auf diese Maske passen. *KProcess* nimmt diese Ersetzung nicht vor. Es wird also der *ls*-Befehl für die Datei **.cpp* ausgeführt, die jedoch wahrscheinlich nicht existiert. Ebenso werden Umgebungsvariablen wie *\$HOME* nicht ersetzt.

Eine andere Klasse zum Starten anderer Programm ist *KRun*. Sie bietet allerdings keine Möglichkeit, die Ein- und Ausgabe des Programms umzuleiten. Eine Beschreibung finden Sie in der Online-Referenz der KDE-Bibliotheken.

Mehrere Threads

Ebenso wie bei der Erzeugung eines neuen Kindprozesses kann man mit dem Thread-Konzept einen parallelen Ausführungsstrang erzeugen, der unabhängig vom ersten Strang abgearbeitet wird. Im Gegensatz zu einem mit *fork* erzeugten Prozess ist ein neuer Thread »schlanker«, da er die Daten des Prozesses nicht kopiert. Threads sind daher schneller und verbrauchen weniger Speicher als Prozesse. Nach der Verdopplung benutzen beide Threads die gleichen Daten. Eine Änderung einer Variable in einem Thread (außer lokaler Variablen auf dem Stack) ändert also auch diese Variable im anderen Thread.

Die Qt-Bibliothek enthält ab der Version 2.2 eine Reihe von Klassen, mit denen Sie plattformunabhängig Threads erzeugen und koordinieren können. Zentrale Klasse ist dabei *QThread*, die einen neuen Thread erzeugt. Dazu müssen Sie eine neue Unterklasse von *QThread* bilden und in dieser die Methode *run* überschreiben. In diese fügen Sie den Code ein, der ausgeführt werden soll. Um einen neuen Thread zu erzeugen, bilden Sie ein Objekt Ihrer Klasse und rufen die Methode *start* auf. Diese führt nun – parallel zum Haupt-Thread – die Berechnung in der Methode *run* aus. Ist diese Methode beendet, wird auch der Kind-Thread beendet. Für die Koordination zwischen Threads enthält die Qt-Bibliothek die Klassen *QMutex*, *QWaitCondition* und *QSemaphore*. Weitere Informationen zu diesen Klassen finden Sie in der Online-Referenz der Qt-Bibliothek.

Auch bei der Erzeugung eines zusätzlichen Threads ist dringend anzuraten, die Aufgaben eindeutig zu teilen: Ein Thread (in der Regel der Haupt-Thread) ist nur für die Anzeige und die Verarbeitung der Events zuständig; der andere Thread führt Berechnungen durch und verzichtet auf Zugriffe auf den X-Server. So vermeiden Sie Inkonsistenzen. Die Übertragung der berechneten Daten kann bei Threads einfach durch eine Variable geschehen, auf die beide Threads zugreifen. Der Kind-Thread schreibt einfach die Ergebnisse in diese Variable und setzt ein Flag, dass die Arbeit beendet ist. Der Vaterprozess fragt in regelmäßigen Abständen das Flag ab und benutzt die Ergebnisse, sobald das Flag gesetzt ist. Das Senden eines Signals von einem Thread zum anderen funktioniert hierbei nicht, da sowohl das Signal als auch alle angeschlossenen Slots im Kindprozess ausgeführt werden. Slots, die dabei auf den X-Server zugreifen – direkt oder indirekt –, können wiederum zu Inkonsistenzen führen. Anstatt ein Flag zu setzen, kann natürlich auch eine Pipe oder ein Socket eingesetzt werden, die bzw. der Vater- und Kind-Thread miteinander verbindet und die bzw. der im Vater-Thread per *QSocketNotifier*-Objekt abgefragt wird.

4.13.5 Übungsaufgabe

Übung 4.12

Schreiben Sie ein Programm *our-less*, das Text von der Standardeingabe *stdin* einliest und ihn in einem Fenster mit Rollbalken darstellt, z.B. in einem *QMultiLineEdit*-Widget. Dieses Programm kann zum Beispiel als grafischer Ersatz des Unix-Programms *less* dienen, indem man es auf der Kommandozeile folgendermaßen einsetzt:

```
% ls -l | our-less
```

Hinweis: Benutzen Sie ein Objekt der Klasse *QSocketNotifier*, um über neue Daten an der Standardeingabe informiert zu werden.

4.14 Audio-Ausgabe

Bisher hatten wir uns in diesem Buch vor allem mit der grafischen Darstellung von Sachverhalten beschäftigt. In der letzten Zeit haben die Fähigkeiten von KDE und Qt im Bereich von Tönen und Klängen deutlich verbessert. Es ist inzwischen möglich, multimediale Programme in KDE zu schreiben. Noch ist die Entwicklung nicht abgeschlossen, und gerade der Multimedia-Bereich von KDE scheint sich noch am stärksten zu wandeln, aber die wichtigsten Weichen sind gestellt. So hat sich als Sound-Server das Programm *aRts* (*analog realtime synthesizer*) durchgesetzt, der über das Protokoll *MCOP* (*Multimedia Communications Protocol*) angesprochen wird. Damit wird unter KDE endlich die bisher recht stiefmütterliche Unterstützung von Audioausgaben unter Linux standardisiert und auf eine leistungsfähige Basis gestellt.

4.14.1 Warntöne aus dem Lautsprecher ausgeben

Um einfache Warntöne zu erzeugen oder die Aufmerksamkeit des Anwenders zu gewinnen, kann die statische Methode *QApplication::beep* benutzt werden. Beim Aufruf der Methode wird ein kurzer Signalton im Lautsprecher des PCs erzeugt. Ein solcher Ton kann zum Beispiel benutzt werden, um eine wichtige Warnmeldung anzukündigen, die umgehend beachtet werden muss. Setzen Sie diesen Warnton aber wirklich nur in wichtigen Fällen ein. Je nach Bauart des PC ist der Warnton unter Umständen unangenehm laut, so dass sich der Anwender schnell gestört fühlen kann.

Der Warnton wird auf dem Terminal ausgegeben, an dem der Anwender sitzt. Bei den meisten Arbeitsplatzrechnern ist es der eigene PC. Läuft das Programm aber über eine Internetverbindung auf einem anderen Rechner, so piept trotzdem der Rechner, an dem der Anwender sitzt.

4.14.2 Klangdateien ausgeben

Wenn Sie differenziertere Audiosignale ausgeben wollen, hilft Ihnen die Methode *beep* nicht weiter. Insbesondere für charakteristische Klänge, die Ihr Programm von anderen unterscheiden, müssen Sie Audio-Dateien über die Soundkarte des Rechners ausgeben.

KAudioPlayer

In den KDE-Bibliotheken ist die Klasse *KAudioPlayer* enthalten, mit der Sie so genannte gesampelte Daten – digitalisierte Daten, die meist über ein Mikrofon aufgezeichnet wurden – über die Soundkarte ausgeben können. Diese Daten müssen in Form einer WAV-Datei vorliegen. WAV-Dateien haben eine große Verbreitung in der Microsoft Windows-Welt, aber auch auf Unix-Systemen finden sie neben den AU-Dateien immer mehr Verbreitung. Je nach Aufzeichnungsqualität der WAV-Datei können die Audiodaten in Mono oder Stereo vorliegen, mit einer Digitalisierungstiefe von 8 oder 16 Bit pro Kanal und einer Abtastfrequenz zwischen 11 kHz und 48 kHz. Alle Formate werden korrekt erkannt und in der Qualität wiedergegeben, die mit der installierten Soundkarte maximal möglich ist. Damit die Soundkarte aber überhaupt angesprochen werden kann, muss sie im Betriebssystem installiert sein. Unter Linux bedeutet das auch heute noch oftmals, dass der Betriebssystem-Kernel neu kompiliert werden muss. Bei einigen Distributionen wird die Soundkarte aber auch automatisch erkannt. Als Test, ob Ihre Soundkarte korrekt installiert ist, aktivieren Sie die KDE-Systemklänge. Im KDE-Kontrollzentrum können Sie bestimmten Ereignissen des KDE Window-Managers *kwin* Klänge zuordnen. Dort können Sie diese Klänge auch testen. Falls Ihre Soundkarte beim Testen nichts ausgibt, ist sie wahrscheinlich nicht korrekt installiert.

Um nun eine WAV-Datei abzuspielen, benutzen Sie einfach die statische Methode *KAudioPlayer::play*, der Sie als Parameter den Namen (evtl. mit Pfad) der Datei übergeben. (Wenn Sie den Pfad weglassen, wird die Datei in den KDE-Standardverzeichnissen gesucht, also in *\$KDEDIR/share/sounds* und *\$HOME/.kde/share/sounds*.) Folgende Zeile reicht also bereits aus:

```
KAudioPlayer::play ("KDE_Startup.wav");
```

Sie haben allerdings nun keine Kontrolle mehr über die abgespielte Sounddatei: Sie können sie nicht mehr anhalten oder die Lautstärke verändern. Die Datei wird sogar noch weitergespielt, wenn Ihr Programm beendet wird. Leider haben Sie auch keine Möglichkeit, von Ihrem Programm aus festzustellen, ob die Soundkarte überhaupt korrekt installiert ist und ob die angegebene Datei gefunden wurde.

Allerdings ist es möglich, mehrere Dateien gleichzeitig abzuspielen – sowohl aus einem einzelnen Programm als auch aus mehreren Programmen gleichzeitig. Sie werden im Rechner gemischt und dann erst ausgegeben. Also auch wenn ein anderes KDE-Programm gerade Musik spielt (z.B. ein MP3-Player), wird Ihre Sound-Datei ausgegeben.

QSound

Auch Qt bietet eine solche einfache Klasse zur Ausgabe von Audio-Dateien namens *QSound* an. Sie benutzt unter Unix das *NAS (Network Audio System)* zur Ausgabe der Datei. Dieses System ist aber nicht auf allen Rechnern installiert. Deshalb sollten Sie diese Klasse nur einsetzen, wenn Sie ein reines Qt-Programm schreiben, also auf *KAudioPlayer* nicht zurückgreifen können. Unter Microsoft Windows wird das Windows-eigene Multimedia-System benutzt. Dort können Sie also sicher sein, dass die Ausgabe funktioniert.

QSound besitzt – genau wie *KAudioPlayer* – die statische Methode *play*, der Sie einfach den Namen der abzuspielenden Audiodatei übergeben. Zusätzlich besitzt *QSound* die statische Methode *available*, mit der Sie testen können, ob die Audioausgabe überhaupt möglich ist.

4.14.3 Beliebige Audioströme ausgeben

Die oben beschriebenen Techniken reichen nur für einfache Warnklänge aus. Multimedia-Anwendungen stellen deutlich höhere Ansprüche an das Audiosystem:

- Es muss möglich sein, kontinuierliche Audioströme ausgeben zu lassen.
- Die Ausgabe von »on the fly« errechneten Daten muss möglich sein – ohne Umweg über eine Datei.

- Die Ausgabe sollte zu jedem Zeitpunkt – auch während der Ausgabe – manipuliert werden können, z.B. in Lautstärke, Qualität oder Geschwindigkeit.
- Die Reaktions- und Latenzzeit des Audio-Systems sollte so kurz wie möglich sein. Wenn also das Programm eine Ausgabe beginnen (oder vorzeitig beenden) will, sollte das Audio-System möglichst direkt reagieren. Der Anwender sollte keine Verzögerung bemerken können.
- Es sollte möglich sein, dass mehrere Programme gleichzeitig Audiodaten ausgeben. Diese sollten vom System gemischt werden. So will der Anwender z.B. während des Abspielens von MP3-Dateien von seinem E-Mail-Programm über hereinkommende Nachrichten informiert werden, ohne dass die Musikausgabe abbricht.

Insbesondere Spiele stellen hohe Anforderungen an das Audio-System. Die Latenzzeit muss so kurz wie möglich sein, damit eine Aktion des Spielers direkt eine Änderung der Klangkulisse nach sich zieht (Explosionen, Schüsse usw.). Das bedeutet aber, dass die Zwischenspeicher für die Audiodaten möglichst klein sein müssen. Das führt aber dazu, dass sie sehr schnell leerlaufen und mit neuen Daten gefüllt werden müssen. Um dann einen Datenunterlauf zu vermeiden (der deutlich hörbare Störgeräusche und Aussetzer produzieren würde), muss die Lieferung neuer Daten gut koordiniert werden.

In KDE 2.0 übernimmt diese Aufgabe der *aRts* (*analog realtime synthesizer*). Er ist modular aufgebaut, d.h. er besteht aus kleinen Modulen, die Spezialaufgaben übernehmen (Lautstärkeregelung, Mischen von mehreren Signalen, Tiefpass- und Hochpassfilter usw.). Diese Module kommunizieren über das speziell hierfür entwickelte Protokoll *MCOP* (*Multimedia Communication Protocol*). Dieses ist ähnlich aufgebaut wie *CORBA* oder *DCOP* (siehe Kapitel 4.20, *Interprozess-Kommunikation mit DCOP*), ist aber hochgradig geschwindigkeitsoptimiert und bietet spezielle Unterstützung für kontinuierliche Datenströme (so genannte *Streams*).

Weitere Informationen zu *aRts*, *MCOP* und der Ausgabe von Audiodaten in KDE finden Sie im Internet auf der Seite <http://www.arts-project.org/>.

4.15 Die Zwischenablage und Drag&Drop

Um Daten zwischen Applikationen auszutauschen, wird in der Regel die Zwischenablage benutzt. In Unix-Systemen mit dem X-Server ist es üblich, dass Text, der mit der linken Maustaste markiert wird, automatisch in die Zwischenablage kopiert wird. Mit der mittleren Maustaste wird der Text der Zwischenablage automatisch in das Widget eingefügt, über dem sich der Mauszeiger zur Zeit befindet. Diese Nutzung der Zwischenablage ist auch bereits in den vordefinierten Qt-Widget-Klassen implementiert.

Die Zwischenablage des X-Servers kann allerdings nicht nur Text enthalten. Alle Daten, die in einem Block im Speicher abgelegt werden können, können in die Zwischenablage kopiert werden. Um den Typ festzulegen, greift Qt auf MIME-Typen zurück: Der Datentyp wird zunächst in grobe Gruppen unterteilt, die die Kategorie der Daten angeben. Benutzte Kategorien sind zum Beispiel *text*, *image*, *url* oder *application*. Der genaue Typ wird durch eine weitere Bezeichnung angegeben, die mit einem Schrägstrich an die Kategorie angehängt wird. So gibt es beispielsweise unter anderem die MIME-Typen *text/plain*, *text/utf8*, *text/utf16* oder *text/rtf* als Untertypen von *text*, *image/bmp*, *image/xpm* oder *image/jpeg* als Untertypen von *image*.

Qt enthält bereits die Möglichkeit, Text und Bilder als MIME-Typ darzustellen und in die Zwischenablage zu kopieren oder aus ihr auszulesen. Wenn Sie andere MIME-Typen oder sogar selbst definierte MIME-Typen benutzen wollen, können Sie eigene Klassen für die Verarbeitung definieren. Die Möglichkeiten der Zwischenablage von Qt werden in Kapitel 4.15.1, *Die Zwischenablage – QClipboard*, beschrieben.

Neben dem Datenaustausch über die Zwischenablage hat sich in der letzten Zeit auch das Drag&Drop-Konzept durchgesetzt. Dabei zieht man ein Element von einer Applikation in eine andere, das dort eingefügt wird. Das Prinzip ist hierbei sehr ähnlich. Auch hier werden die Datentypen, die übertragen werden sollen, in einem Speicherblock abgelegt. Der Typ der Daten wird über einen MIME-Typ festgelegt. Die Zwischenablage wird hierbei aber nicht benutzt. Mehr über das Drag&Drop-Konzept erfahren Sie in Kapiteln 4.15.2, *Drag&Drop*.

4.15.1 Die Zwischenablage – QClipboard

Qt stellt mit Hilfe der Klasse *QClipboard* eine Schnittstelle zur Zwischenablage des X-Servers her. Da es nur eine einzige Zwischenablage gibt, darf auch nur ein einziges *QClipboard*-Objekt in der Applikation existieren. Daher ist der Konstruktor von *QClipboard* als *private* deklariert. Zugriff auf das *QClipboard*-Objekt erhalten Sie über die Methode *QApplication::clipboard()*. Diese Methode liefert einen Zeiger auf das Objekt zurück.

Wenn Sie in Ihrer Applikation einen Text oder ein Bild in die Zwischenablage schreiben wollen, so benutzen Sie die Methoden *QClipboard::setText*, *QClipboard::setImage* oder *QClipboard::setPixmap*. Den aktuellen Inhalt der Zwischenablage können Sie mit den Methoden *QClipboard::text*, *QClipboard::image* und *QClipboard::pixmap* erfragen. Wenn sich allerdings ein anderer Datentyp in der Zwischenablage befindet, als Sie benötigen, oder die Zwischenablage leer ist, bekommen Sie ein Null-Objekt der Klasse *QString*, *QImage* bzw. *QPixmap* zurück. Dieses können Sie im zurückgegebenen Objekt mit der Methode *isNull* erfragen.

Die folgenden beiden Methoden implementieren ein Kopieren von markiertem Text in die Zwischenablage (*copy*) bzw. ein Einfügen von Text aus der Zwischenablage in den Text auf dem Bildschirm (*paste*):

```
void MyWidget::copy () {
    QString text = markedText();
    QApplication::clipboard()->setText (text);
}

void MyWidget::paste () {
    QString text = QApplication::clipboard()->text();
    if (!isNull (text))
        insertText (text);
}
```

Die Zwischenablage wird vollständig über MIME-Typen gesteuert. Auf der Basis der Klasse *QMimeSource* können Sie Daten in die Zwischenablage einfügen oder aus ihr auslesen lassen. Daten werden über die Methode *QClipboard::setData* in die Zwischenablage eingefügt. Übergeben Sie dieser Methode einen Zeiger auf ein Objekt einer Klasse, die von *QMimeSource* abgeleitet ist. Dieses Objekt müssen Sie mit *new* auf dem Heap angelegt haben. Die weitere Verwaltung des Objekts geschieht durch Qt. Sie dürfen das Objekt also nicht selbst wieder löschen. Die Methode *data* liefert einen Zeiger auf das zur Zeit enthaltene *QMimeSource*-Objekt zurück. Ist die Zwischenablage leer, liefert die Methode einen Null-Zeiger zurück.

Qt enthält bereits die Klassen *QTextDrag*, *QImageDrag* und *QURLDrag* (alle von *QMimeSource* abgeleitet), mit denen Textdaten, Bilddaten bzw. URLs (Pfadangaben für lokale Dateien oder Dateien im WWW oder auf FTP-Servern) in die Zwischenablage kopiert werden können. Für Text und Bilder können Sie natürlich viel einfacher die Methode *setText*, *setImage* und *setPixmap* benutzen. Wenn Sie eigene Datentypen definieren wollen, definieren Sie eine eigene Klasse, die Sie von *QMimeSource* ableiten. Überschreiben Sie dabei die virtuellen Methoden *format* und *encodedData*. *format* sollte für die Parameterwerte 0, 1, 2 usw. die MIME-Typen, die von Ihrem Objekt geliefert werden können, als Text-String zurückgeben. *encodedData* sollte ein *QByteArray*-Objekt erzeugen und in dem MIME-Typ-Format zurückgeben, das im Parameter angegeben wurde.

Wenn Sie nur einen einzigen MIME-Typ unterstützen wollen (das ist in der Regel bei selbst definierten Datentypen der Fall), können Sie einfach die Klasse *QStoredDrag* benutzen. Im Konstruktor geben Sie den MIME-Typ an, und mit der Methode *setEncodedData* legen Sie den Datenblock (als Objekt der Klasse *QByteArray*) fest.

Die Methode *QClipboard::clear* löscht den Inhalt der Zwischenablage. Weiterhin ist in *QClipboard* die Signalmethode *dataChanged* definiert. Diese Signalmethode wird jedes Mal aufgerufen, wenn sich der Inhalt der Zwischenablage ändert – sowohl durch die eigene Applikation als auch durch andere Programme. Wenn

Sie beispielsweise markierten Text automatisch in die Zwischenablage einfügen, können Sie dieses Signal mit einer Methode *unselectAll* verbinden. Sobald also eine andere Applikation die Zwischenablage füllt und damit den markierten Text der eigenen Applikation verdrängt, wird die Markierung des Textes aufgehoben.

4.15.2 Drag&Drop

Eine gute Applikation zeichnet sich durch eine intuitive Bedienbarkeit aus. In den letzten Jahren hat sich auf nahezu allen grafischen Benutzersystemen das Drag&Drop-Prinzip durchgesetzt. Man zieht dabei Elemente von einem Fenster in ein anderes, indem man auf dem Element die linke Maustaste drückt und sie gedrückt hält, die Maus an die Stelle bewegt, an der das Element platziert oder eingefügt werden soll (*drag*), und dort die linke Maustaste wieder loslässt (*drop*). Die Fenster können dabei sogar zu verschiedenen Applikationen gehören. Beide Applikationen müssen dafür allerdings Drag&Drop unterstützen.

Das Drag&Drop-Verfahren wird zum Beispiel eingesetzt, um eine Datei aus einem Dateimanager (z.B. dem *Konqueror*) auf eine Applikation zu ziehen. Diese Applikation öffnet nun diese Datei (falls sie vom richtigen Typ ist) und stellt sie dar. Eine andere Anwendung ist die Zusammenstellung eines Dokuments, indem Elemente aus einem anderen Fenster mit Elementen in das Hauptfenster gezogen werden. So kann beispielsweise eine CD-Player-Applikation aus der Liste der vorhandenen Lieder eine Liste der Lieder erstellen, die abgespielt werden sollen. Die Anwendungen von Drag&Drop sind weitaus vielfältiger. Wenn Sie selbst Drag&Drop einsetzen wollen, überlegen Sie, ob die Benutzung intuitiv ist und ob es eine Arbeitserleichterung für den Anwender im Vergleich zu herkömmlichen Prinzipien ist.

Drag&Drop wird speziell genutzt, wenn Elemente von einem Fenster in ein anderes oder sogar in eine andere Applikation verschoben werden sollen. Wenn Sie Elemente nur innerhalb eines Fensters verschieben wollen, also nur die Position verändern, so ist das Prinzip von Drag&Drop wahrscheinlich nicht der richtige Weg. Das erreichen Sie meist einfacher und für den Anwender komfortabler, indem Sie in Ihrem Widget die Maus-Events abfangen und entsprechend die Position der Elemente verändern. *KPresenter* benutzt zum Beispiel zum Verschieben der grafischen Objekte innerhalb des Dokuments nicht das Drag&Drop-Verfahren. Nach jeder Positionsänderung können Sie sofort das Widget neu zeichnen lassen.

Die Qt-Klasse *QWidget* enthält bereits Methoden, um Elemente zu empfangen, die auf dem Fenster fallen gelassen wurden. Die Daten des Elements werden als ein Block von Bytes beliebiger Länge übertragen. Der Typ der enthaltenen Daten wird als MIME-Typ angegeben, also als String, der den Typ spezifiziert. Ein Widget kann angeben, welche Typen von Daten es entgegennehmen kann.

Wenn Sie in Ihrem Widget Elemente per Drop entgegennehmen wollen, so leiten Sie eine neue Klasse ab. Diese Klasse muss durch Mehrfachvererbung sowohl von einer Widget-Klasse als auch von der Klasse *QDropSite* abgeleitet sein. In der neuen Klasse überschreiben Sie die Methoden *dragEnterEvent* und *dropEvent*. *dragEnterEvent* wird aufgerufen, sobald ein Element beim Ziehen über den Bildschirm in das Widget eintritt. In dieser Methode sollten Sie testen, ob Ihr Widget diesen Datentyp entgegennehmen kann, und entsprechend *QDragEnterEvent::accept* oder *QDragEnterEvent::ignore* aufrufen. Die Methode *dropEvent* wird aufgerufen, falls das Element sich innerhalb des Widgets befindet, wenn die linke Maustaste losgelassen wird. Mit der Methode *QDropEvent::encodedData* können Sie sich den Datenblock geben lassen, der zum übertragenen Element gehört. Sie können dabei den MIME-Typ angeben, in den das Element vor der Rückgabe umgewandelt werden soll. Mit der Methode *QDropEvent::pos* können Sie die Stelle ermitteln, an der der Mauszeiger zum Zeitpunkt des Fallenlassens stand (relativ zur linken oberen Fensterecke). Die Methode *QDropEvent::source* liefert Ihnen das Widget zurück, von dem das Element gezogen wurde. Dieser Rückgabewert macht natürlich nur dann Sinn, wenn das Element von einem Widget stammt, das zur gleichen Applikation gehört.

Als ein einfaches Beispiel wollen wir hier ein Widget entwerfen, das Textelemente entgegennehmen kann. Die Klassendefinition der eigenen Widget-Klasse kann zum Beispiel so aussehen:

```
class MyDropClass : public QWidget, QDropSite
{
    Q_OBJECT
public:
    MyDropClass (QWidget *parent = 0, const char *name = 0);
    ~MyDropClass () {}
    ...

protected:
    void dragEnterEvent (QDragEnterEvent *ev);
    void dropEvent (QDropEvent *ev);
};

void MyDropClass::dragEnterEvent (QDragEnterEvent *ev)
{
    // Testen, ob das Element in den Datentyp TEXT/PLAIN
    // umgewandelt werden kann
    if (ev->provides ("TEXT/PLAIN"))
        // Das Element wird überall im Widget akzeptiert
        ev->accept (rect());
    else
        // Das Element wird nirgends im Widget akzeptiert
        ev->ignore (rect());
}
```

```

}

void MyDropClass::dropEvent (QDropEvent *ev)
{
    QString text;
    if (QTextDrag::decode (ev, text))
    {
        // Dekodierung erfolgreich
        // Hier können die ASCII-Daten in text
        // an die Position ev->pos() eingefügt werden.
        ...
        ev->accept ();
    }
}

```

Ob das Element entgegengenommen wird, testen wir in unserem Beispiel mit der Methode `QDragEnterEvent::provides`. Diese Methode ruft man mit dem gewünschten MIME-Typ auf, und sie liefert *true* zurück, wenn das Element in diesen Datentyp umgewandelt werden kann. Man kann auch die Methode `QDragEnterEvent::format` benutzen. Mit dem Parameter 0 liefert diese Methode den String zurück, der den MIME-Typ des Originalformats des Elements darstellt. Für die Parameterwerte 1, 2, 3 usw. liefert sie die MIME-Typen als String zurück, in die das Element umgewandelt werden kann. Liegen keine weiteren Typen vor, wird der Null-Zeiger zurückgegeben.

In unserem Beispiel wird beim `dragEnterEvent` festgelegt, ob das Element entgegengenommen werden kann oder nicht. Ruft man wie hier *accept* oder *ignore* mit den Rechteck-Koordinaten des ganzen Widgets auf, so gilt diese Entscheidung so lange, bis das Element wieder aus dem Fenster herausgezogen wird. Wenn Sie gezielter festlegen wollen, an welchen Stellen das Element fallen gelassen werden darf, lassen Sie am besten die Methode `dragEnterEvent` unverändert und überschreiben stattdessen die Methode `dragMoveEvent`. Diese Methode wird jedes Mal aufgerufen, wenn das Element innerhalb des Widgets bewegt wird (bei gedrückter Maustaste). Sie können hier anhand des MIME-Typs und der aktuellen Mausposition entscheiden, ob das Objekt an dieser Stelle fallen gelassen werden darf. Rufen Sie anschließend *accept* bzw. *ignore* ohne Parameter auf. Sie können auch hier die Rechteck-Koordinaten angeben, für die diese Entscheidung gelten soll. Wenn Sie ein Rechteck angeben, wird die Methode `dragMoveEvent` erst dann wieder aufgerufen, wenn sich die Maus außerhalb des angegebenen Rechtecks befindet. So können Sie verhindern, dass bei jeder kleinen Mausbewegung die Entscheidung neu getroffen werden muss. Sie können auch bereits die Daten des Elements abfragen, um zu entscheiden, ob das Element fallen gelassen werden kann. Benutzen Sie dazu wie in der Methode `dropEvent` die Methode `QDragMoveEvent::encodedData`. Auf diese Möglichkeit sollten Sie aber möglichst verzichten, da die Umwandlung zeitintensiv sein kann.

Um eine Drag&Drop-Operation zu starten, müssen Sie ein Objekt der Klasse *QDragObject* (bzw. einer abgeleiteten Klasse) mit *new* auf dem Heap erzeugen und eine der Methoden *drag*, *dragMove* oder *dragCopy* dieses Objekts aufrufen. In der Regel erzeugen Sie dieses Objekt innerhalb der Methode *mousePressEvent* oder der Methode *mouseMoveEvent* Ihres Widgets. Für die Standard-MIME-Typen Text, Bilder und URLs gibt es bereits die fertigen, von *QDragObject* abgeleiteten Klassen *QTextDrag*, *QImageDrag* und *QURLDrag*.

Es gibt drei verschiedene Möglichkeiten, ein Element per Drag&Drop zu bewegen: Sie können das Element vom alten Fenster in das neue Fenster übertragen, wobei es – bei erfolgreicher Übertragung – im Ursprungsfenster gelöscht wird; Sie können das Element in das neue Fenster kopieren, oder Sie können den Benutzer mit Hilfe der `[Strg]`-Taste wählen lassen, ob verschoben oder kopiert werden soll. Um den Drag&Drop-Vorgang zu starten, rufen Sie eine der drei Methoden der Klasse *QDragObject* auf: *dragMove* (verschiebt das Element), *dragCopy* (kopiert das Element) oder *drag* (verschiebt oder kopiert, abhängig davon, ob `[Strg]` gedrückt ist). Durch diesen Methodenaufruf übernimmt Qt die weitere Kontrolle des Objekts. Erst nachdem das Element fallen gelassen wurde, kehrt die aufgerufene Methode zurück. Das *QDragObject*-Objekt wird vollständig von Qt verwaltet und nach dem Drag&Drop-Vorgang automatisch mit *delete* wieder gelöscht. Die Methoden *dragMove* und *drag* liefern einen Rückgabewert vom Typ *bool* zurück, der genau dann *true* ist, wenn eine erfolgreiche Verschiebung des Elements vorgenommen wurde. Nur in diesem Fall ist also das Element im Ursprungsfenster zu löschen. *dragCopy* liefert keinen Rückgabewert, da das Element im Ursprungsfenster nicht gelöscht werden muss.

Sie können mit der Methode *QDragObject::setPixmap* ein Bild festlegen, das zusammen mit der Maus verschoben wird, um dem Anwender eine grafische Rückmeldung zu geben, dass zur Zeit ein Drag&Drop-Vorgang abläuft. Als zweiten Parameter können Sie den so genannten Hotspot dieses Bildes angeben. Das ist der Punkt auf dem Bild, auf dem der Mauszeiger positioniert wird. Wenn Sie beispielsweise als Hotspot den Punkt (0/0) angeben, so liegt das Bild mit der oberen linken Ecke unter dem Mauszeiger. Meist benutzt man die Koordinaten (-10/-10), so dass die obere linke Bildecke zehn Pixel rechts und zehn Pixel unterhalb des Mauszeigers zu liegen kommt. Wenn die Grafik auf dem Bild nicht genau rechteckig ist, empfiehlt es sich, ein *QPixmap*-Objekt mit Maske zu benutzen, so dass der Rand des Bildes transparent ist.

Das folgende Beispiel erweitert die Widget-Klasse *QListBox*, die Textelemente und Bildelemente in einer Liste darstellt, um die Möglichkeit, einzelne Textelemente aus dem Objekt herauszuziehen. Außerdem können auch Text- und Bildelemente auf diesem Objekt fallen gelassen werden. Die Implementierung ist noch keineswegs voll ausgereift: So besteht bei *QListBox* beispielsweise die Möglichkeit, mehrere Elemente zu markieren, nachdem man *setMultiSelection (true)* aufgerufen hat.

Unsere Beispielimplementierung verschiebt aber immer nur ein Element per Drag&Drop.

```
class DNDListBox : public QListBox, QDropSite
{
    Q_OBJECT

public:
    DNDListBox (QWidget *parent = 0, const char *name = 0)
        : QListBox (parent, name),
          QDropSite (this),
          pressed (false) {}
    ~DNDListBox () {}

protected:
    void mousePressEvent (QMouseEvent *ev);
    void mouseMoveEvent (QMouseEvent *ev);
    void mouseReleaseEvent (QMouseEvent *ev);
    void dragMoveEvent (QDragMoveEvent *ev);
    void dropEvent (QDropEvent *ev);

private:
    QPoint startPos;
    bool pressed;
};

void DNDListBox::mousePressEvent (QMouseEvent *ev)
{
    if (ev->button() == LeftButton) {
        startPos = ev->pos();
        pressed = true;
    }
    QListBox::mousePressEvent (ev);
}

void DNDListBox::mouseReleaseEvent (QMouseEvent *ev)
{
    if (ev->button() == LeftButton)
        pressed = false;
    QListBox::mouseReleaseEvent (ev);
}

void DNDListBox::mouseMoveEvent (QMouseEvent *ev)
{
    if (!pressed)
    {
        QListBox::mouseMoveEvent (ev);
        return;
    }
}
```

```

    }

    // Bewegung der Maus bei gedrückter linker Maustaste
    // um mehr als zehn Pixel startet Drag&Drop-Vorgang
    if (QABS (ev->pos().x() - startPos.x()) > 10 ||
        QABS (ev->pos().y() - startPos.y()) > 10)
    {
        pressed = false;
        QDragObject *d;
        int item = findItem (startPos.y());
        QString contentsText = text (item);
        if (!contentsText.isNull())
        {
            // Falls Textelement, QTextDrag-Objekt erzeugen
            d = new QTextDrag (contentsText, this);
            // Als Bild eine Textseite benutzen
            d->setPixmap (Icon ("text.xpm"), QPoint (-10, 10));
        }
        else
        {
            // Sonst Bildelement, QImageDrag-Objekt erzeugen
            QPixmap *pix = pixmap (item);
            d = new QImageDrag (pix->convertToImage(), this);
            // Als Bild ein Foto benutzen
            d->setPixmap (Icon ("image.xpm"), QPoint (-10, 10));
        }

        // Drag&Drop-Vorgang starten
        // Hier wird dragMove() benutzt, das Element soll
        // also verschoben werden.
        if (d->dragMove())
            // Falls verschoben, Element aus der Listbox
            // entfernen
            removeItem (item);
    }
}

void DNDListBox::dragMoveEvent (QDragMoveEvent *ev)
{
    // Nimmt nur Text- und Bildelemente entgegen
    if (QImageDrag::canDecode (ev) ||
        QTextDrag::canDecode (ev))
        ev->accept (rect());
    else
        ev->ignore (rect());
}

void DNDListBox::dropEvent (QDropEvent *ev)
{

```

```

// Index in der Liste finden
int index = findItem (lastPos.y());
QString text;
QPixmap pix;
if (QImageDrag::decode (ev, pix))
{
    insertItem (pix, index);
    ev->accept();
}
else if (QTextDrag::decode (ev, text)) {
    insertItem (text, index);
    ev->accept();
}
else
    ev->ignore();
}

```

Mit dieser Klasse können Einträge von einem *DNDListBox*-Widget zu einem anderen verschoben werden. Quelle und Ziel des Drag&Drop-Vorgangs können im gleichen Vater-Widget liegen, aber auch in verschiedenen Toplevel-Widgets. Sie können sogar in verschiedenen Programmen sein. Auch andere Widgets, die Elemente von MIME-Typen erzeugen, die von Qt in ein Text- oder Bildobjekt umgewandelt werden können, können als Quelle oder Ziel dienen. Ebenso kann man ein Element innerhalb eines *DNDListBox*-Widgets verschieben. Nachteilig ist, dass Bilder beim Übertragen per Drag&Drop zuerst in das XPM-Dateiformat umgewandelt und anschließend wieder zurückgewandelt werden. Innerhalb einer Applikation wäre es schneller und effizienter, das *QPixmap*-Objekt zu übertragen, da in dem Fall keine Umwandlung nötig wäre.

Wenn Sie diese Klasse in Ihren eigenen Programmen verwenden wollen, müssen Sie sie noch in einigen Punkten verbessern. Zur Zeit wird ein Objekt an der Stelle des Elements eingefügt, an dem der Mauszeiger beim Loslassen der Maustaste stand. Besser wäre es, das Element in dem Zwischenraum zwischen den bestehenden Elementen einzufügen, denen der Mauszeiger am nächsten ist.

Problematischer ist allerdings die Situation, wenn man einen Eintrag innerhalb der gleichen *QListBox* zieht und wieder einfügt. Befindet sich die Einfügestelle vor der Stelle, an der das Element ursprünglich lag, so wird beim Entfernen des alten Elements ein falscher Eintrag gelöscht. Sie können dieses Problem umgehen, indem Sie in der Methode *dropEvent* testen, ob das Ursprungs-Widget das gleiche Widget wie das Ziel-Widget ist:

```

if (ev->source() == this)
{
    // Ursprungs-Widget und Ziel-Widget identisch
    // Noch nicht einfügen, sondern erst zwischenspeichern
    position = index;
}

```

```
    buffer = ev->encodedData();  
}  
else  
{  
    // sonst einfügen wie gehabt  
}
```

Die Variablen *position* (Typ *int*) und *buffer* (Typ *QByteArray*) sind hierbei neue *private*-Variablen der Klasse *DNDListBox*. In der Methode *mouseMoveEvent* kann nach dem Aufruf der Methode *QDragObject::drag* zuerst das ursprüngliche Element gelöscht werden. Danach wird getestet, ob das Element zwischengespeichert wurde. In diesem Fall wird es nun eingefügt.

Wollen Sie andere Daten als Text und Bilder per Drag&Drop übertragen, so definieren Sie eine eigene Klasse, die Sie von *QDragObject* ableiten. Sie müssen dabei insbesondere die virtuellen Methoden *format* und *encodedData* (beide geerbt von *QMimeSource*, der Basisklasse von *QDragObject*) überschreiben. *format* sollte für die Parameterwerte 0, 1, 2 usw. die MIME-Typen zurückgeben, die von Ihrem Objekt geliefert werden können. *encodedData* sollte ein *QByteArray*-Objekt erzeugen und in dem MIME-Typ-Format zurückgeben, das im Parameter angegeben wurde.

Wenn Sie nur einen einzigen MIME-Typ unterstützen wollen (das ist in der Regel bei selbst definierten Datentypen der Fall), so können Sie einfach die Klasse *QStoredDrag* benutzen. Im Konstruktor geben Sie den MIME-Typ an, und mit der Methode *setEncodedData* legen Sie den Datenblock (als Objekt der Klasse *QByteArray*) fest.

4.16 Session-Management

Ein gutes KDE-Programm speichert seinen vollständigen Zustand automatisch ab, wenn der X-Server heruntergefahren wird. Das geschieht in der Regel beim Logout. Wird der X-Server in einer späteren Sitzung wieder gestartet, werden auch automatisch alle KDE-Programme, die geöffnet waren, wieder gestartet. Sie lesen ihren gespeicherten Zustand wieder ein, so dass der Anwender genau an der Stelle fortfahren kann, an der er aufgehört hatte.

Den Zeitraum zwischen dem Login eines Benutzers und dem Logout nennt man *Session*; die automatische Speicherung des Zustands am Ende einer Session bezeichnet man als *Session-Management*.

Beachten Sie, dass das Session-Management nicht benutzt wird, wenn Sie eine Applikation regulär beenden. In diesem Fall wird das Programm auch nicht automatisch beim nächsten Login wieder gestartet. Wenn Sie den Zustand des Programms auch beim regulären Beenden speichern möchten (zum Beispiel die Fenstergrößen und -positionen), benutzen Sie hierfür die normale Konfigurationsdatei, die in Kapitel 4.10, *Konfigurationsdateien*, besprochen wurde.

Die Klassen *KApplication* und *KMainWindow* besitzen bereits einige Methoden, um das Session-Management ohne großen Aufwand zu einem Programm hinzuzufügen. Auch die Qt-Bibliothek bietet eine eigene Klasse, *QSessionManager*, zur Behandlung des Session-Managements an. Obwohl die Qt-Klasse mehr Möglichkeiten bietet als die KDE-Implementierung, sollte man die KDE-Implementierung vorziehen, da sie bereits eine sehr einfache Möglichkeit bietet, alle Hauptfenster der Klasse *KMainWindow* abzuspeichern. Außerdem stellt sie Konfigurationsdateien bereit, in denen Sie den Zustand des Programms speichern können. Somit ist der Aufwand für Sie viel geringer.

4.16.1 Session-Management in KDE

KDE erstellt für das Session-Management eigene Konfigurationsdateien, in denen der aktuelle Programmstatus gespeichert werden kann. Während die normale Konfigurationsdatei einer Applikation *appname.rc* heißt, hat die Konfigurationsdatei für das Session-Management den Dateinamen *appname~1.rc*, *appname~2.rc* und so weiter. Wenn die Applikation mehrmals gestartet wurde, erhält jede Instanz des Programms eine eigene Konfigurationsdatei für das Session-Management. Nach dem nächsten Login werden diese Dateien automatisch wieder gelöscht. Sie erhalten Zugriff auf die Konfigurationsdatei des Session-Managements mit der Methode *KApplication::sessionConfig*. Beachten Sie aber, dass diese Konfigurationsdatei nur während eines Session-Management-Vorgangs existiert, also beim Logout bei laufendem Programm oder beim erneuten Start des Programms beim nächsten Login.

KApplication besitzt die Signalmethode *saveYourself*, die vom Session-Management aufgerufen wird. Dazu müssen Sie aber zunächst das Session-Management mit der Methode *KApplication::enableSessionManagement* (*false*) aktivieren. (Der Parameter vom Typ *bool* gibt an, ob Sie ein eigenes Kommando für die Aktivierung des Session-Managements im Window-Manager angeben wollen. In der Regel ist das nicht nötig, daher wird hier der Wert *false* benutzt.) Sie können einen eigenen Slot mit diesem Signal verbinden und in diesem Slot den Zustand Ihres Programms abspeichern. Beachten Sie dabei aber unbedingt, dass Sie in diesem Slot nicht auf den X-Server zugreifen dürfen. Der X-Server ist während des Session-Management-Logout-Prozesses blockiert. Sie sollten also keine Änderungen an Widgets vornehmen, keine neuen Widgets öffnen und auch keine Pixmaps benutzen. Insbesondere können Sie hier keine Benutzerabfrage vornehmen, ob das aktuelle Dokument gespeichert werden soll und unter welchem Dateinamen dieses geschehen soll. Generell gilt, dass Sie geöffnete und geänderte Daten in einer temporären Hilfsdatei abspeichern sollten und nicht in der Originaldatei. Schließlich können Sie nicht wissen, ob der Anwender die Änderungen wirklich in der Datei abspeichern oder sie verwerfen will.

Mit der Methode *KApplication::isRestored* können Sie erfragen, ob die Applikation automatisch beim Login gestartet wurde – in diesem Fall muss der Status aus der Konfigurationsdatei rekonstruiert werden – oder vom Anwender gestartet wurde. Meist rufen Sie diese Methode unmittelbar nach der Erzeugung des *KApplication*-Objekts auf.

Eine Applikation, die das Session-Management benutzen will, kann nach folgendem Schema aufgebaut sein:

```
void AnyObject::saveYourselfSlot ()
{
    // Wird von saveYourself aufgerufen.
    // Speichert den Status des Programms in der
    // Session-Management-Konfigurationsdatei
    KConfig *conf = kapp->getSessionConfig();
    ....
}

int main (int argc, char **argv)
{
    KApplication app (argc, argv);
    app.enableSessionManagement (false);
    if (app.isRestored())
    {
        KConfig *conf = app.getSessionConfig();
        // Zustand wiederherstellen aus den
        // Daten in conf
    }
    ....
    AnyObject *any = new AnyObject (...);
    QObject::connect (kapp, SIGNAL (saveYourself()),
                     any, SLOT (saveYourselfSlot()));
    ....
}
```

Es gibt eine viel einfachere Möglichkeit, das Session-Management in KDE sehr elegant zu implementieren. Dazu bietet die Klasse *KMainWindow* die beiden Methoden *saveProperties* und *readProperties*. Wenn Sie als Hauptfenster also die Klasse *KMainWindow* (bzw. eine abgeleitete Klasse) benutzen, empfiehlt es sich, mit diesen beiden Methoden zu arbeiten.

Die *KMainWindow*-Klasse speichert automatisch die Fensterkoordinaten und die Positionen von Menüleiste und Werkzeugleisten in der Konfigurationsdatei für das Session-Management ab. In der Methode *saveProperties* kann Ihre eigene, abgeleitete Klasse weitere Attribute speichern, die für den Status des Fensters wichtig sind. Da es mehrere Hauptfenster geben kann, wird für jedes Hauptfenster eine eigene Gruppe in der Konfigurationsdatei angelegt. Die Methode

saveProperties erhält im ersten Parameter das Konfigurationsobjekt, das bereits auf die Gruppe des Hauptfensters eingestellt ist. Wenn Sie Daten in diesem Objekt abspeichern, sollten Sie also nicht die Gruppe ändern.

Analog dazu wird die Methode *readProperties* beim Wiederherstellen der Hauptfenster aufgerufen. Auch hier ist die Gruppe bereits korrekt eingestellt, so dass Sie direkt die benötigten Attribute wieder auslesen können.

Das Wiederherstellen der Hauptfenster muss in der *main*-Funktion noch gestartet werden. Da Sie in der Regel alle Hauptfenster als Instanzen der gleichen Klasse erzeugen, können Sie das Makro *RESTORE* benutzen, das in der Datei *kmainwindow.h* definiert ist. Diesem Makro müssen Sie den Klassennamen als Parameter übergeben.

Wenn Sie die Klasse *KMainWindow* (oder eine abgeleitete Klasse) benutzen, brauchen Sie die Methode *KApplication::enableSessionManagement* nicht aufrufen, da dies automatisch von *KMainWindow* erledigt wird.

Ein Programm, das diese Technik ausnutzt, kann beispielsweise so aussehen:

```
class MyMainWindow : public KMainWindow
{
    Q_OBJECT
    ...
protected:
    void saveProperties (KConfig *conf);
    void readProperties (KConfig *conf);
};

void MyMainWindow::saveProperties (KConfig *conf)
{
    // Zusätzliche Attribute (zum Beispiel die im Fenster
    // eingestellten Daten) können in conf gespeichert
    // werden. Ändern Sie dabei nicht die eingestellte
    // Gruppe.
}

void MyMainWindow::readProperties (KConfig *conf)
{
    // Hier werden die gespeicherten Attribute wieder
    // aus conf eingelesen und im Fenster gesetzt.
}

int main (int argc, char **argv)
{
    KApplication app (argc, argv);
    if (app.isRestored())
        // Alte Fenster rekonstruieren
        RESTORE (MyMainWindow);
}
```

```

else
{
    // Sonst ein erstes, leeres Fenster erzeugen
    MyMainWindow *help = new MyMainWindow ();
    help->show();
}
...
}

```

Falls Ihre Applikation Hauptfenster verschiedener Klassen enthalten kann, müssen Sie beim Wiederherstellen der Fenster selbst Hand anlegen. Die Objekte der Klasse *KMainWindow* oder einer Unterklasse werden mit Indizes von 1 bis *n* abgespeichert. Mit der statischen Methode *KMainWindow::canBeRestored (int i)* kann man erfragen, ob ein Fenster mit dem Index *i* abgespeichert wurde. Mit der statischen Methode *KMainWindow::classNameOfToplevel (int i)* kann man den Klassennamen des Fensters erfragen, der zum Index *i* gehört. Enthält ein Programm beispielsweise die beiden Hauptfensterklassen *MyMainWindow1* und *MyMainWindow2*, kann folgende *main*-Funktion die Fenster wiederherstellen:

```

int main (int argc, char **argv)
{
    KApplication app (argc, argv);
    if (app.isRestored())
    {
        int i = 1;
        while (KMainWindow::canBeRestored (i)) {
            // Solange das Fenster mit Index i abgespeichert
            // ist, Klassenname erfragen
            QString name = KMainWindow::classNameOfToplevel (i);
            // Je nach Klassenname Klasse erzeugen und
            // restaurieren
            if (name == "MyMainWindow1")
                (new MyMainWindow1 ())->restore (i);
            else if (name == "MyMainWindow2")
                (new MyMainWindow2 ())->restore (i);
            else
                debug ("unbekannter Klassenname bei restore");
            i++;
        }
    }
    else
    {
        // Sonst ein erstes, leeres Fenster erzeugen,
        // hier zum Beispiel der Klasse MyMainWindow2
        MyMainWindow2 *help = new MyMainWindow2 ();
        help->show();
    }
    ...
}

```

4.16.2 Session-Management in Qt

Auch Qt bietet zur Implementierung des Session-Managements eine eigene Klasse namens *QSessionManager* an. Im Gegensatz zur KDE-Implementierung können Sie hier den X-Server beim Herunterfahren Ihrer Applikation benutzen, nachdem Sie ihn speziell angefordert haben. So können Sie beispielsweise einen Dialog öffnen lassen, in dem der Anwender gefragt wird, ob er ungesicherte Daten speichern möchte. Die Klasse bietet weiterhin die Möglichkeit, den Logout-Prozess abubrechen. Allerdings bietet die Qt-Implementierung des Session-Managements keine speziellen Konfigurationsdateien an, in denen der Zustand gespeichert werden könnte. Es empfiehlt sich daher, die KDE-Implementierung vorzuziehen.

Der Konstruktor der Klasse *QSessionManager* ist als *private* deklariert. Sie können also kein eigenes Objekt erzeugen. Es gibt nur ein zentrales Objekt, das in *QApplication* erzeugt wird. *QApplication* besitzt die beiden virtuellen Methoden *commitData* und *saveState*. Diese beiden Methoden werden beim Logout-Prozess aufgerufen. Die Default-Implementierungen dieser beiden Methoden bewirken nichts. Um den Status Ihrer Applikation zu speichern, leiten Sie eine eigene Klasse von *QApplication* ab (bzw. von *KApplication*, wenn Sie ein KDE-Programm schreiben) und überschreiben die Methoden. Beide Methoden liefern in einem Parameter das *QSessionManagement*-Objekt, auf das Sie zugreifen können.

Sie können sich innerhalb der Methoden Zugriff auf den X-Server verschaffen, um ein Fenster auf dem Bildschirm anzuzeigen, indem Sie die Methode *allowsInteraction* des *QSessionManagement*-Objekts aufrufen. Ist der Rückgabewert dieser Methode *true*, wird der Zugriff gewährt. So können Sie beispielsweise in einem Dialogfenster den Anwender fragen, ob ungespeicherte Daten gespeichert werden sollen. Beachten Sie aber, dass KDE-Programme ihren Status in einer temporären Datei abspeichern und diesen Status beim Neustart wiederherstellen sollten. Vermeiden Sie also solche Dialoge, wenn möglich. Für besonders wichtige Fälle können Sie auch die Methode *allowsErrorInteraction* aufrufen, die höhere Priorität hat. Auch hier gibt ein Rückgabewert von *true* an, dass Sie auf den X-Server zugreifen dürfen. Nach dem Zugriff sollten Sie den X-Server unbedingt durch Aufruf der Methode *release* wieder freigeben.

Durch den Aufruf der Methode *QSessionManager::cancel* können Sie den Logout-Prozess abbrechen. Beachten Sie aber, dass ein Programm dies nur in Ausnahmefällen tun sollte. Insbesondere kann ein Programm, das den Logout-Prozess immer abbricht, für viel Verwirrung sorgen.

Mit der Methode *QSessionManager::setRestartHint* können Sie festlegen, ob das Programm beim nächsten Login wieder gestartet werden soll. Als Parameter sind die Werte *RestartIfRunning*, *RestartAnyway*, *RestartImmediately* und *RestartNever* erlaubt. *RestartIfRunning* ist die Standardeinstellung. Mit ihr wird das Programm

beim nächsten Login wieder gestartet. *RestartAnyway* gibt an, dass das Programm bei jedem Login automatisch gestartet werden soll, unabhängig davon, ob es beim letzten Logout noch lief. Das ist zum Beispiel für Programme sinnvoll, die permanent im Hintergrund laufen sollen. Auch wenn der Anwender sie explizit beendet, werden sie beim nächsten Login wieder gestartet. Mit *RestartImmediately* legen Sie fest, dass das Programm automatisch nach einem Beenden wieder gestartet werden soll. Auch das kann für Programme sinnvoll sein, die immer im Hintergrund laufen sollen. Seien Sie jedoch vorsichtig mit dieser Einstellung, da sie oft zu Verwirrung führen kann. Mit *RestartNever* geben Sie an, dass das Programm auch beim nächsten Login nicht automatisch gestartet wird. Es muss also vom Anwender explizit erneut gestartet werden. Nicht alle X-Server bzw. Window-Manager unterstützen alle Einstellungen. Es handelt sich hierbei also immer nur um Vorschläge an das System.

Mit der Methode *isRestored* der Klasse *QApplication* können Sie testen, ob Ihr Programm durch Session-Management gestartet (Rückgabewert *true*) oder vom Anwender explizit aufgerufen wurde (Rückgabewert *false*). Am besten rufen Sie diese Methode unmittelbar nach Erzeugung des *QApplication*-Objekts auf. Liefert die Methode *true*, müssen Sie in der Regel den Zustand des Programms wiederherstellen.

4.17 Drucken mit Qt

Qt bietet bereits zwei interessante Klassen an, *QPrintDialog* und *QPrinter*, mit denen eine Applikation einen Drucker nutzen kann. Dieser Drucker kann dabei sowohl direkt angeschlossen als auch über ein Netzwerk erreichbar sein.

In der Qt-Version für Unix-Betriebssysteme erfolgt – wie in Unix üblich – die Druckerausgabe in der Seitenbeschreibungssprache Postscript. Für Drucker, die Postscript nicht direkt umsetzen können, übernimmt in Linux das Program *GhostScript* die Umsetzung in die druckerspezifische Sprache. Die Druckerinstallation ist Teil der Installation des Betriebssystems und ist bei den meisten Linux-Distributionen inzwischen sehr komfortabel möglich.

Die Klasse *QPrinter*, die bereits in Kapitel 4.2.6, *Unterklassen von QPaintDevice*, kurz angesprochen wurde, ist von der Klasse *QPaintDevice* abgeleitet. Mit ihr kann man auf einen Drucker wie auf ein Widget oder in eine Pixmap zeichnen. Dazu nutzt man in der Regel die Klasse *QPainter*, die das Zeichnen von Linien, Flächen oder Text ermöglicht. Diese Klasse wurde ausführlich in Kapitel 4.2, *Zeichnen von Grafikprimitiven*, beschrieben. Alle dort beschriebenen grafischen Elemente und alle Transformationen können auch für das Zeichnen auf ein *QPrinter*-Objekt genutzt werden.

Mit *QPaintDeviceMetrics* kann man die Größe einer Seite in Millimetern oder in Einheiten (standardmäßig in Einheiten von 1/72 Zoll) ermitteln. Um beispiels-

weise eine rote Linie der Dicke 3/72 Zoll diagonal über die Seite zu zeichnen, können Sie folgendes Programmstück benutzen:

```
QPrinter prn; // Erzeuge Druckerobjekt
QPaintDeviceMetrics metrics (&prn); // Metrics-Objekt
QPainter painter (&prn); // Painter-Objekt

painter.setPen (QPen (red, 3));
painter.drawLine (0, 0, metrics.width(), metrics.height());
```

Einen Seitenvorschub erreichen Sie mit der Methode *newPage*. Alle weiteren Zeichenoperationen, die anschließend mit dem *QPainter*-Objekt gezeichnet werden, erscheinen auf der nächsten Seite.

Am *QPrinter*-Objekt kann man eine Reihe von Einstellungen vornehmen. Mit *setPageSize* kann man beispielsweise die Seitengröße einstellen. Dazu muss man die Blattgröße in Form einer Konstanten als Parameter angeben. Die wichtigsten Konstanten sind dabei sicherlich *QPrinter::A4* und *QPrinter::Letter*. Die Standardeinstellung ist die Blattgröße A4. Mit *setOrientation* können Sie zwischen Hochformat (*QPrinter::Portrait*) und Querformat (*QPrinter::Landscape*) wählen.

Mit *setNumCopies* können Sie die Anzahl der Kopien einstellen. Mit *setColorMode* kann man zwischen Farbdruck (*QPrinter::Color*) und Graustufendruck (*QPrinter::Grayscale*) wählen.

Mit *setPrinterName* können Sie den Drucker festlegen, auf den gedruckt werden soll. Benutzen Sie dabei die Druckernamen des Systems. Wenn Sie den Drucker nicht festlegen, wird der Standarddrucker benutzt. Mit der Methode *setPrintProgram* können Sie das Programm einstellen, mit dem der Ausdruck in die Druckerwarteschlange eingereiht werden soll. Unter Linux ist meist das Programm *lpr* dafür zuständig. Dieses Programm ist auch die Standardeinstellung. Wenn Sie ein anderes Programm nutzen wollen, können Sie es hier einstellen.

Statt auf einen Drucker können Sie die Postscript-Ausgabe auch in eine Datei umlenken lassen. Dazu können Sie den Namen der Datei (mit optionaler Pfadangabe) mit der Methode *setOutputFileName* festlegen. Wenn Sie einen leeren String als Dateiname angeben, wird die Ausgabe wieder auf den Drucker umgeleitet.

Eine bereits begonnene Ausgabe können Sie mit der Methode *abort* wieder abbrechen. Der Rückgabewert der Methode ist vom Typ *bool* und gibt an, ob der Abbruch korrekt durchgeführt werden konnte. In Linux startet der Ausdruck auf den Drucker in der Regel erst, wenn die komplette Datei in der Druckerwarteschlange abgelegt ist. Daher ist ein Abbruch noch möglich, ohne dass bereits Teile des Dokuments gedruckt sind.

Im *QPrinter*-Objekt können auch weitere Einstellungen vorgenommen werden, die in der eigenen Applikation abgefragt und berücksichtigt werden müssen.

Zum einen kann man mit *setFromTo* den Bereich der Seiten einstellen, der gedruckt werden soll. Wenn Ihr Programm ein mehrseitiges Dokument ausdruckt, so sollte es mit den Methoden *fromPage* und *toPage* die eingestellten Werte auslesen und nur die entsprechenden Seiten ausgeben. Mit der Methode *setPageOrder* kann die Reihenfolge des Ausdrucks auf den Wert *QPrinter::FirstPageFirst* (die erste Seite zuerst) oder den Wert *QPrinter::LastPageFirst* (die letzte Seite zuerst) eingestellt werden. Ihre Applikation sollte den eingestellten Wert mit *pageOrder* auslesen. Ist *LastPageFirst* eingestellt, so sollten die Seiten in umgekehrter Reihenfolge ausgegeben werden.

Qt stellt ein Dialogfenster zur Verfügung, mit dem viele dieser Einstellungen ganz einfach vom Benutzer vorgenommen werden können. Dieses Dialogfenster ist in der Klasse *QPrintDialog* implementiert (siehe Kapitel 3.7.8, *Dialoge*). In ihm kann der Anwender den Drucker aus der Liste der im System installierten Drucker wählen. Er kann auch die Ausgabe in eine Datei umleiten. Weiterhin kann er den Bereich der auszudruckenden Seiten festlegen, die Reihenfolge des Ausdrucks, ob in Farbe oder Graustufen gedruckt werden soll, die Anzahl der Kopien, die Papiergröße und die Orientierung. Um dieses Dialogfenster zu nutzen, gibt es zwei Möglichkeiten: Entweder benutzen Sie die statische Methode *QPrintDialog::getPrinterSetup*. Dieser Methode übergeben Sie einen Zeiger auf das einzustellende *QPrinter*-Objekt. Besser ist es jedoch, im *QPrinter*-Objekt die Methode *setup* zu benutzen. Diese Methode ruft das *QPrintDialog*-Fenster auf und übernimmt die Einstellungen. Setzen Sie dabei im *QPrinter*-Objekt mit der Methode *setMinMax* den Seitenbereich, den Ihr Dokument hat. Der Anwender kann dann für den zu druckenden Seitenbereich nur einen Ausschnitt aus diesem Seitenbereich wählen. In Ihrem Programm könnte zum Beispiel der Slot, der beim Menüpunkt DATEI | DRUCKEN... aktiviert wird, folgendermaßen aussehen:

```
void MainWindow::filePrint ()
{
    // Druckerobjekt erzeugen
    QPrinter prn;

    // Möglicher Seitenbereich von 1 bis lastPage
    prn.setMinMax (1, lastPage);

    // Dialogfenster für Einstellungen öffnen
    // Bei Rückgabewert false hat der Anwender den Dialog
    // abgebrochen.
    if (prn.setup())
    {
        int numberOfPages = prn.toPage() - prn.fromPage() + 1;
        for (int i = 0; i < numberOfPages; i++) {
            int printNow;    // nächste zu druckende Seite
            if (prn.pageOrder() == QPrinter::FirstPageFirst)
                printNow = prn.fromPage() + i;
```

```

else
    printNow = prn.toPage() - i;
    // Ausgabe der Seite printNow
    ...

    // Seitenvorschub
    prn.newPage();
}
}
}

```

4.18 Dateizugriffe

Das Öffnen, Lesen und Schreiben von Dateien ist zwar bereits im C- und C++-Standard definiert, jedoch ergeben sich oft dennoch Unterschiede zwischen verschiedenen Betriebssystemen. Aus diesem Grund ist in Qt eine Reihe von Klassen definiert, die einen vollständig plattformunabhängigen Zugriff ermöglichen. Durch das konsequent angewandte Klassenkonzept ist der Umgang mit Dateien und Verzeichnissen sehr komfortabel.

Tabelle 4-15 zeigt eine Liste der wichtigsten Klassen. In den weiteren Kapiteln werden diese näher beschrieben.

Klasse	Anwendung
<i>QIODevice</i>	abstrakte Klasse, repräsentiert ein Ein-/Ausgabe-Gerät
<i>QFile</i>	elementare Zugriffe auf eine Datei
<i>QFileInfo</i>	Informationen über Zugriffsrechte, Dateart und Modifikationszeitpunkt
<i>QDir</i>	Verzeichnisinhalt auslesen
<i>QTextStream</i>	Stream-Funktionalität für Textdateien
<i>QDataStream</i>	Stream-Funktionalität für Binärdateien
<i>KTempFile</i>	(KDE) temporäre Datei
<i>KSaveFile</i>	(KDE) sichere, atomare Datei

Tabelle 4-15 Wichtige Klassen für Dateizugriffe

4.18.1 Ein-/Ausgabe-Geräte und elementare Zugriffe

Qt definiert eine abstrakte Klasse *QIODevice*, die ein Ein-/Ausgabe-Gerät repräsentiert. Ein konkretes Gerät wird durch eine Unterklasse von *QIODevice* implementiert. Qt definiert bereits einige eigene Unterklassen: *QFile* (Datei), *QBuffer* (Speicherbereich), *QSocket* und *QSocketDevice* (Socket-Verbindungen).

Allen Ein-/Ausgabe-Geräten ist gemeinsam, dass man sie öffnet und schließt und dass man im geöffneten Zustand Daten lesen oder schreiben kann. Durch diese

gemeinsame abstrakte Klasse *QIODevice* können Sie Funktionen oder Methoden schreiben, die komplexere Zugriffe auf ein Ein-/Ausgabe-Gerät vornehmen, ohne genau zu wissen, ob es sich dabei um eine Datei oder ein anderes Gerät handelt. Hier folgt ein erstes Beispiel für eine Funktion, die die Daten aus einem Gerät liest und die Anzahl der Leerzeichen zählt:

```
int countSpaces (QIODevice *dev)
{
    int counter = 0;
    dev->open (IO_ReadOnly); // Gerät zum Lesen öffnen
    while (!dev->atEnd())    // Solange noch Daten
    {
        char ch = dev->getch(); // Ein Zeichen lesen
        if (ch == ' ')
            counter++;
    }
    dev->close();           // Gerät schließen
    return counter;
}
```

Dieser Funktion übergeben Sie zum Beispiel als Parameter die Adresse eines *QFile*-Objekts. Ebenso gut könnten Sie aber auch ein Objekt einer anderen Unterklasse von *QIODevice* benutzen.

Verschiedene Geräte haben unterschiedliche Eigenschaften. *QIODevice* stellt eine Reihe von Methoden zur Verfügung, um die wichtigsten Eigenschaften eines Geräts in Erfahrung zu bringen:

- Bei einigen Geräten kann man die Position des »virtuellen Schreib-Lese-Kopfes« verändern (beispielsweise bei Dateien), andere kann man nur nacheinander auslesen bzw. beschreiben (beispielsweise Socket-Verbindungen). Sie können mit *isDirectAccess* bzw. *isSequentialAccess* prüfen, welcher der beiden Typen vorliegt. Auch eine Zwischenlösung, die Sie mit *isCombinedAccess* abfragen, kann existieren, wird zur Zeit aber noch nicht benutzt. Besitzt ein Gerät *directAccess*, so können Sie einige Methoden von *QIODevice* benutzen: Sie können die aktuelle Position mit der Methode *at (int)* versetzen. Mit *at ()* (ohne Parameter) erfragen Sie die aktuelle Position. *size()* liefert Ihnen die Gesamtgröße der Daten im Gerät in Byte (also für *QFile* beispielsweise die Dateigröße).
- Bei einigen Geräten werden die Daten vor dem Schreiben zwischengespeichert, so dass größere Datenblöcke geschrieben werden können. Sie können erfragen, ob dieses für ein Gerät der Fall ist, indem Sie *isBuffered* prüfen. Ist ein Gerät vom Typ *isBuffered*, können Sie mit der Methode *flush* das Schreiben der noch gespeicherten Daten erzwingen. Bei vielen Geräten können Sie in der Methode *open* festlegen, ob die Daten zwischengespeichert werden oder nicht (siehe unten).

- Bei einigen Geräten werden Daten beim Lesen oder Schreiben sofort verarbeitet (beispielsweise Dateien), bei anderen Geräten kann der Vorgang längere Zeit in Anspruch nehmen. Zum Teil muss sogar gewartet werden, bis neue Daten vorliegen (beispielsweise beim Lesen von *stdin* oder von einem Socket). Die Methode *isSynchronous* liefert *true* zurück, falls keine Verzögerungen zu befürchten sind.
- Einige Geräte können nur gelesen, andere nur geschrieben werden, wieder andere kann man lesen und schreiben. Die Methoden *isReadable*, *isWritable* und *isReadWrite* geben an, was für ein bestimmtes Gerät möglich ist. (*isReadWrite* ist nur die UND-Verknüpfung der Ergebnisse von *isReadable* und *isWritable*.) In der Regel legen Sie beim Öffnen eines Geräts mit *open* fest, welche Zugriffe erlaubt sind (siehe unten).
- Einige Geräte bieten die Möglichkeit, die in MS-DOS üblichen Zeilenwechsel (die aus einer CR/LF-Kombination bestehen) in die unter Unix üblichen LF-Zeichen umzuwandeln. Ob das Gerät diese Umsetzung vornimmt, können Sie mit *isTranslated* erfragen. Ob diese Umsetzung vorgenommen werden soll, legen Sie in der Regel beim Öffnen des Geräts mit *open* fest; ob das Gerät diese Umsetzung aber auch tatsächlich unterstützt, hängt von der Unterklasse selbst ab. *QIODevice* enthält keinerlei Code für die Unterstützung dieser Umsetzung.

4.18.2 Dateiinformationen

Die Klasse *QFileInfo* kann benutzt werden, um Informationen über Dateien zu erhalten. Die interessantesten Methoden sind in Tabelle 4.16 aufgelistet.

Methoden	Ergebnis
<i>exists</i>	prüft, ob die Datei oder das Verzeichnis existiert
<i>isFile</i>	<i>true</i> , falls »normale« Datei
<i>isDir</i>	<i>true</i> , falls Verzeichnis
<i>isSymLink</i>	<i>true</i> , falls symbolischer Link
<i>readLink</i>	liefert das Ziel eines symbolischen Links
<i>baseName</i>	Dateiname ohne Endung
<i>extension</i>	Dateiendung
<i>lastModified</i>	Zeitpunkt der letzten Dateiänderung
<i>lastRead</i>	Zeitpunkt des letzten Lesezugriffs (wird nicht von allen Systemen unterstützt)
<i>size</i>	Größe der Datei
<i>isReadable</i>	prüft, ob Leserechte (für den aktuellen User) gegeben sind

Tabelle 4-16 Methoden der Klasse *QFileInfo*

Methoden	Ergebnis
<i>isWritable</i>	prüft, ob Schreibrechte (für den aktuellen User) gegeben sind
<i>isExecutable</i>	prüft, ob Ausführungsrechte (für den aktuellen User) gegeben sind
<i>owner</i>	Besitzer der Datei
<i>group</i>	Benutzergruppe, der die Datei zugeordnet ist
<i>permission</i>	genaue Zugriffsrechte für <i>Owner</i> , <i>Group</i> und <i>Others</i>

Tabelle 4-16 Methoden der Klasse *QFileInfo* (Forts.)

Um an die Informationen zu einer bestimmten Datei zu gelangen, erzeugen Sie einfach ein *QFileInfo*-Objekt und übergeben dem Konstruktor den Dateinamen als Parameter. Alternativ können Sie hier auch ein *QFile*-Objekt benutzen. Anschließend können Sie die Methoden aus Tabelle 4.16 anwenden.

Das folgende kleine Beispiel prüft, ob die Unix-Datei */etc/passwd*, die Informationen über die eingetragenen Benutzer enthält, korrekt gegen Manipulation geschützt ist. Dazu muss sie als Besitzer und Gruppe *root* enthalten, und Schreibzugriff darf nur der Besitzer haben.

```
QFileInfo passwd ("/etc/passwd");
if (!passwd.exists())
{
    qDebug ("Datei /etc/passwd existiert nicht!");
}
else
{
    if (passwd.owner() != "root")
        qDebug ("Die Datei sollte als Besitzer root haben!");
    if (passwd.group() != "root")
        qDebug ("Die Datei sollte als Gruppe root haben!");
    if (passwd.permission(QFileInfo::WriteGroup) ||
        passwd.permission(QFileInfo::WriteOther))
        qDebug ("Nur der Besitzer darf Schreibrechte haben!");
}
```

4.18.3 Verzeichnisverwaltung

Speziell um sich den Inhalt von Verzeichnissen anzuschauen, definiert Qt die Klasse *QDir*. Sie bietet Methoden für vier verschiedene Anwendungsbereiche:

- Analyse, Vervollständigung und Vereinfachung von Verzeichnispfaden
- Inhalt eines Verzeichnisses ausgeben
- Anlegen und Löschen von Unterverzeichnissen
- Ermitteln von Hauptverzeichnis, Home-Verzeichnis und aktuellem Verzeichnis

An einem einfachen Beispiel wollen wir zeigen, wie der Inhalt eines Verzeichnisses ausgelesen werden kann. Das folgende Beispiel durchsucht das aktuelle Verzeichnis mit allen Unterverzeichnissen und gibt eine Liste aller gefundenen Dateien mit der Endung ».mp3« zurück.

```
#include <qdir.h>
void addMp3Files (QDir dir, QStringList &files)
{
    // Alle Unterverzeichnisse finden (auch die, die
    // nicht auf .mp3 enden)
    start.setMatchAllDirs (true);

    // Liste mit allen Einträgen ermitteln
    QFileInfoList *list;
    list = dir.entryInfoList (*.mp3);
    QFileInfoListIterator iter (*list);
    while (iter.current())
    {
        // in fname speichern wir den gefundenen Dateinamen
        // (inklusive Verzeichnis) ab
        QString fname = (*iter).filePath();
        if ((*iter).isDir())
        {
            // In Unterverzeichnisse rekursiv hineingehen
            addMp3Files (QDir (fname), files);
        }
        else
        {
            // alle anderen Einträge sind MP3-Dateien
            files.append (fname);
        }
        ++iter;
    }
}

int main()
{
    QStringList list;
    // Liste der MP3-Dateien im aktuellen
    // Verzeichnis und in den Unterverzeichnissen
    addMp3Files (QDir::current(), list);

    // Liste ausgeben
    QStringList::Iterator it;
    for (it = list.begin(); it != list.end(); ++it)
        qDebug ("File: %s", (*it).ascii());
}
```

In diesem Beispiel haben wir die Methode *QDir::entryInfoList* benutzt, die einen Zeiger auf eine Liste von *QFileInfo*-Objekten zurückliefert. Diese Objekte benötigen wir hier, um zwischen Verzeichnissen und anderen Dateien zu unterscheiden. Damit wir alle Verzeichnisse (und nicht nur die Verzeichnisse, die auf den Namensfilter **.mp3* passen) erhalten, muss am Anfang die Methode *setMatchAllDirs (true)* aufgerufen werden. Werden dagegen nur die Dateinamen benötigt, so kann man auch die Methode *QDir::entryList* benutzen. Wenn wir beispielsweise die rekursiven Aufrufe für die Unterverzeichnisse von der Suche nach den MP3-Dateien trennen, kann die Funktion auch so aussehen:

```
void addMp3Files (QDir dir, QStringList &files)
{
    // Iterator-Objekt wird später zum Durchlaufen
    // der QStringList-Objekte benötigt
    QStringList::Iterator it;

    // Zunächst nur alle MP3-Dateien im Verzeichnis dir
    // finden
    QStringList fList = dir.entryList ("*.mp3",
                                       QDir::Files);

    // Liste durchlaufen
    for (it = fList.begin(); it != fList.end(); ++it)
        // Jedem Dateinamen den absoluten Pfad voranstellen
        files += dir.absFilePath (*it);

    // Anschließend die Liste aller Unterverzeichnisse
    // ermitteln und für jedes Unterverzeichnis die
    // Methode rekursiv aufrufen
    QStringList dList = dir.entryList (QDir::Dirs);
    while (it = dList.begin(); it != dList.end(); ++it)
        addMp3Files (QDir (dir.absFilePath (*it)), files);
}
```

Diese zweite Implementierung ist etwas übersichtlicher, aber auch etwas langsamer. Für effiziente Programme ist daher die erste Lösung vorzuziehen.

4.18.4 Stream-basierte Ein-/Ausgabe

QFile sowie die anderen Unterklassen von *QIODevice* bieten zunächst nur die Möglichkeit, die Daten in der Datei byteweise oder blockweise zu schreiben oder zu lesen. Komfortabler ist es jedoch, mit einem Stream-Konzept (deutsch: Strom) zu arbeiten, in dem die einzelnen Datenelemente nacheinander in den Stream geschrieben bzw. aus ihm gelesen werden, wobei als Datenelement nicht nur ein Datenblock, sondern auch ein Objekt von (nahezu) beliebigem Typ in Frage kommt. Qt bietet als Zusatz zu *QIODevice* zwei solcher Stream-Klassen an: *QTextStream* für Textdateien und *QDataStream* für Binärdateien. In beiden Klas-

sen sind für alle interessanten Datentypen die Operatoren << und >> überladen. Mit diesen können wie bei den C++-Streams Daten in den Stream geschrieben oder aus dem Stream gelesen werden.

Die Klasse *QTextStream* wird fast genauso benutzt wie die Klasse *iostream* von C++. Ein wichtiger Unterschied besteht darin, dass *QTextStream* auf einem *QIODevice*-Objekt arbeitet. Daher kann es nicht nur mit Dateien oder der Standard- und -ausgabe benutzt werden, sondern zum Beispiel auch mit einem internen Datenbereich, mit einem Socket oder mit einer selbst geschriebenen Unterklasse von *QIODevice*.

In der Regel geben Sie im Konstruktor von *QTextStream* einen Zeiger auf ein *QIODevice*-Objekt an. Speziell für Dateien können Sie aber auch ein *QFile*-Objekt angeben. Anschließend wenden Sie die Operatoren << und >> an, um Daten in die Textdatei zu schreiben oder aus ihr zu lesen. Dabei sind als Datentypen einzelne Zeichen (*char*, *QChar*), Zeichenketten (*char**, *QString*, *QString*), und Zahlen (*short*, *int*, *long*, *float* und *double*) möglich. Bei der Ausgabe von Zahlen werden diese als Text dargestellt. Die Ausgabe kann dabei durch bestimmte Flags manipuliert werden, z.B. in der Gesamtbreite in Zeichen (*width*) oder der Anzahl der Nachkommastellen (*precision*). Beim Einlesen von Zahlen aus einem Strom werden so lange einzelne Zeichen vom Stream gelesen, wie sie sinnvoll als Zahl interpretiert werden können.

Speziell für das Abspeichern oder Einlesen von Unicode-Daten kann *QTextStream* sehr gut genutzt werden. Standardmäßig benutzt *QTextStream* den lokalen Zeichensatz (also für Deutschland beispielsweise *Latin1*), sie kann aber mit der Methode *setEncoding* auf ein anderes Format umgestellt werden. Diese Methode sollte vor dem ersten Aufruf zum Schreiben oder Lesen benutzt werden. Für echte Unicode-Daten verwenden Sie am besten die Werte *Unicode* (für 16-Bit-kodiertes Unicode) oder *UnicodeUTF8* (für 8-Bit-kodiertes Unicode, ASCII-kompatibel). Nähere Informationen zu Unicode finden Sie in Kapitel 4.8, *Der Unicode-Standard*. Diese Fähigkeit haben wir auch in unserem Beispiel in Kapitel 3.5.6, *Applikation für ein Dokument*, benutzt. Der Texteditor, den wir dort entwickelt haben, speichert und liest seine Dateien im UTF-8-Format.

Wollen Sie größere Datenmengen in einer Datei speichern und sie nachher wieder (unverfälscht) einlesen, so ist *QTextStream* nicht sehr gut geeignet: Zahlen, die als Text abgespeichert werden, belegen mehr Speicherplatz, und beim Einlesen kann es dazu kommen, dass die Daten anders interpretiert werden, als sie gespeichert wurden. Außerdem ist die Umwandlung in Textdaten ineffizient.

Zum Speichern größerer Datenmengen bietet Qt daher die Klasse *QDataStream* an. Mit dieser Klasse können Datentypen direkt als Binärdaten in einer Datei abgelegt und auch wieder eingelesen werden. Außerdem ist für eine enorme Anzahl von Datentypen das Abspeichern und Einlesen bereits definiert (siehe Tabelle 4.17).

<code>Q_INT8</code>	<code>double</code>	<code>QColorGroup</code>	<code>QRegion</code>	<code>QDate</code>
<code>Q_UINT8</code>	<code>char*</code>	<code>QPalette</code>	<code>QSize</code>	<code>QTime</code>
<code>Q_INT16</code>	<code>QBrush</code>	<code>QPen</code>	<code>QVariant</code>	<code>QDateTime</code>
<code>Q_UINT16</code>	<code>QColor</code>	<code>QPixmap</code>	<code>QWMatrix</code>	<code>QMap</code>
<code>Q_INT32</code>	<code>QCursor</code>	<code>QPoint</code>	<code>QBitArray</code>	<code>QString</code>
<code>Q_UINT32</code>	<code>QFont</code>	<code>QPointArray</code>	<code>QByteArray</code>	<code>QValueList</code>
<code>float</code>	<code>QImage</code>	<code>QRect</code>	<code>QString</code>	

Tabelle 4-17 Datentypen, die von *QDataStream* unterstützt werden

QDataStream bietet außerdem den großen Vorteil, dass das Format, in dem die Daten abgelegt werden, plattformunabhängig ist. Dateien, die Sie auf einem Rechner abgespeichert haben, können Sie auf einem anderen wieder einlesen, auch wenn sich die interne Repräsentation der Daten auf den Rechnern unterscheidet. Somit eignet sich *QDataStream* auch, um Datendokumente (Datenbanken und Ähnliches) abzuspeichern, oder um Daten zwischen einem Client und einem Server über eine Socket-Verbindung auszutauschen.

Als Anwendungsbeispiel wollen wir eine kleine Datenbank betrachten, in der Personendaten (bestehend aus Name, Geburtsdatum und Jahreseinkommen) abgelegt werden. Die Daten werden in einer Datei abgelegt und beim Starten des Programms in den Speicher gelesen. Beim Beenden werden sie wieder in die Datei zurückgeschrieben. (Das geht natürlich nur, wenn die Datenmenge nicht zu groß wird.)

Um die Daten einer Person abzulegen, definieren wir eine eigene Klasse *Person* mit drei Attributen. Der Einfachheit halber überladen wir zusätzliche Operatoren `<<` und `>>`, um ein *Person*-Objekt in einem *QDataStream* abzuspeichern oder auszulesen. In diesen Operatoren benutzen wir wiederum die bereits definierten Operatoren `<<` und `>>`, um die einzelnen Bestandteile einer Person (Name = *QString*, Geburtsdatum = *QDate*, Jahreseinkommen = *double*) abzulegen bzw. einzulesen. Im Speicher legen wir die gesamte Liste der Personen in einer *QValueList* ab, so dass recht einfach neue Personen hinzugefügt oder alte gelöscht werden können. Das Abspeichern der Datenbank besteht nun nur noch aus dem Abspeichern der *QValueList*, ist also nur noch eine Zeile. Beachten Sie auch, dass das Dateiformat unserer Datenbank eindeutig und plattformunabhängig definiert ist, also auf jedem Rechner auch wieder eingelesen werden kann.

```
// Diese Klasse speichert Daten zu einer Person
class Person
{
    ...
    // Die Attribute, die die Daten einer Person speichern
    private:
```

```
    QString name;
    QDate geburtsdatum;
    double jahreseinkommen;

    // Die Operatoren << und >> werden als friend deklariert,
    // damit wir auf die Attribute zugreifen können.
    friend QDataStream &operator<< (QDataStream&, const Person&);
    friend QDataStream &operator>> (QDataStream&, Person&);
};

// Der Operator zum Abspeichern einer Person
QDataStream &operator<< (QDataStream &stream, const Person &p)
{
    stream << p.name
        << p.geburtsdatum
        << p.jahreseinkommen;
    return stream;
}

// Der Operator zum Einlesen einer Person
QDataStream &operator>> (QDataStream &stream, Person &p)
{
    stream >> p.name >> p.geburtsdatum >> p.jahreseinkommen;
    return stream;
}

// Diese Klasse verwaltet die Datenbank
class Datenbank
{
public:
    Datenbank ();
    ~Datenbank ();
private:
    QList<Person> daten;
};

// Konstruktor der Datenbank, lädt Daten aus einer Datei
Datenbank::Datenbank ()
{
    // Datenbankdatei öffnen,
    // falls nicht vorhanden, leere Datenbank
    QFile dbfile ("persondb.dat");
    if (!dbfile.exists())
        return;
    dbfile.open (IO_ReadOnly);

    // Stream zur Datei anlegen
    QDataStream stream (dbfile);
```



```
// Daten einlesen (komplette QList<Person>)
stream >> daten;
}

// Destruktor der Datenbank, speichert Daten in Datei
Datenbank::~Datenbank ()
{
    // Datenbankdatei zum Schreiben öffnen,
    QFile dbfile ("persondb.dat");
    dbfile.open (IO_WriteOnly | IO_Truncate);

    // Stream zur Datei anlegen
    QDataStream stream (dbfile);

    // Daten schreiben (komplette QList<Person>)
    stream << daten;
}
```

Das vollständige Listing (mit grafischer Oberfläche) finden Sie auf der CD, die dem Buch beiliegt. Weitere Informationen zum Thema *QValueList* finden Sie in Kapitel 4.7.2, *Container-Klassen*. Zum Schreiben der Datenbank eignet sich außerdem besser die Klasse *KSaveFile* (statt *QFile*), die in Kapitel 4.18.5, *Temporäre und atomare Dateien*, vorgestellt wird.

4.18.5 Temporäre und atomare Dateien

Auch die KDE-Bibliotheken stellen zwei Klassen zur Verfügung, die für den Zugriff auf Dateien sehr gute Dienste leisten können.

Falls Sie zum Abspeichern von Zwischenergebnissen vorübergehend eine temporäre Datei öffnen wollen, können Sie dazu auch die Klasse *KTempFile* benutzen. Sie brauchen sich bei dieser Klasse nicht um einen noch nicht vorhandenen Dateinamen oder ein geeignetes Verzeichnis zu kümmern, das erledigt *KTempFile* für Sie. Das folgende Beispiel verdeutlicht die Anwendung dieser Klasse:

```
// temporäre Datei anlegen
KTempFile temporary;
QFile *f = temporary.file();

// arbeiten mit f, z.B. mit QDataStream oder QTextStream
f->open (IO_ReadWrite);
// ...

// temporäre Datei wieder löschen
temporary.unlink();
```

Beachten Sie, dass Sie die temporäre Datei nicht unbegrenzt lange nutzen sollten. Daten, die Sie bis zum nächsten Aufruf des Programms zwischenspeichern wollen, sollten beispielsweise nicht in einer temporären Datei abgelegt werden, da diese in der Regel in das Verzeichnis */tmp* gelegt wird. Einige Unix-Systeme löschen regelmäßig alle Dateien in diesem Verzeichnis, die für eine bestimmte Zeit nicht mehr benutzt worden sind.

Vor einem anderen Problem stehen Sie, wenn Sie in Ihrem Programm ein Dokument verändern und anschließend das geänderte Dokument wieder in der gleichen Datei speichern wollen. (Dieses ist der typische Vorgang bei einem Befehl *DATEI – SPEICHERN*.) Treten während des Speicherns Fehler auf, so ist der alte Inhalt der Datei verloren, und die neuen Daten sind ebenfalls nicht korrekt geschrieben worden. Eine sicherere Vorgehensweise ist es daher, die neuen Daten zunächst in einer Datei mit einem anderen Namen abzulegen, anschließend erst die alte Datei zu löschen und dann die neue Datei umzubenennen. Man spricht in diesem Zusammenhang auch von einem atomaren (unteilbaren) Zugriff: Entweder sind die neuen Daten vollständig geschrieben, oder die alten sind unverändert erhalten geblieben. Ein Zwischenstadium existiert nicht. Genau diese Vorgehensweise benutzt die Klasse *KSaveFile*. Das folgende Beispiel zeigt, wie Sie diese Klasse einsetzen können.

```
KSaveFile file ("database.dat");
QDataStream *stream = file.dataStream();
(*stream) << ...
(*stream) << ...
if (!file.close())
{
    // Speichern schlug fehl, Originaldatei
    // ist erhalten geblieben
}
```

Neben der Methode *dataStream*, die einen Zeiger auf ein *QDataStream*-Objekt zurückliefert, kann man auch mit *file* ein *QFile*-Objekt oder mit *textStream* ein *QTextStream*-Objekt erhalten.

4.19 Netzwerkprogrammierung

Sowohl die Qt-Bibliothek als auch die KDE-Bibliotheken bieten einige Klassen an, um den Zugriff auf ein vorhandenes Netzwerk auf komfortable Art zu ermöglichen. Insbesondere ist es möglich, Programme unabhängig von der verwendeten Plattform und vom Betriebssystem zu schreiben.

Die Netzwerkklassen von Qt wurden erst mit Version Qt 2.2 in die allgemeine Bibliothek aufgenommen. Sie sind im Modul *Network* enthalten. Beachten Sie also, dass beim Kompilieren der Qt-Bibliothek dieses Modul eingebunden wurde.

Dies ist standardmäßig der Fall, und wird nur dann gezielt abgeschaltet, wenn es beim Kompilieren Schwierigkeiten gibt. Wenn Sie mit einer älteren Version als Qt 2.2 arbeiten, müssen Sie leider auf die Netzwerkklassen verzichten.

In den folgenden Kapiteln werden die wichtigsten Klassen kurz vorgestellt. Dabei gehen wir von den hardware-nahen Klassen zu den Klassen der oberen Protokollschichten vor.

4.19.1 Socket-Verbindungen

Eine Socket-Verbindung ist eine Punkt-zu-Punkt-Verbindung zwischen zwei Prozessen, die über ein Netzwerk verbunden sind. Es gibt verschiedene Arten von Sockets. Am häufigsten werden TCP/IP-Sockets benutzt, die die Verbindung über das Internet oder ein Intranet herstellen. Daneben gibt es aber auch noch andere Socket-Typen, wie zum Beispiel Unix-Domain-Sockets, die zur Verbindungsherstellung das Dateisystem benutzen.

Weiterhin unterscheidet man zwischen Datagram-Sockets, die einzelne Datenpakete verschicken, und Stream-Sockets, die eine feste Verbindung zwischen zwei Prozessen aufbauen und über diese die Daten austauschen. Stream-Sockets werden in der Praxis häufiger genutzt, da sie Mechanismen gegen Datenverfälschung und Datenverlust bieten.

Die Socket-Unterstützung in den Qt- und KDE-Bibliotheken ist dementsprechend am besten für Stream-Sockets auf TCP/IP-Basis ausgebaut, aber auch andere Socket-Arten können mit diesen Klassen zusammen genutzt werden. Auch in diesem Kapitel werden wir uns hauptsächlich den TCP/IP-Sockets im Stream-Modus widmen.

Tabelle 4.18 zeigt eine Übersicht über die von Qt und KDE zur Verfügung gestellten Klassen zu Socket-Verbindungen.

Klasse	Beschreibung
<i>QDns</i>	einfache Klasse zum Zugriff auf einen Name-Server (nicht-blockierend)
<i>QSocketDevice</i>	einfache Klasse zur Abstraktion eines Sockets
<i>QSocket</i>	Socket-Klasse für Clients, Unterklasse von <i>QIODevice</i>
<i>QServerSocket</i>	Socket-Klasse für Server, erstellt neue Verbindungs-Sockets
<i>KSocket</i>	wie <i>QSocket</i> , aber nicht von <i>QIODevice</i> abgeleitet
<i>KSeverSocket</i>	wie <i>QServerSocket</i> , erzeugt <i>KSocket</i> -Objekte beim Verbindungsaufbau

Tabelle 4-18 Klassen zur Socket-Unterstützung

Eine Socket-Verbindung kann wie eine Datei benutzt werden: Man erhält vom Betriebssystem einen Deskriptor und kann unter Angabe dieses Deskriptors Daten in den Socket hineinschreiben oder aus dem Socket auslesen. Stream-Socket-Verbindungen sind dabei immer bidirektional: Das, was der eine Prozess in den Stream hineinschreibt, kann der Prozess am anderen Ende der Stream-Verbindung herauslesen und umgekehrt.

Hardware-nahe Klassen

Die Klassen *QDns* und *QSocketDevice* bieten eine recht hardwarenahe, aber dennoch plattformunabhängige Zugriffsmöglichkeit. *QDns* erlaubt dabei den Zugriff auf einen Name-Server, um die IP-Adresse aus einem symbolischen Rechnernamen zu ermitteln. So kann beispielsweise der Name des WWW-Server-Rechners »www.trolltech.com« in die IP-Adresse dieses Rechners umgewandelt werden. *QDns* stellt die Anfrage an den Name-Server dabei asynchron, d.h. ohne den eigenen Prozess zu blockieren, bis die Antwort vom Name-Server eingetroffen ist.

QSocketDevice ist eine einfache Kapselungsklasse für einen Socket. Über Methoden haben Sie alle Zugriffsmöglichkeiten für den Socket (*bind*, *listen*, *close* und andere). So haben Sie eine komfortablere Schnittstelle, als wenn Sie direkt mit den Betriebssystemfunktionen arbeiten müssten. Um diese Klasse in bereits bestehende Programme einzubauen, gibt es zum einen die Möglichkeit, schon existierende Socket-Deskriptoren in ein *QSocketDevice*-Objekt abzulegen (mit der Methode *setSocket*), und umgekehrt die Möglichkeit, den Deskriptor des Sockets im *QSocketDevice*-Objekt abzufragen (mit der Methode *socket*). *QSocketDevice* unterstützt sowohl Datagram- als auch Stream-Sockets.

Sowohl *QDns* als auch *QSocketDevice* sind für Sie nur dann interessant, wenn Sie sich mit Socket-Programmierung bereits gut auskennen.

Klassen für Client-Server-Verbindungen

Die häufigste Anwendung von Streams ist eine Client-Server-Architektur: Ein Server stellt einen Dienst zur Verfügung und bietet dazu einen so genannten Port an, an den die Clients, die diesen Dienst nutzen wollen, einen Verbindungswunsch herantragen können. Durch die Annahme dieses Verbindungswunsches kommt die Stream-Socket-Verbindung zustande. Über diese Verbindung sendet der Client die Daten für eine Anfrage zum Server, und dieser sendet nach Bearbeitung der Anfrage die Ergebnisse über den Socket zurück an den Client. Der Server kann dabei auch mehrere Verbindungen zu verschiedenen Clients aufbauen.

Damit der Client den Server ansprechen kann, muss er dessen IP-Adresse sowie die Port-Nummer wissen, unter der der Server die Verbindungswünsche erwartet. Für die Standard-Server sind diese Port-Nummern fest vergeben. So befindet sich zum Beispiel ein FTP-Server (ftp-Protokoll) hinter der Port-Nummer 21, ein

WWW-Server (http-Protokoll) hinter Port 80, ein Mail-Server zum Versenden (smtp-Protokoll) hinter Port 25, ein Mail-Server zum Abholen von Mails (pop3-Protokoll) hinter Port 110. Nicht belegte Ports können frei benutzt werden, wobei natürlich Port-Kollisionen zu vermeiden sind.

Sowohl Qt als auch KDE bieten Klassen an, die die Implementierung eines Clients oder eines Servers einfach machen. Wie in vielen anderen Fällen auch sind die KDE-Klassen bereits älter (schon in KDE 1.1), die Qt-Klassen erst später hinzugekommen (ab Qt 2.2). Ob die KDE-Klassen auch in Zukunft noch benutzt werden, ist noch nicht genau abzusehen, eventuell werden sie von den Qt-Klassen vollständig verdrängt werden. Die Qt-Klassen bieten etwas mehr Funktionalität und eine bessere Dokumentation, weshalb sie bei neueren Projekten den Vorzug erhalten sollten.

Im Folgenden werden wir nur die beiden Qt-Klassen *QSocket* und *QServerSocket* besprechen. Für die KDE-Klassen *KSocket* und *KServerSocket* konsultieren Sie bitte die Klassendokumentation.

QSocket bei Clients

Für die Implementierung eines Clients kann die Klasse *QSocket* benutzt werden. Die Anwendung ist sehr einfach: Zunächst wird ein Objekt der Klasse *QSocket* erzeugt, und anschließend wird die Methode *connectToHost* aufgerufen. Dieser Methode wird dabei als erster Parameter der Rechnername und als zweiter Parameter die Port-Nummer des Servers mitgeteilt. Das *QSocket*-Objekt baut nun selbstständig eine Verbindung zum Server auf und schickt anschließend entsprechende Signale: *connected()* nach erfolgreichem Verbindungsaufbau und *error(int)*, falls ein Fehler aufgetreten ist, zum Beispiel weil der Rechnername keiner IP-Adresse zugeordnet werden konnte oder auf dem angegebenen Rechner mit dieser Port-Nummer kein Server installiert ist. Der entsprechende Code kann beispielsweise folgendermaßen aussehen:

```
QSocket *socket = new QSocket ();
connect (socket, SIGNAL (connected()),
        SLOT (verbindungFertig()));
connect (socket, SIGNAL (error(int)),
        SLOT (verbindungFehler(int)));
socket->connectToHost ("www.trolltech.com", 80);
```

Der Verbindungsaufbau zum Server geschieht auf jeden Fall blockierungsfrei. Ihr Programm läuft weiter und bleibt bedienbar. Erst wenn der Verbindungsaufbau beendet ist oder ein Fehler auftrat, wird das entsprechende Signal geschickt. Diese Eigenschaft der Klasse *QSocket* ist besonders bei Programmen mit grafischer Oberfläche wichtig, wie es auch in Kapitel 4.13, *Blockierungsfreie Programme*, erläutert wird. Die KDE-Klasse *KSocket* ist hier leider nicht ganz so komfortabel: Tritt beim Suchen des Rechnernamens im Name-Server ein Problem auf, bleibt

Ihr Programm blockiert. Der Anwender kann daher diesen Vorgang nicht abbrechen oder das Programm beenden, bevor nicht der Timeout – der oftmals 30 Sekunden oder länger beträgt – den Vorgang erfolglos abbricht.

Nachdem die Verbindung erfolgreich aufgebaut wurde, kann das *QSocket*-Objekt benutzt werden, um Daten zum Server zu schicken oder um Daten, die vom Server geschickt wurden, einzulesen. Da *QSocket* eine Unterklasse von *QIODevice* ist, kann man hier ganz einfach die Methoden *writeBlock* oder *readBlock* benutzen, oder – noch komfortabler – die Hilfsklassen *QTextStream* oder *QDataStream* verwenden.

Wollen Sie beispielsweise einen Textstring an den Server schicken, so können Sie folgenden Code benutzen:

```
QTextStream stream (socket); // Textstream zum Socket erzeugen
stream << "Textstring an den Server\n";
```

Diese Daten werden automatisch zwischengepuffert und gesendet, sobald der Socket wieder bereit ist. Mit der Methode *bytesToWrite* können Sie jederzeit ermitteln, wie viele Daten noch zwischengespeichert werden.

Um Daten aus dem Socket zu lesen, verbinden Sie einen eigenen Slot mit dem Signal *readyRead()*. Dieses Signal wird immer dann geschickt, wenn neue Daten vom Server angekommen sind. In diesem Slot lesen Sie dann die Daten aus. Sie können sich dabei mit der Methode *bytesAvailable* vorab informieren, wie viele Bytes an Daten zur Zeit gelesen werden können. Um zu testen, ob bereits eine ganze Text-Zeile (einschließlich des abschließenden Newline-Zeichens) im Socket-Puffer liegt, können Sie die Methode *canReadLine()* benutzen. Ist das noch nicht möglich, können Sie einfach auf den nächsten Aufruf Ihres Slots warten.

Um die Verbindung zum Server zu beenden, rufen Sie die Methode *close* auf. Falls Sie Daten in den Socket geschrieben haben, die noch nicht übertragen werden konnten, wartet das *QSocket*-Objekt, bis diese Daten geschrieben wurden, und beendet erst dann die Verbindung. So werden Datenverluste vermieden. Sie bekommen in diesem Fall das Signal *delayedCloseFinished()*. Nun können Sie das *QSocket*-Objekt problemlos löschen. Lagen dagegen keine zu schreibenden Daten vor, wird die Verbindung direkt getrennt. Sie erkennen das daran, dass die Methode *state()* den Wert *Idle* zurückliefert. Wollen Sie beispielsweise im Anschluss an die erfolgreiche Trennung der Verbindung den eigenen Slot *verbindungBeendet()* aufrufen, so kann das so aussehen:

```
socket->close();
if (socket->state() == QSocket::Idle)
{
    verbindungBeendet();
}
else
{

```

```
connect (socket, SIGNAL (delayedCloseFinished()),  
        SLOT (verbindungBeendet()));  
}
```

In jedem Fall können Sie auch das *QSocket*-Objekt jederzeit löschen. Ebenso können Sie eine neue Verbindung mit der Methode *connectToHost()* aufbauen. Die alte Verbindung wird dann auf der Stelle beendet. Dabei gehen allerdings noch zwischengespeicherte Daten verloren.

Als konkretes Beispiel werden wir uns einen Client anschauen, der den *finger*-Server eines Rechners kontaktiert. Auf Unix-Rechner bietet dieser *finger*-Server unter der Port-Nummer 79 die Möglichkeit, eine Liste der zur Zeit eingeloggten User zu ermitteln. Das dabei benutzte »Protokoll« ist sehr einfach: Nach dem Verbindungsaufbau schickt der Client die User-Namen, zu denen er genauere Informationen erhalten will, als ASCII-String über den Socket an den Server. Die Namen müssen dabei durch Leerschritte getrennt sein, und die Zeile muss mit einem Newline-Zeichen abgeschlossen sein. Schickt der Client einen leeren String, der nur aus dem Newline-Zeichen besteht, so fordert er damit eine Liste aller eingeloggten User an. Direkt im Anschluss daran sendet der Server die Ergebnisse wieder als ASCII-Text an den Client zurück.

Da unser Client eine grafische Oberfläche erhalten soll und wir außerdem eigene Slots definieren müssen, die mit den Signalen des Sockets verbunden werden, benutzen wir eine eigene Klasse. Im Folgenden sind nur die zentralen Stellen der Klasse aufgeführt. Das komplette Listing finden Sie auf der CD, die dem Buch beiliegt. Abbildung 4.55 zeigt unseren Client bei der Arbeit.

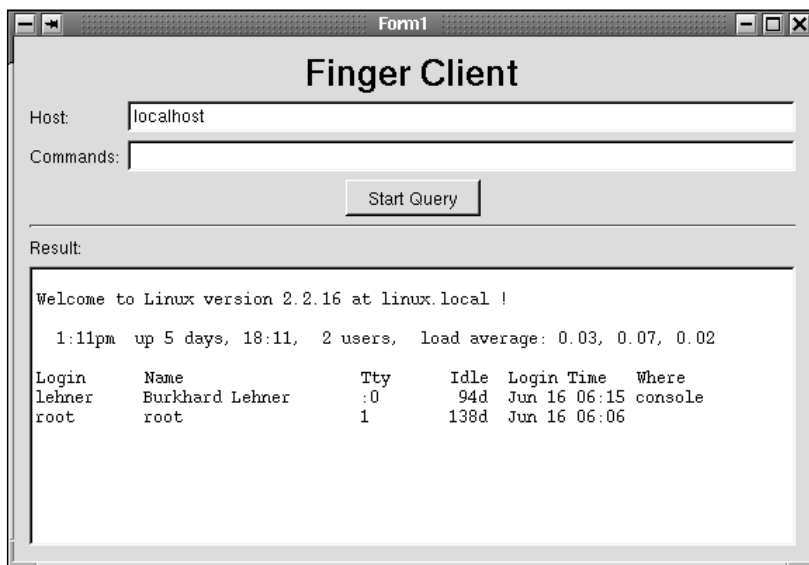


Abbildung 4-55 Der Finger-Client bei der Arbeit

```

class FingerImpl : public Form1
{
    Q_OBJECT
public:
    FingerImpl (QWidget* parent = 0, const char* name = 0);
protected slots:
    void startQuery();
    void sendCommand();
    void readResult();
    void error(int);
private:
    QSocket *socket;
};

// Konstruktor
FingerImpl::FingerImpl (QWidget* parent, const char* name)
    : Form1 (parent, name)
{
    // Socket-Objekt erzeugen
    // Das Fenster ist parent-Objekt, dadurch wird
    // das Socket-Objekt automatisch mit dem Fenster
    // gelöscht.
    socket = new QSocket (this);
    // Signale verbinden
    connect (socket, SIGNAL (connected()),
            SLOT (sendCommand()));
    connect (socket, SIGNAL (readyRead()),
            SLOT (readResult()));
    connect (socket, SIGNAL (error(int)),
            SLOT (error(int)));
}

// Baut eine Verbindung zum Finger-Server auf,
// wird aufgerufen, wenn der Button geklickt wird.
void FingerImpl::startQuery ()
{
    resultWidget->clear();
    // Verbindung aufbauen
    socket->connectToHost (hostInput->text(), 79);
}

// Sendet ein Kommando an den Finger-Server;
// wird aufgerufen, sobald Socket-Verbindung aufgebaut ist.
void FingerImpl::sendCommand ()
{
    // Kommando aus der QLineEdit holen, Newline anhängen
    // (Kommando kann leer sein)
    // Hier wird QString benutzt, um ASCII-Text zu erhalten
    QString command = commandInput->text() + "\n";

```



```
// Diesen String per Socket an den Server senden
socket->writeBlock (command, command.length());
}

// Liest die Daten, die der Server geschickt hat,
// aus dem Socket aus und stellt sie dar.
// Wird aufgerufen, sobald neue Daten ankommen.
void FingerImpl::readResult ()
{
    // Solange noch mindestens eine Textzeile vorhanden ist
    while (socket->canReadLine())
    {
        // Eine Zeile einlesen
        QString line = socket->readLine();
        // Newline am Ende entfernen
        line.truncate (line.length() - 1);
        // Ergebnis anzeigen
        resultWidget->append (line);
    }
}

// Wird aufgerufen, wenn ein Fehler aufgetreten ist
void FingerImpl::error (int err)
{
    switch (err)
    {
        case QSocket::ErrConnectionRefused:
            resultWidget->setText ("Connection Refused!");
            break;
        case QSocket::ErrHostNotFound:
            resultWidget->setText ("Host not found!");
            break;
        case QSocket::ErrSocketRead:
            resultWidget->setText ("Error reading socket!");
            break;
        default:
            resultWidget->setText ("Unknown socket error!");
    }
}
```

QServerSocket und QSocket bei Servern

Auch für die Implementierung eines Servers bietet Qt eine komfortable Klasse: *QServerSocket*. Diese Klasse wartet an einem Port auf hereinkommende Verbindungswünsche von Clients. Außerdem kommt auch hier wieder die Klasse *QSocket* zum Einsatz, die für eine einzelne Verbindung auch auf der Server-Seite benutzt werden kann.

QServerSocket ist eine abstrakte Klasse. Um sie benutzen zu können, müssen Sie eine eigene Unterklasse von *QServerSocket* ableiten. In dieser abgeleiteten Klasse müssen Sie die rein virtuelle Methode *newConnection* überschreiben. Diese Methode wird immer dann automatisch aufgerufen, wenn eine Verbindung mit einem Client erzeugt wurde. Als Parameter erhält diese Methode den Dateidescriptor der erzeugten Socket-Verbindung. Beachten Sie, dass hier nicht automatisch ein *QSocket*-Objekt für die Verbindung erzeugt wird. Sie können hier also auch auf herkömmliche Weise auf den Socket zugreifen. Am einfachsten erzeugen Sie jedoch mit *new* ein neues *QSocket*-Objekt und setzen darin mit der Methode *setSocket* die übergebene Socket-Verbindung ein. Anschließend verbinden Sie die Signale des *QSocket*-Objekts mit eigenen Slots. Die Socket-Verbindung ist zu diesem Zeitpunkt bereits hergestellt, Sie brauchen also nicht auf ein *connected()*-Signal zu warten. (Sie würden ohnehin sehr lange darauf warten.)

Um einen Server zu starten, müssen Sie nur ein Objekt Ihrer abgeleiteten Klasse zu erzeugen. Im Konstruktor von *QServerSocket* wird dazu die Port-Nummer übergeben, unter der der Service angeboten werden soll. Hier können Sie entweder eine feste Portnummer benutzen, die noch nicht von anderen Diensten belegt wird, oder Sie benutzen den Wert 0. In diesem Fall sucht das Betriebssystem automatisch eine noch freie Portnummer heraus. Diese Portnummer muss Ihr Server natürlich irgendwie öffentlich machen, damit ein Client mit ihm in Verbindung treten kann. Möglich wäre beispielsweise, diese Portnummer in einer Datei abzuliegen, auf die der Client Zugriff hat (zum Beispiel auch über eine FTP- oder HTTP-Verbindung). Ein weiterer Parameter heißt *backlog*. Mit diesem Parameter können Sie einstellen, wie viele Verbindungswünsche zwischengespeichert werden, wenn der Server noch mit einem Verbindungsaufbau beschäftigt ist. Sofern Sie keinen Hochverfügbarkeitsserver implementieren wollen, der mit extrem vielen Verbindungswünschen gleichzeitig konfrontiert wird, können Sie es durchaus bei dem Vorgabewert von 0 belassen. Dieser Wert beschränkt übrigens nicht die Anzahl der gleichzeitig möglichen Client-Server-Verbindungen. Diese sind grundsätzlich nicht eingeschränkt. (Eine Einschränkung hierbei können Sie nur erreichen, indem Sie das *QServerSocket*-Objekt löschen, wenn Sie keine weiteren Verbindungen zulassen wollen.)

Als ein einfaches Anwendungsbeispiel wollen wir hier einen »Wurzel-Server« entwickeln. Dieser Server stellt im Netzwerk den Dienst zur Verfügung, aus einer beliebigen *double*-Zahl die Wurzel zu ziehen und das Ergebnis zurückzusenden. Ein solcher Server ist in der Praxis natürlich nicht besonders sinnvoll, da die Wurzelberechnung viel einfacher im Client selbst durchgeführt werden kann. Der Server kann aber auf einfache Weise auch umgeschrieben werden, um zum Beispiel eine kleine Datenbank zu verwalten, E-Mails abzurufen oder einen Drucker zu steuern. Abbildung 4.56 zeigt den Server und einen mit ihm verbundenen Client. Das hier abgedruckte Listing enthält wiederum nur die zentralen

Stellen aus der Server-Implementierung. Den kompletten Quellcode für den Server sowie für den Client – insbesondere die grafische Benutzeroberfläche – finden Sie auf der CD-ROM, die dem Buch beiliegt.

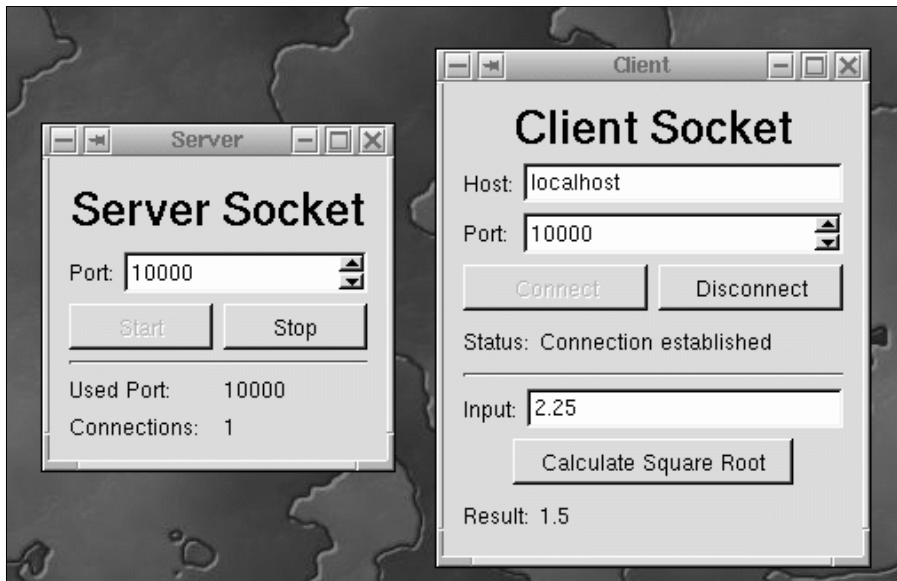


Abbildung 4-56 Server und Client bei der Arbeit

```
// Diese Klasse definiert einen Server-Socket, der
// auf Verbindungen durch Clients wartet. Dazu leiten
// wir die Klasse von QServerSocket ab und überschreiben
// die abstrakte Methode "newConnection".
class SqrtServerSocket : public QServerSocket
{
    Q_OBJECT
public:
    SqrtServerSocket (Q_UINT16 port, int backlog = 0,
                     QObject *parent = 0,
                     const char *name = 0);
    void newConnection (int socket);

signals:
    // Über dieses Signal teilt der Server-Socket
    // die aktuelle Anzahl von Verbindungen mit.
    void connectionCounterChanged (int number);

private slots:
    void oneConnectionClosed ();
```

```

private:
    // Interne Liste aller aktuellen Verbindungen.
    // Wird oftmals nicht benötigt.
    QList <QSocket> connectionlist;
};

/*
    Konstruktor, ruft nur den Konstruktor von
    QServerSocket auf. Es wird unmittelbar mit dem
    Warten auf eine Verbindung begonnen. Für jede
    neue Verbindung wird "newConnection" aufgerufen.
    Mit der Methode "ok" kann getestet werden, ob der
    Socket korrekt initialisiert werden konnte.
*/
SqrtServerSocket::SqrtServerSocket
    (Q_UINT16 port, int backlog,
     QObject *parent, const char* name)
    : QServerSocket (port, backlog, parent, name)
{
}

/*
    Diese Methode wird bei jeder neuen Verbindung
    aufgerufen. Der Parameter "socket" ist der
    Dateideskriptor des neu erzeugten Sockets. Dieser
    Socket ist bereits verbunden.
*/
void SqrtServerSocket::newConnection (int socket)
{
    // Es wird ein neues Socket-Objekt erzeugt, das
    // die Bearbeitung des Auftrags übernimmt. Die Arbeit
    // für den Server-Socket ist damit weit gehend
    // abgeschlossen.
    SqrtSocket *newSocket = new SqrtSocket (this);

    // Das neue Socket-Objekt wird auf den bereits
    // erzeugten Socket gesetzt. Dieser Socket ist bereits
    // verbunden, es ist also kein connect mehr nötig.
    newSocket->setSocket (socket);

    // In diesem Beispiel führen wir eine Liste der
    // Verbindungen im Server-Socket mit. Oftmals ist das
    // nicht nötig, so dass der Verwaltungsaufwand für
    // die Liste entfällt.
    connectionlist.append (newSocket);

    // Wir lassen uns von einem Verbindungsabbau
    // informieren, um die Liste aktuell zu halten.
    connect (newSocket, SIGNAL (connectionClosed()),

```

```

        SLOT (oneConnectionClosed());
// Die neue Anzahl der Verbindungen wird der Außenwelt
// mitgeteilt, damit die Anzeige entsprechend
// korrigiert werden kann.
emit connectionCounterChanged (connectionlist.count());
}

/*
Dieser Slot wird von einem Socket-Objekt aufgerufen,
wenn die Verbindung beendet wurde. Sie wird nicht
benötigt, wenn der Server-Socket keine Liste der
Verbindungen hält.
*/
void SqrtServerSocket::oneConnectionClosed ()
{
    // Um herauszufinden, welcher Socket beendet wurde,
    // benutzen wir die Methode sender.
    QSocket *sock = (QSocket *) sender();

    // Das Socket-Objekt wird aus der Liste entfernt
    connectionlist.removeRef (sock);

    // Die Anzeige wird aktualisiert
    emit connectionCounterChanged (connectionlist.count());

    // Nun können wir das Socket-Objekt löschen (falls es
    // sich - wie hier - nicht selbst löscht).
    // Es sollte allerdings nicht direkt gelöscht werden,
    // da dieser Slot ja vom Socket-Objekt selbst
    // aufgerufen wurde, wir also in dieses Objekt
    // zurückkehren.
    // Stattdessen starten wir einen Timer mit
    // Zeitintervall 0, der den Slot "deleteMyself" vom
    // Socket-Objekt aufruft, sobald die Kontrolle wieder
    // in der Haupt-Event-Schleife ist.
    QTimer::singleShot (0, sock, SLOT (deleteMyself()));
}

```

4.19.2 Netzwerktransparenter Dateizugriff mit KIO

KDE bietet dem Entwickler mit dem *KIO-Konzept* (*KDE Input/Output*) ein sehr mächtiges und dennoch einfach zu bedienendes Werkzeug an, um einen netzwerktransparenten Dateizugriff zu verwirklichen. Dateien können dann nicht nur von der lokalen Festplatte, sondern auch aus anderen Quellen gelesen und geschrieben, kopiert, verschoben, gelöscht oder umbenannt werden. Andere Quellen könnten zum Beispiel FTP- oder WWW-Server sein, die über ein Internet oder Intranet erreichbar sind. Der Anwender eines solchen Programms bekommt (im Idealfall) nichts davon mit, dass die geladene Datei nicht auf der Festplatte liegt.

Die Funktionalität des KIO-Konzepts ist in der Bibliothek *libkio* zusammengefasst. Falls Sie dieses Konzept also nutzen wollen, müssen Sie diese Bibliothek beim Linken Ihres Programms einbinden. Unter Unix müssen Sie dazu bei den gängigen Compilern die Kommandozeilenoption *-lkio* hinzufügen. Wenn Sie Makefiles benutzen, so gibt es für diesen Zweck im Makefile meist eine spezielle Variable `LFLAGS`. Fügen Sie die Option hier hinzu. Wenn Sie *tmake* verwenden, heißt die entsprechende Variable `LIBS`.

Fast alle Klassen und Funktionen, die KIO bereitstellt, sind im Namespace *KIO::* definiert. Tabelle 4.19 gibt einen Überblick über die wichtigsten Klassen und Funktionen, die für Programmentwickler interessant sind.

Bezeichnung	Art	Beschreibung
<i>KURL</i>	Klasse	Analyse und Manipulation von URLs
<i>KIO::NetAccess</i>	Klasse	synchrone Netzwerkoperationen
<i>KIO::Job</i>	Klasse	Informationen über eine gerade laufende Netzwerkoperation
<i>KIO::get</i>	Funktion	Einlesen einer Datei
<i>KIO::put</i>	Funktion	Schreiben einer Datei
<i>KIO::copy</i>	Funktion	Kopieren einer Datei oder eines Verzeichnisses
<i>KIO::move</i>	Funktion	Verschieben einer Datei oder eines Verzeichnisses
<i>KIO::del</i>	Funktion	Löschen einer Datei oder eines Verzeichnisses
<i>KIO::mkdir</i>	Funktion	Anlegen eines Unterverzeichnisses
<i>KIO::rmdir</i>	Funktion	Löschen eines leeren Unterverzeichnisses

Tabelle 4-19 Wichtige Klassen und Funktionen zum Dateizugriff über ein Netzwerk

Festlegen einer URL

Um festzulegen, wo die Datei gefunden werden kann oder wo sie abgespeichert werden soll, benutzt KDE *URLs* (*Uniform Resource Locator*). Eine URL ist dabei ein Textstring, der mit der Protokollbezeichnung beginnt. (Diese endet mit einem Doppelpunkt.) Daran schließen sich – abhängig vom Protokoll – die genaue Bezeichnung des Ortes an, an dem die Datei abgelegt ist, sowie der Dateiname. Danach kann hinter einem Doppelkreuz (#) noch eine nähere Angabe zu einer Position innerhalb der Datei folgen oder auch ein Unterprotokoll, wenn die angegebene Datei selbst eine ganze Verzeichnishierarchie enthält. Hier ein paar Beispiele für den Aufbau einer URL:

- *file:/usr/local/qt/include/qapplication.h*
auf dem lokalen Dateisystem (Protokoll *file:*), im Verzeichnis */usr/local/qt/include/*, mit dem Dateinamen *qapplication.h*

- <http://www.trolltech.com/index.html>
auf einem WWW-Server (Protokoll *http*), auf dem Rechner mit der IP-Adresse *www.trolltech.com*, im Hauptverzeichnis, mit dem Dateinamen *index.html*
- <http://doc.trolltech.com/qlabel.html#101ecb>
auf einem WWW-Server (Protokoll *http*), auf dem Rechner mit der IP-Adresse *doc.trolltech.com*, im Hauptverzeichnis, mit dem Dateinamen *qlabel.html*, an der Stelle des Ankers *101ecb*
- <ftp://pmueller:qwertz@ftp.kde.org/stable/README>
auf einem FTP-Server (Protokoll *ftp*), eingeloggt mit dem User-Namen *pmueller* mit dem Passwort *qwertz*, auf dem Rechner mit der IP-Adresse *ftp.kde.org*, im Verzeichnis */stable*, mit dem Dateinamen *README*
- <file:/home/mueller/tmake-1.6.tar.gz#tar:/doc/tmake.html>
auf dem lokalen Dateisystem (Protokoll *file*), im Verzeichnis */home/mueller*, mit dem Dateinamen *tmake-1.6.tar.gz*, in dieser Archiv-Datei (Unterprotokoll *tar*) im Unterverzeichnis */doc* und dem Dateinamen *tmake.html*

Zum Speichern, Analysieren und Manipulieren solcher URLs benutzt KDE die Hilfsklasse *KURL*. Auch in Qt ist eine solche Hilfsklasse mit dem Klassennamen *QUrl* definiert. Beide Klassen besitzen die gleichen Möglichkeiten und sind zueinander kompatibel. Die Methodenbezeichnungen sind jedoch leicht abweichend. Tabelle 4.20 listet die wichtigsten Methoden der Klassen mit einer kurzen Beschreibung auf. Zu all diesen Methoden zum Analysieren einer URL gibt es in der Regel auch eine entsprechende *set*-Methode, um einen einzelnen Bestandteil der URL zu ändern.

Methode in <i>KURL</i>	Methode in <i>QUrl</i>	Beschreibung
<i>isMalformed</i>	<i>isValid</i>	Test, ob syntaktischer Aufbau der URL korrekt ist (Bedeutung in <i>KURL</i> und <i>QUrl</i> entgegengesetzt!)
<i>isLocalFile</i>	<i>isLocalFile</i>	<i>true</i> für URL von lokalem Dateisystem (<i>file</i> .)
<i>protocol</i>	<i>protocol</i>	Protokoll-String (einschließlich Doppelpunkt)
<i>user</i>	<i>user</i>	Login-Name (z.B. für FTP-Server)
<i>pass</i>	<i>password</i>	Passwort (z.B. für FTP-Server)
<i>host</i>	<i>host</i>	Rechneradresse
<i>port</i>	<i>port</i>	Portnummer (eingeleitet durch einen Doppelpunkt)
<i>path</i>	<i>path</i>	Pfad (Verzeichnis + Dateiname)
<i>query</i>	<i>query</i>	Anfrage (eingeleitet durch ein Fragezeichen)

Tabelle 4-20 Wichtige Methoden der Klassen *KURL* und *QUrl*

In der Regel benötigt man diese Methoden jedoch nur selten. Meist erstellt man ein *KURL*-Objekt, dem man per Konstruktor direkt den gesamten URL-String mitgibt, und verwendet dieses *KURL*-Objekt dann in den verschiedenen Methoden zum Zugriff auf die angegebene Datei. Um beispielsweise eine Datei von einem FTP-Server in eine lokale Datei zu kopieren, können Sie folgenden Code verwenden (die Klasse *NetAccess* wird später näher erläutert):

```
KURL fileurl ("http://www.trolltech.com/index.html");
KIO::NetAccess::download (fileurl, "/tmp/index.html");
```

Wollen Sie dagegen eine URL aus mehreren Teilen zusammenbauen, greifen Sie auf die *set*-Methoden zurück. Wenn Sie beispielsweise in einem Dialog in mehreren Eingabeelementen die Daten für den Zugriff auf einen FTP-Server eingelesen haben, können Sie die URL folgendermaßen zusammenstellen:

```
KURL ftpurl;
ftpurl.setProtocol ("ftp");
ftpurl.setHost (inputHost->text());
if (!inputLogin->text().isEmpty())
{
    ftpurl.setUser (inputLogin->text());
    ftpurl.setPass (inputpassword->text());
}
ftpurl.setPath (inputPath->text());
```

Einfache synchrone Zugriffe mit *KIO::NetAccess*

Die Klasse *KIO::NetAccess* bietet eine Reihe von statischen Methoden, mit denen man sehr einfach eine Datei bearbeiten kann. Die am häufigsten benutzten Methoden sind dabei *download* und *upload*. Die erste Methode lädt eine durch eine URL spezifizierte Datei in eine temporäre Datei auf dem lokalen Dateisystem herunter. Die zweite kopiert eine Datei des lokalen Dateisystems zurück auf die angegebene URL. Beide Methoden haben den Rückgabetypp *bool*, der zurückgibt, ob die Operation erfolgreich war (*true*) oder nicht (*false*). Trat ein Fehler auf, so wird außerdem automatisch ein Meldungsfenster mit näheren Informationen angezeigt. Benötigen die Methoden für die Operation längere Zeit, so wird automatisch ein Fortschrittsbalken angezeigt, der den aktuellen Stand sowie die geschätzte Restzeit anzeigt.

Die Methode *download* lässt sich sehr gut einsetzen, wenn die Applikation ein Dokument verarbeiten soll, das nicht im lokalen Dateisystem liegt. Nach dem Herunterladen kann man die temporäre Datei öffnen, lesen, verarbeiten und auch wieder speichern. Anschließend kann die Datei mit *upload* wieder an die Ursprungsstelle kopiert werden. (Nicht alle Protokolle erlauben allerdings einen solchen Upload ohne weiteres zum Beispiel WWW-Server, oder FTP-Server ohne Schreibberechtigung.) Auch Dateien aus dem lokalen Dateisystem kann man »downloaden«, allerdings sollte man darauf verzichten, da dieses Vorgehen nur

unnötig Plattenspeicher belegt und Rechenzeit verbraucht. Man sollte also vorher testen, ob die URL der gewünschten Datei nicht bereits auf das lokale Dateisystem verweist.

Hier wollen wir zunächst eine mögliche Implementierung eines Downloads betrachten:

```
void MyDocument::openURL (const KURL &url)
{
    // Test auf korrekte URL
    if (url.isMalformed())
        return;

    QString filename;
    if (url.isLocalFile())
    { // lokale Datei, kein Download nötig
        filename = url.path();
    }
    else
    { // nicht-lokale Datei, download
        // erzeugt eine temporäre Datei, deren Dateiname
        // in filename gespeichert wird
        if (!KIO::NetAccess::download (url, filename))
            return; // download fehlgeschlagen
    }

    // Einlesen und Verarbeiten der (nun lokalen) Datei,
    // zum Beispiel mit QFile
    QFile file (filename);
    file.open (IO_ReadOnly);
    ....

    // temporäre Datei wird nicht mehr benötigt.
    // removeTempFile löscht sie nur dann, falls sie
    // durch download erzeugt wurde
    KIO::NetAccess::removeTempFile (filename);
}
```

In unserem Beispiel wurde die temporäre Datei am Ende der Methode wieder gelöscht. Dazu kann die Methode *removeTempFile* benutzt werden. Sie kann gefahrlos angewendet werden, auch wenn die URL bereits eine lokale Datei war, da nur mit *download* erzeugte Dateien mit ihr auch wieder gelöscht werden. Falls Sie die temporäre Datei auch über die Methode hinweg benutzen wollen (zum Beispiel um Änderungen zunächst in dieser Datei zwischenspeichern, bevor sie mit *upload* wieder zurückgeschrieben wird), können Sie *removeTempFile* natürlich auch zu einem späteren Zeitpunkt aufrufen. Vergessen Sie den Aufruf allerdings, bleiben die temporären Dateien erhalten und müssen von Hand gelöscht werden.

Beim Zurückschreiben mit *upload* müssen Sie die Daten zunächst in eine lokale Datei schreiben (falls das nicht ohnehin bereits geschehen ist). Mit Hilfe der Klasse *KTempFile* können Sie eine temporäre Datei anlegen, die automatisch anschließend wieder gelöscht wird:

```
void MyDocument::saveToLocalFile (QFile *file)
{
    // Speichert die Daten in der lokalen Datei ab, die vom
    // Parameter file angegeben wird.
    ...
}

void MyDocument::saveURL (const KURL &url)
{
    if (url.isMalformed())
        return;

    if (url.isLocalFile())
    {
        // Die Daten direkt in der lokalen Datei speichern
        QFile file (url.path());
        file.open (IO_WriteOnly);
        saveToLocalFile (&file);
    }
    else
    {
        // temporäre Datei zum Speichern anlegen
        KTempFile tempfile;
        saveToLocalFile (tempfile.file());
        KIO::NetAccess::upload (tempfile.name(), url); // upload
        tempfile.unlink(); // temporäre Datei wieder löschen
    }
}
```

KIO::NetAccess enthält neben *download* und *upload* noch Methoden zum Kopieren (*copy*) oder Löschen (*del*) einer einzelnen Datei sowie zum Prüfen, ob die URL tatsächlich auf eine existierende Datei verweist (*exists*) und welchen Dateityp diese hat (*mimetype*).

Asynchrone Zugriffe mit *KIO::Jobs*

Alle Methoden von *KIO::NetAccess* sind synchrone Operationen. Das heißt, sie kehren erst zum Aufrufer zurück, nachdem die Operation vollständig ausgeführt wurde. Während der Operationen werden zwar die Events weiterhin verarbeitet, so dass die Fenster der Applikation korrekt wiederhergestellt werden, falls sie verdeckt wurden oder sich ihr Inhalt ändert. Ebenso hat der Anwender die Möglichkeit, den Vorgang abubrechen. Dennoch bleiben die Fenster – wie bei einem modalen Dialogfenster – blockiert; das Programm ist also während der Zeit der

Operation nicht bedienbar. In vielen Situationen ist das auch durchaus sinnvoll, da nicht weitergearbeitet werden kann, bevor die Operation vollständig abgeschlossen ist. So kann man eine Textdatei erst dann bearbeiten, wenn sie komplett geladen wurde.

In anderen Fällen wäre es allerdings besser, wenn die Dateizugriffe im Hintergrund ablaufen würden, so dass der Anwender weiterhin das Programm bedienen kann. Hier einige typische Beispiele:

- Beim Upload einer Datei könnte der Anwender bereits eine andere Datei öffnen und bearbeiten, ohne auf das Ende des Uploads warten zu müssen.
- Bei einem Multiple Document Interface (MDI) sollten die anderen Fenster durch den Download eines Dokuments nicht blockiert sein.
- Beim Download kann es sinnvoll sein, den bereits gelesenen Teil anzuzeigen. Der Anwender hat dann die Möglichkeit, den Download vorzeitig abubrechen, wenn er sieht, dass er die falsche Datei ausgewählt hat.

Da gerade Netzwerkzugriffe oft sehr langsam sein können, kann eine Operation häufig mehrere Minuten dauern. Umso wichtiger wird es dann, dass die Operationen asynchron – also im Hintergrund – ablaufen. Auch dazu bietet *KIO* entsprechende Konzepte an, die allerdings komplizierter zu nutzen sind. Die Operationen werden dabei auf so genannte Jobs verteilt, die bei einer Änderung des Zustands ein entsprechendes Signal senden. Dieses Konzept wird auch intern in *KIO::NetAccess* genutzt, ist dort aber für den Entwickler nicht sichtbar.

Die Dateizugriffe werden beim Job-Konzept mit Hilfe der Funktionen gestartet, die in Tabelle 4.18 aufgelistet wurden. Die Funktionen sind im Namespace *KIO* definiert und erhalten als Parameter in der Regel eine oder mehrere URLs in Form von *KURL*-Objekten. Die Funktionen starten zwar die gewünschte Operation, warten aber nicht, bis diese beendet ist, sondern kehren direkt zurück. Der Rückgabewert ist ein Zeiger auf ein automatisch erzeugtes Objekt der Klasse *KIO::Job*. Dieses Objekt repräsentiert die soeben angestoßene Operation.

Das war eigentlich schon alles, was Sie tun mussten. Sobald das Programm in die Haupt-Event-Schleife zurückkehrt, werden im Hintergrund die entsprechenden Daten übertragen bzw. Operationen ausgeführt. Oftmals wollen Sie aber über den Fortgang der Operation informiert werden, insbesondere um über auftretende Fehler unterrichtet zu werden. Verbinden Sie dazu einige der Signale, die in *KIO::Job* definiert sind, mit eigenen Slots.

Das wichtigste Signal ist *KIO::Job::result(KIO::Job*)*. Dieses wird immer zum Ende der Operation aufgerufen, egal ob die Operation erfolgreich oder aufgrund eines Fehlers beendet wurde. Es wird Ihnen ein Zeiger auf das *KIO::Job*-Objekt übergeben, das die Operation durchführte. Dieses Objekt hat einige Methoden für die Fehlerbehandlung: Die Methode *error()* liefert Ihnen einen Fehlercode zurück

oder den Wert 0, falls die Operation erfolgreich war. Die Methode *errorString()* liefert Ihnen einen ausführlichen Fehlerstring (übersetzt in die eingestellte Landessprache) zurück, den Sie zum Anzeigen einer Fehlermeldung benutzen können. Noch mehr Komfort bietet die Methode *showErrorDialog()*, die direkt einen Dialog mit entsprechender Fehlermeldung auf dem Bildschirm ausgibt.

Neben dem Signal *result* sind noch einige andere definiert, die Sie zum Beispiel zur Anzeige von Zusatzinformationen oder eines Fortschrittbalkens nutzen können. In Tabelle 4.21 sind diese Signale aufgelistet.

Signal	Beschreibung
<i>result</i>	wird nach dem Ende der Operation gesendet
<i>canceled</i>	wird bei einem Abbruch gesendet (auch <i>result</i> wird in diesem Fall gesendet)
<i>infoMessage</i>	Zusatzinformationen als Text (z.B. aktueller Status)
<i>percent</i>	Prozent der bereits erledigten Arbeit
<i>totalSize</i>	Gesamtgröße der Operation (in Byte)
<i>processedSize</i>	Bereits bearbeitete Größe (in Byte)
<i>speed</i>	Geschwindigkeit der Operation (in Byte pro Sekunde)

Tabelle 4-21 Signale der Klasse KIO: : Job

Als ein einfaches Beispiel wollen wir hier ein ganzes Verzeichnis von einem FTP-Server mitsamt allen enthaltenen Dateien und Unterverzeichnissen auf unsere lokale Platte kopieren lassen. Dazu benötigen wir nur einen einzigen Funktionsaufruf. Um aber auch über auftretende Fehler informiert zu werden, benötigen wir einen selbst definierten Slot:

```
void MyObject::startCopy()
{
    // Starten des Kopierauftrags
    KIO::Job *job;
    KURL quelle ("ftp://ftp.kde.org/pub/kde/stable/latest/"
                "distribution/tar/generic/source/bz2");
    KURL ziel ("file:/home/mueller/kde20");
    job = KIO::copy (quelle, ziel);

    // Ergebnissignal mit eigenem Slot verbinden
    connect (job, SIGNAL (result (KIO::Job*)),
            SLOT (copyResult (KIO::Job*)));
}

void MyObject::copyResult (KIO::Job* job)
{
    if (job->error() != 0)
        job->showErrorDialog ();
}
```

Bei der Anwendung der KIO-Funktionen müssen einige Dinge beachtet werden:

- Das *KIO::Job*-Objekt wird automatisch erzeugt und auch wieder gelöscht. Rufen Sie daher niemals selbst *delete* für ein solches Objekt auf. Das Objekt wird in der Regel nach Ausführung des Signals *result* gelöscht. Anschließend sollten Sie also nicht mehr darauf zugreifen. Ebenso sollten Sie *KIO::Job*-Objekte niemals selbst erzeugen, sondern immer nur über die Funktionen in *KIO* erzeugen lassen.
- In den meisten Fällen ist es nicht nötig, einen eigenen Fortschrittsbalken anzuzeigen, da automatisch ein entsprechendes Fenster mit Informationen geöffnet wird. Wollen Sie dieses selbst übernehmen, so übergeben Sie als weiteren Parameter *showProgress* den Wert *false*. (Der Default-Wert dieses Parameters ist immer *true*.) In diesem Fall empfiehlt es sich, den Fortschritt selbst anzuzeigen und auch eine Abbruchmöglichkeit zu geben.
- Ihr Programm bleibt während der Durchführung der Operation bedienbar. Das ist zwar unbestreitbar ein echter Vorteil, überlegen Sie aber auch, welche Funktionen Ihres Programms vor dem Ende der Operation überhaupt sinnvoll sind, und deaktivieren Sie die anderen. Verhindern Sie zum Beispiel unbedingt, dass der Anwender die gleiche Operation mehrmals startet. Was käme zum Beispiel dabei heraus, wenn er zwei verschiedene Dateien gleichzeitig in einen Editor lädt?
- Die Operationen werden nicht von Ihrer eigenen Applikation durchgeführt, sondern vom KDE-Daemon *klauncher*. Dieser sowie die Programme *dcopserver*, *kded*, *kio_file* und *kio_uiserver* müssen bereits gestartet sein, damit ein Netzwerkzugriff erfolgen kann. Dieses sollte aber in einem korrekt installierten KDE2-System immer der Fall sein.
- Die Funktionen im *KIO*-Namespace geben in der Regel einen Zeiger auf eine Unterklasse von *KIO::Job* zurück. Die Signale aus Tabelle 4.20 werden natürlich an alle Unterklassen vererbt, so dass Sie sie in jedem Fall benutzen können. Einige Unterklassen besitzen aber noch zusätzliche Signale, die interessant oder auch unbedingt nötig sein können. Ein Beispiel dafür sind die Funktionen *get* und *put*, die einen Zeiger auf ein *KIO::TransferJob*-Objekt zurückliefern. Diese beiden Funktionen werden weiter unten genauer beschrieben.
- Die Funktionen *copy* und *move* bilden weit gehend die Unix-Befehle *cp* und *mv* nach. Ist das Ziel ein existierendes Verzeichnis, so werden die Daten in dieses Verzeichnis kopiert bzw. verschoben. Ist es dagegen kein existierendes Verzeichnis und besteht die Quelle nur aus einer einzelnen Datei bzw. einem einzelnen Verzeichnis, so wird die Quelle unter diesem neuen Namen kopiert bzw. umbenannt. Alternativ können Sie auch die Funktionen *file_copy* und *file_move* benutzen, die jeweils nur eine einzelne Datei kopieren oder verschie-

ben, oder Sie können *copyAs* bzw. *moveAs* benutzen, die in jedem Fall die Zielangabe als neuen Namen benutzen, auch wenn die Ziel-URL ein bereits existierendes Verzeichnis ist.

- Für eine Auflistung aller Funktionen schauen Sie in der Dokumentation zur KIO-Bibliothek nach. Die Erläuterungen sind zur Zeit zwar noch etwas dürftig, aber die meisten Funktionen sind selbsterklärend.
- Die Funktionen sind sehr mächtig, und es spricht nichts dagegen, sie auch einzusetzen, wenn ganze Verzeichnisse auf dem lokalen Dateisystem kopiert oder verschoben werden sollen.

Als Letztes wollen wir uns noch ein Beispiel anschauen, in dem mit den Befehlen *KIO::get* und *KIO::put* Daten einer Datei direkt in den Speicher gelesen bzw. aus dem Speicher geschrieben werden. Diese Funktionen sind damit der asynchrone Ersatz für die Methoden *download* und *upload* der Klasse *KIO::NetAccess*, die wir im letzten Abschnitt besprochen haben.

Die beiden Funktionen geben jeweils einen Zeiger auf ein *KIO::TransferJob*-Objekt zurück. Dieses Objekt hat unter anderem zwei zusätzliche Signale: zum einen das Signal *data*, mit dem die *get*-Operation meldet, dass neue Daten beim Download angekommen sind, zum anderen das Signal *dataReq*, mit dem die *put*-Operation meldet, dass sie bereit ist, weitere Daten im Upload zu verschicken. Sie müssen also das jeweilige Signal mit einem eigenen Slot verbinden, damit die Operationen auch sinnvoll arbeiten.

Hier folgt der Beispielcode aus dem vorherigen Abschnitt, angepasst an eine asynchrone Datenübertragung mit *put* und *get*. Beachten Sie, dass wir in diesem Fall keine temporären Hilfsdateien anlegen müssen.

Hier folgt zunächst das Programmstück zum Einlesen einer Datei:

```
void MyDocument::openURL (const KURL &url)
{
    // Test auf korrekte URL
    if (url.isMalformed())
        return;

    // Download starten (auch für lokale Dateien)
    KIO::TransferJob *job = KIO::get (url);

    // Und Signale verbinden
    connect (job, SIGNAL (data(KIO::Job*, const QByteArray&)),
            SLOT (appendData(KIO::Job*, const QByteArray&)));
    connect (job, SIGNAL (result(KIO::Job*)),
            SLOT (downloadReady(KIO::Job*)));

    // Speicher zum Sammeln der Daten löschen
```

```

docData = QByteArray();

// Menüpunkt "Open" (und evtl. andere Menüpunkte) deaktivieren
....
}

void MyDocument::appendData(KIO::Job *, const QByteArray& data)
{
    // Ein weiterer Datenblock trifft ein, der an das Dokument
    // angehängt und evtl. bereits angezeigt werden kann.
    // Liegen die Dokument-Daten beispielsweise im QByteArray
    // docData, so können Sie folgenden Code benutzen:
    QDataStream stream (docData, IO_WriteOnly | IO_Append);
    stream << data;
    ....
}

void MyDocument::downloadReady(KIO::Job* job)
{
    // Fehler melden
    if (job->error() != 0)
    {
        job->showErrorDialog ();
        // Dokument ungültig, also löschen
        docData = QByteArray();
    }
    // Menüpunkt "Open" wieder aktivieren
    ....
}

```

Und hier ist der Quellcode, den Sie zum Speichern eines Dokuments benutzen können:

```

void MyDocument::saveURL (const KURL &url)
{
    // Test auf korrekte URL
    if (url.isMalformed())
        return;

    // Upload starten (auch für lokale Dateien)
    KIO::TransferJob *job = KIO::put (url);

    // Und Signale verbinden
    connect (job, SIGNAL (dataReq(KIO::Job*, QByteArray&)),
            SLOT (sendMoreData(KIO::Job*, QByteArray&)));
    connect (job, SIGNAL (result(KIO::Job*)),
            SLOT (downloadReady(KIO::Job*)));

    // Indexzeiger auf aktuelle Dokumentposition an den Anfang

```

```

    // setzen. Bei jedem geschriebenen Block Zeiger weitersetzen.
    docPosition = 0;

    // Menüpunkt "Save" (und evtl. andere Menüpunkte)
    // deaktivieren
    ....
}

void MyDocument::sendMoreData(KIO::Job *, QByteArray& data)
{
    // Ein weiterer Datenblock kann gesendet werden. Als
    // Blockgröße wählen wir hier 1024 Byte. Sie können hier
    // auch das ganze Dokument auf einmal speichern. Wir nehmen
    // an, das Dokument liegt in docData, die aktuelle Position
    // ist docPosition.
    int rest = docData.size() - docPosition;
    if (rest == 0)
        data = QByteArray(); // Übertragung beendet
    else
    {
        rest = QMIN (rest, 1024); // Blockgröße begrenzen
        data.duplicate (docData.data() + docPosition, rest);
    }
    docPosition += rest;
}

void MyDocument::downloadReady(KIO::Job* job)
{
    // Fehler melden
    if (job->error() != 0)
        job->showErrorDialog ();

    // Menüpunkt "Save" wieder aktivieren
    ....
}

```

Beispielprogramme finden Sie auf der CD-ROM, die dem Buch beiliegt.

Unterstützung weiterer Protokolle in KIO

Die Dateioperationen der eben besprochenen Funktionen werden nicht von Ihrer Applikation ausgeführt, sondern von einem anderen Programm, dem Programm *kio_file*. Ihr Programm wendet sich mit einer Anfrage per DCOP an dieses Programm und überträgt den Befehl sowie die URLs. *kio_file* bestimmt anhand der URL, welches Protokoll benutzt werden soll. Der Code für die Behandlung der einzelnen Protokolle ist in dynamischen Bibliotheken gespeichert, die erst bei Bedarf dynamisch zur Laufzeit eingebunden werden. Das entlastet den Speicher, da nicht benutzte Protokolle nicht eingebunden werden, und ermöglicht es außerdem, im laufenden Betrieb weitere Protokolle hinzuzufügen.

Jedes unterstützte Protokoll wird durch eine Datei im Verzeichnis *\$KDEDIR/share/config/protocols* beschrieben. Neben der Protokollbezeichnung finden sich dort auch Informationen darüber, ob Dateien über dieses Protokoll nur gelesen oder auch geschrieben werden können, und andere Dinge. Die eigentlichen Bibliotheken finden sich im Verzeichnis *\$KDEDIR/lib/* in den Dateien mit den Namen *kio_<PROTOKOLLNAME>.so*.

Wollen Sie eigene Protokolle entwickeln und in das System integrieren, so orientieren Sie sich am besten an den bereits in KDE integrierten Protokollen. Sie finden den entsprechenden Quellcode im KDE-Paket *kdelibs* im Unterverzeichnis *kio*. Jedes Protokoll ist in einem eigenen Unterverzeichnis gespeichert. So ist das FTP-Protokoll beispielsweise im Unterverzeichnis *ftp* zu finden und enthält die Dateien *ftp.h*, *ftp.cc* sowie *ftp.protocol*. Die Dateien *ftp.h* und *ftp.cc* definieren die Klasse *FtpProtocol* (als Unterklasse von *KIO::SlaveBase*) sowie die C-Funktion *kdemain*, die eine Instanz dieser Klasse erzeugt. Die Datei *ftp.protocol* wird bei der Installation in das Verzeichnis *\$KDEDIR/share/config/protocols* kopiert und erhält dabei den Namen *ftp.desktop*. Weitere Informationen können Sie auch der Datei *DESIGN* im Unterverzeichnis *kio* entnehmen.

4.19.3 Netzwerktransparenter Dateizugriff in Qt

Auch Qt bietet ab der Version Qt 2.2 ein Konzept zur Unterstützung von Dateizugriffen über verschiedene Protokolle an. Es lässt sich in etwa eine Zuordnung der einzelnen Klassen zwischen der Qt-Lösung und dem KIO-Konzept von KDE herstellen, die in Tabelle 4.22 aufgeführt ist.

KIO-Konzept	Qt-Network-Konzept
<i>KURL</i>	<i>QUrl</i>
Funktionen in <i>KIO</i> (<i>get</i> , <i>put</i> , <i>copy</i>)	Methoden in <i>QUrlOperator</i> (<i>get</i> , <i>put</i> , <i>copy</i>)
<i>KIO::Job</i>	<i>QNetworkOperation</i>
<i>KIO::SlaveBase</i>	<i>QNetworkProtocol</i>
<i>KIO::NetAccess</i>	keine Entsprechung

Tabelle 4-22 Gegenüberstellung von KIO und Qt-Network

Das Speichern, Analysieren und Manipulieren von URLs geschieht unter Qt mit der Klasse *QUrl*, die ja bereits in Tabelle 4.20 vorgestellt wurde. Bis auf einige Namensunterschiede von Methoden entspricht sie der Klasse *KURL*. Deshalb wollen wir hier nicht weiter auf diese Klasse eingehen.

Asynchrone Zugriffe mit *QUrlOperator*

Ähnlich wie bei KIO gibt es die Möglichkeit, Operationen auf Dateien auszuführen, die durch URLs gekennzeichnet werden, zum Beispiel kopieren (*copy*), umbe-

nennen (*rename*) oder löschen (*remove*). Während beim KIO-Konzept diese Operationen durch Funktionen im Namespace *KIO* gestartet wurden, sind es beim Qt-Network-Konzept Methoden der Klasse *QUrlOperator*. Um sie nutzen zu können, müssen Sie daher zunächst ein Objekt der Klasse *QUrlOperator* erzeugen.

Die Methoden liefern ebenfalls als Rückgabewert einen Zeiger auf ein Objekt der Klasse *QNetworkOperation*, der Informationen über die Operation enthält, ebenso wie bei den KIO-Operationen Zeiger auf Objekte der Klasse *KIO::Job* zurückgeliefert wurden. Es gibt jedoch einen Unterschied: Während die Klasse *KIO::Job* die Signale verschickte, um über den Stand der Operation zu berichten, sind diese Signale beim Qt-Network-Konzept in der Klasse *QUrlOperator* definiert. Daher hat der zurückgelieferte Zeiger auf das *QNetworkOperation*-Objekt auch eine weit geringere Bedeutung. In der Regel kann man ihn einfach ignorieren. Abbildung 4.57 verdeutlicht den Unterschied.

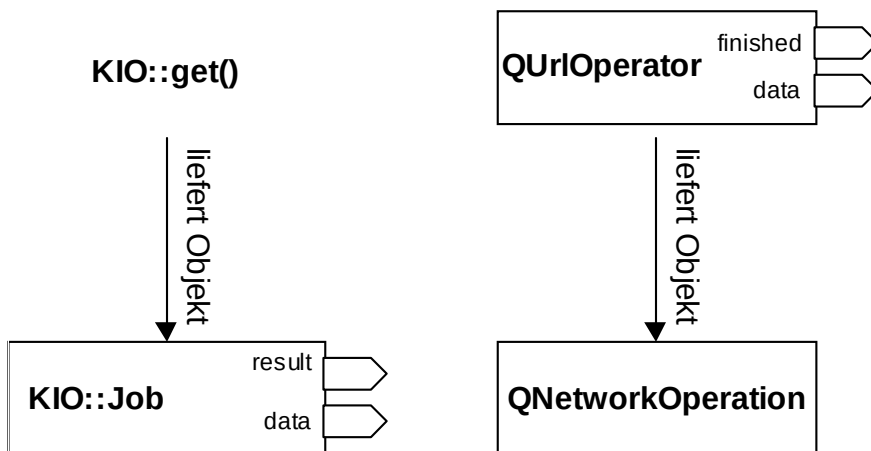


Abbildung 4-57 Unterschiede zwischen dem Qt-Network- und dem KIO-Konzept

Ein einfaches Beispiel soll die Anwendung der Klasse *QUrlOperator* verdeutlichen: Wir wollen eine Datei von einem FTP-Server auf unsere lokale Festplatte kopieren. Der Quellcode dafür sieht folgendermaßen aus:

```

QUrlOperator op;
connect (op, SIGNAL (finished (QNetworkOperation*)),
        myObject, SLOT (copyReady (QNetworkOperation*)));
op.copy ("ftp://ftp.kde.org/pub/kde/README",
        "/home/mueller");
  
```

In diesem Fall ignorieren wir den Rückgabewert der Methode *copy*, da wir zunächst keine weiteren Informationen über die Operation benötigen. Diese

erhalten wir nach dem Ende der Operation im eigenen Slot, der vom Signal *finished* aufgerufen wird. Dieser Slot kann beispielsweise so aussehen:

```
void MyObject::copyReady (QNetworkOperation* operation)
{
    if (operation->state() != QNetworkProtocol::StDone)
    {
        switch (operation->errorCode())
        {
            case QNetworkProtocol::ErrFileNonExisting:
                qDebug ("Verzeichnis oder Datei nicht gefunden");
                ....
        }
    }
}
```

Leider gibt es keine Möglichkeit, den Fehlercode einfach in einen aussagekräftigen Fehlertext umzuwandeln. Diese Aufgabe müssen Sie selbst übernehmen.

Sie haben übrigens auch die Möglichkeit, bereits im Konstruktor von *QUrlOperator* eine Basis-URL anzugeben. (*QUrlOperator* ist sogar von *QUrl* abgeleitet, d.h. Sie können alle Methoden von *QUrl* zur Analyse und Manipulation der Basis-URL nutzen.) In den einzelnen Operationen können Sie dann relative Angaben machen. Nötig ist das zum Beispiel, wenn Sie ein neues Verzeichnis anlegen lassen wollen. Geben Sie dann als Basisadresse das Verzeichnis an, in dem das neue Unterverzeichnis erstellt werden soll. Der Code dazu sieht beispielsweise folgendermaßen aus:

```
QUrlOperator op ("ftp://mueller:xyz@ftp.kde.org/Pub/kde");
connect (op, SIGNAL (finished (QNetworkOperation*)),
        SLOT (mkdirReady (QNetworkOperation*)));
op.mkdir ("kde45");
```

Die möglichen Operationen im Qt-Network-Konzept sind nicht so mächtig wie die von KIO: Die *copy*-Operation kopiert nur eine Datei (oder eine Liste von Dateien) unter Beibehaltung des Namens, während die *copy*-Operation in KIO auch ganze Verzeichnisbäume kopiert und beim Kopieren einer Datei diese auch gleich umbenennen kann. Auch stellt Qt nicht automatisch einen Fortschrittsbalken dar. Falls Sie einen solchen Balken anzeigen möchten (und das sollten Sie auf jeden Fall bei Operationen, die längere Zeit dauern können), müssen Sie diesen mit Hilfe des Signals *QUrlOperator::dataTransferProgress* implementieren. Für die meisten einfachen Aufgaben reichen die Operationen, die Qt bietet, aber vollkommen aus.

Unterstützung weiterer Protokolle in Qt

Das Network-Konzept von Qt ist noch sehr neu und nicht so umfangreich wie das KIO-Konzept von KDE. Das macht sich unter anderem auch in der Zahl der bisher unterstützten Protokolle bemerkbar: Qt kennt bisher nur zwei Protokolle, nämlich das lokale Dateisystem (in der Klasse *QLocalFS* implementiert) sowie das FTP-Protokoll (in der Klasse *QFtp* implementiert). In naher Zukunft wird voraussichtlich auch das HTTP-Protokoll hinzukommen.

Es gibt jedoch auch beim Qt-Network-Konzept die Möglichkeit, eigene Protokolle zu implementieren und hinzuzufügen. Leiten Sie dazu eine eigene Klasse von der Klasse *QNetworkProtocol* ab, und überschreiben Sie die virtuellen Methoden mit dem für Ihr Protokoll spezifischen Code. Dieses Protokoll müssen Sie dann noch mit der Methode *QNetworkProtocol::registerNetworkProtocol* installieren, und schon können Sie Ihr eigenes Protokoll mit *QUrlOperator* zusammen benutzen.

Das Erstellen eines neuen Protokolls ist damit unter Qt-Network ein Stück einfacher als unter KIO. Während Sie beim KIO-Konzept allerdings das Protokoll während der Laufzeit für alle KDE-Programme installieren konnten, können Sie es bei Qt-Network zunächst nur in eigenen Programmen verwenden. Um es auch anderen Programmen zugänglich zu machen, müssen Sie es in die Qt-Bibliothek hineinkompilieren.

4.19.4 Zusammenfassung Netzwerkprogrammierung

Sowohl Qt als auch KDE bieten eigene Konzepte an, um weit gehend plattformunabhängig auf ein Netzwerk zuzugreifen. Der Entwickler steht wie bereits öfter vor der Wahl, welches der Konzepte er vorzieht. Daher listen wir hier einige Entscheidungshilfen auf:

- Bei der Kommunikation über Sockets – also auf den unteren Netzwerkschichten – stellen die Klassen *QSocket* und *QServerSocket* die bessere Alternative dar, da sie konsequenter in der Schnittstelle und auch ausgereifter sind als *KSocket* und *KServerSocket*. Insbesondere bieten sie blockierungsfreie Nameserver-Zugriffe – eigentlich eine unverzichtbare Eigenschaft für GUI-Programme.
- Bei den höheren Protokollschichten – dem Dateizugriff mit URLs über verschiedene Protokolle hinweg – hat zur Zeit das KIO-Konzept der KDE-Bibliotheken deutlich die Nase vorn. Es unterstützt mehr Protokolle, bietet mächtigere Operationen und stellt automatisch den Fortschritt einer Operation auf dem Bildschirm dar.
- Für ein Programm, das auch auf Nicht-KDE-Systemen laufen soll, kommt das KIO-Konzept natürlich nicht in Frage, da es sehr viele KDE-Komponenten voraussetzt, unter anderem den KDE-Daemon (*kded*) und den DCOP-Server.

- Für echte KDE-Programme ist dagegen das KIO-Konzept schon fast Pflicht: Ein KDE-Programm sollte eine Datei nur durch Angabe der URL über ein beliebiges unterstütztes Protokoll verwenden können. Das ist natürlich mit Hilfe des KIO-Konzepts sehr einfach realisierbar. In einfachen Fällen kann man das Öffnen oder Speichern einer Datei über die synchronen Methoden der Klasse *KIO::NetAccess* erledigen. Muss es dagegen asynchroner Zugriff sein – das heißt, die Operation läuft im Hintergrund ab, während Ihr Programm bedienbar bleibt –, so kommen die Funktionen im Namespace *KIO* zur Anwendung.
- Beide Konzepte ermöglichen das Erweitern um eigene Protokolle. Während bei Qt-Network die Erstellung der eigenen Netzwerkklass und die Nutzung in eigenen Programmen einfacher ist, ist es bei KIO sehr viel leichter möglich, Protokolle in das KDE-System zu integrieren, die auf der Stelle von allen laufenden Programmen genutzt werden können.

4.20 Interprozesskommunikation mit DCOP

Eine der wichtigsten Neuerungen in KDE 2.0 ist die Möglichkeit, Nachrichten und auch große Datenmengen auf einfache Weise zwischen verschiedenen KDE-Programmen auszutauschen.

Zur Realisierung wurde zunächst der Standard *CORBA* in Betracht gezogen. *CORBA* ist ein inzwischen sehr ausgereiftes Konzept und bietet eine ungeheure Menge an Diensten an. *CORBA* ist plattform- und programmiersprachenunabhängig. Es gibt bereits für viele Programmiersprachen *CORBA*-Pakete. So ist auch eine Kommunikation zwischen einem C++- und einem Java-Programm möglich, die auf verschiedenen Rechnerplattformen laufen, die über das Internet verbunden sind. Die Programme bekommen davon nichts mit.

Das freie Softwarepaket *mico* bot sich als konkrete Implementierung des *CORBA*-Standards an. Nachdem eine Reihe von KDE-Programmen testweise um eine *CORBA*-Schnittstelle erweitert worden waren, stellte sich jedoch heraus, dass *CORBA* zu umfangreich ist, dadurch sehr viel Speicher verbraucht und das System bremst.

So wurde entschieden, für KDE ein eigenes Protokoll zur Interprozesskommunikation zu schreiben, das *Desktop Communications Protocol (DCOP)*. Dieses Protokoll bietet nur einen Bruchteil der Möglichkeiten von *CORBA*, ist aber sehr klein und einfach gehalten. Es belegt daher kaum zusätzlichen Speicher und arbeitet sehr effizient.

4.20.1 Einsatzgebiete des DCOP

DCOP dient in erster Linie zum Austausch von kleinen Nachrichten zwischen KDE-Programmen. Hier nur drei kleine Beispiele:

- Ein Programm kann beim Start testen, ob bereits eine weitere Instanz dieses Programms läuft. In diesem Fall kann es seine Kommandozeilenparameter an das bereits laufende Programm per DCOP übergeben und sich dann selbst beenden. Auf genau diese Weise arbeitet die Klasse *KUniqueApplication*, die in Kapitel 3.3.2, *KDE-Applikationen*, vorgestellt wurde.
- Ein Programm kann per DCOP Nachrichten an die Server von KDE schicken, um beispielsweise andere KDE-Programme starten zu lassen oder um sich über andere laufende KDE-Programme zu informieren.
- Ein Programm kann per DCOP durch andere Programme »ferngesteuert« werden (*scripting*). So kann man beispielsweise einen laufenden Editor von außen anweisen, eine neue Datei zu öffnen oder die aktuelle Datei zu speichern.
- Dieses Skripting kann natürlich auch auf alle KDE-Programme angewendet werden, die ohnehin im Hintergrund tätig sind, zum Beispiel auf den Window-Manager (*kwin*), den Desktop-Manager (*kdesktop*) oder die Startleiste (*kicker*).

Jedes Programm, das DCOP nutzen will, muss sich zunächst beim DCOP-Server (*dcopserver*) anmelden. Abbildung 4.58 verdeutlicht diesen Zusammenhang. Dort sind drei Applikationen beim DCOP-Server angemeldet. Jede bekommt vom Server eine eindeutige Applikationsidentifikation (*appId*) zugeordnet, die in der Regel dem Programmnamen entspricht. Gibt es jedoch mehrere angemeldete Programme mit dem gleichen Namen, so wird eine abweichende, eindeutige ID generiert. Beachten Sie, dass ein Programm – wie hier *KMyApp* – mehrfach gestartet worden sein kann. Jede gestartete Instanz baut dabei ihre eigene Verbindung zum Server auf.

Wie die *appId* gewählt wird, kann jede Applikation selbst bestimmen. Der Dateimanager *Konqueror* hängt beispielsweise an seinen Namen die Prozessnummer an, unter der er zur Zeit läuft, und erzeugt damit automatisch eine eindeutige *appId*.

Damit ein Programm nun einem anderen Programm eine Nachricht zukommen lassen kann, müssen einige Voraussetzungen erfüllt sein:

- Der DCOP-Server muss laufen. Das erledigt KDE normalerweise automatisch beim Start des X-Servers. Auf Nicht-KDE-Systemen muss er dagegen von Hand gestartet werden.
- Beide Programme müssen laufen und müssen sich beim DCOP-Server registriert haben.
- Das sendende Programm muss den eindeutigen ID-String des empfangenden Programms kennen sowie das genaue Format, das die Nachricht haben muss.

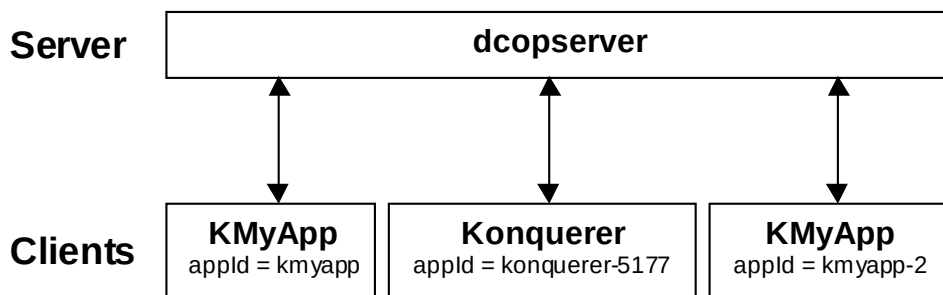


Abbildung 4-58 Verbindungen zwischen KDE-Programmen und dem DCOP-Server

Bisher war von Nachrichten die Rede. Genauer betrachtet handelt es sich dabei um den Aufruf einer Methode eines Objekts im anderen Programm. Die Daten für die Parameter der Methode werden dabei in einem Datenblock gespeichert und an das aufgerufene Programm übergeben. Dieses Programm extrahiert aus dem übergebenen Datenblock wieder die Werte für die einzelnen Parameter und ruft mit diesen die gewünschte Methode auf. Der Rückgabewert dieser Methode wird wiederum in einen Datenblock gepackt und an das aufrufende Programm übergeben. Bei *DCOP* handelt es sich daher um einen *Remote-Procedure-Call-Mechanismus (RPC)*, der auf Objekte erweitert wurde.

4.20.2 Aufruf von DCOP-Methoden von der Kommandozeile

Auf einem KDE-System ab Version 2.0 werden automatisch auch zwei hilfreiche Tools namens *dcop* und *kdcop* installiert. Mit diesen Programmen kann man sich anzeigen lassen, welche Programme sich zur Zeit beim DCOP-Server angemeldet haben, welche Objekte diese zur Verfügung stellen und welche Methoden für eines der Objekte aufgerufen werden können. Man kann sogar die Methoden aufrufen lassen. *dcop* ist dabei ein Kommandozeilen-Tool, während *kdcop* alles grafisch und übersichtlich anzeigt. Diese Tools ist daher sehr gut geeignet, um erste Erfahrungen mit DCOP zu sammeln. Ihre Haupteinsatzgebiete sind jedoch das Debuggen des DCOP-Systems sowie die Fernsteuerung von KDE-Programmen in Shell-Skripten. Abbildung 4.59 zeigt *kdcop* im Einsatz. Tabelle 4.23 zeigt die Syntax des Aufrufs von *dcop*.

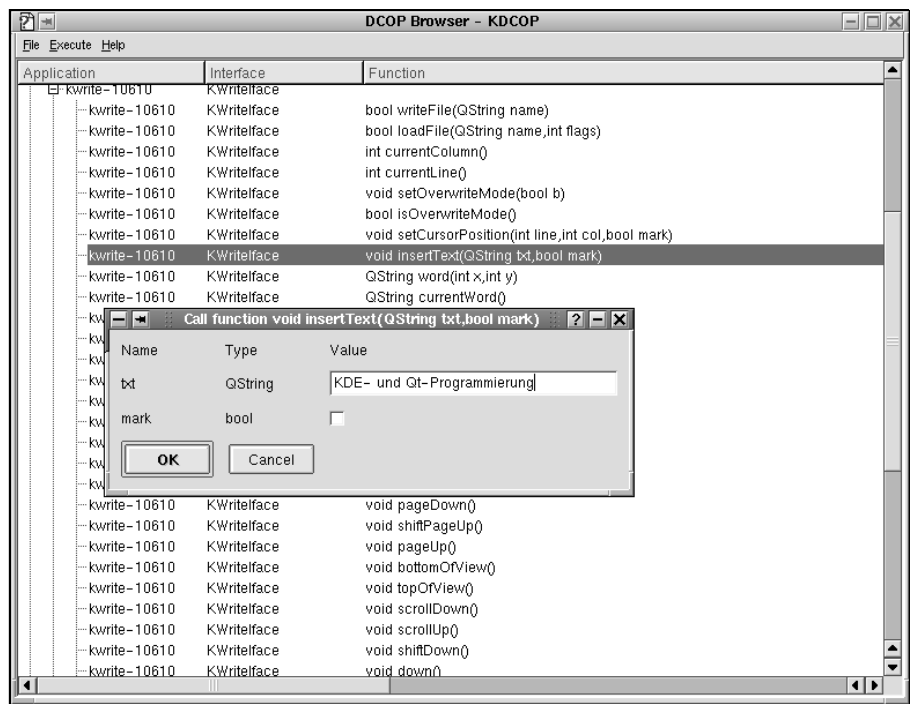


Abbildung 4-59 kdcop beim Aufruf einer DCOP-Methode

Aufruf	Bedeutung
<i>dcop</i>	zeigt eine Liste aller registrierten Applikationen an
<i>dcop appld</i>	zeigt eine Liste aller Objekte der Applikation <i>appld</i> an
<i>dcop appld objld</i>	zeigt eine Liste aller DCOP-Methoden des Objekts <i>objld</i> des Programms <i>appld</i> an
<i>dcop appld objld methode [parameter]</i>	ruft die angegebene Methode mit den Parametern auf

Tabelle 4-23 Aufrufsyntax von *dcop*

Wir wollen hier an einem konkreten Beispiel den Einsatz von *dcop* besprechen. Starten Sie zunächst das Programm *KWrite*. Ein Aufruf von *dcop* führt zu folgender Liste der registrierten Applikationen:

```
% dcop
kwin
klauncher_linux_500
kicker
kwrite
```



```
kded
knotify
kio_uiserver
kcookiejar
konqueror-8108
khotkeys
kmail
kxmlrpcd
kdesktop
klipper
kwrite-10625
konqueror-10480
```

Diese Liste kann auf Ihrem System natürlich auch anders aussehen.

In dieser Liste sind unter anderem einige der Hintergrundprogramme von KDE enthalten, aber zum Beispiel auch das gestartete *kwrite*-Programm. Die *appld* besteht in diesem Fall aus dem Programmnamen sowie der Prozessnummer des Programms. Wenn Sie das Programm *kwrite* mehrmals gestartet haben, so erscheint jede laufende Instanz mit ihrer eigenen Prozessnummer in der Liste.

Das Programm *dcop* kann uns nun auch anzeigen, welche Objekte das Programm *kwrite* für DCOP zur Verfügung stellt:

```
% dcop kwrite
qt
KWriteIFace
```

Da nur ein *kwrite*-Programm läuft, brauchen wir die Prozessnummer nicht anzugeben. In unserem Fall haben wir nur zwei Objekte. Das erste angegebene Objekt, *qt*, ist dabei gar kein echtes Objekt, sondern nur eine so genannte *Brücke* zwischen DCOP und dem Qt-Signal-Slot-Konzept. Diese Brücke werden wir uns später noch etwas genauer anschauen.

Das zweite Objekt trägt die Bezeichnung *KWriteIFace*. Dieses Objekt stellt also die Schnittstelle (Interface) für den Zugriff auf das Programm *KWrite* dar. Lassen wir uns nun auflisten, welche Methoden dieses Objekts wir aufrufen können:

```
% dcop kwrite KWriteIFace
QStringList interfaces()
QStringList functions()
void cursorLeft()
void shiftCursorLeft()
void cursorRight()
void shiftCursorRight()
void wordLeft()
void shiftWordLeft()
void wordRight()
void shiftWordRight()
```

```

void home()
void shiftHome()
void end()
void shiftEnd()
void up()
void shiftUp()
void down()
void shiftDown()
void scrollUp()
void scrollDown()
void topOfView()
void bottomOfView()
void pageUp()
void shiftPageUp()
void pageDown()
void shiftPageDown()
void top()
void shiftTop()
void bottom()
void shiftBottom()
int numLines()
QString text()
QString currentTextLine()
QString textLine(int num)
QString currentWord()
QString word(int x,int y)
void insertText(QString txt,bool mark)
void setCursorPosition(int line,int col,bool mark)
bool isOverwriteMode()
void setOverwriteMode(bool b)
int currentLine()
int currentColumn()
bool loadFile(QString name,int flags)
bool writeFile(QString name)

```

Die meisten Methodennamen sind sicher selbsterklärend. Wie Sie sehen, haben wir sehr viele Möglichkeiten, Informationen abzufragen oder das Programm zu manipulieren. Die folgende Zeile gibt beispielsweise den gesamten Text im Editorfenster aus:

```
% dcop kwrite KWriteIface text
```

Und mit der folgenden Zeile fügen wir einen Text an der aktuellen Cursorposition ein:

```
% dcop kwrite KWriteIface insertText "Hier ist der Text" false
```

Kommen wir nun noch einmal auf die bereits erwähnte DCOP-Qt-Brücke zurück. Sie ermöglicht es Ihnen, nicht nur die speziellen DCOP-Methoden von registrier-

ten Objekten aufrufen zu lassen, Sie können über diese Brücke auch von allen *QObject*-Instanzen (sowie den davon abgeleiteten Klassen) alle Slots aufrufen lassen und außerdem auf alle Properties zugreifen. Sehen wir uns zunächst einmal die Funktionen an, die uns das Pseudo-Objekt *qt* zur Verfügung stellt:

```
% dcop kwrite qt
QCStringList functions()
QCStringList interfaces()
QCStringList objects()
QCStringList find(QCString)
```

Die Funktion *objects* liefert uns eine Liste aller Objekte:

```
% dcop kwrite qt objects
qt/_ptrpriv
qt/unnamed1(QSignal, 0x80b2c14)
qt/unnamed2(QObject, 0x80b2bf0)
qt/unnamed3(QSignal, 0x80ec47c)
qt/unnamed4(QObject, 0x80ec458)
...
qt/unnamed92(HIManager, 0x8094928)
qt/_ptrpriv
qt/global pixmap cache
qt/unnamed93(QTimer, 0x8092d08)
qt/unnamed94(QTimer, 0x8092cdc)
qt/toolTipManager
qt/toolTipManager/unnamed1(QTimer, 0x808fab8)
qt/kwrite-mainwindow#1
qt/kwrite-mainwindow#1/hide-dock
qt/kwrite-mainwindow#1/unnamed1(QTimer, 0x8094868)
qt/kwrite-mainwindow#1/unnamed2(KWrite, 0x8098938)
...
qt/kwrite
qt/kwrite/session manager
qt/kwrite/kdetranslator
```

Die Ausgabe ist hier gekürzt, denn es sind in der Tat sehr viele Objekte, die definiert sind.

Auf jedes der Objekte können wir unter Angabe des angezeigten »Pfades« zugreifen. Schauen wir uns zum Beispiel einmal das Hauptfenster an:

```
% dcop kwrite qt/kwrite-mainwindow#1
QCStringList functions()
QCStringList interfaces()
QCStringList properties()
bool setProperty(QCString,QVariant)
QVariant property(QCString)
void writeConfig()
void newCaption()
```

```
void timeout()
void newStatus()
void newCurPos()
void printDlg()
void printNow()
void editToolbars()
void editKeys()
void toggleStatusbar()
void toggleToolbar()
void togglePath()
void configure()
void newView()
void newWindow()
void appHelpActivated()
void whatsThis()
void lower()
void raise()
void close()
void constPolish()
void polish()
void showNormal()
void showFullScreen()
void showMaximized()
void showMinimized()
void iconify()
void hide()
void show()
void repaint()
void update()
void clearFocus()
void setFocus()
```

Als Beispiel geben wir den Inhalt des Editors auf dem Drucker aus:

```
% dcop kwrite qt/kwrite-mainwindow#1 printNow
```

Ebenso kann man auf einzelne Properties des Hauptfensters zugreifen, zum Beispiel auf den Text der Titelleiste:

```
% dcop kwrite qt/kwrite-mainwindow#1 setProperty caption "KDE"
true
```

Hier aber eine dringende Warnung: Es ist zwar über diese Brücke möglich, jeden Slot und jede Property in jedem Objekt eines Programms von außen zu nutzen, jedoch führt es oft auch zu unvorhersehbaren Effekten, wenn man Slots aus dem normalen Aufrufzusammenhang herausgelöst aktiviert. Zu Debugging-Zwecken kann es ganz praktisch sein, diese Brücke zu nutzen, der normale Weg sollte aber immer über spezielle DCOP-Methoden führen. (Diese sind ohnehin viel effizienter als der Umweg über die DCOP-Qt-Brücke.)

4.20.3 Aufruf von DCOP-Methoden aus einem Programm heraus

Während im letzten Abschnitt die DCOP-Methoden durch spezielle Programme aufgerufen wurden, wenden wir uns nun dem »normalen« Einsatz von DCOP zu: der Kommunikation von Programmen miteinander.

Wenn Sie aus Ihrem Programm heraus auf ein anderes Programm zugreifen wollen, so müssen Sie folgende Dinge beachten:

- Sie müssen eine Verbindung zum DCOP-Server hergestellt haben (das machen Sie in der Regel am Anfang des Programms).
- Sie müssen die Application-ID des Programms kennen, das Sie ansprechen wollen.
- Sie müssen die genaue Objekt-ID des Objekts kennen, das Sie im anderen Programm ansprechen wollen.
- Sie müssen die genaue Signatur der Methode kennen, die Sie aufrufen wollen, also den Methodennamen sowie die Datentypen der Parameter.

Um eine Verbindung zum DCOP-Server herzustellen, wird ein Objekt der Klasse *DCOPClient* benötigt. Das Anlegen und Verwalten dieses Objekts kann man in der Regel der Klasse *KApplication* überlassen. So haben Sie von allen Stellen Ihres Programms einen leichten Zugriff auf das Objekt:

```
DCOPObject *client = kapp->dcopClient();
```

Diese Zeile erzeugt eine *DCOPObject*-Instanz automatisch, falls sie noch nicht vorhanden ist, und liefert den Zeiger darauf zurück.

Als Erstes muss sich nun Ihr Programm beim DCOP-Server anmelden. Dazu benutzen Sie die Methode *DCOPClient::attach()*. Sie liefert *true* zurück, falls die Verbindung aufgebaut werden konnte, und *false* für den Fall, dass ein Fehler auftrat. Diese Verbindung muss in der Regel nur einmal für Ihr Programm ausgeführt werden, am besten direkt beim Start des Programms. Sie können diese Verbindung daher in der *main*-Funktion herstellen:

```
int main (int argc, char **argv)
{
    KApplication app (argc, argv, "myapp");
    if (kapp->dcopClient()->attach() == false)
    {
        qDebug ("Connection to DCOP-Server "
                "failed! Aborting!");
        exit (1);
    }
    ....
}
```

In diesem Beispiel wird das Programm beendet, wenn keine Verbindung hergestellt werden konnte. Oftmals kann ein Programm aber auch sinnvoll ohne DCOP-Zugriffe ausgeführt werden. In diesem Fall empfiehlt es sich, nur eine Warnung auszugeben und das Programm danach normal auszuführen.

Nach dem Aufbau der Verbindung ist auch dieses Programm in der Liste der beim DCOP-Server registrierten Programme aufgeführt, und zwar unter der Applikations-ID *myapp*-<Prozessnummer>. Wollen Sie eine andere Applikations-ID verwenden, so benutzen Sie statt *attach* die Methode *registerAs*, wie es in Kapitel 4.20.4, *Entwickeln eigener DCOP-Klassen*, beschrieben wird.

Nach der Registrierung können wir nun die Methoden von anderen Programmen aufrufen. Im weiteren Verlauf dieses Abschnitts wird die klassische, rudimentäre Art des Aufrufs beschrieben. Sie geschieht mit Hilfe eines Aufrufs der Methode *send* bzw. *call* des *DCOPClient*-Objekts. Diese Vorgehensweise ist aber sehr umständlich und fehleranfällig. Ein Aufruf über so genannte *Stub-Objekte* (übersetzt etwa »Stummel-Objekte«) ist sehr viel einfacher und komfortabler. Diese Stub-Objekte können automatisch generiert werden. Die Vorgehensweise wird in Kapitel 4.20.4, *Entwickeln eigener DCOP-Klassen*, genau erläutert. Leider bieten bisher kaum KDE-Programme solche vordefinierten Stub-Klassen an, so dass Sie dort den hier beschriebenen Ansatz wählen müssen.

Ist man nicht am Rückgabewert der Methode interessiert (insbesondere, wenn der Rückgabotyp *void* ist), so benutzt man für den Aufruf die Methode *DCOPClient::send*. Sie sendet alle nötigen Werte an den DCOP-Server, der sich dann um das Weiterleiten kümmert. Das eigene Programm wird unmittelbar danach fortgesetzt.

DCOPClient::send benötigt vier Parameter: Als Erstes die *applId* des Programms, das wir ansprechen wollen, als Zweites die Objekt-ID des Objekts, als Drittes die Methodensignatur (Methodenname und Liste der Parametertypen) und als Viertes die Werte für die Parameter. Alle Parameter werden dazu mit Hilfe von *QDataStream* in einem *QByteArray* abgelegt. Das Erzeugen dieses *QByteArray* ist etwas aufwendiger und sollte deshalb vorher in einer lokalen Variablen erfolgen.

Wollen wir beispielsweise im Programm *myapp* im Objekt *mainobject* die DCOP-Methode *setMinMax* aufrufen, die zwei Integer-Zahlen erhalten soll, so sieht der Aufruf folgendermaßen aus:

```
// Die Werte dieser beiden Variablen sollen
// als Parameter benutzt werden:
int min = 5;
int max = 17;

// Zuerst werden die Werte für die Parameter mit Hilfe
// von QDataStream in einem QByteArray abgelegt
QByteArray params;
```

```
QDataStream (params, IO_WriteOnly) << min << max;

// Nun erfolgt der Aufruf
DCOPClient *client = kapp->dcopClient();
bool ok = client->send ("myapp", "mainobject",
                        "setMinMax(int,int)", params);
if (ok == false)
{
    qDebug ("Verbindung zum DCOP-Server unterbrochen!");
}
```

Schauen wir uns dieses Code-Stück noch einmal genauer an: Die ersten beiden Zeilen speichern die Werte für die beiden Parameter in einer lokalen Variable *params* vom Typ *QByteArray*. Dazu wird ein temporäres Hilfsobjekt vom Typ *QDataStream* angelegt, das auf der Variablen *params* arbeitet. Mit Hilfe der überladenen Operatoren << schreiben wir in derselben Zeile noch die beiden Zahlen 5 und 17 in den Stream und damit in die Variable *params* hinein.

Nachdem nun die Parameterwerte gespeichert worden sind, können wir den Aufruf starten. Dazu benötigen wir aber zunächst einen Zeiger auf das *DCOPClient*-Objekt, den wir in der lokalen Variable *client* ablegen. Schließlich rufen wir von diesem Objekt die Methode *send* auf. Sie bekommt die bereits oben erwähnten vier Parameter mit auf den Weg. Wichtig ist, dass beim dritten Parameter nicht nur der Methodenname, sondern die gesamte Signatur angegeben wird. Sie muss exakt mit der Signatur der aufgerufenen DCOP-Methode übereinstimmen (zusätzliche Leerschritte sind erlaubt). Der vierte Parameter ist schließlich unser vorbereiteter Parameterspeicher *params*. Es ist natürlich enorm wichtig, dass Sie beim Speichern der Parameterwerte in *params* die korrekte Anzahl und die richtigen Datentypen verwenden, da sonst die aufgerufene Methode unsinnige Werte erhält. Da keine weitere Typprüfung stattfindet, sind solche Fehler oft schwer zu entdecken.

send liefert einen Wert vom Typ *bool* zurück, der aussagt, ob der Befehl an den DCOP-Server weitergeleitet werden konnte. Er sagt aber nichts darüber aus, ob der Befehl auch ausgeführt werden konnte. Haben Sie sich bei einem der Parameter vertippt, so läuft der Befehl ins Leere, ohne dass Sie eine Fehlermeldung erhalten.

Wollen Sie eine DCOP-Methode aufrufen, die keine Parameter besitzt, so vereinfacht sich übrigens das Aufrufschema, da wir keine lokale Variable *params* benötigen. Ein Aufruf kann dann beispielsweise so aussehen:

```
// Nun erfolgt der Aufruf, dieses Mal ohne Parameter,
// stattdessen mit einem temporären QByteArray-Objekt
DCOPClient *client = kapp->dcopClient();
bool ok = client->send("myapp", "mainobject",
                      "recalculate()", QByteArray());
```

```

if (ok == false)
{
    qDebug ("Verbindung zum DCOP-Server unterbrochen!");
}

```

Liefert die DCOP-Methode, die Sie aufrufen wollen, einen Rückgabewert, an dem Sie interessiert sind, so müssen Sie statt *send* die Methode *call* benutzen. Ihr Programm wird dann so lange angehalten, bis die aufgerufene DCOP-Methode beendet ist und der Rückgabewert vom DCOP-Server an das eigene Programm zurückgemeldet wurde. Während dieser Zeit ist das Programm nicht bedienbar, die Benutzeroberfläche wird blockiert.

DCOPClient::call besitzt zwei weitere Parameter, in denen der Datentyp des Rückgabewerts (als Text vom Typ *QString*) sowie der eigentliche Wert (ebenfalls in einem *QByteArray* gepackt) zurückgegeben werden. In einem weiteren Parameter, *useEventLoop*, vom Typ *bool* können Sie außerdem festlegen, ob während der Abarbeitung der DCOP-Methode das gesamte Programm blockiert sein soll (Defaultwert *false*) oder ob nur die Eingabe blockiert sein soll, während andere Events wie zum Beispiel der Bildschirmaufbau weiterhin abgearbeitet werden (*true*). Es empfiehlt sich – insbesondere bei DCOP-Methoden, deren Laufzeit man nicht genau kennt – hier den Wert *true* zu benutzen.

Betrachten wir als Beispiel ein externes Programm, das eine Temperatur-Umrechnung von Celsius nach Fahrenheit vornimmt. Diesen Dienst stellt es als DCOP-Methode zur Verfügung, so dass auch andere Programme darauf zugreifen können. Das Programm läuft unter der *appId temperature*. Das Objekt, das für die Umrechnung zuständig ist, ist unter dem Namen *calculate* ansprechbar. Die Umrechnungsmethode heißt *celsiusToFahrenheit*. Sie erwartet einen Parameter vom Typ *double* und gibt das Ergebnis als Rückgabewert vom Typ *double* zurück. Der Aufruf dieser DCOP-Methode sieht dann folgendermaßen aus:

```

// Parameter speichern
QByteArray params;
QDataStream (params, IO_WriteOnly) << celsius;

// Variablen zum Abspeichern des Rückgabetyps und -werts
QString replyType;
QByteArray replyData;

// DCOP-Methode aufrufen
DCOPClient *client = kapp->dcopClient();
bool ok = client->call
    ("temperature",           // Applikations-ID
     "calculate",             // Objekt-ID
     "celsiusToFahrenheit(double)", // Methode
     params,                  // Parameterwerte
     replyType,               // nach Aufruf: Rückgabetyp

```



```

        replyData,    // nach Aufruf: Rückgabewert
        true);        // nicht blockierend
// Fehlerbehandlung
if (ok == false)
    qDebug ("DCOP-Aufruf fehlgeschlagen!");
if (replyType != "double")
    qDebug ("Rückgabetyt double erwartet!");

// Rückgabewert ermitteln
double fahrenheit;
QDataStream (replyData, IO_ReadOnly) >> fahrenheit;

```

Wir benötigen in diesem Fall zwei zusätzliche Variablen, um den Rückgabetyt und den Rückgabewert zu speichern. Sie sollten in jedem Fall den Rückgabetyt prüfen, ob er mit dem von Ihnen erwarteten Typ übereinstimmt. Um den Rückgabewert aus dem *QByteArray* wieder herauszubekommen, wird ganz analog ein temporäres *QDataStream*-Objekt benutzt. In diesem Fall lesen wir allerdings aus. Daher ist der Zugriffsmodus *IO_ReadOnly*, und als Operator benutzen wir *>>* statt *<<*.

4.20.4 Entwickeln eigener DCOP-Klassen

Bisher haben wir DCOP-Methoden in anderen Programmen aufgerufen, indem wir die Parameter in ein *QByteArray* kopiert haben und dann in einem Aufruf von *call* oder *send* die genaue Methodensignatur angegeben haben. Das ist jedoch fehleranfällig und umständlich. Eigene Klassen auf diese Art zu entwickeln, die per DCOP ferngesteuert werden können, ist noch komplizierter.

Aus diesem Grund wurden die Programme *dcopidl* und *dcopidl2cpp* entwickelt. Sie erstellen automatisch aus einer einfachen Klassendeklaration den Code, der zur Analyse von DCOP-Anfragen nötig ist, und verteilen die Anfragen auf die einzelnen Methoden. Weiterhin erzeugen sie Code für eine so genannte Stub-Klasse, die den Aufruf einer DCOP-Methode extrem vereinfacht.

Wir wollen uns den Einsatz dieser Hilfsprogramme anhand eines einfachen Beispiels anschauen: Wir erstellen ein Programm, das im Hintergrund läuft und als Dienst für andere Programme die Möglichkeit bietet, aus einer Zahl die Quadratwurzel zu ziehen. (Die Funktionalität ist also die gleiche wie in unserem Beispiel in Kapitel 4.19.1, *Socket-Verbindungen*, aber dieses Mal benutzen wir das DCOP-Protokoll.) In der Praxis ist ein solches Programm natürlich nicht sinnvoll, da das Berechnen einer Quadratwurzel sehr einfach ist. Eine Erweiterung unseres Beispiels auf komplexere Aufgaben ist jedoch leicht möglich.

Wir benötigen zunächst wie gehabt eine neue Klasse, die die DCOP-Anfragen für uns bearbeitet. Sie muss eine Unterklasse von *DCOPObject* sein, kann aber durch Mehrfachvererbung auch noch Unterklasse einer anderen Klasse sein.

Wir erweitern die Klassendeklaration noch um zwei Bestandteile: An den Anfang der Klasse setzen wir das Makro *K_DCOP* (ähnlich, wie bei von *QObject* abgeleiteten Klassen das Makro *Q_OBJECT* eingefügt wird, also ohne abschließendes Semikolon), und wir definieren alle Methoden, die DCOP-Methoden werden sollen, in einem eigenen Abschnitt *k_dcop*: (ähnlich, wie Signale und Slots in Qt in einem eigenen Abschnitt *signals*: oder *public slots*: definiert sind). Das war eigentlich auch schon alles.

Bei allen DCOP-Methoden ohne Rückgabewert können Sie als Rückgabetypp das Makro *ASYNc* statt eines *void* benutzen. Es dient dabei noch einmal als deutliches Zeichen, dass diese DCOP-Methode keinen Rückgabewert liefert und daher mit *send* statt mit *call* aufgerufen werden kann. (Das Makro *ASYNc* wird übrigens vom Präprozessor tatsächlich durch *void* ersetzt.)

Für unser Beispiel des Quadratwurzel-Dienstes sieht die Klassendeklaration dann folgendermaßen aus (Datei *sqrtoject.h*):

```
#ifndef _MYTEST_H_
#define _MYTEST_H_

#include <dcopobject.h>

class SqrtObject : virtual public DCOPObject
{
    K_DCOP

public:
    SqrtObject (const QString &objID);

    k_dcop:
        double calculateSquareRoot (double z);

};

#endif
```

Unsere Klasse ist damit definiert. Sie enthält eine einzige DCOP-Methode namens *calculateSquareRoot*. Sie bekommt als Parameter einen Wert vom Typ *double* und liefert auch einen *double*-Wert als Rückgabetypp.

Welche Datentypen sind als Parameter eigentlich erlaubt? Erlaubt sind alle, die mit *QDataStream* in ein *QByteArray* geschrieben werden können. Nicht erlaubt sind allerdings Zeiger oder Referenzen, denn es können nur Werte per DCOP-Aufruf übergeben werden. Adressen von Speicherbereichen sind nicht möglich. Eine Ausnahme gibt es allerdings: Konstante Referenzen sind erlaubt. In diesem Fall wird der Wert übergeben und keine Referenz.

Nun sind nur zwei weitere Programmaufrufe nötig:

```
% dcpidl sqrtoobject.h >sqrtoobject.kidl
% dcpidl2cpp sqrtoobject.kidl
```

Der erste Befehl liest die relevanten Daten aus unserer Header-Datei aus und speichert sie in der Datei *sqrtoobject.kidl*. (*kidl* steht übrigens als Abkürzung für *KDE Interface Definition Language*, Sie können hier aber auch eine beliebige andere Endung benutzen.) Diese Datei ist ein XML-Dokument. Der zweite Befehl erzeugt aus dieser XML-Datei drei C++-Dateien: *sqrtoobject_skel.cpp*, *sqrtoobject_stub.h* und *sqrtoobject_stub.cpp*. Die letzten beiden Dateien sind im Moment noch nicht wichtig, sie werden erst beim Aufruf der DCOP-Methode von anderen Programmen aus benötigt.

Jetzt müssen wir noch die deklarierten Methoden mit Code füllen. Dazu erstellen wir die Datei *sqrtoobject.cpp*. Sie enthält in diesem Fall nur den Konstruktor sowie die Methode *calculateSquareRoot* der Klasse *SqrtObject*. Der Code sieht folgendermaßen aus:

```
#include "sqrtoobject.h"

SqrtObject::SqrtObject (const QCString &objID)
    : DCOPObject (objID)
{
}

double SqrtObject::calculateSquareRoot (double z)
{
    return sqrt (z);
}
```

Beachten Sie, dass wir im Konstruktor den Konstruktor von *DCOPObject* aufrufen. Um die Objekt-ID setzen zu können, benötigt also auch der Konstruktor von *SqrtObject* einen Parameter.

Um unser DCOP-Objekt einzusetzen, müssen wir nur noch in unserem Programm ein Objekt dieser Klasse erzeugen und ihm im Konstruktor die gewünschte Objekt-ID übergeben. Ein entsprechendes Hauptprogramm kann dann zum Beispiel so aussehen:

```
#include "sqrtoobject.h"
#include <kapp.h>
#include <dcpclient.h>
#include <qpushbutton.h>

int main (int argc, char **argv)
{
```

```

KApplication app (argc, argv, "SqrtServer");

// Registrierung als Applikation "SqrtServer"
QString appId = app.dcopClient()->registerAs (app.name(), false);
QDebug ("Registered as %s", appId.ascii());

// Erzeugen des DCOP-Objekts mit Namen "sqrtobject"
SqrtObject server ("sqrtobject");

// Ein Button, mit dem der Server beendet werden kann
QPushButton *bt = new QPushButton ("Quit Server", 0);
bt->show();
QObject::connect (bt, SIGNAL (clicked()),
                  &app, SLOT (quit()));
app.setMainWidget (bt);

return app.exec();
}

```

Um das Programm zu erstellen, müssen Sie nun sowohl die von Ihnen geschriebene Datei *sqrtobject.cpp* als auch die automatisch generierte Datei *sqrtobject_skel.cpp* in die Projektdatei bzw. das Makefile mit aufnehmen.

In vielen Fällen schreiben Sie in Ihrem Programm ohnehin eine eigene Klasse für das Hauptfenster, beispielsweise abgeleitet von *KMainWindow*. In dieser Klasse können Sie auch die DCOP-Methoden deklarieren. Dabei müssen Sie allerdings folgende Punkte beachten:

- Leiten Sie Ihre eigene Klasse per Mehrfachvererbung sowohl von einer Widget-Klasse (z.B. *KMainWindow*) als auch von *DCOPObject* ab. Beachten Sie dabei, dass die Widget-Klasse in der Liste der Vaterklassen unbedingt an der ersten Stelle stehen muss.
- Am Anfang der Klassendeklaration müssen Sie beide Makros *Q_OBJECT* und *K_DCOP* einsetzen.
- Sie können in den Methodendeklarationen beliebig Blöcke mit normalen Methoden, Signal-Methoden, Slot-Methoden und DCOP-Methoden mischen. Es ist aber nicht möglich, eine Slot- oder Signal-Methode auch gleichzeitig zur DCOP-Methode zu machen.
- Im Konstruktor müssen Sie auch die Konstruktoren der Widget-Klasse und der Klasse *DCOPObject* aufrufen. Wird in Ihrem Programm nur ein einziges Hauptfenster angelegt, kann die Objekt-ID als konstanter String eingesetzt werden; ansonsten empfiehlt es sich aber, die Objekt-ID als Parameter des Konstruktors Ihrer eigenen Klasse zu benutzen.
- Auf die Header-Datei müssen Sie sowohl das Programm *moc* als auch die Programme *dcopidl* und *dcopidl2cpp* anwenden, und Sie müssen beide erzeugten Programmdateien zur ausführbaren Datei hinzulinken.

Die entsprechende Klasse kann dann beispielsweise so aussehen:

```
#include <kmainwindow.h>
#include <dcopobject.h>

class MyMainWindow : public KMainWindow, public DCOPObject
{
    Q_OBJECT
    K_DCOP
public:
    MyMainWindow (QWidget *parent, const char *name,
                  const QString &objId);

    ...
    k_dcop:
    ...
    signals:
    ...
};
```

Um nun die DCOP-Methoden unseres Programms von einem anderen Programm aus aufzurufen, bedient sich dieses der Stub-Klasse (Stummel-Klasse). Diese Klasse wird durch die von *dcopidl2cpp* erzeugten Dateien *sqrtoject_stub.h* und *sqrtoject_stub.cpp* definiert. Diese Stub-Klasse bietet genau die gleichen Methoden, die wir als DCOP-Methoden deklariert haben, füllt sie aber nicht mit dem Code zur Berechnung der Wurzel, sondern füllt sie mit Code zum Aufruf der entsprechenden Methode im Server per DCOP. Im Konstruktor der Stub-Klasse geben Sie dazu die Application-ID und die Objekt-ID an, unter der das Objekt vom DCOP-Server angesprochen wird. Abbildung 4.60 verdeutlicht den Ablauf eines Aufrufs nochmals.

Der Code dazu sieht folgendermaßen aus:

```
// Stub-Objekt anlegen
SqrtObject_stub *server_stub =
    new SqrtObject_stub ("SqrtServer", "sqrtoject");

// DCOP-Aufruf ausführen
result = server_stub->calculateSquareRoot (value);

// Prüfen, ob Aufruf erfolgreich war
if (!server_stub->ok())
    QMessageBox::warning (this, "DCOP Error",
                           "Fehler beim Aufruf des Wurzelservers");
else
    cout << "Die Wurzel aus " << value
          << " ist " << result << endl;
```

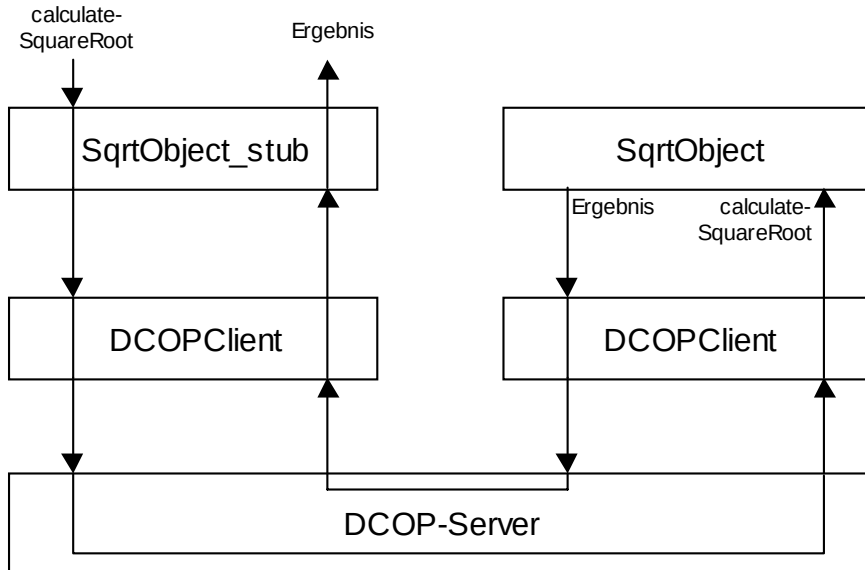


Abbildung 4-60 Aufrufreihenfolge bei einem DCOP-Aufruf

Das Stub-Objekt brauchen Sie natürlich nicht für jeden Aufruf neu anzulegen. In der Regel legen Sie es einmal an und benutzen es dann für beliebig viele DCOP-Zugriffe.

Die Weiterleitung des Aufrufs durch die *DCOPClient*-Objekte und über den DCOP-Server erfolgt in der Regel sehr schnell. Die Bearbeitung des Befehls kann aber natürlich eine unbestimmte Zeit in Anspruch nehmen. Bis das Endergebnis nicht zurückübermittelt wurde, bleibt das Client-Programm aber blockiert. Der Aufruf der Stub-Methode kehrt erst dann zum Aufrufer zurück, wenn das Ergebnis feststeht. Es wird allerdings eine zusätzliche Event-Schleife gestartet, so dass zwar Maus- und Tastatureingaben nicht bearbeitet werden, alle anderen Events (z.B. das Neuzeichnen von Fensterinhalten) aber weitergeleitet und ausgeführt werden.

Anders ist das bei DCOP-Methoden, die als Rückgabetypp den Spezialtyp *ASYNC* erhalten haben. Sie liefern keinen Wert zurück, und die Stub-Methode kehrt direkt nach dem Absetzen des Befehls an den DCOP-Server zum Aufrufer zurück. Das Programm wird dabei also nicht blockiert. Sie haben allerdings auch keine Möglichkeit festzustellen, ob der Befehl erfolgreich ausgeführt werden konnte.

4.20.5 DCOP-Signale

Will ein Programm eine wichtige Änderung bekanntgeben, ohne zu wissen, welche anderen Programme an dieser Information interessiert sind, sind die Möglichkeiten von DCOP, die wir bisher kennen gelernt haben, nicht ausreichend. Stattdessen ist es nötig, dass sich ein Programm für eine solche Information anmeldet. Zu diesem Zweck wurde der DCOP-Server um die Möglichkeiten von so genannten *DCOP-Signalen* erweitert. Ähnlich wie beim Signal-Slot-Konzept von Qt geschieht eine solche Anmeldung mit Hilfe eines *connect*-Befehls.

Beachten Sie aber, dass dieses Konzept *nicht* auf dem Signal-Slot-Konzept von Qt basiert. Es wurden nur die Bezeichnungen übernommen, da es sich um ähnliche Vorgänge handelt.

DCOP-Signale müssen nicht deklariert oder angemeldet werden. Sie werden einfach durch einen Aufruf von *DCOPClient::emitDCOPSignal* an den DCOP-Server gesendet, der entsprechend alle Programme benachrichtigt, die sich für dieses Signal angemeldet haben. Dementsprechend tauchen DCOP-Signale auch nicht in einer Klassendeklaration auf. Umso wichtiger ist es deshalb, dass die Dokumentation eines Programms auch genaue Informationen darüber enthält, welche DCOP-Signale es verschickt und welches Datenformat die übermittelten Parameter besitzen. Diese Informationen müssen sorgfältig geprüft werden. Stimmen sie nicht genau – zum Beispiel weil einer der Parametertypen falsch angegeben wurde –, so verbinden sich die anderen Programme mit einem falschen Signal und werden daher nie benachrichtigt.

Betrachten wir zunächst als Beispiel ein Programm, das den Akku-Ladezustand eines Notebooks überwacht. Sinkt der Ladezustand unter einen kritischen Wert, soll ein entsprechendes DCOP-Signal ausgesendet werden. Alle Programme, die kritische Daten enthalten, können dieses Signal abfangen und entsprechend darauf reagieren.

Zunächst müssen wir dem Signal einen treffenden Namen geben. Wir wählen hier »powerLow«. Als Parameter sollen im Signal noch der Ladezustand in Prozent und die geschätzte Restdauer in Minuten übertragen werden. Der Aufruf sieht dann etwa folgendermaßen aus:

```
// Werte für die Parameter ermitteln
int percent = akku.percent();
int restTime = akku.restTime();
// Parameter - wie üblich - in einem QByteArray speichern
QByteArray params;
QDataStream (params, IO_WriteOnly) << percent << restTime;
// DCOP-Signal senden
DCOPClient *client = kapp->dcopClient();
client->emitDCOPSignal ("powerLow(int,int)", params);
```

Ist ein anderes Programm nun daran interessiert, über ein solches Ereignis informiert zu werden, so muss es sich beim DCOP-Server mit diesem Signal verbinden lassen. Dabei gibt es eine DCOP-Methode an, die aufgerufen werden soll. Diese Methode benötigt exakt die gleichen Parametertypen wie das Signal, damit eine Verbindung zustande kommen kann. Genau wie bei Signal-Slot-Verbindungen von Qt darf dabei die DCOP-Methode beliebig viele (auch alle) Signal-Parameter am Ende der Parameterliste ignorieren.

Schauen wir uns an, wie beispielsweise ein Editor so erweitert werden kann, dass er dieses Signal empfängt und entsprechend darauf reagieren kann (mit einer automatischen Speicherung der Daten beispielsweise). Dazu benötigen wir zunächst einmal eine DCOP-Methode. Diese kann entweder keinen, einen oder zwei *int*-Parameter besitzen. In diesem Fall wollen wir beide Werte (Prozentsatz des Ladestandes und Restdauer) erhalten, stattdessen die Methode also mit zwei *int*-Parametern aus. Diese Methode fügen wir in die zentrale Editor-Klasse ein. Im Konstruktor dieser Klasse verbinden wir dann diese Methode mit dem Signal.

```
class MyEditor : public KMainWindow, virtual public DCOPObject
{
    Q_OBJECT
    K_DCOP
public:
    MyEditor ();
    k_dcop:
        void emergencySaveData (int, int);
        ....
};

MyEditor::MyEditor ()
    : KMainWindow (), DCOPObject ("myEditor")
{
    ....
    DCOPClient *client = kapp->dcopClient();
    bool ok = client->connectDCOPSignal
        ("kpowerchecker",      // appId der sendenden Applikation
         "powerLow(int,int)",  // Signal
         objId(),              // Empfänger-Objekt
         "emergencySaveData(int,int)", // Empfänger-Methode
         true);                // Volatile-Wert
    if (ok == false)
    {
        // Fehlermeldung, weil das Programm
        // kpowerchecker nicht läuft
    }
}

void MyEditor::emergencySaveData (int percent, int restTime)
```



```
{  
    // Hier können nun die Daten gesichert werden, evtl.  
    // abhängig von den Werten in percent oder restTime  
    ....  
}
```

Da diese Klasse die Deklaration für eine DCOP-Methode enthält, muss natürlich mit den Programmen *dcopidl* und *dcopidl2cpp* der Code für den Aufruf der Methoden erzeugt werden, so wie es im letzten Abschnitt erläutert wurde.

Die Verbindung wird durch die Methode *DCOPClient::connectDCOPSignal* hergestellt. Wir haben hier angenommen, dass unser Programm, das den Ladezustand überwacht, sich unter dem Namen *kpowerchecker* beim DCOP-Server registriert hat. Der letzte Parameter der Methode ist der so genannte *Volatile-Wert* (*Flüchtigkeitswert*). Er gibt an, ob die Verbindung auf die Laufzeit des sendenden Programms beschränkt ist (*true* = flüchtige Verbindung) oder nicht (*false* = nichtflüchtige Verbindung).

Eine *flüchtige Verbindung* benutzen Sie dann, wenn das sendende Programm permanent im Hintergrund läuft. Für eine flüchtige Verbindung muss das sendende Programm bereits zum Zeitpunkt des Verbindungsaufbaus gestartet und beim DCOP-Server registriert sein. Ist dies nicht der Fall, so zeigt der Rückgabewert einen Fehler an. Eine *nicht-flüchtige* Verbindung bleibt dagegen auch erhalten, wenn das sendende Programm beendet und neu gestartet wird. Auch braucht das sendende Programm zum Zeitpunkt des Verbindungsaufbaus noch nicht gestartet worden zu sein. Eine nichtflüchtige Verbindung benutzen Sie dann, wenn das sendende Programm nur bei Bedarf gestartet wird und dann eventuell seine Signale schickt, wenn es also nicht permanent im Hintergrund läuft.

In unseren Beispielen oben wurden die Signale von der Applikation versendet, nicht von einem speziellen DCOP-Objekt. Sie haben auch die Möglichkeit, eine Verbindung gezielt zu einem DCOP-Signal eines speziellen Objekts aufzubauen. Dazu gibt es überladene Varianten von *DCOPClient::emitDCOPSignal* und *DCOPClient::connectDCOPSignal*. Diese Varianten benötigen als zusätzlichen Parameter die Objekt-ID eines speziellen DCOP-Objekts. Zur Zeit ist diese Angabe jedoch noch etwas aus der Luft gegriffen, da die übergebene Objekt-ID mit keinem existierenden DCOP-Objekt korrespondieren muss. Voraussichtlich wird jedoch in einer der nächsten Versionen der Programme *dcopidl* und *dcopidl2cpp* die Möglichkeit gegeben sein, auch für eigene DCOP-Objekte Signale zu deklarieren, die dann genau wie Qt-Signale einfach aufgerufen werden, um das entsprechende Signal zu versenden. Ein Aufruf von *DCOPClient::emitDCOPSignal* wird dann hin-fällig.

4.20.6 Zusammenfassung DCOP

Ab KDE 2.0 ist im Standard das einfache Protokoll *DCOP* enthalten, das den Austausch von Nachrichten zwischen verschiedenen KDE-Programmen ermöglicht.

Für DCOP-Aufrufe gilt:

- Im Hintergrund muss der DCOP-Server laufen.
- Jedes Programm, das kommunizieren will, muss sich mit Hilfe eines Objekts der Klasse *DCOPClient* beim Server anmelden. Die Verwaltung dieses Objekts übernimmt die Klasse *KApplication*.
- Zum Aufruf einer DCOP-Methode in einem anderen Programm verwendet man die Methoden *DCOPClient::send* für einen einfachen Aufruf ohne Rückgabewert bzw. die Methode *DCOPClient::call*, um auch den Rückgabewert der Methode zu erhalten.
- Beim Aufruf müssen Sie die *appId* der Applikation, die Objekt-ID, die Methodensignatur (Name und Liste der Parametertypen) sowie die Werte für die Parameter (gepackt in einem *QByteArray*) angeben. Beim Aufruf von *call* erhalten Sie den Datentyp des Rückgabewerts sowie den Wert selbst (ebenfalls gepackt in einem *QByteArray*) zurück.
- Als Parametertypen sind alle Datentypen erlaubt, die mit *QDataStream* verarbeitet werden können. Nicht erlaubt sind Zeiger und Referenzen. Für weitere Datentypen können die Operatoren *<<* und *>>* von *QDataStream* entsprechend überladen werden, so dass auch sie benutzt werden können.
- Referenzen auf DCOP-Objekte können mit Hilfe des Datentyps *DCOPRef* als Parameter übertragen werden.
- Um eigene Methoden zu erstellen, die von anderen Programmen per DCOP aufgerufen werden können, benötigt man eine von der Klasse *DCOPObject* abgeleitete Klasse, in der die rein virtuelle Methode *process* überschrieben wird. Es empfiehlt sich, die Programme *dcopidl* und *dcopidl2cpp* zu benutzen, um eine solche Klasse zu erstellen.
- Um eine eigene Klasse um DCOP-Methoden zu erweitern, leiten Sie sie (in der Regel per Mehrfachvererbung) von *DCOPObject* ab. An den Anfang der Klasse stellen Sie das Makro *K_DCOP*. Die Methoden, die per DCOP aufgerufen werden sollen, stellen Sie in einen eigenen Block der Kategorie, den Sie mit *k_dcop:* einleiten. Bei Methoden ohne Rückgabewert, die nur mit *send* und nicht mit *call* aufgerufen werden sollen, geben Sie als Rückgabebetyp *ASYNc* an.
- Im Konstruktor dieser eigenen Klasse rufen Sie den Konstruktor der Klasse *DCOPObject* auf, dem Sie als Parameter einen innerhalb der Applikation eindeutigen Namen mitgeben. Dieser Name kann auch durch die Adresse im Speicher oder aus dem Qt-Objektnamen automatisch gebildet werden.

- Wenden Sie das Programm *dcopidl* auf die Datei an, die die Deklaration Ihrer Klasse enthält (also in der Regel eine Header-Datei). Dies erzeugt eine XML-Datei mit den notwendigen Informationen. Aus dieser Datei können Sie mit dem Programm *dcopidl2cpp* automatisch eine *cpp*-Datei erzeugen, die den notwendigen zusätzlichen Code enthält. Diesen lassen Sie ebenfalls kompilieren und linken.
- Mit Hilfe von *DCOP-Signalen* kann eine Applikation eine Information an die Außenwelt weitergeben, ohne zu wissen, welche anderen Programme sich für diese Information interessieren (Methode *DCOPClient::emitDCOPSignal*). Andere Programme, die diese Signale erhalten wollen, müssen sich beim DCOP-Server anmelden (Methode *DCOPClient::connectDCOPSignal*).

4.21 Komponenten-Programmierung mit KParts

In der Software-Branche ist Wiederverwertbarkeit (Reusability) ein zentraler Begriff geworden. Viele Programme sind inzwischen so komplex, dass es unmöglich ist, sie vollständig neu zu programmieren. Stattdessen wird ein Programmierstil angestrebt, bei dem man im besten Fall ein Programm nur noch aus einer Hand voll Komponenten »zusammensteckt«. Dieses Konzept wird durch den Einsatz von Klassen bereits teilweise genutzt. Widget-Klassen bilden die vordefinierten Komponenten, und mit nur wenigen Zeilen Code kann man sehr komplexe Benutzeroberflächen zusammenstellen.

Klassen haben aber noch einen kleinen Nachteil: Sie besitzen eine ganz individuelle Schnittstelle, die aus den öffentlichen Methoden gebildet wird. Diese Methoden muss man als Nutzer einer Klasse genau kennen. Dynamisch zur Laufzeit eine vorher nicht weiter bekannte Komponente zum Programm hinzuzubinden ist damit weit gehend unmöglich. (Das Property-System der *QObject*-Klasse ermöglicht es zumindest teilweise, dass man auch unbekannte Klassen nutzen kann; für echte Komponenten ist das aber noch zu wenig. Siehe auch Kapitel 3.1.4, *Informationen über die Klassenstruktur*.)

4.21.1 Das KParts-Komponentensystem

KParts ist ein System, das diesen Nachteil behebt. Eine Komponente in *KParts* besteht aus einem Widget und einer Menge von Einträgen für die Menüleiste und die Werkzeugleiste, mit denen der Anwender die Daten im Widget manipulieren kann.

Das Programm, das wohl am meisten von *KParts*-Komponenten profitiert, ist der KDE-Dateimanager *Konqueror*. Wenn Sie beispielsweise eine Textdatei im Dateimanager anklicken, öffnet sich nicht ein eigenes Fenster mit dem Inhalt der Datei, sondern der Editor (im Defaultfall *KWrite*) wird im *Konqueror*-Fenster selbst

geöffnet (siehe Abbildung 4.61). Das *KWrite*-Editorfenster ist dabei eine Komponente, die bei Bedarf von *Konqueror* nachgeladen wird. Auch die Menüs von *Konqueror* werden entsprechend angepasst. So bekommt der Menüpunkt BEARBEITEN zusätzliche Einträge wie SUCHEN..., WEITERSUCHEN oder GEHE ZU ZEILE.... Dieser Vorgang ähnelt dem *OLE (Object Linking and Embedding)*, wie es unter Microsoft Windows bekannt ist.

Konqueror selbst weiß nicht, welche Komponenten vorhanden sind und welche Komponente für welchen Dateityp eingesetzt werden soll. Insbesondere werden die Komponenten nicht beim Linken eingebunden (was einen enormen Speicherbedarf zur Folge hätte), sondern sie werden dynamisch bei Bedarf nachgeladen. Dazu werden dynamische Bibliotheken verwendet.

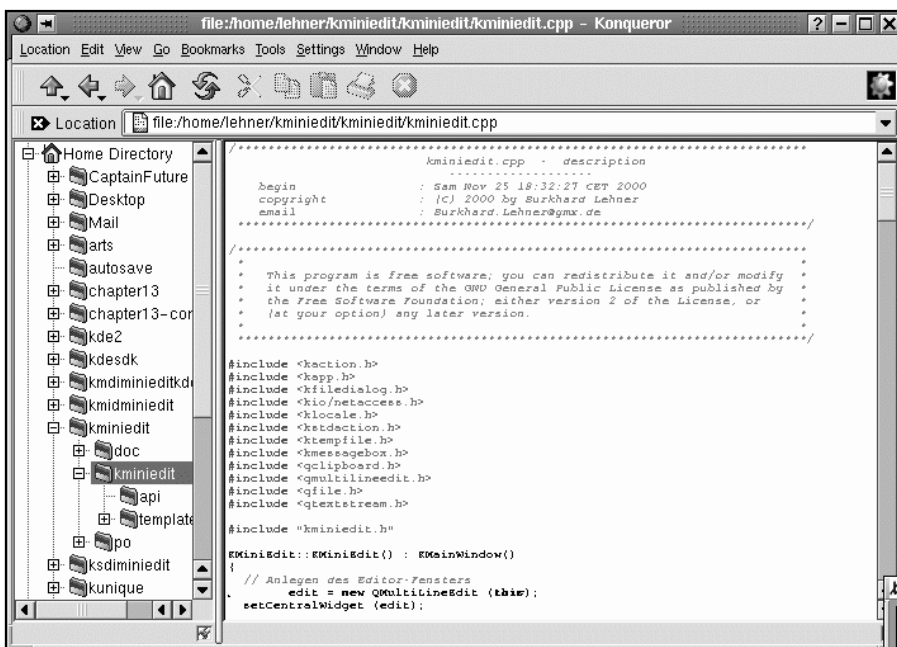


Abbildung 4-61 Konqueror mit einer eingebetteten KWrite-Komponente

Für einen Entwickler einer Anwendung gibt es drei typische Einsatzgebiete für Komponenten:

- Eine existierende Komponente wird in das eigene Programm eingebunden. Ähnlich wie mit den vordefinierten Widgetklassen kann man so mit wenigen Zeilen Code ein leistungsfähiges Programm erstellen. Auf einfache Weise kann man so zum Beispiel den Texteditor *KWrite* oder den Bildbetrachter *KView* im eigenen Programm nutzen. Die Komponenten kann man dabei entweder fest einbinden, also zum eigenen Programm hinzulinken, oder aber auch dynamisch zur Laufzeit nur bei Bedarf einbinden lassen.

- Eine neue Komponente wird entwickelt, die für andere Programme interessant sein könnte. In der Regel implementiert man zu dieser Komponente außerdem noch eine ganz einfache Applikation, die diese Komponente nutzt. Neue Komponenten sind zum Beispiel nützlich, wenn Sie einen neuen Dokumenttyp bearbeiten wollen.
- Eine neue Komponente wird in den Dateimanager *Konqueror* eingebettet, um Dateien in bestimmten Formaten anzeigen zu können. Dazu müssen die Informationen über die neue Komponente in die Konfigurationsdateien von KDE eingefügt werden.

4.21.2 Einsatz existierender Komponenten in eigenen Programmen

Um eine fremde Komponente im eigenen Programm einsetzen zu können, brauchen Sie nicht viel von dieser Komponente zu wissen: den Namen der Bibliothek, die die Komponente enthält, und den Klassentyp, den die Komponente bietet. Insbesondere benötigen Sie keine speziellen Header-Dateien für die Komponente.

Tabelle 4.24 enthält einige Komponenten und die dazugehörige Bibliothek.

Komponente	Aufgabe	enthalten in Bibliothek
<i>KWrite</i>	Anzeigen und Ändern von Textdateien	<i>libkwritepart</i>
<i>KView</i>	Anzeigen von Bildern	<i>libkviewpart</i>
<i>KHTML</i>	Anzeigen von HTML-Dateien	<i>libkhtml</i>
<i>KGhostView</i>	Anzeigen von PostScript-Dateien	<i>libkghostview</i>

Tabelle 4-24 Einige in KDE enthaltene Komponenten

Als Klassentyp werden hauptsächlich zwei Klassen verwendet: *KParts::ReadOnlyPart* für Komponenten, die nur anzeigen, aber nicht ändern können (Betrachter), und *KParts::ReadWritePart* für Komponenten, die anzeigen und auch verändern können (Editoren). Viele Komponenten stellen beide Klassentypen zur Verfügung, so dass Sie selbst entscheiden können, welchen Klassentyp Sie benutzen wollen. Beide Klassen sind übrigens von *KParts::Part* abgeleitet, und *KParts::ReadWritePart* ist eine Unterklasse von *KParts::ReadOnlyPart*.

Einbinden von Komponenten

Komponenten werden in nahezu allen Fällen erst zur Laufzeit des Programms bei Bedarf eingebunden. Dazu wird die Klasse *KLibLoader* verwendet. Diese Klasse lädt die Komponenten-Bibliothek in den Speicher und liefert ein so genanntes *Factory*-Objekt zurück. Dieses Objekt kann nun benutzt werden, um beliebig viele Objekte der Komponente zu erzeugen. (In der Regel benötigt man nur ein Objekt.)

Um eine *KView*-Komponente anzulegen, wird folgender Code benutzt:

```
// Zunächst die Bibliothek suchen und laden.
// Rückgabe ist ein Objekt vom Typ KLibFactory.
KLibFactory *factory;
factory = KLibLoader::self()->factory ("libkviewpart");

// Fehlerfall abfangen (NULL-Zeiger)
if (!factory)
{
    qDebug ("Fehler: libkviewpart nicht gefunden!");
    return;
}

// Nun ein Komponenten-Objekt erzeugen
QObject *help;
help = factory->create (this, "view-Komponente",
                      "KParts::ReadOnlyPart");

// Zur Sicherheit testen, ob auch tatsächlich ein
// Objekt der richtigen Klasse erzeugt wurde.
if (!help->inherits ("KParts::ReadOnlyPart"))
{
    qDebug ("Fehler: Objekt hat falsche Klasse!");
    return;
}

// Nun kann problemlos gecastet werden
KParts::ReadOnlyPart *komponente;
komponente = (KParts::ReadOnlyPart *) help;
```

Der Aufruf von *create* benötigt drei Parameter: Der erste Parameter legt das Vaterobjekt der Komponente fest, der zweite Parameter den Objektnamen. Der dritte Parameter gibt an, von welchem Klassentyp das erzeugte Objekt sein soll. In unserem Fall wollen wir also ein *KParts::ReadOnlyPart*-Objekt erzeugen lassen. (Das tatsächlich erzeugte Objekt kann – und wird wahrscheinlich auch – einer Unterklasse angehören. In unserem Fall ist die zurückgelieferte Klasse beispielsweise *KViewPart*.)

Das erzeugte Komponenten-Objekt enthält alle wichtigen Informationen zur Bearbeitung. Außerdem erzeugt es automatisch ein *QWidget*-Objekt, das für die Anzeige zuständig ist. Dieses Widget erhält als Vater-Widget ebenfalls das Vaterobjekt. Daher sollte das Vaterobjekt immer das Widget sein, das die Komponente einbetten soll, in der Regel also das Hauptfenster. Über die Methode *KParts::widget* können Sie sich die Adresse dieses Widgets besorgen. Das ist aber in der Regel gar nicht nötig, da die Komponente selbst dafür sorgt, dass alles korrekt bearbeitet wird.

Dennoch müssen einige Operationen mit der Komponente durchgeführt werden. *KParts::ReadOnlyPart* und *KParts::ReadWritePart* stellen ein Dokument dar, das in einer Datei abgelegt ist (oder auch per FTP oder HTTP von einem Server geladen werden kann). Die Klassen stellen daher einige Methoden zur Verfügung, um die Datei festzulegen oder sie zu speichern. Tabelle 4.25 enthält eine Liste der wichtigsten Methoden von *KParts::ReadOnlyPart*. Tabelle 4.26 enthält die zusätzlichen Methoden, die in *KParts::ReadWritePart* noch hinzukommen (zum Speichern oder um festzustellen, ob das Dokument geändert wurde).

Methoden	Bedeutung
<i>openURL</i>	Öffnet das angegebene Dokument in der Komponente
<i>closeURL</i>	Schließt das Dokument wieder
<i>url</i>	Liefert die URL des aktuellen Dokuments zurück

Tabelle 4-25 Wichtige Methode von KParts: : ReadOnlyPart

Methoden	Bedeutung
<i>save</i>	Speichert das Dokument unter dem aktuellen Namen
<i>saveAs</i>	Speichert das Dokument unter einem anderen Namen
<i>isModified</i>	Liefert zurück, ob das Dokument geändert wurde (nur selten benötigt)

Tabelle 4-26 Wichtige zusätzliche Methoden von KParts: : ReadWritePart

Die angegebenen Methoden rufen Sie in der Regel in den üblichen Slots auf, die von den Menüpunkten OPEN, CLOSE, SAVE oder SAVEAS aufgerufen werden. (Dazu müssen Sie natürlich einen Zeiger auf das Komponenten-Objekt speichern.) Die Komponente selbst macht dabei fast alles Weitere: Sie liest die Datei ein, kopiert sie gegebenenfalls in eine lokale Datei oder speichert sie ab. Wenn Sie *closeURL* für ein geändertes Dokument in *KParts::ReadWritePart* aufrufen, fragt die Komponente automatisch nach, ob Sie die Datei speichern möchten.

Um die Speicherfreigabe brauchen Sie sich übrigens nicht zu kümmern: Sobald das Hauptfenster geschlossen wird, wird das Komponenten-Widget mitgelöscht, und das bewirkt gleichzeitig ein Löschen der Komponente selbst. Wenn Sie die Komponente vorzeitig entfernen möchten (z.B. um stattdessen eine andere Komponente zu benutzen), so können Sie auch gezielt die Komponente löschen.

Was nun noch fehlt, ist die Integration der speziellen Befehle der Komponente in die Menüstruktur des Hauptfensters. Aber auch das ist kein großes Problem. Es sind nur zwei Schritte notwendig:

- Ihr Hauptfenster muss nun nicht mehr von der Klasse *KMainWindow*, sondern von *KParts::MainWindow* abgeleitet sein.

- Nach dem Erzeugen der Komponente (mit dem Hauptfenster als Vater-Wid-
get) muss noch die Methode *KParts::MainWindow::createGUI (komponente)* auf-
gerufen werden, wobei *komponente* der Zeiger auf die neue Komponente ist.

Der Code für die Hauptfensterklasse sieht dann etwa folgendermaßen aus:

```
#include <kparts/mainwindow.h>
class Shell : public KParts::MainWindow
{
    Q_OBJECT
public:
    Shell();
    ~Shell();

    void openURL( const KURL & url );

protected slots:
    void slotFileOpen();

private:
    KParts::ReadOnlyPart *m_part;
};

Shell::Shell()
{
    KAction * paOpen = KStdAction::open
        (this, SLOT (slotFileOpen()), actionCollection());
    KAction * paQuit = KStdAction::quit
        (this, SLOT (close()), actionCollection());

    // Bibliothek libkghostview suchen
    KLibFactory *factory =
        KLibLoader::self()->factory( "libkghostview" );
    if (factory)
    {
        // Komponenten-Objekt erzeugen
        m_part = (KParts::ReadOnlyPart *)
            factory->create (this, "kgvpart",
                            "KParts::ReadOnlyPart");
        // Anzeigebereich setzen
        setView (m_gvpart->widget());
        // Befehle in die Menüs integrieren
        createGUI (m_gvpart);
    }
    else
        kdFatal() << "No libkghostview found !" << endl;
}

Shell::~~Shell()
```



```
{
    delete m_part;
}

void Shell::openURL( const KURL & url )
{
    m_part->openURL( url );
}
```

Hier sind nur die wichtigsten Teile aufgeführt, das vollständige Listing finden Sie auf der CD-ROM, die dem Buch beiliegt.

Wollen Sie mehrere Komponenten gleichzeitig innerhalb eines Hauptfensters geöffnet haben, so muss koordiniert werden, welche Komponente gerade aktiv ist. Danach entscheidet sich nämlich, welche Befehle in der Menüleiste und in den Werkzeugleisten enthalten sind. Dazu dient die Klasse *KParts::PartManager*.

4.22 Programme von Qt 1.x auf Qt 2.x portieren

Die Firma Trolltech geht bei der Erstellung einer neuen Version sehr behutsam vor, so dass alte Programme auch unter der neuen Version lauffähig bleiben. Man unterscheidet dabei zwei Fälle:

- *Binärkompatibel (binary compatible)* ist eine neue Bibliothek, wenn die alten kompilierten Programme ohne jede Änderung lauffähig bleiben.
- *Source-kompatibel (source compatible)* ist eine neue Bibliothek, wenn die alten Programme ohne Änderung neu kompiliert werden können und dann lauffähig sind.

Mit der Version 2.0 liegt nun aber eine Qt-Version vor, die weder binär- noch source-kompatibel zu den Versionen vor 2.0 ist. Trolltech tut diesen Schritt, der auch an der höheren Hauptversionsnummer zu erkennen ist, um ein paar grundlegende Änderungen einzuführen. Ebenso ändern sich ab der Version 2.0 die *Lizenzbestimmungen* für die Unix-Edition.

Es ist problemlos möglich, zwei verschiedene Versionen der *Qt-Bibliothek* auf dem Rechner installiert zu haben. Die alten Programme, die noch mit der alten Bibliothek gebunden worden sind, werden beim Starten automatisch nach der alten Bibliothek suchen. Programme, die mit der neuen Bibliothek gebunden wurden, greifen auch nur auf diese zu. Beim Umsteigen auf eine neue Qt-Version ist es also nicht nötig, alle Programme zu ändern und neu zu kompilieren.

Wenn Sie nun ein Programm im Quelltext vor sich haben und dieses an die neue Qt-Version anpassen wollen, müssen Sie eine Reihe von Punkten beachten.

4.22.1 Installieren der neuen Qt-Version

Für die eigene Programmentwicklung reicht es nicht aus, nur eine neue *Bibliothek* zu installieren. Sie benötigen darüber hinaus auch die neuen *Header-Dateien* und die neue Version des *Meta-Object-Compilers (moc)*. Diese sind im *Developers-Paket* enthalten, in dem Sie auch die neue Klassendokumentation sowie die Sourcen der Qt-Bibliothek finden. Achten Sie also darauf, dass Sie die Developers-Version vollständig installieren.

Achten Sie auch darauf, dass die neue Qt-Bibliothek (Dateiname *libqt*) in Ihrem Bibliothekspfad installiert ist. Die alte Bibliothek kann weiterhin installiert bleiben, sie unterscheidet sich von der neuen durch die Versionsnummer.

Beim Kompilieren müssen die neuen Header-Dateien im *Include*-Pfad vorhanden sein. Dort sollten unter keinen Umständen auch noch alte Header-Dateien gefunden werden. Am besten benennen Sie den Pfad mit den alten Header-Dateien um, wenn Sie das Verzeichnis nicht löschen wollen.

Achten Sie darauf, dass im Suchpfad für ausführbare Dateien die neue *moc*-Version zuerst gefunden wird. Der *moc* der Version 2.0 ist *nicht* zur alten Version kompatibel. Auch die späteren Qt-Versionen ergänzen *moc* um besondere Eigenschaften. Am sichersten ist es, wenn Sie die alte Version löschen oder umbenennen.

4.22.2 Kompilieren alter Programme

Nun können Sie versuchen, ein altes Programm mit der neuen Bibliothek zu übersetzen. Löschen Sie dazu alle *moc*-Dateien – meist reicht dazu ein Aufruf von »`make clean`«. In vielen Fällen wird sich das Programm einwandfrei übersetzen lassen, wobei die Wahrscheinlichkeit groß ist, dass das fertige Programm auch fehlerfrei läuft. Achten Sie aber auf jeden Fall auf Warnungen, die der Compiler meldet. Sie sind oft ein Hinweis auf kleinere Änderungen an den Typen der Übergabeparameter. Besonders bei Zeigern ergeben sich daraus leicht Fehler, die zum Absturz des Programms führen und die oftmals nicht einfach zu entdecken sind.

Aber auch, wenn keine Warnungen gemeldet werden und das Programm fehlerfrei läuft (oder zu laufen scheint), sollten Sie das Programm an die neue Bibliothek anpassen, um die neuen Möglichkeiten besser auszunutzen und das Programm effizienter zu machen. Schauen Sie sich die wichtigsten Neuerungen in Qt 2.0 an, die in den folgenden Unterkapiteln beschrieben werden.

4.22.3 Die neue Klasse *QString*

Eine der wichtigsten Änderungen in Qt 2.0 ist die Umstellung der *QString*-Klasse auf *Unicode-Zeichensätze*. Ein Zeichen eines Strings wird nun nicht mehr durch ein Byte im ASCII-Code dargestellt, sondern durch zwei Byte. Die alte Klasse ist

jedoch weiterhin vorhanden, nun aber unter dem Klassennamen *QCString*. Das »C« steht dabei für die normale C-Repräsentation einer Zeichenkette als Array von *char*, die durch das '\0'-Zeichen abgeschlossen wird.

Obwohl sie auf ganz unterschiedlichen Daten beruhen, sind die Schnittstellen der Klasse *QString* und der Klasse *QCString* fast identisch. In vielen Fällen wird also ein altes Programm auch unter Qt 2.0 kompilierbar sein. Die häufigste Ursache dafür, dass es nicht funktioniert, ist die Methode *data*, die in *QString* nicht mehr existiert. Diese Methode lieferte einen Zeiger auf die String-Daten zurück und umging dadurch auch die implizite gemeinsame Nutzung der String-Daten zwischen verschiedenen *QString*-Objekten.

Die einfachste Methode, Probleme mit der neuen Klasse *QString* zu beheben, besteht darin, jedes Auftreten von *QString* im Programmcode textuell durch *QCString* zu ersetzen. Danach sollte das Programm wieder kompilierbar sein, und zwar mit dem gleichen Verhalten wie zuvor. Ein automatischer Typecast zwischen *QCString* und *QString* sorgt dafür, dass an den Stellen, wo ein *QString* erwartet wird, auch ein *QCString* stehen kann.

Diese Methode macht aber keinen Gebrauch von den Möglichkeiten der neuen *QString*-Klasse, also insbesondere der Nutzung von Unicode. Eine etwas aufwendigere, aber bessere Methode ist es, jedes Auftreten der Methode *data* anzupassen. In den Fällen, in denen ein `const char*`-Typ erwartet wird, kann der Methodenaufruf einfach entfallen. Ein automatischer Typecast sorgt für die Umwandlung. Zum Beispiel kann

```
QString s = "ABCDE";  
QObject *obj = new QObject (this, s.data());
```

ersetzt werden durch:

```
QString s = "ABCDE";  
QObject *obj = new QObject (this, s);
```

An Stellen, an denen kein expliziter Typ erwartet wird, z.B. in den Ellipsenparametern einer Funktion oder Methode (das sind die Parameter einer Funktion, die durch die Auslassung »...« gekennzeichnet sind, z.B. in `printf` und ähnlichen Funktionen), kann man die Umwandlung durch einen *expliziten Typecast* oder Aufruf der Methode *ascii* erzielen. Zum Beispiel kann man

```
QString s = "ABCDE";  
printf ("String enthält %s\n", s.data());
```

durch

```
QString s = "ABCDE";  
printf ("String enthält %s\n", (const char *) s);
```

oder durch

```
QString s = "ABCDE";  
printf ("String enthält %s\n", s.ascii());
```

ersetzen. An Stellen, an denen ein `char*`-Typ erwartet wird, obwohl der `String` nicht verändert wird (das ist bei einigen alten C-Funktionen der Fall, bei denen vergessen wurde, in die Header-Datei ein `const` einzutragen), kann man sich damit behelfen, dass man die Methode `ascii` aufruft und das Ergebnis einem expliziten Typecast auf `(char*)` unterwirft. An Stellen, an denen ein `char*`-Typ erwartet wird *und* die `String`-Daten verändert werden, war bereits die alte Version nicht sauber programmiert, und man sollte sich eine sauberere Lösung überlegen.

Auch diese zweite Methode hat noch einen Schönheitsfehler: Sie benutzt überall die neue *QString*-Klasse auf Unicode-Basis, auch wenn das gar nicht nötig wäre, zum Beispiel weil es sich ausschließlich um einen internen `String` handelt, der nie ausgegeben wird, also auch keine internationalen Sonderzeichen enthalten wird. Optimal ist es daher sicherlich, wenn man in jedem Fall genauer entscheidet, ob es sich um einen internen `String` handelt oder nicht. Im ersten Fall wird man die Klasse *QCString* einsetzen, im zweiten Fall den `String` an die Methoden der neuen Klasse *QString* anpassen. Dabei sollte auch beachtet werden, dass die Anzahl der Umwandlungen von *QCString* nach *QString* und umgekehrt möglichst klein bleibt.

4.22.4 Aufzählungstypen und Konstanten werden in der Klasse *Qt* zusammengefasst

In der Vergangenheit kam es häufiger zu Problemen mit Bezeichnern, die in den *Qt*-Header-Dateien definiert wurden. Um diese »Verschmutzung« des Namensraums einzudämmen, sind alle ehemals globalen *Aufzählungstypen* und *Konstanten* in der Klasse *Qt* definiert, die in der Header-Datei *qnamespace.h* enthalten ist. Die Klasse *QObject* sowie einige andere Klassen, die besonders häufig auf diese Bezeichner zugreifen, sind nun von dieser Klasse abgeleitet, so dass sich innerhalb dieser Klassen keine Änderung gegenüber vorher ergibt. Außerhalb dieser Klassen kann auf die Aufzählungstypen und Konstanten jedoch nicht mehr direkt zugegriffen werden. An den Stellen, an denen der Compiler also einen unbekannten Bezeichner meldet, reicht es aus, die Klassenangabe »*Qt::*« voranzustellen. Aus `black` wird so zum Beispiel `Qt::black`.

Die Klasse *Qt* enthält nun:

- die *Farbkonstanten* (*color0*, *color1*, *black*, *white*, ...)
- die *Cursor-Konstanten* (*arrowCursor*, *upArrowCursor*, *crossCursor*, ...)

- den Aufzählungstyp *ButtonState* (*NoButton*, *LeftButton*, *RightButton*, ...)
- den Aufzählungstyp *Orientation* (*Horizontal*, *Vertical*)
- den Aufzählungstyp *AlignmentFlags* (*AlignLeft*, *AlignTop*, ...)
- den Aufzählungstyp *WidgetFlags* (*WType_TopLevel*, *WStyle_Title*, ...)
- den Aufzählungstyp *ImageConversionFlags* (*AutoColor*, *ColorOnly*, ...)
- den Aufzählungstyp *BGMode* (*TransparentMode*, *OpaqueMode*)
- den Aufzählungstyp *PaintUnit* (*PixelUnit*, *LoMetricUnit*, ...)
- den Aufzählungstyp *GUIStyle* (*WindowsStyle*, *MotifStyle*)
- den Aufzählungstyp *Modifier* (*SHIFT*, *CTRL*, ...)
- den Aufzählungstyp *Key* (*Key_A*, *Key_B*, *Key_Up*, *Key_F1*, ...)
- den Aufzählungstyp *ArrowType* (*UpArrow*, *DownArrow*, ...)
- den Aufzählungstyp *RasterOp* (*CopyROP*, *OrROP*, ...)
- den Aufzählungstyp *PenStyle* (*NoPen*, *SolidLine*, ...)
- den Aufzählungstyp *BrushStyle* (*NoBrush*, *SolidPattern*, ...)
- den Aufzählungstyp *WindowsVersion* (*WV_NT*, *WV_95*, ...).

4.22.5 Vereinfachtes Layout-Management in Qt 2.0

In Qt 2.0 ist das *Layout-Management* mit den Klassen *QBoxLayout* und *QGridLayout* stark vereinfacht worden. Code, der für eine alte Qt-Version geschrieben wurde, ist zwar fehlerfrei kompilierbar, in einigen (seltenen) Fällen kann die Darstellung auf dem Bildschirm aber mit Qt 2.0 anders aussehen. Der Grund dafür ist, dass Widgets nun über eine unterschiedliche *SizePolicy* verfügen und daher vom Layout-Manager mit unterschiedlicher Priorität bei der Verteilung des freien Platzes berücksichtigt werden. Dieses Problem löst man in den meisten Fällen dadurch, dass man Elemente, die nun zu klein (da andere Elemente bevorzugt wurden), mit einem höheren *Stretch-Faktor* versieht. (Der *Stretch-Faktor* ist der letzte Parameter in der Methode *addWidget*. Für Elemente, die nun unter Qt 2.0 zu klein sind, ist er meist 0 oder wird weggelassen.)

Auch beim Layout-Management empfiehlt es sich, das Programm nicht nur erneut lauffähig zu machen, sondern die neuen Möglichkeiten auszunutzen. Im neuen Layout-Management machen Widgets selbst Angaben darüber, wie sie sich verhalten wollen, wenn sich der zur Verfügung stehende Platz ändert. Dazu gibt es ab Qt 2.0 die Methode *QWidget::sizePolicy*. Für die bereits fertigen Widgets aus der Qt-Bibliothek ist diese Methode bereits sinnvoll überschrieben, so dass sich schon automatisch ein meist sinnvolles Verhalten im Layout-Management

ergibt. Wenn Sie selbst neue Widgets definieren, sollten Sie ebenfalls diese Methode sinnvoll überschreiben. (Ab Qt 2.2 können Sie stattdessen auch die Methode `setSizePolicy` benutzen.) Ganz besonders wichtig ist es dann auch, die Methode `QWidget::sizeHint` zu überladen. Genauere Informationen zu diesem Thema finden Sie in Kapitel 4.4, *Entwurf eigener Widget-Klassen*, sowie in Kapitel 3.6, *Anordnung von GUI-Elementen in einem Fenster*.

Auch bei der Benutzung der fertigen Widgets können Sie Ihr Programm an die neuen Möglichkeiten anpassen. Da die Widgets selbst Angaben darüber machen, wie sie ihre Größe ändern wollen, ist es meist nicht mehr nötig, den Widgets eine *Mindestgröße* oder eine *feste Größe* zu geben. Zeilen der Form

```
w->setMinimumSize (w->sizeHint());
```

oder

```
w->setFixedSize (w->sizeHint());
```

oder auch komplexere Ausdrücke wie

```
w->setFixedHeight (w->sizeHint().height());  
w->setMinimumWidth (w->sizeHint().width());
```

können daher in den meisten Fällen einfach gelöscht werden, ohne dass sich die Darstellung auf dem Bildschirm ändert. (Oftmals ist die Layout-Aufteilung sogar günstiger als vorher.)

In manchen Programmen ist ein Makro `MINSIZE` definiert, das genau die oberen Zeilen vereinfacht. Auch diese Makro-Aufrufe können daher in der Regel einfach gelöscht werden.

Bei besonders einfachen Aufgaben können Sie den Einsatz von `QHBoxLayout` und `QVBoxLayout` durch die neuen Widgets `QHBox` und `QVBox` ersetzen sowie `QGridLayout` durch das Widget `QGrid`. Das vereinfacht das Listing und macht es etwas übersichtlicher. Informationen über die Anwendung der neuen Widgets finden Sie auch in Kapitel 3.6.3, *Einfache Layouts mit QHBox, QVBox und QGrid*.

4.22.6 QStyle als Ersatz für GUIStyle

Ab Qt 2.0 wird die Darstellung der Widgets – ob in einer Windows-ähnlichen oder einer an Motif angelehnten Art – nicht mehr durch den Aufzählungstyp `GUIStyle` festgelegt, sondern durch die Klasse `QStyle` und davon abgeleitete Klassen. Der Vorteil dieser Änderung besteht zum einen in einer einfacheren Erweiterung durch neue Darstellungsarten, da nun nicht mehr der Code der Widgets geändert werden muss. Zum anderen lassen sich auch neue Widgets, die zwischen den Darstellungen unterscheiden, einfacher implementieren, da sie von den virtuellen Methoden von `QStyle` Gebrauch machen können.

In diesem Zusammenhang wurde auch die Methode `QWidget::setStyle` entfernt. Ab Qt 2.0 ist die Darstellungsart global festgelegt und für alle Widgets einer Applikation gleich. Wenn der Compiler also Aufrufe von `QWidget::setStyle` nicht auflösen kann – das sollte in einem guten Programm aber eigentlich nicht vorkommen –, so ist im Einzelfall zu entscheiden, ob dieser Aufruf einfach gelöscht oder durch einen Aufruf von `QApplication::setStyle` ersetzt werden sollte.

Der Rückgabewert der Methode `QWidget::style` sowie `QApplication::style` ist jetzt nicht mehr der Aufzählungstyp `GUIStyle`, sondern die Klasse `QStyle`. Diese Klasse besitzt die Methode `QStyle::GUIStyle`, mit der man wiederum an die alten Werte herankommt. Ersetzen Sie zum Beispiel einfach

```
if (style() == MotifStyle)
```

durch:

```
if (style().GUIStyle() == MotifStyle)
```

Besser ist es jedoch auch hier wieder, etwas genauer hinzuschauen und zu prüfen, ob Sie nicht die Methoden der Klasse `QStyle` für die Darstellung nutzen können.

4.22.7 Zusammenfassung

Bei der Umstellung auf die Qt-Version 2.0 sind folgende Punkte zu beachten:

- Es ist nicht nötig, alle Programme neu zu kompilieren, da automatisch die korrekte Bibliothek benutzt wird.
- Vor der Neukompilierung sollte gewährleistet sein, dass sowohl die neue *Bibliothek* als auch die neuen *Header-Dateien* und der neue *moc* benutzt werden. Alle vom *moc* generierten Dateien sollten gelöscht werden.
- Die Klasse `QString` kann einfach durch `QCString` ersetzt werden. Oder man kann die neuen Möglichkeiten von `QString` (*Unicode-Unterstützung*) nutzen, indem man die Aufrufe der Methode `QString::data` ersetzt. Am besten entscheidet man von Fall zu Fall, welche der beiden Möglichkeiten die bessere ist.
- Viele Konstanten und Aufzählungstypen sind nun in der Klasse `Qt` definiert, von der unter anderem jetzt die Klasse `QObject` abgeleitet ist. Außerhalb der Methoden einer von `Qt` abgeleiteten Klasse muss die Klassenbezeichnung »Qt::« vorangestellt werden.
- Sollte das *Layout* eines alten Programms nach der Neukompilierung nicht mehr optimal sein, kann man das meist mit ein paar zusätzlichen *Stretch-Faktoren* korrigieren. Auch hier sollte man die Aufrufe von `setMinimumSize` und `setFixedSize` am besten dort entfernen, sie unnötig geworden sind.

- Die Methode `QWidget::setStyle` wurde entfernt, da die Darstellungsart jetzt global für die ganze Applikation gilt. Die Methode `QWidget::style` hat jetzt den Rückgabewert `QStyle`. Durch die Methode `QStyle::GUIStyle` kann man noch die alte Repräsentation durch den Aufzählungstyp `GUIStyle` ermitteln. Bei selbst definierten Widgets sollte möglichst auf die virtuellen Methoden der Klasse `QStyle` zurückgegriffen werden.

4.23 Programme von KDE 1.x auf KDE 2.x portieren

Wollen Sie ein altes Programm von KDE 1.0 oder KDE 1.1 auf KDE 2.0 kompilieren, so stoßen Sie an vielen Stellen auf Inkompatibilitäten. In diesem Kapitel wollen wir auf die häufigsten Probleme eingehen und Lösungsmöglichkeiten vorschlagen.

Da KDE 2.0 auf Qt 2.2 basiert, ist es zunächst notwendig, die Portierungen von Qt 1.x auf Qt 2.x zu portieren, wie es im vorangegangenen Kapitel 4.22, *Programme von Qt 1.x auf Qt 2.x portieren*, beschrieben wurde. Anschließend müssen noch die Änderungen in den KDE-Klassen berücksichtigt werden.

Bei einer Reihe von KDE-Klassen wurden die Methodennamen geändert, um ein einheitliches Namensschema zu erhalten. Insbesondere wurden die Methodennamen, die mit *get* begannen (z.B. `KApplication::getKApplication`) durch einen Namen ohne *get*, jedoch mit kleinem Anfangsbuchstaben ersetzt (also `KApplication::kApplication`). Wenn Sie beim Kompilieren Ihres Programms auf Fehlermeldungen der Art »undefined method call« oder Ähnliches stoßen, so schauen Sie in der Klassendokumentation nach, ob sich der entsprechende Methodenname geändert hat.

Viele Header-Dateien zu KDE-Klassen werden nun nicht mehr automatisch mit *kapp.h* eingebunden. Der Compiler wird daher häufig undefinierte Klassen melden. Oftmals meldet der Compiler aber auch nur eine fehlende Methode, wenn die Klasse vorab deklariert worden ist, aber die Header-Datei nicht eingebunden wurde. Fügen Sie die entsprechende *#include*-Anweisung ein, und versuchen Sie es erneut. Generell gilt die Regel, dass zu jeder KDE-Klasse, die im Programm benutzt wird, auch die Header-Datei eingebunden werden muss.

Eine weitere Änderung an der Schnittstelle der KDE-Klassen ist die Umstellung von *char** auf *QString*, wo es möglich war. Da es jedoch eine automatische Konvertierung gibt, werden alte Programme voraussichtlich ohne Änderung auch mit den neuen Schnittstellen funktionieren. Es empfiehlt sich jedoch trotzdem, das gesamte Programm noch einmal zu durchforsten und an jeder Stelle, an der ein Text als String übergeben wird, zu entscheiden, welche Klasse benutzt werden sollte.

4.23.1 Prüfen von Kommandozeilenparametern

In KDE 2.0 wird üblicherweise mit einer neuen Klasse, *KCmdLineArgs*, noch vor dem Anlegen des *KApplication*-Objekts eine Analyse der Kommandozeilenparameter vorgenommen. Der Konstruktor von *KApplication* benötigt daher die Parameter *argc* und *argv* nicht mehr. Aus Kompatibilitätsgründen ist der alte Konstruktor aber immer noch vorhanden, verlangt nun aber zwingend als dritten Parameter den Applikationsnamen. Diesen Konstruktor sollten Sie möglichst nicht mehr nutzen. Ändern Sie das Programm am besten so ab, wie es in Kapitel 3.3.2, *KDE-Applikationen*, beschrieben ist.

Ändern Sie also die folgende Zeile (in der Regel in der *main*-Funktion zu finden)

```
KApplication app (argc, argv);
```

in

```
KApplication app (argc, argv, "applikationsname");
```

ab, wobei statt *applikationsname* natürlich der Name Ihres Programms stehen muss. Oder ändern Sie am besten die Struktur folgendermaßen:

```
KCmdLineArgs::init (argc, argv, "applikationsname",  
                    "Beschreibung", "Version");  
KApplication app;
```

4.23.2 Konfigurationsdateien

Der Zugriff auf die Standardkonfigurationsdatei eines KDE-Programms geschieht nun nicht mehr über das *KApplication*-Objekt, sondern über die Klasse *KGlobal*. Ein Zugriff der folgenden Art

```
KConfig* config = kapp->getConfig();
```

muss daher durch

```
KConfig* config = KGlobal::config();
```

ersetzt werden. Beachten Sie, dass die Header-Datei *kconfig.h* nicht automatisch mit *kglobal.h* eingebunden wird. Ergänzen Sie also bei Bedarf eine entsprechende *#include*-Anweisung.

Eine weitere Änderung, die die Konfigurationsdateien betrifft, ist der Wegfall der Iteratoren, um alle Gruppen der Konfigurationsdatei oder alle Einträge einer Gruppe zu durchlaufen. Stattdessen können Sie mit den Methoden *groupList* eine Liste aller Gruppenbezeichnungen erhalten, und mit der Methode *entryMap* ein *QMap*-Objekt bekommen, das alle Einträge einer Gruppe enthält. Falls im alten Programm also Iteratoren verwendet wurden – was eher selten der Fall ist –, müssen Sie die Programmstruktur hier entsprechend ändern.

4.23.3 Internationalisierung und Lokalisierung

Ältere Programme benutzen oft statt des Makros *i18n* den Aufruf *klocale->translate*. Dieser Aufruf ist nun nicht mehr möglich. Ändern Sie also jedes Vorkommen von

```
klocale->translate (".....");
```

in:

```
i18n (".....");
```

Weiterhin können Sie das *KLocale*-Objekt nun nicht mehr über *KApplication* erhalten, sondern nur noch über die Klasse *KGlobal*. Ersetzen Sie also weiterhin alle Zugriffe der Form

```
kapp->getLocale()
```

oder der Form

```
klocale
```

durch:

```
KGlobal::locale()
```

i18n ist nun kein Makro mehr, sondern eine Funktion. Auf die Form des Aufrufs hat das allerdings keine Auswirkung, so dass sich daraus keine Änderungen ergeben.

Weitere Informationen finden Sie in Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*.

4.23.4 Icons

Das Einlesen von Icons aus dem entsprechenden KDE-Verzeichnis hat sich geändert. Die Dateiendung wird nun automatisch angefügt, so dass sie nicht mehr angegeben werden muss. Auch die beiden Makros *ICON* und *Icon* wurden entfernt. Benutzen Sie stattdessen für Icons der Werkzeugleiste das Makro *BarIcon*, das die selbe Aufgabe erfüllt. Wenn Sie Ihr Programm auf *KAction*-Objekte umstellen, benötigen Sie dieses Makro gar nicht mehr (siehe Kapitel 3.5.3, *Definition von Aktionen für die Menü- und Werkzeugleisten*).

Einen Aufruf von

```
ICON ("filenew.xpm")
```

ersetzen Sie also durch

```
BarIcon ("filenew")
```

4.23.5 Neue Hauptfensterklasse *KMainWindow*

KDE 2.0 enthält eine neue Widget-Klasse zur Implementierung des Hauptfensters: *KMainWindow*. Diese Klasse ist nun von *QMainWindow* abgeleitet und ersetzt die alten Klassen *KTopWidget* und *KMainWindow*. *KMainWindow* existiert nur noch aus Kompatibilitätsgründen. Sie sollten daher Ihr Programm an *KMainWindow* anpassen. Als Konsequenz der Umstellung ergeben sich einige Änderungen der Bezeichner. Die wichtigsten sind in Tabelle 4.27 aufgeführt.

in <i>KMainWindow</i>	in <i>KMainWindow</i>
<code>#include <ktmainwindow.h></code>	<code>#include <kmainwindow.h></code>
<code>new KMainWindow ()</code>	<code>new KMainWindow (0)</code>
<code>setView ()</code>	<code>setCentralWidget ()</code>
<code>enableStatusBar (true)</code>	<code>statusBar()->show()</code>
<code>enableStatusBar (false)</code>	<code>statusBar()->hide()</code>
<code>setMenu ()</code>	entfällt
<code>addToolBar ()</code>	entfällt

Tabelle 4-27 Änderungen von *KMainWindow* nach *KMainWindow*

Einige kleine Änderungen ergeben sich auch beim Füllen der Menüleiste. Das automatisch erzeugte Popup-Menü zum Eintrag *HELP* wird nun nicht mehr von *KApplication*, sondern von *KMainWindow* erzeugt. Günstiger ist es aber, wenn Sie das Programm komplett umstellen auf das *KAction*-Konzept und der Anordnung der Befehle in den Menü- und Werkzeugleisten durch eine XML-Datei.

Weitere Informationen finden Sie auch in Kapitel 3.5, *Das Hauptfenster*.

4.23.6 Debugging-Umstellung von *kDebug* auf *kdDebug*

Das alte System zum Ausgeben von Debug-Meldungen mit Hilfe des Makros *KDEBUG* bzw. der Funktion *kDebug* wurde umgestellt auf Funktionen namens *kdDebug*, *kdWarning*, *kdError* und *kdFatal*. Falls das alte Programm *kDebug* benutzte, ändern Sie die entsprechenden Aufrufe um. Ein automatisches Skript dazu mit dem Namen *kDebug2kdDebug.sh* finden Sie im Paket *kdesdk*. Die meisten Programme benutzten aber *qDebug*. Dort ist keine Änderung nötig. Sie sollten aber evtl. in Erwägung ziehen, auf das KDE-Konzept umzusteigen.

4.23.7 Drag&Drop

Das Drag&Drop-System von KDE 1.1 existiert nicht mehr. Stattdessen wird nun das Drag&Drop von Qt benutzt. Sollte Ihr Programm noch das alte System nutzen, müssen Sie wahrscheinlich eine etwas aufwendigere Umstrukturierung vornehmen. Genaue Informationen erhalten Sie in Kapitel 4.15.2, *Drag&Drop*.

4.23.8 Netzwerkzugriffe mit *kfm*

Die Klasse *KFM* zum Netzwerkzugriff auf Dateien wurde aufgespalten und auf verschiedene andere Klassen verteilt.

Beachten Sie auch, dass Sie nun statt der Bibliothek *libkfm* die Bibliothek *libkio* sowie eventuell *libkonq* einbinden müssen. Passen Sie das Makefile dementsprechend an.

Der häufigste Einsatz dieser Klasse war der Download von Dateien anhand einer URL. Diese Funktionalität ist nun in der Klasse *KIO::NetAccess* enthalten. Ersetzen Sie also

```
QString tempfilename = KFM::download (url);  
...  
KFM::removeTempFile (tempfilename);
```

durch:

```
QString tempfilename = KIO::NetAccess::download (url);  
...  
KIO::NetAccess::removeTempFile (tempfilename);
```

Das Kopieren und Verschieben von Dateien und Verzeichnissen ist nun ebenfalls in *KIO* implementiert. Dort gibt es jetzt die Möglichkeit, die Operation synchron (also blockierend) oder asynchron (also im Hintergrund) ausführen zu lassen. Eine genaue Beschreibung finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Zugriff mit KIO*.

Das Ausführen von anderen Programmen (*KFM::exec*) sowie das Öffnen von Dateien mit dem zugeordneten Programm (*KFM::openURL*) sind nun durch die Klasse *KRun* implementiert. Ersetzen Sie einfach einen der folgenden Aufrufe

```
KFM::openURL (url);
```

oder

```
KFM::exec (url);
```

durch folgende Zeile:

```
new KRun (url);
```

4.23.9 Weggefallene Klassen

Eine ganze Reihe von KDE-Klassen wurden aus den Bibliotheken entfernt, weil ihre Funktionen entweder in neuen Qt-Klassen implementiert sind oder sie durch andere KDE-Klassen ersetzt wurden. Tabelle 4.28 enthält eine Liste der wichtigsten Klassen:

alte Klasse	kann ersetzt werden durch
<i>KClipboard</i>	<i>QClipboard</i>
<i>KPanner</i> , <i>KNewPanner</i>	<i>QSplitter</i>
<i>KMessageBox</i>	<i>QMessageBox</i> oder <i>KMessageBox</i>
<i>KCombo</i>	<i>KComboBox</i> (Umbenennung)
<i>KQuickHelp</i>	<i>QWhatsThis</i>
<i>KRadioGroup</i>	<i>KToggleAction::setExclusiveGroup</i> oder <i>QActionGroup</i>
<i>KString</i>	<i>QString</i>
<i>KButton</i>	<i>QToolButton</i>
<i>KLed</i> , <i>KLedLamp</i>	<i>KLed</i> (geänderte Schnittstelle)
<i>KHTMLView</i>	<i>KHTMLWidget</i>
<i>KSpinBox</i> , <i>KNumericSpinBox</i>	<i>QSpinBox</i>
<i>KTreeList</i> , <i>KTabListBox</i>	<i>QListView</i> oder <i>KListView</i>

Tabelle 4-28 Alte KDE-Klassen und Alternativklassen

Die neuen Klassen haben oftmals eine andere Schnittstelle, so dass es in den meisten Fällen nicht ausreicht, nur die Klassennamen umzubenennen. Konsultieren Sie die Klassendokumentation zu den einzelnen Klassen.

5 Hilfsmittel für die Programmerstellung

Es gibt inzwischen eine ganze Reihe von Hilfsmitteln, die das Erstellen einer eigenen KDE-Applikation sehr vereinfachen. Mit ihnen kann die Entwicklungszeit zum Teil drastisch reduziert werden. Alle hier beschriebenen Programme sind auch auf der CD enthalten, die diesem Buch beiliegt. Für die jeweils aktuellsten Versionen besuchen Sie bitte die Webseiten der jeweiligen Tools.

In Kapitel 5.1, *tmake*, und Kapitel 5.2, *automake* und *autoconf*, werden zwei Möglichkeiten vorgestellt, automatisch Makefiles generieren zu lassen. Ein gutes Makefile erfordert – wenn es von Hand geschrieben wird – viel Arbeit, kann aber später beim Entwickeln des Programms viel Zeit sparen, wenn beim Kompilieren die Abhängigkeiten zwischen den Dateien richtig bestimmt waren. Das Konzept von *automake* und *configure* geht aber noch über die Entwicklungsarbeit hinaus. Mit ihnen erzeugen Sie Source-Pakete, die auf vielen Systemen kompiliert werden können, ohne dass Anpassungen nötig wären.

Kapitel 5.3, *kapptemplate* und *kappgen*, beschreibt zwei Tools aus dem Paket *kdesdk*, die es erlauben, sehr schnell ein Grundgerüst für ein Programm zu erstellen. Sie benutzen dabei *automake* und *autoconf*, so dass Sie sehr einfach professionelle KDE-Programme erzeugen können.

In Kapitel 5.4, *Qt Designer*, beschreiben wir kurz die Möglichkeiten des Programms *Qt Designer*. Mit ihm kann man Bildschirmdialoge sehr einfach aus den vorhandenen GUI-Elementen zusammenstellen. Das Programm erzeugt daraus den nötigen Quellcode.

Kapitel 5.5, *KDevelop*, stellt kurz die integrierte Entwicklungsumgebung *KDevelop* vor. Sie versucht, alle Tools zur Programmentwicklung miteinander zu verbinden, so dass der Programmierer sehr einfach an verschiedenen Problemstellungen arbeiten kann, ohne das Programm wechseln zu müssen.

5.1 tmake

Da es sehr lästig ist, den Compiler von Hand aufzurufen – insbesondere da KDE-Programme viele Compiler-Optionen benötigen –, kann ein Makefile eine große Hilfe sein. Da Makefiles aber ein sehr mächtiges und für den Anfänger schwer zu handhabendes Konzept darstellen, hat die Firma Trolltech das Programm *tmake* entwickelt, mit dem Sie, ausgehend von einer sehr einfachen Projekt-Datei (meist reichen drei bis fünf einfache Zeilen aus), automatisch ein Makefile generieren können. Dabei werden auch die systemspezifischen Besonderheiten berücksichtigt, wie zum Beispiel der Dateiname des Compilers, die Pfade zu den Header-Dateien usw. Es werden insbesondere Aufrufe für den *moc* in das Makefile mit aufgenommen, so dass automatisch für alle Header-Dateien, die eine neue, von

QObject abgeleitete Klasse definieren, die zugehörige *moc*-Datei erzeugt und in das ausführbare Programm eingebunden wird. *tmake* ist damit ein ideales Werkzeug, wenn Sie erste Versuche in Sachen KDE-Programmierung unternehmen. Informationen zu *tmake* und die Möglichkeit zum Download finden Sie unter <http://www.trolltech.com/freebies/tmake.html>. *tmake* läuft dabei sowohl unter den gängigsten Unix-Varianten als auch unter Microsoft Windows.

Die Benutzung von *tmake* ist sehr einfach: Sie schreiben eine Projekt-Datei, in der Sie die Header- und Source-Dateien Ihres Programms sowie den Programmnamen der ausführbaren Datei angeben. Als Beispiel benutzen wir hier eine Applikation, die zwei Klassen selbst definiert: *MyMainWindow* und *MyDialog*. Diese Klassen sind in jeweils einer Source- und einer Header-Datei mit den Dateinamen *mymainwindow.cpp*, *mymainwindow.h*, *mydialog.cpp* und *mydialog.h* definiert. Die *main*-Funktion steht in der Datei *main.cpp*. Die Projekt-Datei namens *myapp.pro* sieht dann wie folgt aus:

```
HEADERS = mymainwindow.h mydialog.h
SOURCES = mymainwindow.cpp mydialog.cpp main.cpp
TARGET  = myapp
```

Ein Aufruf von *tmake* mit der Zeile

```
% tmake myapp.pro -o Makefile
```

erzeugt nun aus der Projekt-Datei *myapp.pro* die Datei *Makefile*. Um nun das Programm zu kompilieren, müssen Sie nur *make* aufrufen:

```
% make
```

Alle Source-Dateien werden nun kompiliert und zu einer ausführbaren Datei *myapp* zusammengelinkt. Die Abhängigkeiten der Dateien voneinander werden automatisch im *Makefile* gespeichert. Wenn Sie also eine der Source-Dateien ändern und anschließend wieder *make* aufrufen, werden nur die betroffenen Dateien neu kompiliert. Das spart insbesondere bei großen Projekten mit vielen Source-Dateien sehr viel Zeit. Ein weiterer großer Vorteil: Alle notwendigen *moc*-Dateien werden automatisch erzeugt und ebenfalls kompiliert und eingebunden.

Wenn Sie während der Entwicklung Ihres Programms neue Dateien zu Ihrem Projekt hinzufügen wollen, so brauchen Sie nur die Projekt-Datei (im Beispiel *myapp.pro*) zu ändern. Bei einem erneuten Aufruf von *make* wird automatisch zunächst aus der Projekt-Datei das aktuelle *Makefile* erzeugt.

Wollen Sie KDE-Programme mit *tmake* erstellen, so müssen Sie zusätzlich den Suchpfad für die KDE-Header-Dateien und die KDE-Bibliotheken angeben. Fügen Sie dazu die folgenden beiden Zeilen der Projekt-Datei hinzu:

```
INCLUDEPATH = $(KDEDIR)/include
LIBS = -L$(KDEDIR)/lib -lkdeui -lkdecore
```


Voraussetzung dafür ist natürlich, dass die Umgebungsvariable *\$KDEDIR* korrekt gesetzt ist. Falls Sie noch weitere KDE-Bibliotheken einbinden müssen (z.B. *kio*, *kfile* oder *khtml*), so fügen Sie diese ebenfalls in *LIBS* ein.

Die Fähigkeiten von *tmake* sind hiermit noch nicht erschöpft. Wir wollen hier stichpunktartig noch einige andere Möglichkeiten aufzählen. Ausführliche Informationen finden Sie in der Online-Dokumentation, die dem Programm *tmake* selbst beiliegt.

- In einer zusätzlichen Zeile *CONFIG = ...* können Sie zusätzliche Optionen festlegen, wie kompiliert werden soll. Geben Sie dazu die gewünschten Optionen durch Leerzeichen getrennt hintereinander an. So können Sie mit *warn_on* Compiler-Warnungen anschalten oder mit *warn_off* abschalten. Mit der Option *release* wird die ausführbare Datei optimiert, mit *debug* werden Debug-Informationen hinzugebunden.
- Durch einen Aufruf von *make clean* können Sie alle automatisch generierten Dateien löschen lassen (alle Objekt-Dateien, *moc*-Dateien und das ausführbare Programm). Auf diese Weise können Sie dafür sorgen, dass alle Dateien garantiert neu erzeugt werden, z.B. nachdem Sie die Zeile *CONFIG = ...* geändert haben.
- Durch einen Aufruf von *make dist* können Sie alle Quellcode-Dateien zusammen mit der Projekt-Datei automatisch in eine Archiv-Datei packen lassen. Das ist praktisch, um den aktuellen Zustand des Projekts zu archivieren oder um das Projekt als Quellcode zu verschicken. Wollen Sie zusätzliche Dateien in die Archiv-Datei aufnehmen (Icons, *README* o. Ä.), so fügen Sie in der Projekt-Datei eine Zeile *DISTFILES = ...* hinzu.
- Ein *make install*, wie man es von anderen Makefiles kennt, gibt es in von *tmake* erzeugten Makefiles nicht. Sie können das Ziel *install* von Hand in das Makefile einfügen. Beachten Sie aber, dass *tmake* bei einer Änderung der Projekt-Datei das Makefile automatisch überschreibt.
- Wollen Sie statt einer ausführbaren Datei eine Bibliothek erzeugen, so wählen Sie eine andere Vorlage, indem Sie in der Projekt-Datei als erste Zeile *TEMPLATE = lib* einfügen. Die Versionsnummer der Bibliothek (wichtig für dynamische Bibliotheken) können Sie beispielsweise mit einer Zeile *VERSION = 0.3.0* festlegen. Standardmäßig wird eine dynamische Bibliothek (**.so*) erzeugt. Eine statische Bibliothek (**.a*) erhalten Sie durch Angabe von *CONFIG = staticlib*.
- Haben Sie mehrere Projekte auf mehrere Unterverzeichnisse verteilt, so können Sie mit einer einzigen Projekt-Datei im Vaterverzeichnis ein Makefile erzeugen lassen, das *make* in allen Unterverzeichnissen aufruft. Geben Sie dazu in der Projekt-Datei im Vaterverzeichnis als erste Zeile *TEMPLATE =*

subdirs an, und geben Sie in der zweiten Zeile *SUBDIRS* = ... an, in der Sie alle Unterverzeichnisse auflisten. Nachdem Sie im Väterverzeichnis mit *tmake* das Makefile erzeugt haben, können Sie *make* oder *make clean* aufrufen. Dieser Aufruf wird in allen angegebenen Unterverzeichnissen ausgeführt. *make dist* funktioniert leider nicht.

- Wenn Sie das Programm *Qt Designer* der Firma Trolltech benutzen, um Fenster und Dialoge zu entwerfen, so können Sie den Aufruf des *uic* (User Interface Compiler) ebenfalls automatisieren, indem Sie in der Projekt-Datei die Zeile *INTERFACES* = ... einfügen (siehe Kapitel 5.4, *Qt Designer*).
- Mit ein wenig Wissen über die Programmiersprache Perl können Sie auch eigene Templates entwerfen. Informationen finden Sie in der Online-Dokumentation von *tmake*.

Die Firma Trolltech hat außerdem noch das Tool *progen* herausgebracht, das sogar die Erstellung einer Projekt-Datei automatisch vornimmt. Wenn also alle Dateien (und sonst keine anderen Source- oder Header-Dateien), die zu einem Projekt gehören, in einem Verzeichnis liegen, erzeugt die folgende Zeile automatisch die Projekt-Datei *myapp.pro*, die wir oben noch von Hand eingegeben hatten:

```
% progen -o myapp.pro
```

Hinter *-o* gibt man den Namen der Projekt-Datei an, die von *progen* erstellt werden soll. In dieser Projekt-Datei werden automatisch alle Source- und Header-Dateien im aktuellen Verzeichnis aufgelistet. Anschließend kann man *tmake* für *myapp.pro* aufrufen, das das Makefile erzeugt. Durch den Aufruf von *make* schließlich werden alle Dateien kompiliert. Es wird eine ausführbare Datei erzeugt, die den gleichen Namen wie die Projekt-Datei (allerdings ohne die Endung *.pro*) hat.

Die Programme *tmake* und *progen* können Sie von der Homepage der Firma Trolltech herunterladen. Sie finden sie ebenfalls auf der CD-ROM, die diesem Buch beiliegt. Sie können kostenlos benutzt und weitergegeben werden. Die beiden Programme benötigen die Skriptsprache Perl. Perl ist aber in den meisten neueren Linux-Distributionen vorhanden. Achten Sie also darauf, dass Sie das Perl-Paket installiert haben.

5.2 automake und autoconf

Die KDE-Programme aus den KDE-Paketen benutzen ein anderes Konzept, um Makefiles erstellen zu lassen. Sie verwenden die Tools *automake* und *autoconf*. Diese Tools sind sehr viel mächtiger, aber auch sehr viel schwieriger zu benutzen als *tmake*. Für die Entwicklung kleiner Testprogramme sind sie sicherlich viel zu umfangreich. Sie werden in der Regel für große Projekte genutzt.

Mit dem Konzept von *automake* und *autoconf* kann man auch mehrere Applikationen in einem Paket zusammenstellen. Die Verzeichnisstruktur eines Projekts sieht dann folgendermaßen aus: Im Projektverzeichnis selbst befinden sich nur Hilfsdateien, die ausführbare Skriptdatei *configure*, ein Unterverzeichnis *admin* mit benötigten Shell-Skripten, ein Unterverzeichnis pro Applikation sowie ein Unterverzeichnis *po*, das die Übersetzungsdateien in andere Sprachen enthält. In jedem Applikationsverzeichnis befinden sich in der Regel weitere Unterverzeichnisse, zum Beispiel mit Icons für die Werkzeugleiste, oder verschiedene Verzeichnisse mit dem Online-Handbuch in verschiedenen Sprachen. Abbildung 5.1 verdeutlicht noch einmal die Verzeichnisstruktur. Diese Struktur ist nicht vorgeschrieben, hat sich aber für KDE-Programme etabliert und wird so von Hilfsprogrammen wie *kapptemplate* (siehe Kapitel 5.3, *kapptemplate* und *kappgen*) oder *KDevelop* (siehe Kapitel 5.5, *KDevelop*) erzeugt.

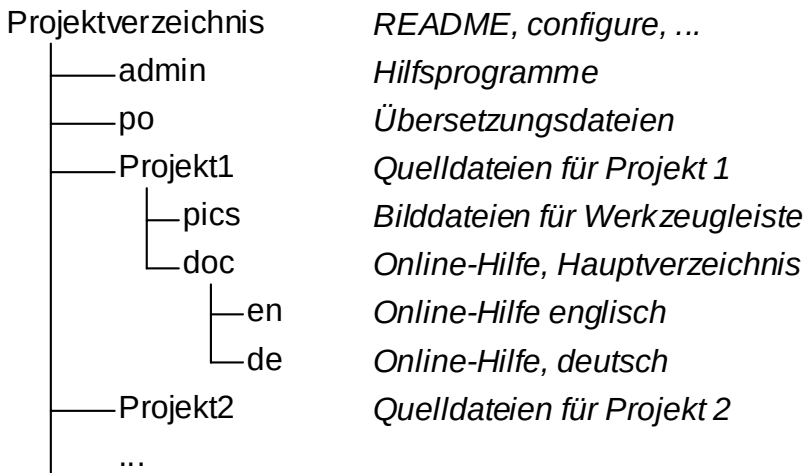


Abbildung 5-1 Verzeichnisstruktur eines *automake/autoconf*-Projekts

Diese gesamte Verzeichnisstruktur kann zu einem Sourcen-Paket zusammengefasst werden, das man anderen Anwendern schicken kann. Zum Kompilieren und Installieren dieses Sourcen-Pakets benötigt man *automake* und *autoconf* nicht.

Der Anwender entpackt nun dieses Paket und startet zunächst die Skriptdatei *configure*. (Diese Skriptdatei läuft auf allen gängigen Shells, zum Beispiel *bash* oder *tcsh*.) Das Skript ermittelt einige Systemeinstellungen, insbesondere die Verzeichnisse, den zu verwendenden Compiler, die vorhandenen Bibliotheken sowie einige Besonderheiten des Compilers. Anschließend generiert es in jedem Unterverzeichnis aus der Datei *Makefile.in* die Datei *Makefile*. Dem *configure*-

Skript kann man eine Reihe von Optionen mitgeben, um den Kompilervorgang zu beeinflussen, z.B. ob Debug-Informationen eingebaut werden sollen oder nicht, oder in welchen Verzeichnissen das Programm installiert werden soll. Eine Liste aller möglichen Optionen erhält der Anwender, wenn er *configure --help* aufruft.

Anschließend kann der Anwender im Projektverzeichnis *make* aufrufen. Alle Source-Dateien in den Applikationen werden nun kompiliert und gelinkt. Zum Abschluss ruft der Anwender *make install* auf, das alle Dateien an die richtige Stelle im Verzeichnisbaum installiert. Für diesen letzten Schritt braucht der Anwender allerdings oft Systemrechte, wenn die Dateien in Verzeichnissen gespeichert werden sollen, für die ein normaler Anwender keine Schreibberechtigung hat.

Die erzeugten Makefiles sind sehr umfangreich und helfen dem Entwickler auch bei anderen Aufgaben. Eine Liste der wichtigsten *make*-Aufrufe finden Sie in Tabelle 5.1. So übernehmen sie zum Beispiel auch die Generierung von Übersetzungsdateien aus dem Sourcecode (siehe Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*). Rufen Sie dazu *make package-messages* auf. Anschließend wechseln Sie in das Verzeichnis *po/*, das die Übersetzungsdateien enthält, und fügen die neuen Übersetzungen in die Dateien ein. (Anmerkung: In der zur Zeit aktuellen Version von *kapptemplate* scheint *make package-messages* nicht korrekt zu funktionieren.)

Aufruf	Wirkung
<i>make</i>	Programme kompilieren und linken
<i>make install</i>	Programme und Daten in die richtigen Verzeichnisse kopieren
<i>make clean</i>	durch <i>make</i> generierte Dateien wieder löschen
<i>make maintainer-clean</i>	alle automatisch generierten Dateien löschen (auch <i>configure</i> und Makefiles)
<i>make dist</i>	ein Sourcen-Paket für Anwender erstellen
<i>make package-messages</i>	alle zu übersetzenden Texte mit <i>xgettext</i> herausfiltern und in das Verzeichnis <i>po</i> bringen

Tabelle 5-1 Nützliche Aufrufe von *make*

In jedem Verzeichnis muss eine Datei *Makefile.in* existieren. In der Regel lässt man diese Datei aber ebenfalls aus jeweils einer (kürzeren und übersichtlicheren) Datei *Makefile.am* automatisch erzeugen.

Um selbst Programme mit diesem Konzept auszustatten, empfiehlt es sich, das Grundgerüst bereits fertig zu übernehmen oder erzeugen zu lassen. Dazu können Sie entweder ein fertiges KDE-Programm benutzen (zum Beispiel *kexample* aus dem Paket *kdesdk*) oder das Grundgerüst von den Programmen *kapptemplate* oder

kappgen erzeugen lassen (siehe Kapitel 5.3, *kapptemplate* und *kappgen*), die ebenfalls im Paket *kdesdk* enthalten sind. (Falls dieses Paket nicht in Ihrer Linux-Distribution enthalten ist, können Sie es vom KDE-FTP-Server unter <ftp://ftp.kde.org/> herunterladen. Meist befindet sich dieses Paket aber nicht in der Standard-KDE-Distribution, sondern nur in den Snapshots im Verzeichnis */pub/kde/unstable/CVS/snapshots/current/.*)

Noch günstiger ist es in der Regel, die integrierte Entwicklungsumgebung *KDevelop* zu nutzen (siehe Kapitel 5.5). Sie verwaltet die Struktur sowie die diversen Makefile-Einträge selbstständig, ohne dass Sie die Dateien von Hand modifizieren müssten. Gerade für Anfänger empfiehlt es sich, diesen Weg zu wählen, da ein Erlernen von *automake* und *autoconf* viel Zeit in Anspruch nimmt.

Wenn Sie *KDevelop* nicht verwenden wollen oder können, so müssen Sie die Dateien mit dem Namen *Makefile.am*, die in jedem Unterverzeichnis vorkommen, von Hand mit einem beliebigen Editor modifizieren. Ein anschließender Aufruf von *make* erzeugt zunächst alle notwendigen Dateien automatisch neu, bevor das Kompilieren gestartet wird. Die häufigsten Modifikationen werden wir im Folgenden kurz beschreiben. Grundlage ist dabei die Struktur, die von *kapptemplate* erzeugt wird. Andere Tools benutzen unter Umständen eine etwas andere Struktur, so dass die Modifikationen mitunter anders aussehen können.

- Wenn Sie das Gefühl haben, dass die verschiedenen Makefiles nicht auf dem aktuellsten Stand sind, diese aber bei einem *make* nicht automatisch neu erzeugt werden, so lassen Sie am besten alle automatisch erzeugten Dateien erneut generieren. (Bei einigen – insbesondere älteren – Projekten heißt die Datei *Makefile.dist* statt *Makefile.cvs*.) Rufen Sie dazu folgende Befehle in dieser Reihenfolge auf:

```
% make -f Makefile.cvs
% rm config.cache
% ./configure
% make
```

- Wenn Sie ein eigenes Unterverzeichnis hinzufügen wollen, fügen Sie in dieses Verzeichnis ebenfalls eine Datei *Makefile.am* ein, die Sie zum Beispiel aus einem anderen Verzeichnis übernehmen. Passen Sie diese Datei an die jeweiligen Bedürfnisse an. Außerdem müssen Sie dieses Unterverzeichnis in der Datei *Makefile.am* im übergeordneten Verzeichnis unter dem Punkt *SUBDIRS* = einfügen. Anschließend sollten Sie, wie oben beschrieben, zur Sicherheit alle automatisch generierten Dateien neu erzeugen lassen.
- Wollen Sie eine neue Quelldatei zu Ihrem Projekt hinzufügen, so kopieren Sie die Datei in das Quellverzeichnis. Eine Code-Datei fügen Sie anschließend in der Datei *Makefile.am* dem Punkt *<programmname>_SOURCES =* hinzu. *programmname* ist dabei der Name der ausführbaren Datei, die erzeugt werden soll. Eine Header-Datei fügen Sie dagegen zum Punkt *noinst_HEADERS =* hinzu.

- Wenn Sie weitere Dateien in Ihr Projekt einbinden wollen (Icons, Sound-Dateien, Desktop-Dateien o. Ä.), die mit *make install* in ein Verzeichnis kopiert werden sollen, so legen Sie die Dateien in einem der Verzeichnisse ab (z.B. im Verzeichnis der Quellcodes). Benutzen Sie im Folgenden einen treffenden Namen für diese Dateien (z.B. *pics* oder *desktopfiles*). Um beispielsweise die Dateien *img1.xpm* und *img2.xpm* im Verzeichnis *\$KDEDIR/share/apps/myapp/pictures* installieren zu lassen, fügen Sie folgende zwei Zeilen in die Datei *Makefile.am* ein:

```
pics_DATA = img1.xpm img2.xpm
picmdir = $(kde_datadir)/myapp/pictures
```

- Wollen Sie Dateien in verschiedene Verzeichnisse installieren lassen, so können Sie auch mehrere dieser zweizeiligen Blöcke in die Datei *Makefile.am* einfügen. Benutzen Sie aber unterschiedliche Bezeichnungen. Besser ist es meist aber, wenn Sie weitere Unterverzeichnisse anlegen, in denen jeweils die Dateien stehen, die in ein gemeinsames Verzeichnis kopiert werden sollen. Jedes der Unterverzeichnisse benötigt dann eine eigene Datei *Makefile.am*, in der dieser zweizeilige Block steht.
- Wie bereits oben erwähnt wurde, sollten Sie für die Installation von Dateien bei einem KDE-Projekt möglichst keine absoluten Pfade angeben. Verwenden Sie stattdessen die vordefinierten Variablen, die in Tabelle 5.2 aufgelistet sind. Dateien, die Sie keinem der Verzeichnisse zuordnen würden und die nur für Ihr eigenes Programm relevant sind, speichern Sie am besten im Verzeichnis *\$(kde_datadir)/myappname*, wobei *myappname* der Name Ihres Programms ist (also in der Regel der Dateiname der ausführbaren Datei).

Wenn Sie eigene Tests in der Skriptdatei *configure* durchführen wollen – zum Beispiel wenn Sie eine Bibliothek benutzen wollen, die auf verschiedenen Unix-Systemen unterschiedlich heißt –, so fügen Sie ein kleines Testprogramm zur Datei *acinclude.m4* im Projektverzeichnis hinzu. Rufen Sie dieses Programm in *configure.in* auf. *configure* startet dann den Compiler mit diesem Programm. Sie können nun in Abhängigkeit davon, ob dabei ein Fehler aufgetreten ist oder nicht, verschiedene Variablen setzen lassen oder *#define*-Direktiven in die generierte Datei *config.h* eintragen. Die Datei *config.h* können Sie beispielsweise in Ihre eigene Datei einbinden, um je nach Plattform verschiedene Teile Ihres Codes kompilieren zu lassen. Dazu müssen Sie sich allerdings sehr genau mit der Materie von *automake* und *autoconf* auseinander setzen. Es würde den Rahmen dieses Buchs sprengen, hier genauer darauf einzugehen.

Variable	üblicher Wert	Bedeutung
<code>\$(kde_appsdir)</code>	<code>\$KDEDIR/share/applnk</code>	desktop-Dateien für Applikationen
<code>\$(kde_bindir)</code>	<code>\$KDEDIR/bin</code>	ausführbare Programme
<code>\$(kde_confdir)</code>	<code>\$KDEDIR/share/config</code>	Konfigurationsdateien
<code>\$(kde_datadir)</code>	<code>\$KDEDIR/share/apps</code>	Datenverzeichnisse für jede Applikation
<code>\$(kde_htmldir)</code>	<code>\$KDEDIR/share/doc/HTML</code>	Hilfdateien
<code>\$(kde_icondir)</code>	<code>\$KDEDIR/share/icons</code>	Icons
<code>\$(kde_includes)</code>	<code>\$KDEDIR/include</code>	KDE-Include-Dateien
<code>\$(kde_libraries)</code>	<code>\$KDEDIR/lib</code>	KDE-Bibliotheken
<code>\$(kde_locale)</code>	<code>\$KDEDIR/share/locale</code>	Übersetzungsdateien
<code>\$(kde_mimedir)</code>	<code>\$KDEDIR/share/mimelnk</code>	desktop-Dateien für Mimetypes
<code>\$(kde_servicesdir)</code>	<code>\$KDEDIR/share/services</code>	desktop-Dateien für Dienste und Protokolle
<code>\$(kde_servietypesdir)</code>	<code>\$KDEDIR/share/servicetypes</code>	desktop-Dateien für Komponententypen
<code>\$(kde_sounddir)</code>	<code>\$KDEDIR/share/sounds</code>	Sound-Dateien
<code>\$(kde_templatesdir)</code>	<code>\$KDEDIR/share/templates</code>	desktop-Dateien für Vorlagen
<code>\$(kde_wallpaperdir)</code>	<code>\$KDEDIR/share/wallpapers</code>	Hintergrundbilder

Tabelle 5-2 Variablen für KDE-Verzeichnisse

5.3 kapptemplate und kappgen

Im Paket *kdesdk* finden Sie die beiden Tools *kapptemplate* und *kappgen*. Diese beiden Tools erstellen automatisch ein Verzeichnisgerüst mit allen nötigen Verzeichnissen unter Verwendung des *automake*-Konzepts. *kapptemplate* ist dabei ein Kommandozeilenprogramm, während *kappgen* eine grafische Benutzeroberfläche hat (siehe Abbildung 5.2). Beide Tools haben den gleichen Funktionsumfang. Es wird automatisch ein Hauptprogramm erstellt, in das Sie nur noch den Code einfügen müssen. *kapptemplate* und *kappgen* sind damit optimal geeignet, um eine Basis für die Entwicklung eines größeren Projekts zu erzeugen. Mit diesen Tools ist es möglich, *automake* und *autoconf* zu nutzen, ohne viel von diesen Programmen zu verstehen. Erst wenn Sie tiefer in die Möglichkeiten eintauchen wollen, müssen Sie sich mit der Materie auseinander setzen.

Ein Dialog mit *kapptemplate* kann zum Beispiel so aussehen:

```
KAppTemplate v0.3.2 (C) 1999 Kurt Granroth <granroth@kde.org>
```

```
What is the application's proper name [default: KMyApp]
: KTest
```

```
What is the application's version [default: 0.1]
```

```
:  
Going with default version '0.1'  
  
Where should I create this [default: /usr/src/ktest-0.1]  
: /home/lehner/helphelp/ktest-0.1  
  
What is your name [default: Your Name]  
: Burkhard Lehner  
  
What is your email address [default: lehner@]  
: b_lehner@informatik.uni-kl.de  
  
Here is what I have:  
The app: KTest v0.1  
Installed in: /home/lehner/helphelp/ktest-0.1  
Author: Burkhard Lehner <b_lehner@informatik.uni-kl.de>  
  
Is this correct (Y/n)?  
: Y  
  
OK, Here we go!!  
  
Creating directory /home/lehner/helphelp/ktest-0.1...  
Creating directory /home/lehner/helphelp/ktest-0.1/ktest...  
Creating directory /home/lehner/helphelp/ktest-0.1/ktest/doc...  
Creating directory /home/lehner/helphelp/ktest-0.1/ktest/doc/en...
```

Es schließen sich noch viele Angaben über kopierte oder erzeugte Programme an. Anschließend wird automatisch *automake* ausgeführt und das »Programm« ein erstes Mal testweise kompiliert.

kapptemplate legt eine umfangreiche Verzeichnisstruktur an. Die für *automake* und *autoconf* nötigen Dateien werden in das Projektverzeichnis kopiert sowie zusätzlich die Dateien *README*, *AUTHORS*, *NEWS* und *COPYING*. Die Datei *README* muss natürlich noch mit den nötigen Informationen gefüllt werden. *COPYING* enthält eine Kopie der GPL (*General Public License*, siehe Anhang B.3). Wenn Sie Ihr Programm unter einer anderen Lizenz verbreiten wollen, müssen Sie diese Datei natürlich durch ein anderes Lizenzdokument ersetzen. Im Unterverzeichnis der Applikation werden ebenfalls die Dateien *Makefile.am* und *Makefile.in* erzeugt sowie das Hauptprogramm *main.cpp* und einige Klassendateien, die eine eigene Hauptfensterklasse definieren. Zusätzlich wird noch das (zunächst leere) Unterverzeichnis *pics/* angelegt. Dort können Sie eigene Icons ablegen, die installiert werden sollen. Diese Bilder müssen Sie dann zur Datei *Makefile.am* in diesem Verzeichnis hinzufügen. Außerdem legt *kapptemplate* das Verzeichnis *doc/* mit dem Unterverzeichnis *en/* an, in das es eine erste, rudimentäre SGML-Datei schreibt. Hier können Sie das Online-Handbuch Ihrer Applikation ergänzen.

Damit Sie nicht vergessen, eine wichtige Datei an Ihr Projekt anzupassen, folgt hier eine Liste der wichtigsten Änderungen, die notwendig werden:

- *COPYING* – Diese Datei sollte die Lizenzbestimmungen für Ihr Programm enthalten. Standardmäßig ist hier der Text der GPL zu finden.
- *INSTALL* – Diese Datei sollte die Anweisungen zur Installation des Programms enthalten. Standardmäßig sind hier allgemeine Anweisungen zum Umgang mit *configure*, *make* und *make install* zu finden. Hier sollten Sie etwas spezifischere Anweisungen geben.
- *NEWS* – In dieser Datei sollten aktuelle Neuerungen des Programms aufgelistet werden.
- *README* – Diese Datei sollte das Programm beschreiben. Gehen Sie dabei insbesondere darauf ein, was das Programm leistet, aber auch darauf, was es nicht oder noch nicht leistet. Für eine Bedienungsanleitung können Sie hier auf die Online-Hilfe verweisen.
- *VERSION* – Diese Datei enthält die aktuelle Versionsnummer. Wenn Sie eine neue Version veröffentlichen, sollten Sie diese Versionsnummer erhöhen. Beachten Sie dabei, dass Sie hier die gleiche Versionsnummer verwenden wie in der Datei *configure.in.in*.
- *myapp.lsm* (wobei *myapp* der Name des Programms ist) – Diese Datei enthält Angaben zum Programm im Linux-Software-Map-Format. Sie wird von vielen Distributoren verwendet, wenn ein Programm in ein Paket aufgenommen wird. Ergänzen Sie hier die entsprechenden Angaben.
- *myapp.spec* (wobei *myapp* der Name des Programms ist) – Diese Datei enthält wichtige Daten, die für die Erstellung eines RPM-Pakets wichtig sind. Ergänzen Sie auch hier entsprechende Angaben.
- *configure.in.in* – Achten Sie hier besonders darauf, dass die Versionsnummer im Eintrag *AM_INIT_AUTOMAKE* korrekt ist.
- *ChangeLog* – Diese Datei sollte möglichst alle Änderungen mit Datum, Uhrzeit und (falls mehrere Entwickler am Projekt arbeiten) Urheber der Änderung enthalten. Falls Sie CVS oder ein anderes Versionskontroll-Tool benutzen, können Sie diese Datei automatisch erweitern lassen. Sonst reicht es meist aus, die wichtigsten Unterschiede zwischen verschiedenen Versionen des Programms aufzulisten.
- *Makefile.am* (im Projektverzeichnis) – Diese Datei enthält eine Reihe wichtiger Informationen darüber, welche Quelldateien kompiliert werden sollen und wo die Resultate abgespeichert werden sollen. Fügen Sie eigene Code-Dateien dem Punkt *myapp_SOURCES* hinzu (wobei *myapp* der Name der ausführbaren Datei ist), Header-Dateien kommen zum Punkt *noinst_HEADERS*. Im Punkt

kdelnkdir legen Sie fest, in welches Verzeichnis die Desktop-Datei kopiert werden soll, und damit, in welcher Rubrik im Startmenü Ihr Programm erscheinen soll. Standardmäßig steht hier das Unterverzeichnis *Utilities*, aber je nach Anwendung können Sie hier auch *Applications*, *Development*, *Editors*, *Games*, *Graphics*, *Internet*, *Multimedia*, *Office* oder *Toys* benutzen. Sie können auch eine eigene Rubrik anlegen lassen, indem Sie hier ein Verzeichnis angeben, das noch nicht existiert. Im Punkt *KDE_ICON* legen Sie die Namen der Applikations-Icons fest, die installiert werden (siehe auch den nächsten Punkt). Dazu werden alle Dateien im Verzeichnis benutzt, die den Programmnamen enthalten und die Endung *.xpm* oder *.png* besitzen. Wenn Sie hier *KDE_ICON = AUTO* setzen, werden alle Dateien benutzt, die auf *.xpm* oder *.png* enden.

- *myapp.desktop* (im Projektverzeichnis) – Diese Datei enthält Informationen über Ihre Applikation, die für das Einbinden in das KDE-System wichtig sind. Aktualisieren Sie hier die Einträge in den *Comment*-Feldern, so dass eine Beschreibung Ihres Programms angezeigt wird. Da Sie wahrscheinlich nicht alle Sprachen beherrschen, löschen Sie alle übrigen *Comment*-Einträge. Falls Ihr Programm mit einem oder mehreren Dateinamen oder URLs aufgerufen werden kann, müssen Sie auch die *Exec*-Zeile ergänzen, indem Sie *%f* für einen Dateinamen mit Pfad, *%n* für einen reinen Dateinamen oder *%u* für eine URL anhängen. Kann Ihr Programm eine Liste von solchen Parametern entgegennehmen, so benutzen Sie den entsprechenden Großbuchstaben (*%F*, *%N* oder *%U*). Setzen Sie auch den Pfad, in dem diese *desktop*-Datei gespeichert werden soll, in der Datei *Makefile.am* korrekt ein.
- Icon-Dateien (im Projektverzeichnis) – Diese Dateien enthalten das Icon, das die Applikation repräsentiert, in verschiedener Auflösung und Farbanzahl. Die Farbanzahl, Auflösung und Art des Icons wird dabei im Dateinamen kodiert. Er setzt sich dabei wie folgt zusammen:
 - *hi* (für Highcolor- oder Truecolor-Farben) oder *lo* (maximal 40 Farben aus der KDE-Farbpalette, siehe Anhang C)
 - der Breite in Pixeln (üblicherweise 16, 22, 32 oder 48)
 - anschließend ein Minuszeichen
 - die Icon-Art (*app* für Applikationen, *action* für Icons der Menü- und Werkzeugleiste)
 - nochmals ein Minus
 - der Applikationsname
 - die Dateiendung *.xpm* oder *.png*

kapptemplate erzeugt automatisch zwei Icons mit den Namen *lo16-app-myapp.png* und *lo32-app-myapp.png*. Sie sollten diese Dateien durch eigene Icons ersetzen.

Falls möglich sollten Sie dabei vier Icons zeichnen lassen, und zwar mit den Auflösungen 16 und 32 Pixeln und sowohl in *locolor* als auch in *hicolor*. Am einfachsten zeichnen Sie das Icon in 32x32 Pixeln mit vielen Farben und lassen die anderen Icons durch Umrechnung erzeugen. Anschließendes Nachbearbeiten kann die Qualität erhöhen. Beachten Sie, dass Sie auf diese Weise *nur* die Applikations-Icons in Ihr Projekt integrieren sollten! Die Icons für die Werkzeugleiste usw. sollten Sie in einem eigenen Verzeichnis ablegen, z.B. in dem von *kapptemplate* angelegten *pics*-Verzeichnis. Dort müssen Sie die Icons noch in die dortige Datei *Makefile.am* eintragen.

- Quellcode-Dateien (im Projektverzeichnis) – *kapptemplate* erzeugt bereits ein einfaches Beispielpogramm, das Sie als Anhaltspunkt für Ihr eigenes Projekt benutzen können. *kapptemplate* erzeugt hier den Code für eine Applikation, die DCOP unterstützt (*myapp.h*, *myapp.cpp*, *myappiface.h* und *main.cpp*), sowie ein weiteres Programm, das diese DCOP-Schnittstelle aufrufen kann (*myapp_client.cpp*). Es enthält eine eigene Klasse zum Anzeigen der Dokumente (*myappview.h* und *myappview.cpp*) nach dem Document-View-Konzept. Ebenso gibt es eine eigene Klasse für ein Dialogfenster für die Einstellungen (*myapppref.h* und *myapppref.cpp*), die Sie an eigene Bedürfnisse anpassen können.
- Handbuch-Dateien (im Verzeichnis *myapp/doc/*) – In diesem Verzeichnis finden Sie im Unterverzeichnis *en* bereits eine rudimentäre SGML-Datei *index.docbook*, die als Grundlage für Ihr eigenes Handbuch dienen kann. Ein Aufruf von *make html* erzeugt aus dieser Datei eine Reihe von *html*-Dateien. Dazu müssen Sie allerdings das Programm *ksgml2html* installiert haben. Passen Sie die Datei *index.docbook* an Ihr eigenes Programm an. Um das Handbuch auch in andere Sprachen zu übersetzen, legen Sie weitere Unterverzeichnisse an, z.B. *de* für Deutsch und *fr* für Französisch. Kopieren Sie *index.docbook* und *Makefile.am* aus dem *en*-Verzeichnis in die neuen Verzeichnisse. Ändern Sie die Datei *Makefile.am* jeweils so ab, dass dort das richtige Installationsverzeichnis in *datadir* = sowie im Aufruf von *ksgml2html* angegeben ist. Ergänzen Sie die neuen Verzeichnisse im Punkt *SUBDIRS* = in der Datei *myapp/doc/Makefile.am*.

Das Programm *kappgen* erzeugt genau wie *kapptemplate* die Grundstruktur eines KDE-Programms, bietet aber eine grafische Oberfläche mit einem Wizzard-Fenster, in dem man die Einstellungen vornehmen kann (siehe Abbildung 5.2).

Die beiden Tools *kapptemplate* und *kappgen* sind im Paket *kdesdk* enthalten, das Sie unter <ftp://ftp.kde.org/pub/kde/unstable/CVS/snapshots/current/> (oder besser von einem Spiegel-Server) unter dem Dateinamen *kdesdk-<Versionsdatum>.tar.bz2* herunterladen können. Zum Entpacken benötigen Sie das Programm *bzip2* (bzw. *bunzip2*), das in den meisten Linux-Distributionen inzwischen enthalten ist oder das Sie auch im Internet (zum Beispiel auf dem KDE-FTP-Server) finden können.

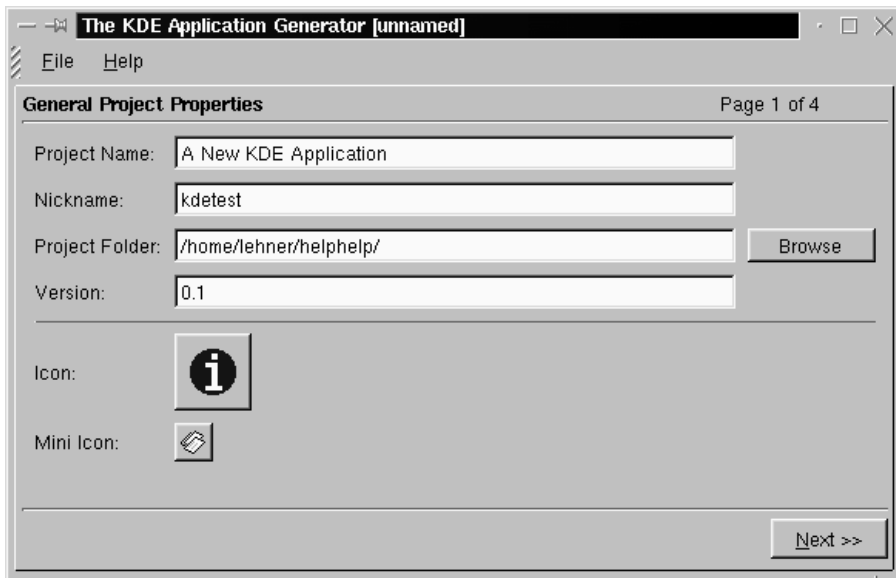


Abbildung 5-2 Einstellungsdialog von *kappgen*

Es scheint so, als würde *kappgen* nicht weiterentwickelt werden. Zumindest ist es im aktuellen *kdesdk*-Paket nicht mehr enthalten. Eventuell wird es in eine spätere Version wieder aufgenommen.

5.4 Qt Designer

Mit dem Programm *Qt Designer* der Firma Trolltech kann man sehr einfach ein Dialogfenster aus GUI-Elementen zusammenstellen (siehe Abbildungen 5.3 und 5.4) und den C++-Quellcode dazu erzeugen lassen. Es ist im normalen Lieferumfang der Qt-Bibliothek enthalten, befindet sich mit großer Wahrscheinlichkeit also schon auf Ihrer Festplatte. (Der Name der ausführbaren Datei lautet *designer*. Sie befindet sich in der Regel im Verzeichnis *\$QTDIR/bin*.) Obwohl *Qt Designer* speziell für Qt entwickelt wurde, unterstützt es auch die wichtigsten GUI-Elemente aus den KDE-Bibliotheken. Es ist auch leicht möglich, weitere Elemente in das Programm aufzunehmen.

In der Regel verläuft die Erzeugung eines Widgets oder eines Dialogs in folgenden Schritten:

1. Nach Auswahl des Menüpunkts FILE-NEW wählen Sie ein Template aus, das als Grundgerüst dienen soll. Dort stehen Ihnen neben einem rudimentären Widget auch ein Dialog, ein Dialog mit Standardbuttons und einige Gerüste mehr zur Verfügung.



Abbildung 5-3 Das ABOUT-Fenster von Qt Designer

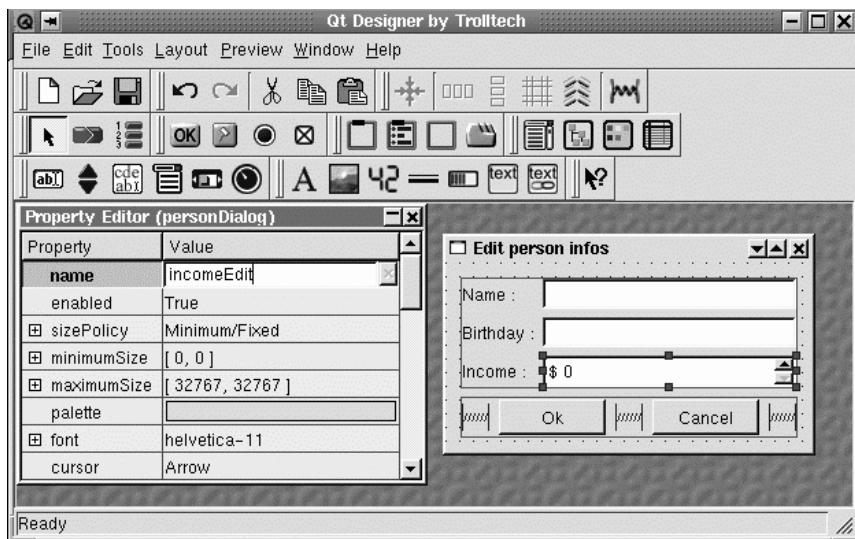


Abbildung 5-4 Qt Designer beim Entwurf eines einfachen Dialogs

2. Um einzelne GUI-Elemente einzufügen, klicken Sie zunächst auf das entsprechende Symbol in der Werkzeugleiste und ziehen anschließend in Ihrem Fenster einen Rahmen auf, in dem das Element platziert werden soll. An dieser Stelle brauchen Sie die Position und Größe nur ungefähr zu bestimmen.
3. Im Fenster *Property Editor* können Sie zusätzlich die Eigenschaften der GUI-Elemente verändern. Für alle Elemente, auf die Sie später noch vom Programm aus zugreifen wollen, sollten Sie in der Eigenschaft *name* einen sinnvollen und leicht wiedererkennbaren Namen wählen.
4. Fügen Sie Layout-Anweisungen ein. Setzen Sie dazu einen Spacer-Objekt (dargestellt durch eine Feder) an alle Stellen, an denen nachher im Fenster zwischen Elementen Platz gelassen werden soll. (Spacer sind entweder horizontal oder vertikal). Markieren Sie anschließend die Gruppe von Elementen, die nebeneinander, untereinander oder in einem Gitter angeordnet werden sollen. (Mehrere Elemente markieren Sie, indem Sie  zusammen mit der linken Maustaste verwenden oder indem Sie einen Rahmen um die Elemente ziehen.) Wählen Sie anschließend oben aus der Werkzeugleiste die gewünschte Anordnung aus. Wiederholen Sie das so lange, bis alle Elemente korrekt angeordnet sind.
5. Deklarieren Sie eigene Slots, und verbinden Sie die Signals der GUI-Elemente mit den passenden Slots. Klicken Sie dazu auf das Signal-Slot-Symbol in der Werkzeugleiste, und ziehen Sie in Ihrem Fenster eine Linie zwischen den beiden Elementen, die miteinander verbunden werden sollen (vom Signal zum Slot). Um eigene Slots für Ihr Fenster zu deklarieren, ziehen Sie die Verbindung vom GUI-Element auf den Rahmen Ihres Fensters. In dem Fenster, das sich daraufhin öffnet, können Sie das Signal und den Slot auswählen, die Sie verbinden wollen, oder auch eigene Slots hinzufügen.
6. Testen Sie Ihr Fenster mit PREVIEW-PREVIEW FORM. Testen Sie insbesondere die Aktivierung von Slots und das Verhalten des Fensters beim Verändern der Größe.
7. Speichern Sie Ihren Entwurf per FILE-SAVE in einer Datei mit der Endung *ui* (*User Interface*). Nun können Sie *Qt Designer* verlassen.
8. Erzeugen Sie daraus automatisch den Quellcode, der Ihr Fenster erzeugt, mit Hilfe des Programms *uic* (*User Interface Compiler*). (*tmake* erledigt diese Aufgabe für Sie automatisch, siehe unten.) Rufen Sie dazu *uic* zweimal auf:

```
% uic -o mywindow.h mywindow.ui  
% uic -o mywindow.cpp -impl mywindow.h mywindow.ui
```

Binden Sie die beiden Dateien in Ihr Projekt ein. Theoretisch können Sie die automatisch erzeugte Klasse bereits nutzen. In den meisten Fällen müssen Sie aber noch eigenen Code einfügen, zum Beispiel um die deklarierten Slots mit

Aktionen zu versehen, um die GUI-Elemente mit Inhalt zu füllen oder um die Klasse mit weiteren Attributen zu versehen. Es empfiehlt sich hierbei nicht, die Originaldateien zu verändern, denn Ihre Modifikationen gehen verloren, sobald Sie *uic* erneut aufrufen. Stattdessen sollten Sie eine neue Klasse erstellen, die von Ihrer Fensterklasse abgeleitet ist und die entsprechenden Methoden und Attribute enthält. Auch hierbei kann Ihnen das Programm *uic* helfen, indem es das Grundgerüst (*Skelett*) für diese abgeleitete Klasse erzeugt. Dieser Klasse müssen Sie einen Namen geben. Meist benutzt man hier den alten Klassennamen und hängt die Endung *Impl* (für *Implementation*) an. Folgende Aufrufe des *uic* erzeugen die entsprechende Klasse:

```
% uic -o mywindowimpl.h -subdecl MyWindowImpl mywindow.h mywindow.ui
% uic -o mywindowimpl.cpp -subimpl MyWindowImpl mywindowimpl.h mywindow.ui
```

Die letzten drei Kommandozeilenparameter der Aufrufe geben an, wie der Klassenname lauten soll, welche Header-Datei eingebunden werden muss und wie die zugehörige *ui*-Datei lautet. Beachten Sie, dass in *mywindowimpl.h* die Datei *mywindow.h* eingebunden werden muss, in *mywindowimpl.cpp* aber *mywindowimpl.h*. In den beiden erzeugten Dateien können Sie nun eigene Erweiterungen vornehmen. Beachten Sie aber, dass Sie diese Dateien nur einmal automatisch generieren lassen. Wenn Sie später weitere Slots deklarieren, so müssen Sie diese Slots von Hand einfügen. Lassen Sie auf keinen Fall diese Dateien neu erzeugen, da dann Ihr eigener Code wieder verloren geht.

Nach jeder Änderung an der *ui*-Datei müssen Sie den Quellcode der Klasse neu erzeugen lassen. Damit Sie nicht jedes Mal das Programm *uic* von Hand aufrufen müssen, bietet Ihnen *tmake* die Möglichkeit, dieses automatisch vornehmen zu lassen (siehe auch Kapitel 5.1, *tmake*). Fügen Sie dazu in Ihrer Projekt-Datei den Punkt *INTERFACES = ...* hinzu. In diesem Punkt geben Sie alle *ui*-Dateien an, die zu Ihrem Projekt gehören. Die entsprechenden Klassendateien werden nun automatisch generiert, sobald die *ui*-Datei verändert wurde. Die Code-Dateien erhalten dabei den gleichen Dateinamen wie die *ui*-Datei, allerdings mit den Endungen *.h* und *.cpp*. Diese Dateien brauchen Sie auch nicht in der *SOURCES*- oder *HEADERS*-Zeile einzutragen, da sie automatisch berücksichtigt werden. Das Grundgerüst für die abgeleitete Klasse müssen Sie aber weiterhin von Hand mit *uic* erzeugen und auch in die *SOURCES*- und *HEADERS*-Zeile eintragen.

Zu *Qt Designer* gehören ein Online-Handbuch sowie ein ausgezeichnetes Tutorial (beides zur Zeit leider nur in Englisch), die Sie als Einstieg in den Umgang mit *Qt Designer* unbedingt anschauen sollten. Sie finden beides im Menüpunkt *HELP*.

Wenn Sie *Qt Designer* für die Entwicklung eines KDE-Programms benutzen, sollten Sie *uic* immer mit dem Parameter *-tr i18n* aufrufen, damit für die Übersetzung der festen Textkonstanten die Funktion *i18n* anstatt der Methode *tr* benutzt wird (siehe Kapitel 4.9.1, *KDE-Übersetzungen – die Funktion i18n*).

5.5 KDevelop

KDevelop ist ein Programm, das im Rahmen des KDE-Projekts entstanden ist. Es bietet eine vollständige graphische Entwicklungsumgebung (siehe Abbildung 5.5). Mit *KDevelop* können auch große Projekte einfach verwaltet werden. *KDevelop* ist zwar für die Entwicklung von KDE-Applikationen geschrieben worden, kann jedoch auch ohne große Probleme für andere Programme benutzt werden. Durch seine intuitive Bedienung ist es insbesondere auch für Einsteiger in die KDE- und Qt-Programmierung sehr gut geeignet.

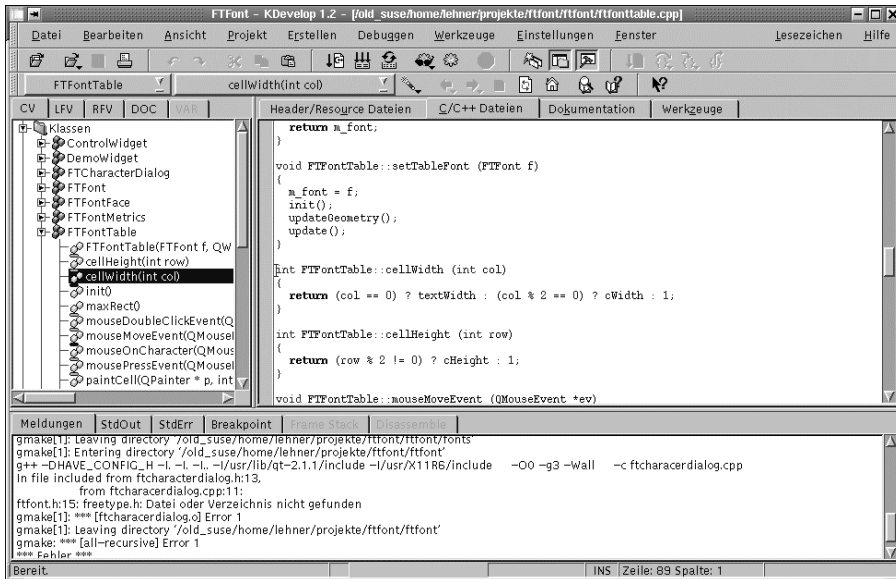


Abbildung 5-5 KDevelop mit Editorfenster (rechts), Klassenbrowser (links) und Meldungen (unten)

KDevelop stellt folgende Funktionen zur Verfügung:

- *Integrierter Editor* mit Syntax-Highlighting (siehe Abbildung 5.5, rechtes Teilfenster)
- *Klassenbrowser* (siehe Abbildung 5.5, linkes Teilfenster)
- *Projekt-Wizard* – Mit ihm kann man Grundgerüste für Projekte anlegen lassen, ähnlich wie mit *kapptemplate* und *kapppen*. Es gibt eine Reihe von Templates für verschiedene Projekte, für KDE-Projekte, für reine Qt-Projekte, leere Projekte und anderes.
- *Projekt-Manager* – Er verwaltet die Dateien, die zu einem Projekt gehören, und passt automatisch die Dateien *Makefile.am* in den Unterverzeichnissen an.

Das umfasst nicht nur die Quelldateien, sondern auch Icons, das Online-Handbuch und beliebige andere Dateien.

- *Icon-Editor* – Mit ihm kann man die notwendigen Icons zeichnen lassen.
- *Dokumentationsbrowser* – Mit ihm kann man die Online-Dokumentation zu den KDE- und Qt-Bibliotheken durchsuchen und kann auch eigene Dokumentationen im HTML-Format einbinden. Ebenso sind eine ausführliche Anleitung für die Arbeit mit *KDevelop* sowie weitere Hilfetexte zum Entwurf von KDE-Programmen bereits enthalten.
- *Integrierter einfacher Debugger* – Er bietet die wichtigsten Grundfunktionen zum Debuggen des Programms, wie z.B. Breakpoints, schrittweise Ausführung u.ä. Es ist geplant, den Debugger des KDE-Projekts *KDbg* hier einzubinden. (In der aktuellen Version 1.2 kann dieser nur als externer Debugger aufgerufen werden.)
- *CVS-Verwaltung* – Mit ihr kann das aktuelle Projekt leicht mit einer Version auf einem CVS-Server abgeglichen werden, so dass mehrere Entwickler gleichzeitig am gleichen Projekt arbeiten können.
- *Dialog-Editor* – Mit ihm kann man per Maus sehr einfach Dialoge zusammenstellen, die im eigenen Projekt benutzt werden können. Die Funktionsweise ähnelt der von *Qt Designer* (siehe Kapitel 5.4, *Qt Designer*), der *Dialog-Editor* ist allerdings zur Zeit noch nicht so ausgereift. Insbesondere fehlt eine gute Unterstützung des Layout-Konzepts von Qt. In der nächsten Version soll statt dieses Dialog-Editors eine Unterstützung für *Qt Designer* enthalten sein.

Mit nur einem Klick kann man viele Aktionen ausführen lassen:

- Das Programm kompilieren, linken oder ausführen lassen
- Ein API-Referenzhandbuch erstellen lassen (mit *kdoc*; *doxygen* wird noch nicht unterstützt, siehe Kapitel 4.6, *Klassendokumentation mit doxygen*)
- Durch Klicken auf eine Fehlermeldung im Editor direkt an die fehlerhafte Stelle springen
- Ein Sourcen-Paket erstellen

Die aktuelle Version 1.2 von *KDevelop* basiert noch auf der Version KDE 1.1. Um diese Version nutzen zu können, müssen Sie also die Bibliotheken von Qt 1.44 und KDE 1.1 installiert haben. Es kann aber trotzdem problemlos auch Programme für Qt 2.x und KDE 2.0 erstellen und verwalten. Die nächste Version wird bereits unter KDE 2.0 laufen.

Nähere Informationen zum Programm *KDevelop* finden Sie auf der Homepage <http://www.kdevelop.org/>. Das *KDevelop*-Paket können Sie von dort, aber auch von dem FTP-Server von KDE herunterladen. Die CD, die diesem Buch beiliegt, enthält ebenfalls die zum Zeitpunkt der Drucklegung des Buches aktuellste Version.

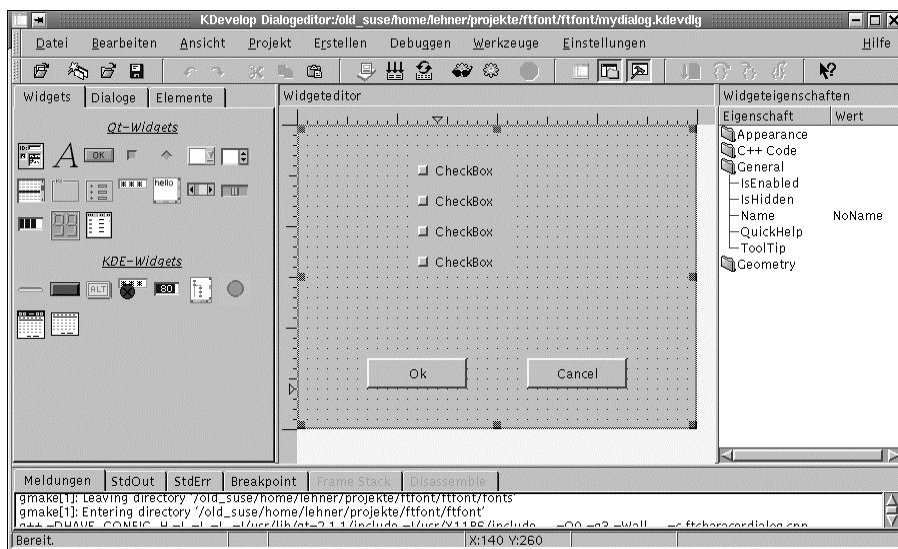


Abbildung 5-6 Integrierter Dialog-Editor von KDevelop

6 Ausgewählte, kommentierte Klassenreferenz

In diesem Kapitel werden die wichtigsten Klassen der KDE- und Qt-Bibliotheken beschrieben. Zu jeder Klasse wird vermerkt, in welcher Header-Datei sie definiert ist und von welcher Klasse sie abgeleitet ist. Das Einsatzgebiet und die Funktionsweise der Klassen werden beschrieben und in vielen Fällen mit einem Beispiel verdeutlicht. Besonderheiten bei der Benutzung der Klasse werden nochmals herausgestellt.

Bei Klassen, die bereits in den ersten Kapiteln des Buches ausführlich beschrieben worden sind, verweisen wir nur auf das entsprechende Kapitel. Die weniger wichtigen oder internen Klassen werden hier nicht aufgeführt, da das den Rahmen des Buches sprengen würde. Wir verweisen Sie hier auf die Referenzen der entsprechenden Bibliotheken. Diese sind auch auf der CD-ROM, die dem Buch beiliegt, enthalten (sowohl im HTML- als auch in Postscript-Format). Die jeweils aktuellste Version dieser Referenzen finden Sie auf der Homepage der Firma Trolltech (<http://www.trolltech.com/>) bzw. des KDE-Projekts (<http://www.kde.org/>). Am besten kopieren Sie sich die HTML-Versionen auf Ihre Festplatte und entpacken Sie dort. So können Sie während des Programmierens leicht mit einem Browser Ihrer Wahl in den Referenzen blättern. Wenn Sie KDevelop benutzen (siehe Kapitel 5.5, *KDevelop*), sind diese Referenzen automatisch im integrierten Dokumentations-Browser enthalten.

In Kapitel 6.1 werden alle Klassen der KDE-Bibliotheken, in Kapitel 6.2 alle Klassen der Qt-Bibliothek aufgeführt. Innerhalb der beiden Kapitel sind die Klassen nach Namen alphabetisch sortiert.

6.1 KDE-Klassen in kdeui, kdeccore, kfile und kio

In diesem Kapitel werden die wichtigsten Klassen aus den KDE-Bibliotheken beschrieben. Bei jeder Klasse wird aufgeführt, in welcher Bibliothek sie enthalten ist. Beim Linken des Programms müssen alle Bibliotheken angegeben sein, aus denen Klassen benutzt werden.

KAction

Definiert in *kaction.h*, abgeleitet von *QObject*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert eine Interaktion des Anwenders mit dem Menü oder der Werkzeugleiste. Ein solches Aktionsobjekt kann mit der Methode *plug* in ein Popup-Menü oder eine Werkzeugleiste eingefügt werden. Es enthält Informationen über die Beschriftung, das Icon und den aufzurufenden Slot. Genauere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KActionMenu

Definiert in *kaction.h*, abgeleitet von *KAction*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert ein Untermenü innerhalb der Menü- oder Werkzeugleiste. Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KAudioPlayer

Definiert in *kaudioplayer.h*, abgeleitet von *QObject*, enthalten in *kdecore*.

Diese Klasse bietet eine sehr einfache Möglichkeit, eine Audio-Datei auf der Soundkarte auszugeben. Benutzen Sie einfach die statische Methode *play*, der Sie den Dateinamen übergeben. Weitere Informationen finden Sie in Kapitel 4.14, *Audio-Ausgabe*.

KApplication

Definiert in *kapp.h*, abgeleitet von *QApplication*, enthalten in *kdecore*.

Jedes KDE-Programm muss genau eine Instanz dieser Klasse erzeugen. Im Konstruktor werden viele Initialisierungen vorgenommen und die Verbindung zum X-Server aufgebaut. Bevor kein *KApplication*-Objekt erzeugt worden ist, können keine Widgets oder *QPixmap*-Objekte erzeugt werden.

KApplication ist von *QApplication* abgeleitet. Neben den Initialisierungen von *QApplication* lädt *KApplication* die Übersetzungsdateien für die eingestellte Landessprache sowie die Konfigurationsdateien.

Weitere Informationen finden Sie in Kapitel 3.3.2, *KDE-Applikationen*.

KColorButton

Definiert in *kcolorbtn.h*, abgeleitet von *QPushButton*, enthalten in *kdeui*.

Dieses Widget erzeugt einen *QPushButton*, der eine Farbe anzeigt. Der Anwender kann die Farbe wählen, indem er auf den Button klickt. In diesem Fall öffnet sich ein *QColorDialog*-Fenster, in dem die Farbe eingestellt werden kann.

Weitere Informationen finden Sie in Kapitel 3.7.3, *Buttons*.

Beispiel:

Der folgende Code erzeugt ein *KColorButton*-Widget, wie es in Abbildung 6.1 zu sehen ist. Die gewählte Farbe kann mit *KColorButton::color* ausgelesen werden.

```
KColorButton *z = new KColorButton (this);  
// Anfängliche Farbe "rot" setzen  
z->setColor (red);
```



Abbildung 6-1 *KColorButton*

Besonderheiten:

KColorButton erzeugt ein *KColorDialog*-Fenster. Dieses Fenster wird im Destruktor wieder gelöscht. Das kann zu einem Laufzeitfehler führen, wenn *KColorButton* automatisch im Destruktor von *QApplication* gelöscht wird, da das *KColorDialog*-Fenster eventuell bereits vorher gelöscht worden ist. Das *KColorButton*-Widget muss also auf jeden Fall vor dem Beenden des Programms gelöscht werden.

KColorCells

Definiert in *kcolordlg.h*, abgeleitet von *QTableView*, enthalten in *kdeui*.

Dieses Widget stellt eine Auswahl von Farben in einer Tabelle aus Spalten und Zeilen dar, aus der der Anwender eine Farbe durch Anklicken auswählen kann.

Weitere Informationen finden Sie in Kapitel 3.7.5, *Auswahlelemente*.

Beispiel:

Der folgende Code erzeugt ein *KColorCells*-Widget wie es in Abbildung 6.2 zu sehen ist. Der Anwender kann durch Anklicken eines Feldes eine der Farben auswählen. Daraufhin wird das Signal *colorSelected* aufgerufen. Die Signalmethode hat einen Parameter vom Typ *int*, der den Index der gewählten Farbe angibt. Der Index der ausgewählten Farbe kann auch mit der Methode *getSelected* ermittelt werden.

```
// KColorCells-Widget mit 5 Zeilen und 5 Spalten
KColorCells *z = new KColorCells (this, 5, 5);

// Farben eintragen
z->setColor (0, QColor (124, 88, 15));
z->setColor (1, QColor (73,210, 38));
....
connect (z, SIGNAL (colorSelected (int)),
        this, SLOT (selection (int)));
```

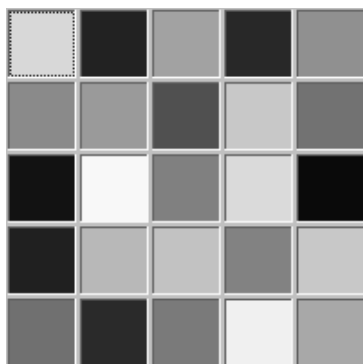


Abbildung 6-2 *KColorCells*

KColorCombo

Definiert in *kcolordlg.h*, abgeleitet von *QComboBox*, enthalten in *kdeui*.

Dieses Widget implementiert eine *QComboBox* mit einer Reihe von Farben, aus der der Anwender eine Farbe auswählen kann.

Weitere Informationen finden Sie in Kapitel 3.7.5, *Auswahlelemente*.

KColorDialog

Definiert in *kcolordlg.h*, abgeleitet von *QDialog*, enthalten in *kdeui*.

Das Dialogfenster, das durch *KColorDialog* implementiert wird, ermöglicht es dem Anwender, eine Farbe anhand des Rot-, Grün- und Blauanteils festzulegen (siehe Abbildung 6.3). Außerdem enthält das Dialogfenster eine Palette von Farbeinträgen, die der Anwender selbst festlegen und wiederverwenden kann.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

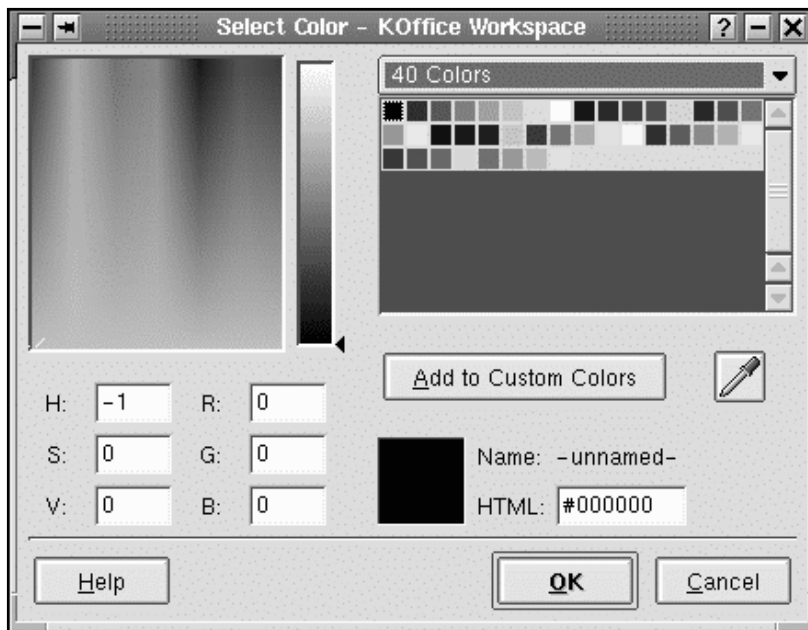


Abbildung 6-3 KColorDialog

KConfig

Definiert in *kconfig.h*, abgeleitet von *KConfigBase*, enthalten in *kdecore*.

Diese Klasse lädt zwei Konfigurationsdateien – eine globale und eine lokale. Die Einstellungen der lokalen Datei haben dabei Vorrang vor den Einträgen in der

globalen Datei. Das *KApplication*-Objekt erzeugt automatisch ein *KConfig*-Objekt, auf das Sie mit der Methode *KApplication::getConfig* zugreifen können.

Weitere Informationen finden Sie in Kapitel 4.10, *Konfigurationsdateien*.

KConfigBase

Definiert in *kconfigbase.h*, abgeleitet von *QObject*, enthalten in *kdecore*.

Diese abstrakte Klasse bildet die Basisklasse für KDE-Konfigurationsdateien. In dieser Klasse sind Methoden zum Eintragen und Auslesen von Schlüssel/Wert-Paaren in ein Konfigurationsobjekt enthalten.

Weitere Informationen finden Sie in Kapitel 4.10, *Konfigurationsdateien*.

KFileDialog

Definiert in *kfiledialog.h*, abgeleitet von *KFileBaseDialog*, enthalten in *kfile*.

In einem Dialogfenster dieser Klasse kann der Anwender eine Datei aus dem Filesystem auswählen (siehe Abbildung 6.4). Die Bibliothek *kfile* enthält weitere Klassen, zum Beispiel *KDirDialog*, mit der der Anwender ein Verzeichnis auswählen kann, und *KPreviewDialog*, bei der der Anwender in der rechten Hälfte des Fensters eine Vorschau auf den Inhalt einer ausgewählten Datei sieht.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

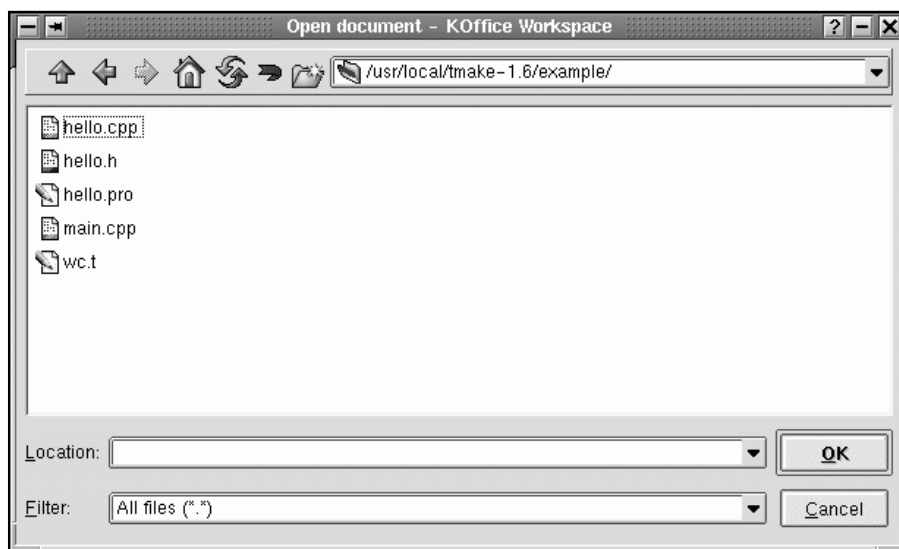


Abbildung 6-4 KFileDialog

KFontDialog

Definiert in *kfontdialog.h*, abgeleitet von *QDialog*, enthalten in *kdeui*.

In einem Dialogfenster dieser Klasse kann der Anwender einen Zeichensatz festlegen (siehe Abbildung 6.5).

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

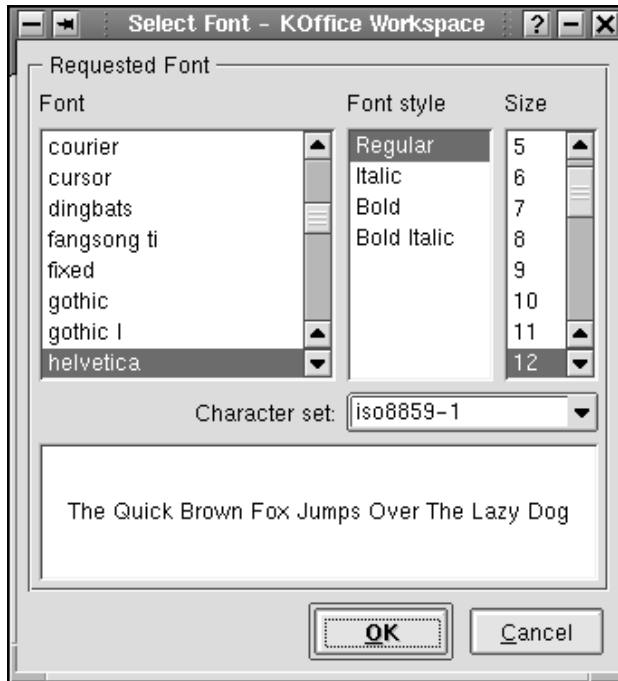


Abbildung 6-5 KFontDialog

Beispiel:

Die einfachste Anwendung des *KFontDialog*-Fensters ist der Aufruf der statischen Methode *getFont*:

```
QFont f;
int result = KFontDialog::getFont (f);
if (result == QDialog::Accepted)
{
    // Zeichensatz auf f setzen
    ....
}
```


KGradientSelector

Definiert in *kselect.h*, abgeleitet von *KSelector*, enthalten in *kdeui*.

In einem Widget dieser Klasse kann der Anwender – ähnlich wie in einem *QSlider*-Objekt – einen Wert festlegen. Im Hintergrund des Widgets wird ein Farbverlauf dargestellt.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungselemente*.

KIO

Enthalten in *kio*.

Genau genommen ist KIO keine Klasse, sondern ein Namensraum. In diesem Namensraum ist eine Reihe von Klassen und Funktionen definiert, die den Zugriff auf Dateien über ein Netzwerk mit Hilfe einer URL erlauben. Eine detaillierte Einführung finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*.

KLed

Definiert in *kledlamp.h*, abgeleitet von *QFrame*, enthalten in *kdeui*.

Ein Widget dieser Klasse stellt eine kleine Lampe auf dem Bildschirm dar, die den Zustand »ein« oder »aus« anzeigt (siehe Abbildung 6.6).

Weitere Informationen finden Sie in Kapitel 3.7.2, *Anzeigeelemente*.



Abbildung 6-6 *KLedLamp*

KLocale

Definiert in *klocale.h*, enthalten in *kdecore*.

Mit dieser Klasse wird in KDE eine Applikation an die Landeseinstellungen angepasst. Mit der Methode *translate* wird ein Text in eine andere Sprache übersetzt. Mit Hilfe des Makros *i18n* wird die Einbindung in ein Programm sehr einfach.

Weitere Informationen finden Sie in Kapitel 4.9, *Mehrsprachige Anwendungen und Internationalisierung*.

KMainWindow

Definiert in *kmainwindow.h*, abgeleitet von *QMainWindow*, enthalten in *kdeui*.

Diese Widget-Klasse implementiert ein Hauptfenster einer Applikation. Es enthält bereits Methoden zum Anlegen von Menüleiste, Werkzeugleisten und Statuszeile, Methoden zur Verwaltung mehrerer Hauptfenster sowie Methoden zum Speichern der Konfiguration in einem Session-Management-Ablauf.

Detaillierte Informationen über diese Klasse und ihre Anwendungen finden Sie in Kapitel 3.5, *Das Hauptfenster*. Informationen über das Session-Management mit der Klasse *KMainWindow* finden Sie in Kapitel 4.16.1, *Session-Management in KDE*.

KMenuBar

Definiert in *kmenubar.h*, abgeleitet von *QFrame*, enthalten in *kdeui*.

Diese Widget-Klasse implementiert eine Menüleiste, die in der Regel am oberen Rand des Hauptfensters dargestellt wird. Diese Menüleiste kann jedoch auch am unteren Rand oder frei beweglich dargestellt werden.

Die KDE-Hauptfensterklasse *KMainWindow* kann ein *KMenuBar*-Objekt selbstständig erzeugen und verwalten.

Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KMessageBox

Definiert in *kmessagebox.h*, enthalten in *kdeui*.

Diese Klasse bietet eine Reihe von statischen Methoden, mit denen sehr einfach ein Meldungsfenster geöffnet werden kann. Rufen Sie dazu eine der statischen Methoden *yesNo*, *yesNoCancel* oder *information* auf. Die erste Methode besitzt zwei Buttons, die zweite Methode drei Buttons. Diese Methoden werden eingesetzt, wenn der Anwender auf eine Ja/Nein-Frage antworten soll. Die dritte Methode erzeugt ein Informationsfenster mit nur einem Button. Diese Methode wird oft eingesetzt, um wichtige Meldungen darzustellen, die der Anwender erst bestätigen muss, bevor er weiterarbeiten kann. *KMessageBox* enthält noch eine Reihe weiterer interessanter Methoden. Der Anwender hat dabei die Möglichkeit, das Feld »Dieses Fenster nicht mehr anzeigen« anzuwählen. Dieses Informationsfenster wird dann in Zukunft unterdrückt. *KMessageBox* speichert in der Konfigurationsdatei ab, welche Informationsfenster unterdrückt werden sollen.

Alternativ kann man auch die Klasse *QMessageBox* verwenden. Welche Klasse man für eine bestimmte Aufgabe vorzieht, muss man von Fall zu Fall entscheiden.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

KProcess

Definiert in *kprocess.h*, abgeleitet von *QObject*, enthalten in *kdecore*.

Mit einem Objekt dieser Klasse kann man andere Programme und Applikationen starten. Dabei sind mehrere Modi möglich: Das Programm kann warten, bis das Unterprogramm abgearbeitet ist, es kann sich informieren lassen, wann das

Unterprogramm beendet ist, oder es kann völlig unabhängig vom Unterprogramm ablaufen.

Weitere Informationen finden Sie in Kapitel 4.13.4, *Mehrere Prozesse und Threads*, im Abschnitt *Ausführen anderer Programme*.

KRadioAction

Definiert in *kaction.h*, abgeleitet von *KToggleAction*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert eine Benutzeraktion, die eingeschaltet sein kann und die ausgeschaltet wird, indem ein anderes Objekt ausgeschaltet wird. Dieses Aktionsobjekt kann in Menü- und Werkzeugleisten benutzt werden. Um festzulegen, welche *QRadioAction*-Objekte zusammengehören und sich gegenseitig auslösen sollen, trägt man mit der Methode *setExclusiveGroup* den gleichen Identifikationsstring ein. Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KRecentFilesAction

Definiert in *kaction.h*, abgeleitet von *KListAction*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert eine Benutzeraktion, bei der der Anwender aus einer Liste von den zuletzt benutzten Dateien eine auswählt. (Diese wird dann in der Regel eingelesen und angezeigt.) Die Liste kann in der Konfigurationsdatei gespeichert werden, so dass sie beim nächsten Programmstart die Daten beibehält. Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KSelectAction

Definiert in *kaction.h*, abgeleitet von *KAction*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert eine Benutzeraktion, bei der der Anwender eine von mehreren Möglichkeiten aktivieren kann. Dieses Aktionsobjekt kann in Menü- und Werkzeugleisten benutzt werden. Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KServerSocket

Definiert in *ksock.h*, abgeleitet von *QObject*, enthalten in *kdecore*.

Mit einem Objekt dieser Klasse können Sie einen Socket (sowohl einen TCP/IP-Socket als auch einen Unix-Domain-Socket) erzeugen, der auf eine Verbindung wartet. Sobald eine Verbindung eintrifft, wird das Signal *accepted* aufgerufen, das als Parameter ein *KSocket*-Objekt enthält, das die neue Verbindung enthält.

Auch die Qt-Bibliothek enthält inzwischen eine Klasse, *QServerSocket*, mit der gleichen Funktionalität. Informationen über diese Klasse finden Sie in Kapitel 4.19.1, *Socket-Verbindungen*.

KSimpleConfig

Definiert in *ksimpleconfig.h*, abgeleitet von *KConfigBase*, enthalten in *kdecore*.

In einem Objekt dieser Klasse wird der Inhalt einer einzelnen Konfigurationsdatei abgelegt. Sie bietet – im Gegensatz zu *KConfig* – die Möglichkeit, einzelne Schlüssel/Wert-Paare oder sogar ganze Gruppen zu löschen. *KSimpleConfig* wird häufig benutzt, um eigene Dateien im Format der KDE-Konfigurationsdateien zu erstellen.

Weitere Informationen finden Sie in Kapitel 4.10.4, *Eigene Konfigurationsdateien*.

KSocket

Definiert in *ksock.h*, abgeleitet von *QObject*, enthalten in *kdecore*.

Ein Objekt der Klasse *KSocket* kann eine Socket-Verbindung (sowohl TCP/IP-Sockets als auch Unix-Domain-Sockets) verwalten. Man kann die Socket-Verbindung sowohl zum Schreiben als auch zum Lesen (und auch zu beidem) benutzen. Das Lesen oder Schreiben muss man mit der Methode *enableRead* bzw. *enableWrite* zunächst aktivieren. Wenn neue Daten ankommen bzw. der Socket zum Senden weiterer Daten bereit ist, wird das Signal *readEvent* bzw. *writeEvent* gesendet. Wenn die Verbindung unterbrochen wird, wird das Signal *closeEvent* gesendet.

Auch die Qt-Bibliothek besitzt Klassen zum Umgang mit Sockets, insbesondere *QSocket*, *QSocketDevice* und *QServerSocket*. Diese Klassen bieten eine einfachere Schnittstelle. Genauere Informationen finden Sie in Kapitel 4.19.1, *Socket-Verbindungen*.

KStatusBar

Definiert in *kstatusbar.h*, abgeleitet von *QStatus*, enthalten in *kdeui*.

Diese Widget-Klasse implementiert eine Statuszeile, die üblicherweise am unteren Rand eines Applikationsfensters steht. Sie kann Informationen über den aktuellen Status des Programms enthalten, zum Beispiel die Cursor-Position, Informationen über ausgewählte Objekte oder den aktuellen Status einer längeren Berechnung.

Die Statuszeile kann in mehrere Bereiche unterteilt werden. *KStatusBar* bietet auch die Möglichkeit, einen Mausklick auf einen der Bereiche festzustellen.

Die Hauptfensterklasse *KMainWindow* enthält bereits die Methode *statusBar*, mit der ein Objekt dieser Klasse automatisch erzeugt und unten in das Fenster eingefügt werden kann.

Weitere Informationen finden Sie in Kapitel 3.5.5, *Die Statuszeile*.

KMainWindow

Definiert in *ktmainwindow.h*, abgeleitet von *QWidget*, enthalten in *kdeui*.

Diese Klasse wurde früher als Hauptfensterklasse benutzt und ist nur noch aus Kompatibilitätsgründen vorhanden. Neue Programme sollten stattdessen die Klasse *KMainWindow* benutzen.

KToggleAction

Definiert in *kaction.h*, abgeleitet von *KAction*, enthalten in *kdeui*.

Ein Objekt dieser Klasse repräsentiert eine Benutzeraktion, die ein- oder ausgeschaltet sein kann. Dieses Aktionsobjekt kann in Menü- und Werkzeugleisten benutzt werden. Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KToolBar

Definiert in *ktoolbar.h*, abgeleitet von *QFrame*, enthalten in *kdeui*.

Diese Widget-Klasse implementiert eine Werkzeugleiste. Eine Werkzeugleiste enthält Buttons mit kleinen Icons, über die die wichtigsten Befehle der Menüleiste noch schneller zu erreichen sind.

Die Klasse *KToolBar* erzeugt eine Werkzeugleiste, die oben oder unten im Fenster angeordnet werden kann oder auch frei beweglich verschoben werden kann.

Die Hauptfensterklasse *KMainWindow* enthält bereits die Methode *toolBar*, mit der eine oder mehrere Werkzeugleisten erzeugt werden können.

Weitere Informationen finden Sie in Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*.

KUniqueApplication

Definiert in *kuniqueapp.h*, abgeleitet von *KApplication*, enthalten in *kdecore*.

Setzt man diese Klasse anstelle von *KApplication* ein, so wird nur maximal eine Instanz des Programms gestartet. Startet man das Programm erneut, werden alle wichtigen Daten an die laufende Instanz gesendet, und das zweite Programm wird beendet.

Genaue Informationen über den Einsatz von *KUniqueApplication* finden Sie in Kapitel 3.3.2, *KDE-Applicationen*.

KURL

Definiert in *kurl.h*, enthalten in *kdecore*.

In dieser Klasse kann eine URL (eine eindeutige Adresse eines Dokuments, entweder lokal im Dateisystem oder im Internet) gespeichert werden. Eine URL ist

dabei ein String, der aus einer Protokollangabe (z.B. *http:*, *ftp:* oder *file:*), einer Pfadangabe und einem Dateinamen besteht.

KURL besitzt eine Reihe von Methoden, mit denen man die gespeicherte URL analysieren und aufspalten kann. Weitere Informationen finden Sie in Kapitel 4.19.2, *Netzwerktransparenter Dateizugriff mit KIO*.

KXYSelector

Definiert in *kselect.h*, abgeleitet von *QWidget*, enthalten in *kdeui*.

Mit einem Widget dieser Klasse kann der Anwender einen Punkt in einem zweidimensionalen Koordinatensystem festlegen. Dabei kann das Widget mit einem Hintergrundbild versehen werden. Dieses Widget wird beispielsweise in *KColorDialog* benutzt, um gleichzeitig die Sättigung und den Farbton einer Farbe einzustellen.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungs-elemente*.

6.2 Qt-Klassen

In diesem Kapitel werden die wichtigsten Klassen aus der Qt-Bibliothek beschrieben. Beim Linken muss die Bibliothek *qt* eingebunden werden. Einige der Klassen sind in getrennten Modulen definiert. Beachten Sie, dass Sie eine Bibliothek benutzen, die dieses Modul einkompiliert hat. (Für die frei erhältliche Qt Free Edition sind üblicherweise alle Module eingebunden. Nur die Qt Professional Edition (die »Light«-Version der kommerziellen Qt-Lizenz), enthält nicht alle Module. Alle Klassen, bei denen keine Angabe zum Modul steht, sind in der Hauptbibliothek enthalten.

QAccel

Definiert in *qaccel.h*, abgeleitet von *QObject*.

QAccel verwaltet Tastatur-Kurzkommandos, wie sie in vielen Programmen benutzt werden, um die Auswahl von wichtigen Menüpunkten zu beschleunigen, z.B. **[Strg]+[P]**, um das aktuelle Dokument zu drucken, oder **[F1]**, um die Online-Hilfe zu starten. *QAccel* wird dazu einem Fenster zugeordnet. Wird in diesem Fenster (oder einem der anderen Fenster im gleichen Toplevel-Widget) nun eine der Kurzkommandotasten betätigt, so sendet das *QAccel*-Objekt das Signal *activated(int id)* mit der zugehörigen ID. Stattdessen kann auch zu jedem eingetragenen Kurzkommando ein parameterloser Slot angegeben werden, der aufgerufen wird.

Beim Entwurf eines KDE-Programms wird diese Klasse selten direkt benötigt. Kurzkommandos lassen sich am besten beim Einfügen eines Menüpunktes in ein *KMenuBar*-Objekt festlegen (siehe Kapitel 3.5.5, *Definition von Aktionen für die Menü- und Werkzeugleisten*).

QAction

Definiert in *qaction.h*, abgeleitet von *QObject*.

Ein *QAction*-Objekt repräsentiert eine Benutzeraktion, insbesondere in Menü- und Werkzeugleisten. In KDE-Programmen sollte stattdessen die Klasse *KAction* oder eine Unterklasse benutzt werden (siehe Kapitel 3.5.3, *Definition von Aktionen für Menü- und Werkzeugleisten*).

Eine Aktion enthält dabei Informationen über die Bezeichnung und das Icon einer Aktion. Sie kann mit *setToggleAction(true)* als ein-/ausschaltbare Aktion definiert werden. Mit den Methoden *addTo* und *removeFrom* kann man die Aktion in Elemente des Typs *QPopupMenu*, *QMenuBar* oder *QToolBar* einfügen lassen.

Weitere Informationen finden Sie in Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*.

QActionGroup

Definiert in *qaction.h*, abgeleitet von *QAction*.

Ein Objekt dieser Klasse kann *QAction*-Objekte sammeln und verwalten. Dazu benutzt man das Objekt als Vaterobjekt der *QAction*-Objekte. Man kann sie auch manuell mit *insert* einfügen. Mit einer solchen Gruppierung können beispielsweise mehrere *QAction*-Objekte mit einem Aufruf von *setEnabled* aktiviert oder deaktiviert werden. Die häufigste Anwendung ist allerdings, wenn mehrere ein-/ausschaltbare *QAction*-Objekte sich gegenseitig ausschließen sollen. Diese Aktionen können dann in einem *QActionGroup*-Objekt gesammelt werden. Anschließend rufen Sie *setExclusive(true)* für dieses *QActionGroup*-Objekt auf. Weitere Informationen finden Sie in Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*.

In KDE-Programmen benutzen Sie stattdessen die von *KAction* abgeleitete Klasse *KRadioAction* und setzen dort mit *setExclusiveGroup* für alle Aktionen einen gemeinsamen String ein.

QApplication

Definiert in *qapplication.h*, abgeleitet von *QObject*.

Diese Klasse stellt das zentrale Kontroll-Objekt eines KDE- oder Qt-Programms dar. Von dieser Klasse muss es in jeder Applikation *genau ein* Objekt geben. Bevor dieses Objekt nicht erzeugt ist, dürfen viele andere Klassen (wie z.B. *QWidget*) nicht benutzt werden. Das Objekt dieser Klasse ist für die Verbindung mit dem X-Server und die Verteilung von Events zuständig.

In der KDE-Bibliothek ist die Klasse *KApplication* definiert, die von *QApplication* abgeleitet ist. Sie übernimmt vollständig deren Funktion, implementiert aber

zusätzlich noch weitere Methoden, z.B. für den Umgang mit Konfigurationsdateien. Eine KDE-Applikation sollte daher *immer* ein Objekt der Klasse *KApplication* benutzen.

Eine genauere Beschreibung der Klassen *QApplication* und *KApplication* ist in Kapitel 3.3, *Grundstruktur einer Applikation*, zu finden.

QArray

Definiert in *qarray.h*, abgeleitet von *QGArray* (*protected*).

Das Klassen-Template *QArray* verwaltet eine Array-Datenstruktur eines beliebigen einfachen Typs (keine Typen mit Konstruktoren, Destruktoren oder virtuellen Methoden). Die Datenelemente werden in einem Block im Speicher unmittelbar hintereinander abgelegt. Das gesamte Array wird als *explizit gemeinsame Daten* behandelt (siehe Kapitel 4.7.1, *Gemeinsame Daten*), d.h. bei einer Zuweisung von einem *QArray* an ein anderes werden die Daten nicht kopiert, sondern es wird nur der Referenzzähler erhöht. Man kann die Anzahl der enthaltenen Elemente zur Laufzeit mit der Methode *resize* ändern, das ganze Array mit einem festen Wert initialisieren und im Array nach Elementen suchen.

Die Klasse *QArray* ist in Kapitel 4.7.2, *Container-Klassen*, genauer beschrieben.

QAsciiCache

Definiert in *qasciicache.h*, abgeleitet von *QGCache*.

In dieser Template-Klasse *QAsciiCache* <type> wird ein Cache implementiert, der Schlüsseln vom Typ *char** Werte vom Typ *type** zuordnet.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie im Abschnitt zur Klasse *QCache* hier im Referenzteil.

QAsciiCachelterator

Definiert in *qasciicache.h*, abgeleitet von *QGCachelterator*.

Mit dieser Template-Klasse können Sie ein Iterator-Objekt zu einem *QAsciiCache*-Objekt erzeugen lassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QAsciiDict

Definiert in *qasciidict.h*, abgeleitet von *QGDict*.

In dieser Template-Klasse *QAsciiDict* <type> wird eine Hash-Tabelle implementiert, die Schlüsseln vom Typ *char** Werte vom Typ *type** zuordnet.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie im Abschnitt zur Klasse *QDict* hier im Referenzteil.

QAsciiDictIterator

Definiert in *qasciidict.h*, abgeleitet von *QGDictIterator*.

Mit dieser Template-Klasse können Sie ein Iterator-Objekt zu einem *QAsciiDict*-Objekt erzeugen lassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie in Kapitel 4.7.3, *Iterator-Objekte*.

QAsyncIO

Definiert in *qasyncio.h*, abgeleitet von *QObject*.

Diese Klasse ist die Basisklasse für *QDataSource* und *QDataSink*, zwei Klassen, die zur asynchronen Ein-/Ausgabe benutzt werden. Mit ihrer Hilfe lassen sich große Dateien oder langsame Streams in einer Applikation leicht auslesen bzw. beschreiben, ohne dass der Prozess blockiert. Eine Beschreibung zu diesem Konzept finden Sie in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

QBitArray

Definiert in *qbitarray.h*, abgeleitet von *QByteArray*.

Diese Klasse verwaltet ein Array von Bits oder Boolean-Werten, die im Speicher direkt aneinander gepackt werden, um möglichst wenig Speicher zu verschwenden. Intern greift diese Klasse auf ein *QByteArray* zurück (von dieser Klasse ist es abgeleitet) und schreibt je acht Bits des Bit-Arrays in ein Byte des Byte-Array. Durch Überladen der Zugriffsooperatoren wird die Umrechnung auf die richtige Stelle im Byte-Array vorgenommen.

Eine Beschreibung von *QBitArray* und *QByteArray* finden Sie auch in Kapitel 4.7.2, *Container-Klassen*.

Besonderheiten:

Der Zugriff auf ein einzelnes Bit ist mit den Methoden *setBit*, *clearBit*, *toggleBit* und *testBit* viel effizienter als ein Zugriff mit dem *operator[]*. (Letzterer erzeugt zunächst ein Element der Klasse *QBitVal*, so dass das Bit sowohl geschrieben als auch ausgelesen werden kann, ohne die anderen Bits im Byte zu verändern.) Verwenden Sie daher besser die Methoden als den Operator.

QBitmap

Definiert in *qbitmap.h*, abgeleitet von *QPixmap*.

Die *QBitmap*-Klasse ist eine Kapselung einer X-Bitmap, also einer QPixmap der Farbtiefe 1 (nur zwei Farben), die im X-Server gespeichert wird. Zum Zeichnen in ein *QBitmap*-Objekt mit *QPainter* sollten Sie die Farben *color0* und *color1* benut-

zen, die die Bits entsprechend auf 0 oder 1 setzen. Bitmaps werden vor allem als Cursor-Formen und als Transparenz-Masken benutzt.

QBitmap ist von *QPixmap* abgeleitet und stellt nur eine Spezialisierung auf die Farbtiefe 1 dar.

Eine genaue Beschreibung von *QPixmap* und *QBitmap* finden Sie in Kapitel 4.3.2, *QPixmap*, sowie in Kapitel 4.3.4, *Transparenz in Teilbildern*.

QBitVal

Definiert in *qbitarray.h*.

Diese Hilfsklasse dient nur dazu, den *operator[]* der Klasse *QBitArray* zu implementieren. In einem Objekt dieser Klasse werden ein Zeiger auf das *QBitArray* und der adressierte Bit-Index gespeichert. Beim Lesezugriff wird nun mit *QBitArray::testBit* der Wert ermittelt und beim Schreibzugriff der Wert mit *QBitArray::setBit* gesetzt. Vermeiden Sie möglichst den Zugriff auf einzelne Bits eines *QBitArray*-Objekts mit *operator[]*, da es ineffizient ist, zunächst ein *QBitVal*-Objekt anzulegen (siehe auch die Klasse *QBitArray*).

QBoxLayout

Definiert in *qlayout.h*, abgeleitet von *QLayout*.

Diese Klasse ermöglicht die automatische Anordnung von Widgets nebeneinander (horizontal) oder untereinander (vertikal). Die Widgetgröße und -position wird automatisch angepasst, wenn das Hauptwidget geändert wird. Ein gutes Layout ist für eine intuitive Benutzeroberfläche unverzichtbar, und dafür bietet diese Klasse ein sehr mächtiges, aber leider auch schwierig anzuwendendes Hilfsmittel.

Eine genaue Beschreibung der Anwendung der Layout-Klassen finden Sie in Kapitel 3.6.4, *Die Layout-Klassen QBoxLayout und QGridLayout*.

Im Konstruktor dieser Klasse kann man angeben, ob die Unter-Widgets nebeneinander (*QBoxLayout::LeftToRight* oder *QBoxLayout::RightToLeft*) oder untereinander (*QBoxLayout::Up* oder *QBoxLayout::Down*) angeordnet werden sollen. In den meisten Fällen ist es jedoch einfacher, die Klassen *QHBoxLayout* und *QVBoxLayout* zu verwenden, die jeweils nur eine Richtung benutzen, da sie einen einfacheren Konstruktor besitzen.

Einfachere, aber nicht so mächtige Klassen für die Anordnung von Unter-Widgets sind die Klassen *QVBox* und *QHBox* (siehe Kapitel 3.6.3, *Einfache Layouts mit QHBox, QVBox und QGrid*).

QBrush

Definiert in *qbrush.h*.

Diese Klasse definiert und verwaltet Füllmuster, die beim Zeichnen mit *QPainter* benutzt werden. Ein Füllmuster besteht aus einem Muster und einer Füllfarbe. Besondere Muster sind *NoBrush* (nicht ausfüllen), *SolidPattern* (vollständig mit der Farbe ausfüllen) und *CustomPattern* (mit dem Muster eines *QPixmap*- oder *QBitmap*-Objekts füllen). Mehr über die Klasse *QBrush* erfahren Sie in Kapitel 4.2.2, *Linienarten und Füllmuster*.

QBuffer

Definiert in *qbuffer.h*, abgeleitet von *QIODevice*.

Diese Klasse ist von *QIODevice* abgeleitet. Ein Objekt dieser Klasse kann daher wie eine Datei benutzt werden, indem es mit *open* geöffnet und mit *close* wieder geschlossen wird. Mit *writeBlock* und *readBlock* kann man Datenblöcke in ein Objekt dieser Klasse schreiben bzw. aus ihm auslesen. Intern werden die Daten in einem *QByteArray*-Objekt abgelegt. Mit dieser Klasse kann man ein Dokument statt in einer Datei in einem *QByteArray*-Objekt speichern, um es zum Beispiel über die Zwischenablage oder per Drag&Drop zu kopieren. Mit der Methode *buffer* erhalten Sie Zugriff auf das *QByteArray*-Objekt, mit der Methode *setBuffer* können Sie ein *QByteArray*-Objekt als Daten in das *QBuffer*-Objekt schreiben. Nähere Informationen finden Sie in Kapitel 4.18, *Dateizugriffe*.

Eine häufige Anwendung benutzt *QBuffer*, um mit Hilfe der Klasse *QDataStream* Objekte in einen Datenblock im Speicher statt in eine Datei zu schreiben. Außerdem kann *QBuffer* in einem *QIODeviceSource* benutzt werden, um als Quelle einer asynchronen Datenübertragung zu dienen (siehe auch Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*).

QPushButton

Definiert in *qbutton.h*, abgeleitet von *QWidget*.

Diese Klasse stellt die abstrakte Basisklasse für Buttons aller Art dar. Alle Gemeinsamkeiten von Buttons sind hier realisiert: Ein Button enthält einen Text oder eine Pixmap in der Darstellung, kann in den Modus *disabled* gesetzt werden und aktiviert Signale, wenn er gedrückt (*pressed*) oder losgelassen (*released*) wird. Das Signal *clicked* wird gesendet, wenn er gedrückt und anschließend losgelassen wurde. (In der Regel benutzen Sie dieses Signal.) Es gibt zwei Varianten von Buttons: solche, die beim Betätigen eine einmalige Aktion hervorrufen (normale Buttons), und solche, die beim Betätigen zwischen zwei Zuständen wechseln (Toggle-Buttons). Toggle-Buttons versenden das Signal *toggled*, wenn sich ihr Zustand ändert. Vier spezielle Button-Klassen, die von *QPushButton* abgeleitet sind, sind bereits in Qt implementiert: *QPushButon*, *QToolButton*, *QRadioButton* und *QCheckBox*. Die ersten beiden sind standardmäßig normale Buttons, die letzten beiden Toggle-Buttons.

QButtonGroup

Definiert in *qbuttongroup.h*, abgeleitet von *QGroupBox*.

Diese Klasse fasst eine Sammlung von Buttons zusammen. Sie ist insbesondere nötig, um mehrere, sich ausschließende *QRadioButton*-Objekte einander zuzuordnen. Weiterhin kann *QButtonGroup* Signale mehrerer Buttons zu einem Signal mit einer Button-ID zusammenfassen.

Es gibt zwei Möglichkeiten, Buttons zusammenzufassen: Man kann die *QButtonGroup* als Vater-Widget für alle Buttons benutzen. Da *QButtonGroup* von *QGroupBox* abgeleitet ist, wird in diesem Fall ein Rahmen gezeichnet, in dem die Buttons stehen. Somit ist auch grafisch sichtbar, dass diese Buttons zusammengehören. Man kann aber auch die Buttons einzeln mit der Methode *insert* in das *QButtonGroup*-Objekt einfügen. Die *QButtonGroup* kann dann unsichtbar bleiben. Sie verwaltet in diesem Fall nur die eingefügten Objekte vom Typ *QRadioButton*.

Weitere Informationen zu *QButtonGroup* erhalten Sie in Kapitel 3.7.3, *Buttons*, Kapitel 3.7.7, *Verwaltungselemente*, und Kapitel 3.8.4, *Optionsdialoge*.

QByteArray

Definiert in *qcstring.h*, definiert als *QArray<char>*.

QByteArray ist keine Klasse, sondern ein Typ, der als *QArray<char>* definiert ist. Er realisiert also ein flexibles Array von Bytes und Zugriffsmethoden darauf. *QByteArray* wird hauptsächlich als spezielles *QIODevice* im Speicher (siehe *QBuffer*) und als Basisklasse für *QCString* benutzt. Sie können *QByteArray* natürlich ebenfalls überall dort benutzen, wo Sie binäre Daten zwischenspeichern müssen.

Eine Beschreibung von *QArray* und *QByteArray* finden Sie auch in Kapitel 4.7.2, *Container-Klassen*.

QCache

Definiert in *qcache.h*, abgeleitet von *QGCACHE*.

Dieses Template implementiert einen Cache-Speicher für einen beliebigen Datentyp. Ein Cache wird verwendet, um bereits geladene Daten abzuspeichern, damit sie bei einem erneuten Zugriff schneller erreicht werden können. Typische Anwendungen sind z.B. ein Cache für WWW-Seiten (Zugriff mit der URL als Schlüssel) oder ein Cache von Grafikdateien (Zugriff mit dem Dateinamen als Schlüssel).

Die Elemente des Cache müssen mit *new* auf dem Heap angelegt worden sein. Das Löschen der Elemente übernimmt das *QCache*-Objekt. Die Schlüssel zum Auffinden eines Cache-Elements sind Strings vom Typ *QString*. Jedem Element wird ein Kostenwert zugeordnet (meist ist das der verbrauchte Speicherplatz, es kann aber auch eine beliebige andere positive ganze Zahl sein). Sobald die

Summe der Kostenwerte einen Maximalwert überschreitet, werden die Elemente, die am längsten nicht benutzt worden sind, aus dem Cache gelöscht. Intern wird für die Suche nach einem Element im Cache ein Hash-Verfahren benutzt, so dass das gesuchte Element in der Regel sofort gefunden wird.

Weitere Informationen über *QCache* finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QCacheIterator

Definiert in *qcache.h*, abgeleitet von *QGCacheIterator*.

Diese Klasse erzeugt ein Iterator-Objekt für ein *QCache*-Objekt, um alle Elemente des Cache der Reihe nach durchzugehen. (Siehe auch *QCache* sowie Kapitel 4.7.2, *Container-Klassen*, und Kapitel 4.7.3, *Iterator-Objekte*.)

QCanvas

Definiert in *qcanvas.h*, abgeleitet von *QObject*, enthalten im Modul *Canvas*.

Diese Klasse verwaltet ein zweidimensionales Feld mit grafischen Objekten, die sich auf diesem Feld bewegen können und sich gegenseitig überdecken können. Durch Speichern der Grafik in einem zusätzlichen *QPixmap*-Objekt wird ein Flimmern beim Verschieben von Objekten vermieden.

Beachten Sie, dass *QCanvas* nur die Objekte verwaltet. Das Anzeigen der Objekte in einem *QCanvas*-Objekt geschieht mit einem *QCanvasView*-Objekt. Die einzelnen Objekte sind Instanzen der Klasse *QCanvasItem* bzw. einer davon abgeleiteten Klassen (zur Zeit *QCanvasEllipse*, *QCanvasLine*, *QCanvasPixmap*, *QCanvasPixmapArray*, *QCanvasPolygon*, *QCanvasRectangle*, *QCanvasSprite* und *QCanvasText*). Für eigene Objekte können Sie eine eigene Klasse von *QCanvasItem* oder meist einfacher von *QCanvasPolygonalItem* ableiten.

Weitere Informationen finden Sie in Kapitel 4.5.4, *Bewegte Objekte mit QCanvas*. Beachten Sie, dass die Klasse *QCanvas* und alle zugehörigen Klassen in einem gesonderten Modul *Canvas* definiert sind. Dieses Modul ist in der kommerziellen »light«-Version Qt-Professional-Edition nicht enthalten, wohl aber in der freien Version.

QCanvasItem

Definiert in *qcanvas.h*, abgeleitet von *Qt*.

Diese abstrakte Klasse repräsentiert ein Objekt in einem *QCanvas*-Objekt. Nähere Informationen finden Sie unter *QCanvas* sowie in Kapitel 4.5.4, *Bewegte Objekte mit QCanvas*.

QCanvasView

Definiert in *qcanvas.h* abgeleitet von *QScrollWidget*.

Ein Widget dieser Klasse stellt den Inhalt eines *QCanvas*-Objekts auf dem Bildschirm dar. Sie können mehrere *QCanvasView*-Objekte verschiedene Ausschnitte eines *QCanvas*-Objekts anzeigen lassen.

Nähere Informationen finden Sie unter *QCanvas* sowie in Kapitel 4.5.4, *Bewegte Objekte mit QCanvas*.

QCDEStyle

Definiert in *qcdestyle.h*, abgeleitet von *QMotifStyle*.

Diese Klasse definiert einen Darstellungs-Style, der das Look&Feel der CDE-Oberfläche auf dem Bildschirm erzeugt. Er basiert auf der Klasse *QMotifStyle*. Es sind nur folgende virtuelle Methoden neu definiert: *defaultFrameWidth*, *drawArrow*, *drawIndicator* und *drawExclusiveIndicator*.

QChar

Definiert in *qstring.h*.

In einem Objekt dieser Klasse wird ein *Unicode*-Zeichen (16 Bit) abgespeichert. Die Umwandlung eines ASCII-Zeichens in ein Unicode-Zeichen und umgekehrt geschieht automatisch an den notwendigen Stellen durch einen Cast-Operator und einen Konstruktor. Bei der Umwandlung von *QChar* nach *char* können allerdings Informationen verloren gehen. Diese Umwandlung können Sie auch explizit mit der Methode *QChar::latin1* vornehmen lassen, um die automatische Typumwandlung zu vermeiden und so das Programm leichter lesbar zu machen. Soweit wie möglich sollten Sie alle angezeigten Texte mit *QChar* und *QString* bearbeiten. Weitere Informationen finden Sie in Kapitel 4.8.1, *Unicode-Zeichen*.

QCharRef

Definiert in *qstring.h*.

Diese Hilfsklasse für *QString* enthält einen Verweis auf ein Zeichen eines *QString*-Objekts. Diese Klasse wird vom *operator[]* der Klasse *QString* zurückgeliefert. Sie können sie wie die herkömmliche *QChar*-Klasse benutzen.

QCheckBox

Definiert in *qcheckbox.h*, abgeleitet von *QButton*.

Diese Klasse erzeugt einen Toggle-Button, wie er oft in Optionenmenüs benutzt wird. Er kann die Zustände »an« und »aus« annehmen und kann einen Text oder eine Pixmap enthalten. Der Unterschied zwischen *QCheckBox* und *QRadioButton* liegt in der grafischen Darstellung und im Verhalten, wenn das Widget in ein *QButtonGroup*-Objekt eingefügt wird.

Weitere Informationen finden Sie in Kapitel 3.7.3, *Buttons*.

Beispiel:

Der folgende Code erzeugt ein *QCheckBox*-Widget, wie es in Abbildung 6.7 zu sehen ist. Beim Klicken auf das Widget ändert sich der Zustand. Der Slot *setShowWindow* wird mit einem booleschen Parameter aufgerufen, der den aktuellen Zustand angibt.

```
QCheckBox *z =  
    new QCheckBox ("Show this window next time", this);  
connect (z, SIGNAL (toggled (bool)),  
        this, SLOT (setShowWindow (bool)));  
// Anfangszustand "ein"  
z->setChecked (TRUE);
```



Abbildung 6-7 *QCheckBox*

QCheckListItem

Definiert in *qlistview.h*, abgeleitet von *QListViewItem*.

Ein Objekt dieser Klasse können Sie in ein *QListView*-Objekt einfügen. Es zeichnet auf dem Bildschirm einen kleinen Button mit Beschriftung in die Tabelle ein, der den Zustand »an« oder »aus« besitzen kann, ähnlich wie *QCheckBox* oder *QRadioButton*. Der Zustand kann durch ein Anklicken mit der Maus, das Drücken der Leertaste oder mit der Methode *setOn* geändert werden. Den aktuellen Zustand können Sie mit der Methode *isOn* abfragen. Ein *QCheckListItem*-Objekt kann vom Typ *RadioButton*, *CheckBox* oder *Controller* sein. Der Typ *Controller* zeichnet ein kleines Quadrat auf dem Bildschirm, das je nach Zustand entweder ein Minus- oder ein Plus-Zeichen enthält.

QChildEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

Ein Event-Objekt dieser Klasse wird der Methode *childEvent* (definiert in *QObject*) übergeben. Diese Methode wird aufgerufen, wenn ein neues Objekt als Kindobjekt eingefügt wird oder ein existierendes Kindobjekt entfernt wird. Mit der Methode *child* können Sie im *QChildEvent*-Objekt erfragen, welches Objekt eingefügt bzw. gelöscht wurde. Die Methoden *inserted* und *removed* liefern TRUE zurück, falls dieses Objekt hinzugefügt bzw. entfernt wurde.

Weitere Informationen finden Sie in Kapitel 3.4, *Hintergrund: Event-Verarbeitung*, sowie im Abschnitt zur Klasse *QEvent*.

QClipboard

Definiert in *qclipboard.h*, abgeleitet von *QObject*.

Über diese Klasse erhält man Zugriff auf die Zwischenablage des X-Servers, die benutzt wird, um Copy&Paste zwischen verschiedenen Applikationen zu ermöglichen. In der Zwischenablage können unter anderem Texte und Bilder gespeichert werden. Der Datentyp wird durch seinen Mime-Typ festgelegt.

Weitere Informationen finden Sie in Kapitel 4.15.1, *Die Zwischenablage – QClipboard*.

Besonderheiten:

Es kann pro Applikation nur ein *QClipboard*-Objekt geben. Dieses Objekt wird automatisch von *QApplication* (bzw. *KApplication*) angelegt. Man kann über die Methode *QApplication::clipboard* darauf zugreifen.

QCloseEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

Diese Klasse speichert die Daten eines Close-Events, der vom X-Server geschickt wird, wenn ein Widget geschlossen wird. Dieser Event wird ausgelöst, wenn ein Toplevel-Widget mit dem X-Button geschlossen wird. Er kann auch mit der Methode *QWidget::close* ausgelöst werden. Dieser Event wird an die Event-Methode *QWidget::closeEvent* weitergeleitet.

Wenn Sie in einem selbst geschriebenen Widget auf diesen Event reagieren wollen, überschreiben Sie die Methode *closeEvent*. Wenn Sie die Methode *accept* des *QCloseEvent*-Objekts aufrufen, wird das Fenster anschließend mit *hide* versteckt.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, Abschnitt *showEvent*, *hideEvent*, *closeEvent*. Allgemeine Informationen finden Sie in Kapitel 3.4, *Hintergrund: Event-Verarbeitung*, sowie im Abschnitt zur Klasse *QEvent*.

QCollection

Definiert in *qcollection.h*.

Diese Klasse ist die Basisklasse für alle Container-Klassen von Qt (auch Collection-Klassen genannt) wie z.B. *QDict* und *QList*. Diese Basisklasse verwaltet nur die Einstellungen der Container-Klasse – insbesondere die Einstellung *autoDelete* – und definiert virtuelle Funktionen zum Erzeugen eines neuen Elements oder zum Löschen eines bestehenden. Die eigentliche Datenstruktur wird in den abgeleiteten Klassen definiert. Sie benutzen sie in der Regel nie selbst, sondern immer eine der abgeleiteten Klassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QColor

Definiert in *qcolor.h*.

Diese Klasse speichert einen Farbwert (im RGB-Farbmodell) ab und ermittelt einen passenden Wert für die Darstellung auf dem Bildschirm, abhängig vom Grafikmodus (Palettenindex für Grafikkarten mit 256 Farben oder eine passende Kodierung für Real-Color und True-Color mit 32.768 Farben oder mehr). Qt definiert einige Farbkonstanten vom Typ *QColor*. Eine Liste dieser Konstanten finden Sie in Kapitel 4.1, *Farben unter Qt*. *QColor* wird zum Beispiel benutzt, um eine Zeichenfarbe in *QPen* oder eine Füllfarbe in *QBrush* zu setzen.

Eine Struktur, die einen RGB-Wert mit einem geringeren Overhead abspeichert, ist *QRgb*. Dort wird keine Zuordnung der Farbinformation zu einem Farbwert der Grafikkarte vorgenommen. *QRgb* wird beispielsweise zum Zeichnen in ein *QImage*-Objekt benutzt.

Weitere Informationen zu *QColor* finden Sie in Kapitel 4.1, *Farben unter Qt*.

QColorDialog

Definiert in *qcolordialog.h*, abgeleitet von *QDialog*.

Mit dieser Klasse kann ein Dialogfenster geöffnet werden, in dem eine Farbe (anhand ihres RGB-Werts) ausgewählt werden kann. Am einfachsten benutzen Sie die statische Methode *QColorDialog::getColor*, die den Dialog öffnet und als Rückgabewert die ausgesuchte Farbe vom Typ *QColor* zurückgibt.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*. Eine Alternative für KDE-Programme ist die Klasse *KColorDialog*.

QColorDrag

Definiert in *qdragobject.h*, abgeleitet von *QStoredDrag*.

Diese Klasse dient zum Übertragen einer Farbe (Rot/Grün/Blau) per Drag&Drop in ein anderes Fenster. Nähere Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*.

QColorGroup

Definiert in *qpalette.h*.

Die Farben der Standard-Widgets von Qt sind nicht fest vorgegeben, sondern können durch die Angabe einer Palette festgelegt werden. Eine Palette der Klasse *QPalette* besteht aus drei Objekten des Typs *QColorGroup* für die jeweiligen Widget-Zustände *Active*, *Inactive* und *Disabled*.

In jedem der *QColorGroup*-Objekte sind (zur Zeit) 14 Farben festgelegt, die zum Zeichnen benutzt werden. Diese Farben – die nicht notwendigerweise unterschiedlich sein müssen – werden für verschiedene Zeichenoperationen benutzt.

Eine Liste der Farben und ihre Anwendungen finden Sie in Kapitel 4.1.8, *Die Widget-Farbpalette*.

Nahezu alle Qt-Widgets benutzen ausschließlich die hier definierten Palettenfarben. So ist gewährleistet, dass sich der Benutzer die Farben nach eigenem Geschmack zusammenstellen kann. Die angegebenen Bezeichnungen für die Paletteneinträge sind im Aufzählungstyp `QColorGroup::ColorRole` aufgelistet.

Die Palettenfarben werden intern nicht als `QColor`-Objekte, sondern als `QBrush`-Objekte gespeichert, die aus Farbe und Füllmuster bestehen. Auf diese Weise kann man zum Beispiel auch Pixmaps als allgemeinen Widget-Hintergrund definieren. Beachten Sie aber, dass nicht für alle Zeichenoperationen in den Widgets das Füllmuster benutzt wird. Zum Zeichnen von Text und Linien wird nur der Farbwert des `QBrush`-Objekts benutzt.

QComboBox

Definiert in `qcombobox.h`, abgeleitet von `QWidget`.

Ein Auswahlfeld (engl. *combo box*) ist ein Widget, das es dem Benutzer ermöglicht, aus einer Anzahl von Möglichkeiten eine auszuwählen. Es benötigt dabei nur sehr wenig Bildschirmplatz. Die momentan ausgewählte Option wird zusammen mit einem Pfeil angezeigt. Klickt der Benutzer auf den Pfeil, so erscheint ein Popup-Menü mit allen Optionen, aus denen dann die gewünschte ausgewählt werden kann. Die Optionen können Textstrings oder Pixmaps (auch gemischt in einem Auswahlfeld) sein.

Das Auswahlfeld kann zusätzlich noch editierbar sein. In diesem Fall kann der Benutzer die gewünschte Option selbst eintragen (natürlich nur Text, keine Pixmaps), falls sie in der Liste nicht vorhanden ist. Dieser neu eingegebene Eintrag kann automatisch in die Liste mit aufgenommen werden. Ein typisches Anwendungsbeispiel ist ein Eingabefeld mit so genannter »History«, also einem Speicher der zuletzt eingegebenen Einträge.

Eine Beschreibung der Klasse `QComboBox` finden Sie auch in Kapitel 3.7.5, *Auswahlelemente*.

Beispiel:

Der folgende Code erzeugt ein `QComboBox`-Widget, wie es in Abbildung 6.8 zu sehen ist.

```
// QComboBox-Widget erzeugen, nicht editierbar
QComboBox *z = new QComboBox (FALSE, this);
z->insertItem ("Germany");
z->insertItem ("Austria");
z->insertItem ("Switzerland");
z->insertItem ("Sweden");
z->insertItem ("Denmark");
z->insertItem ("France");
z->insertItem ("Poland");
```



Abbildung 6-8 QComboBox

QCommonStyle

Definiert in *qcommonstyle.h*, abgeleitet von *QStyle*.

Diese abstrakte Klasse definiert einige Gemeinsamkeiten der häufigsten *QStyle*-Klassen. Sie werden in der Regel eine der abgeleiteten Klassen verwenden, es sei denn, Sie wollen ein eigenes *QStyle*-Objekt entwerfen, um eigene Einstellungen zu erhalten. In diesem Fall kann es von Vorteil sein, von *QCommonStyle* statt von *QStyle* abzuleiten, da dann viele Methoden bereits vernünftig überschrieben sind.

QConnection

Definiert in *qconnection.h*.

Diese Klasse repräsentiert eine Verbindung zwischen einem Signal und einem Slot. Sie wird ausschließlich intern benutzt.

QConstString

Definiert in *qstring.h*, abgeleitet von *QString* (nur *private*).

Diese Klasse verwaltet einen *QString*, der aus konstanten Unicode-Daten erzeugt wird. Die Daten werden dabei nicht kopiert. Daher ist die Erzeugung etwas effizienter. Während der Lebensdauer des Objekts sollten die Daten, auf denen die Klasse basiert, nicht verändert werden, da sonst auch der Inhalt des Strings verändert wird.

Auf den String kann ausschließlich über die Methode *string* zugegriffen werden. Sie liefert eine konstante Referenz zurück.

Besonderheiten:

Obwohl *QConstString* von *QString* abgeleitet ist, ist die Oberklasse *QString* nur als *private* deklariert. Auf die Methoden von *QString* kann also nicht zurückgegriffen werden.

Die Anwendung dieser Klasse ist auf ganz seltene Fälle beschränkt, bei denen große Datenmengen, die nicht weiter verändert werden (zum Beispiel aus einer Datei), im Speicher direkt in einen *QString* umgewandelt werden sollen, so beispielsweise beim Einlesen einer Übersetzungsdatei für eine andere Landessprache. In den meisten Fällen reicht die Effizienz von *QString* völlig aus.

QString

Definiert in *qcstring.h*, abgeleitet von *QByteArray*.

Diese Klasse repräsentiert einen String in C-Notation, als Array von *char*, das durch ein Null-Byte abgeschlossen wird. *QString* kann man als einfache String-Klasse für alle Texte benutzen, die nur intern benutzt werden – die also nie auf den Bildschirm ausgegeben werden – und nur ASCII-Zeichen enthalten. Für angezeigten Text sollte *QString* benutzt werden, da nur *QString* Unicode-Texte enthalten kann. *QString* benötigt nur etwa halb so viel Speicherplatz wie *QString*.

Die Strings können eine beliebige Länge haben. Es wird zwischen einem Null-String (entspricht intern einem NULL-Zeiger) und einem leeren String (entspricht intern einem *char*-Array der Länge 1 mit Inhalt 0) unterschieden. Strings werden automatisch verlängert, wenn es nötig sein sollte. Im Gegensatz zu einfachen C-Strings muss man also nicht die maximale Länge vorher festlegen. Wird ein *QString* automatisch verlängert, muss evtl. neuer Speicher allokiert werden, in den die Daten kopiert werden. Es kann daher unter Umständen ineffizient sein, an einen sehr langen *QString* viele kurze Strings (zum Beispiel zeichenweise) anzufügen. *QStrings* bis einige tausend Zeichen sind aber in der Regel unproblematisch.

Die Klasse *QString* benutzt weit gehend implizit gemeinsame Daten. Beim Zuweisen eines Strings an einen anderen wird also nur die Referenz auf die Daten kopiert und der Referenzzähler erhöht. Fast alle bearbeitenden Methoden und Operatoren machen eine echte Kopie von den String-Daten, wenn die Daten mehr als einem String zugeordnet sind. Die einzigen Ausnahmen sind der Index-Operator *operator[]*, mit dem man auf ein einzelnes Zeichen zurückgreifen kann, und die Methode *data*, die einen Zeiger auf den reinen Text-String zurückgibt. (Beide sind in *QByteArray* definiert.)

QString besitzt eine Reihe von automatischen Type-Casts, so dass *QStrings* problemlos in Funktionsaufrufen benutzt werden können, die *const char** erwarten. Umgekehrt lässt sich eine C-Zeichenkette überall dort benutzen, wo *QString* erwartet wird.

Weitere Informationen finden Sie in Kapitel 4.7.4, *Die String-Klassen – QString und QString*.

QCursor

Definiert in *qcursor.h*.

Die Klasse *QCursor* repräsentiert die Darstellung eines Maus-Cursors auf dem Bildschirm. Die Gestalt des Maus-Cursors kann geändert werden, indem sie mit *QWidget::setCursor* für ein Widget gesetzt wird oder indem sie mit *QApplication::setOverrideCursor* für den gesamten Bildschirm festgelegt wird.

Die Gestalt eines Cursors wird in einem *QCursor*-Objekt in Form einer Cursor-Bitmap und einer Maske abgespeichert. Beide sind vom Typ *QBitmap* und müssen die gleiche Größe haben. An den Stellen, an denen die Maske den Farbwert *color1* hat, wird die Farbe der Cursor-Bitmap gezeichnet; dort, wo die Maske *color0* ist, ist der Cursor transparent. Zusätzlich kann noch die Position des *Hot-Spots* definiert werden, also der Stelle im Cursor, auf die der Cursor »zeigt«. Bei einem Pfeil-Cursor ist das beispielsweise die Pfeilspitze, bei einem Kreuz ist es der Mittelpunkt des Kreuzes. Dieser Punkt des Cursors muss nicht unbedingt innerhalb der Bitmap liegen, ist jedoch sinnvollerweise nicht weit außerhalb.

Qt enthält bereits eine Reihe von vordefinierten Cursor-Konstanten, die in der Klasse *Qt* definiert sind. Diese sollten so weit wie möglich benutzt werden. In Abbildung 6.9 sehen Sie von links nach rechts die vordefinierten Cursor *arrowCursor*, *upArrowCursor*, *crossCursor*, *waitCursor*, *ibeamCursor*, *sizeVerCursor*, *sizeHorCursor*, *sizeBDiagCursor*, *sizeFDiagCursor*, *sizeAllCursor*, *blankCursor* (nicht sichtbarer Cursor), *splitVCursor*, *splitHCursor* und *pointingHandCursor*. Die vordefinierten Cursor sehen auf unterschiedlichen X-Servern unterschiedlich aus.

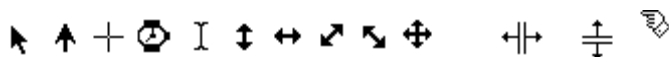


Abbildung 6-9 Vordefinierte Cursorformen

QCursor enthält außerdem die Methode *pos*, die die aktuelle Maus-Cursor-Position als *QPoint*-Objekt zurückliefert. Die Koordinaten sind dabei die absoluten Bildschirmkoordinaten, relativ zur oberen linken Bildschirmcke. Mit der Methode *setPos* kann die Maus bewegt werden.

Beispiele:

Während einer längeren Berechnung sollte der Maus-Cursor in eine Uhr verwandelt werden. Das geschieht mit dem folgenden Code:

```
QApplication::setOverrideCursor (waitCursor);
// Berechnung
...
QApplication::restoreOverrideCursor();
```

Weitere Informationen über Probleme bei blockierenden Berechnungen und ihre Lösungen finden Sie in Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*.

Mit der Methode `QWidget::setCursor` kann die Cursor-Form für ein einzelnes Widget geändert werden. Ein selbst definiertes Widget, das ein Fadenkreuz als Cursor benutzen soll, kann folgendermaßen implementiert werden:

```
class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget (QWidget *parent = 0, const char *name = 0);
    ...
};

MyWidget::MyWidget (QWidget *parent, const char *name)
    : QWidget (parent, name)
{
    setCursor (crossCursor);
    ...
}
```

QCustomEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

Dieser Event-Typ kann für eigene Erweiterungen der Qt-Event-Verarbeitung benutzt werden. Das sollte aber nur in den seltensten Fällen nötig sein. Für selbst definierte Events definieren Sie zunächst eine Zahl, die den Typ repräsentiert. Benutzen Sie dabei keinen der bereits vorgegebenen Event-Typen, die in *qevent.h* aufgelistet sind. Ein Event-Objekt erzeugen Sie mit dem Konstruktor, dem Sie zusätzlich noch einen Zeiger auf Daten übergeben können. Zum Empfangen eines Events überschreiben Sie die Methode *event* in der empfangenden Klasse, die von *QObject* abgeleitet sein muss.

Weitere Informationen finden Sie in Kapitel 3.4, Hintergrund: *Event-Verarbeitung*.

QDataPump

Definiert in *qasyncio.h*, abgeleitet von *QObject*.

Diese Klasse verbindet ein Objekt *QDataSource* mit einem Objekt *QDataSink* und startet die asynchrone Datenübertragung.

QDataPump fragt die Objekte ab, ob sie bereit sind, Daten zu senden bzw. zu empfangen. Sind beide bereit, so initiiert *QDataPump* die Datenübertragung durch einen Aufruf von *QDataSource::sendTo*. Ist eines der beiden Objekte nicht bereit, so kehrt *QDataPump* direkt in die Haupt-Event-Schleife zurück und wird

erst wieder aktiviert, wenn das Objekt mit seinem *Ready*-Signal anzeigt, dass es wieder bereit ist. Außerdem kehrt *QDataPump* nach jedem übertragenen Datenblock in die Haupt-Event-Schleife zurück, damit die anderen Aktivitäten im Programm nicht blockiert werden, aktiviert sich aber über einen Timer-Event automatisch neu.

Wenn Sie selbst eine asynchrone Datenübertragung benutzen wollen, so leiten Sie eigene Klassen von *QDataSource* und *QDataSink* ab. (Wenn Sie eine Datei als Quelle benutzen wollen, können Sie zum Beispiel die Klasse *QIODeviceSource* benutzen.) Erzeugen Sie dann ein Objekt der Klasse *QDataPump* mit den beiden Objekten für Quelle und Ziel im Konstruktor.

Weitere Informationen finden Sie in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

QDataSink

Definiert in *qasyncio.h*, abgeleitet von *QAsyncIO*.

Diese abstrakte Klasse kann asynchron Datenblöcke empfangen. Dazu wird das Objekt einer abgeleiteten Klasse als zweiter Parameter im Konstruktor der Klasse *QDataPump* eingefügt.

Um eine eigene Klasse zu erstellen, müssen Sie die virtuellen Methoden *readyToReceive*, *receive* und *eof* definieren.

Weitere Informationen finden Sie in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

QDataSource

Definiert in *qasyncio.h*, abgeleitet von *QAsyncIO*.

Diese abstrakte Klasse repräsentiert eine Quelle für eine asynchrone Datenübertragung mit Hilfe der Klasse *QDataPump*. Um eine eigene Klasse abzuleiten, müssen Sie die Methoden *readyToSend* und *sentTo* überschreiben.

Von dieser Klasse ist die Klasse *QIODeviceSource* abgeleitet, die ein beliebiges Objekt der Klasse *QIODevice* (meist *QFile*, aber auch *QBuffer* kann sinnvoll sein) als Datenquelle benutzt.

Weitere Informationen finden Sie in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

QDataStream

Definiert in *qdatastream.h*.

Diese Klasse ermöglicht es, Daten von den meisten Qt-Typen in ein *QIODevice*-Objekt zu schreiben bzw. aus ihm zu lesen. Das Datenformat ist dabei plattform-unabhängig. Insbesondere wird eine einheitliche Byte-Reihenfolge benutzt. So ist

es zum Beispiel möglich, Daten auf einer Intel-Maschine in eine Datei zu schreiben und sie auf einer HP-Maschine wieder einzulesen. Für die meisten Qt-Klassen existieren die Operatoren << und >>, um ein Objekt dieser Klasse zu schreiben oder zu lesen.

Eine ausführliche Beschreibung der Klasse finden Sie in Kapitel 4.18.4, *Stream-basierte Ein-/Ausgabe*.

Beispiel:

Das erste Codestück legt drei *int*-Zahlen, ein *QColor*-Objekt und ein *QFont*-Objekt in der Datei *x.cfg* ab:

```
// Schreiben der Datei x.cfg
QFile a ("x.cfg");
if (!a.open(IO_WriteOnly))
{
    // Fehlermeldung und Abbruch
}
QDataStream s (&a);
a << 10 << 20 << 30 << red << green << myFont;
```

Das zweite Codestück lädt diese Datei wieder:

```
// Lesen der Datei x.cfg
QFile a ("x.cfg");
if (!a.open(IO_ReadOnly))
{
    // Fehlermeldung und Abbruch
}
QDataStream s (&a);
int x1, x2, x3;
QColor c1, c2;
QFont f1;
a >> x1 >> x2 >> x3 >> c1 >> c2 >> f1;
```

Besonderheiten:

Einige Klassen wurden vor Version Qt 2.0 anders abgespeichert als vorher. Sie können mit der Methode *setVersion* festlegen, ob das alte (1) oder das neue Format (2) benutzt werden soll. Es empfiehlt sich, das Format in Form einer Zahl als ersten Wert in der Datei abzuspeichern und einzulesen (am besten vom Typ *uchar*, der in allen Formaten identisch gespeichert wird). Beim Einlesen lesen Sie zunächst diesen Wert wieder ein und setzen das Format mit *setVersion* entsprechend. So können Sie auch alte Dateien noch einlesen, die mit einer alten Qt-Version gespeichert wurden.

QDate

Definiert in *qdatetime.h*.

Ein Objekt dieser Klasse kann ein Kalenderdatum speichern und damit einige Berechnungen anstellen. Mit der statischen Methode *currentDate* kann außerdem das aktuelle Datum aus der Echtzeituhr des Rechners ermittelt werden.

Weitere Informationen finden Sie in Kapitel 4.12.1, *Die Klassen QDate, QTime und QDateTime*.

QDateTime

Definiert in *qdatetime.h*.

Diese Klasse enthält ein Objekt der Klasse *QDate* und ein Objekt der Klasse *QTime* und kann somit einen Zeitpunkt, der aus einem Datum und einer Uhrzeit besteht, speichern. Zu dieser Klasse gibt es eine Reihe neuer Berechnungsmethoden.

Weitere Informationen finden Sie in Kapitel 4.12.1, *Die Klassen QDate, QTime und QDateTime*.

QDial

Definiert in *qdial.h*, abgeleitet von *QWidget* und *QRangeControl*.

Diese Widget-Klasse erzeugt einen Drehknopf, an dem man mit Hilfe der Maus oder der Cursor-Tasten einen Wert einstellen kann, ähnlich wie bei einem *QSlider*-Widget.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungs-elemente*.

QDialog

Definiert in *qdialog.h*, abgeleitet von *QWidget*.

Diese Klasse bildet die Grundlage für Bildschirmdialoge, also für Fenster, die Elemente für Einstellungen und Auswahlmöglichkeiten bieten. Da sie von der Klasse *QWidget* abgeleitet ist, kann sie wie diese benutzt werden, um andere Widgets einzufügen.

Ein *QDialog*-Objekt kann *modal* oder *nichtmodal* sein. Ein *modales* Dialogfenster besitzt eine eigene Event-Schleife, so dass in der Zeit, in der das Fenster sichtbar ist, die anderen Fenster der Applikation nicht benutzt werden können. Maus- und Tastatur-Events werden an diese Fenster nicht mehr weitergeleitet. Der Benutzer muss den Dialog zuerst wieder schließen, bevor er mit den anderen Fenstern weiterarbeiten kann. *QDialog* wird daher besonders oft für Eingaben benutzt, die zunächst abgeschlossen werden müssen, bevor das Programm fortfahren kann. Typische Beispiele sind Abfragen der Art »Das Dokument enthält noch ungespeicherte Änderungen. Wollen Sie sie speichern?« oder zum Beispiel

ein Auswahldialog für Dateinamen. Ein *nichtmodales* Fenster verhält sich dagegen ganz ähnlich wie ein gewöhnliches *QWidget*, abgesehen von den beiden folgenden Besonderheiten:

- In einem *QDialog*-Widget kann ein *QPushButton*-Element zum Default-Button erklärt werden. Dazu ruft man beim *QPushButton*-Objekt die Methode *setDefaultButton* auf. Wird innerhalb des *QDialog*-Fensters die Eingabetaste gedrückt, so wird automatisch dieser Button betätigt. Für modale Dialog-Widgets hat weiterhin die `[ESC]`-Taste eine Sonderbedeutung: Wird sie betätigt, wird der Dialog geschlossen und der Rückgabewert ist *Rejected*.
- Ein *QDialog*-Widget ist immer ein *Toplevel*-Widget, also immer ein eigenes Fenster mit eigener Titelleiste und eigenem Rahmen, auch wenn im *parent*-Parameter ein anderes Widget angegeben ist. Das in *parent* angegebene Widget gibt das Basis-Widget des *QDialog*-Fensters an. Das Fenster wird zentriert über diesem Widget dargestellt und bleibt auch immer vor diesem Widget, kann also von diesem nicht verdeckt werden. Wird 0 im *parent*-Parameter angegeben, so wird das *QDialog*-Fenster auf dem Bildschirm zentriert.

Nichtmodale Dialoge werden wie normale *QWidget*-Fenster erstellt, mit *show* dargestellt und mit *hide* wieder versteckt. Modale Dialoge dagegen aktiviert man mit *exec*. Diese Methode kehrt erst wieder zum Aufrufer zurück, wenn der Dialog mit *done* beendet wurde, und liefert einen Wert zurück, der das Resultat des Dialogs darstellt.

Weitere Informationen zur Klasse *QDialog* finden Sie in Kapitel 3.8, *Der Dialogentwurf*. Kapitel 3.7.8, *Dialoge*, enthält eine Liste von bereits definierten Bildschirm-dialogen.

QDict

Definiert in *qdict.h*, abgeleitet von *QGDict*.

Diese Template-Klasse implementiert ein »Wörterbuch« (engl. Dictionary): Zu einem *Schlüssel* wird sehr effizient der zugehörige *Eintrag* im Wörterbuch gefunden. Schlüssel sind dabei vom Typ *QString*, die Einträge sind Zeiger auf einen beliebigen Typ, der durch die Typangabe im Template festgelegt ist.

Realisiert wird die Suche intern über eine *Hash-Tabelle*. Einem *QDict*-Objekt muss beim Erzeugen eine Größe zugewiesen werden. Diese Größe ist jedoch nicht die Maximalanzahl von Einträgen – prinzipiell ist die Anzahl der Einträge unbegrenzt –, sondern die Anzahl der Hash-Positionen. Dieser Wert sollte möglichst größer sein als die Anzahl der zu erwartenden Einträge, da ansonsten die Suche in der Hash-Tabelle ineffizienter werden kann. Weiterhin sollte dieser Wert eine Primzahl sein, da dann die Verteilung der Einträge in der Hash-Tabelle meist günstiger ist.

QDict ist eine Qt-Container-Klasse, d.h. alle Einträge sind Zeiger auf einen Datentyp. Mit der Methode *setAutoDelete* (geerbt von *QContainer*) kann man festlegen, dass diese Einträge von *QDict* verwaltet werden und mit *delete* freigegeben werden, sobald sie aus dem Wörterbuch entfernt werden. Eine Ausnahme ist dabei die Methode *take*, die die Elemente in keinem Fall freigibt.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QDictIterator

Definiert in *qdict.h*, abgeleitet von *QGDictIterator*.

Ein Objekt dieser Klasse kann genutzt werden, um alle Einträge eines *QDict*-Objekts der Reihe nach anzuschauen, zum Beispiel, um den Inhalt eines Wörterbuchs in eine Datei zu schreiben.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie in Kapitel 4.7.3, *Iterator-Objekte*.

QDir

Definiert in *qdir.h*.

Ein Objekt dieser Klasse repräsentiert ein Verzeichnis im Dateiverzeichnisbaum. Man kann die Einträge im Verzeichnis auslesen. Dabei kann man verschiedene Filter anwenden. Der verwendete Pfad kann dabei relativ zum aktuellen Verzeichnis oder absolut zum Wurzelverzeichnis angegeben werden. Die Klasse *QDir* dient hauptsächlich dazu, auf allen Plattformen eine komfortable und effiziente Schnittstelle zu bieten, um die Dateinamen und -rechte der Dateien zu lesen.

Weitere Informationen finden Sie in Kapitel 4.18.3, *Verzeichnisverwaltung*.

QDns

Definiert in *qdns.h*, abgeleitet von *QObject*, enthalten im Modul *Network*.

Diese Klasse ermöglicht einen asynchronen Zugriff auf die Namensauflösung (*Domain Name Service*). So kann das Programm die IP-Adresse zu einem symbolischen Domain-Namen herausfinden, ohne dass das Programm blockiert wird.

Nähere Informationen finden Sie in Kapitel 4.19, *Netzwerkprogrammierung*, sowie in Kapitel 4.13, *Blockierungsfreie Programme*.

QDomDocument

Definiert in *qdom.h*, abgeleitet von *QDomNode*, enthalten im Modul *XML*.

Diese Klasse repräsentiert ein gesamtes Dokument im XML-Format. Sie können eine XML-Datei einlesen und in einem *QDomDocument*-Objekt speichern lassen. Anschließend haben Sie Zugriff auf die einzelnen Elemente, die jeweils abgeleitete Klassen der zentralen Klasse *QDomNode* sind. Diese Klasse wird beispielsweise

vom KDE-Action-System benutzt, um die XML-Beschreibungsdateien für die Anordnung der Aktionen in Menü- und Werkzeugleisten festzulegen (siehe Kapitel 3.5.4, *Aufbau der Menü- und Werkzeugleiste per XML-Datei*).

QDoubleValidator

Definiert in *qvalidator.h*, abgeleitet von *QValidator*.

Diese Validator-Klasse prüft, ob eine Texteingabe (zum Beispiel in einem *QLineEdit*-Widget) eine korrekte Darstellung einer reellen Zahl zwischen einer Ober- und Untergrenze ist, wobei die Anzahl der Nachkommastellen begrenzt werden kann. Die Zahl kann ein Vorzeichen und einen Dezimalpunkt (kein Dezimalkomma!) haben, ein Exponent kann jedoch nicht angegeben werden.

Dieser Validator bietet keine *fixup*-Methode.

QDragEnterEvent

Definiert in *qevent.h*, abgeleitet von *QDragMoveEvent*.

Diese Klasse repräsentiert die Daten eines Drag-Events, der an ein Widget geschickt wird, wenn der Benutzer ein Objekt zieht (drag) und dabei in dieses Widget kommt. Diese Klasse ist mit der Klasse *QDragMoveEvent* identisch.

QDragLeaveEvent

Definiert in *qevent.h*, abgeleitet von *QDragMoveEvent*.

Diese Klasse repräsentiert die Daten eines Drag-Events, der an ein Widget geschickt wird, wenn der Benutzer ein Objekt zieht (drag) und dabei aus diesem Widget herausgeht. Diese Klasse ist mit der Klasse *QDragMoveEvent* identisch.

QDragMoveEvent

Definiert in *qevent.h*, abgeleitet von *QEvent* und *QMimeSource*.

Diese Klasse repräsentiert die Daten eines Drag-Events, der an ein Widget geschickt wird, wenn der Benutzer ein Objekt zieht (drag) und sich dabei innerhalb dieses Widgets bewegt (ähnlich einem *mouseMoveEvent*). Dazu muss das Widget allerdings auch von *QDropSite* abgeleitet sein. Wenn Sie eine eigene Klasse von *QWidget* ableiten, die Drag-Events entgegennehmen soll, überschreiben Sie diese Methode, und testen Sie, ob das Format des verschobenen Objekts von Ihrem Widget akzeptiert werden kann. Dazu können Sie die Methode *provides* benutzen, die Ihnen zurückmeldet, ob das Objekt in einem bestimmten Format vorliegt oder in dieses Format umgewandelt werden kann. Außerdem können Sie die Daten des Objekts mit der Methode *encodedData* einsehen. Da die Umwandlung in das passende Format aber rechenintensiv sein kann, sollten Sie möglichst darauf verzichten.

Wenn Ihr Widget das Objekt entgegennehmen kann, so bestätigen Sie den Event mit *accept*, ansonsten lehnen Sie ihn mit *ignore* ab (das ist die Default-Einstellung). Sie können weiterhin einen Widget-Bereich angeben, für den diese Entscheidung gelten soll, um eine aufwendige Entscheidung nicht bei jeder Mausbewegung durchführen zu müssen. Im Regelfall können Sie einfach *accept(frect())* aufrufen, wenn das Objekt überall im Widget fallengelassen werden kann.

Ausführliche Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*.

QDragObject

Definiert in *qdragobject.h*, abgeleitet von *QObject* und *QMimeSource*.

Ein Objekt dieser Klasse enthält ein Datenobjekt, das mit Drag&Drop verschoben werden soll. Für ein konkretes Datenformat müssen Sie eine eigene Klasse von *QDragObject* ableiten.

Ausführliche Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*.

QDropEvent

Definiert in *qevent.h*, abgeleitet von *QEvent* und *QMimeSource*.

Dieser Event wird an ein Widget geschickt, wenn ein Objekt mit der Maus von einem Widget aus per Drag&Drop auf dieses Widget gezogen wird und dort die Maustaste losgelassen wird (drop). Wenn Sie ein Widget entwerfen, das als Ziel einer Drag&Drop-Operation dienen soll, so überschreiben Sie diese Methode und verarbeiten in ihr die Daten, die Sie mit der Methode *encodedData* ermitteln lassen können. Um den Event *dropEvent* überhaupt empfangen zu können, muss die Widget-Klasse auch von *QDropSite* abgeleitet sein.

Ausführliche Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*.

QDropSite

Definiert in *qdropsite.h*.

Diese Basisklasse erweitert ein Widget um die Fähigkeit, Events von Drag&Drop-Operationen zu empfangen. Wenn Sie selbst ein solches Widget entwerfen wollen, müssen Sie eine neue Klasse definieren, die von *QWidget* und *QDropSite* abgeleitet ist. Beachten Sie, dass bei der Mehrfachvererbung die Klasse *QWidget* (bzw. eine davon abgeleitete Klasse) vor der Klasse *QDropSite* stehen muss (siehe auch Kapitel 3.1.3, *Selbst definierte Klassen von QObject ableiten*), damit *moc* diese Klasse korrekt erkennt.

Ausführliche Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*.

QEvent

Definiert in *qevent.h*.

Die Klasse *QEvent* stellt Informationen über einen internen Qt-Event zur Verfügung. Alle Events, die Qt versendet, enthalten ein von *QEvent* abgeleitetes Objekt, das die näheren Umstände der Auslösung angibt. Allen Event-Objekten ist der Typ gemeinsam, der in einer *int*-Variable gespeichert wird. Dieser Wert wird im Konstruktor von *QEvent* gesetzt und kann mit der Klasse *QEvent::type* abgefragt werden. Qt definiert bereits eine Reihe von Konstanten für die definierten Event-Typen. Eine Liste aller Konstanten finden Sie in der Datei *qevent.h*.

Die abgeleiteten Klassen können weitere Informationen enthalten, z. B. die Mausposition oder die gedrückte Taste, und können Methoden anbieten, mit denen man z. B. den Event akzeptieren oder zurückweisen kann.

Es ist möglich, sich eigene Events zu definieren. Im einfachsten Fall erzeugt man einen neuen Event, indem man ein Objekt der Klasse *QEvent* mit einem selbst definierten Event-Typ erzeugt. Der Event-Typ ist ein *int*-Wert und sollte größer als 1000 sein, damit keine Kollision mit den vordefinierten Qt-Event-Typen auftreten kann. Alternativ können Sie auch eine eigene Unterklasse ableiten, die den Event-Typ setzt und weitere Methoden enthalten kann. Einen solchen Event können Sie dann z. B. mit *QApplication::sendEvent* verschicken und in der Methode *QObject::event* einer selbst definierten Klasse abfangen.

Nähere Informationen zur Event-Verarbeitung finden Sie in Kapitel 3.4, *Hintergrund: Event-Verarbeitung*.

QFile

Definiert in *qfile.h*, abgeleitet von *QIODevice*.

Ein Objekt dieser Klasse repräsentiert eine Datei im Dateisystem. Die Klasse bietet Methoden zum Öffnen und Schließen sowie zum Schreiben und Lesen der Datei. Ebenso existieren statische Methoden zum Testen, ob eine Datei mit einem bestimmten Namen existiert sowie zum Löschen einer Datei.

QFile ist von der abstrakten Basisklasse *QIODevice* abgeleitet, die ein Ein-/Ausgabegerät darstellt. In *QFile* sind die Methoden zum Lesen und Schreiben von Blöcken oder einzelnen Zeichen mit konkretem Code zum Lesen und Schreiben in einer Datei überschrieben.

Außerdem enthält die Klasse *QFile* die statische Methode *remove*, mit der Dateien aus dem Dateisystem gelöscht werden können, sofern die Zugriffsrechte auf die Datei und das Verzeichnis das erlauben.

Weitere Informationen finden Sie in Kapitel 4.18, *Dateizugriffe*.

Beispiel:

Der folgende Code kopiert eine Datei *a.txt* in die Datei *b.txt*. Dazu liest er Blöcke von maximal 1024 Byte ein und speichert sie in der anderen Datei.

```
QFile a ("a.txt");
QFile b ("b.txt");
if (b.exists ())
{
    // Datei b.txt existiert bereits.
    // Anwender fragen, ob die Datei
    // überschrieben werden soll.
    // Falls nicht, abbrechen.
}

if (!a.open (IO_ReadOnly))
{
    // Datei a.txt kann nicht geöffnet werden.
    // Fehlermeldung und Abbruch
}
if (!b.open (IO_WriteOnly))
{
    // Datei b.txt kann nicht geöffnet werden.
    // Fehlermeldung und Abbruch
}

char buffer [1024];
while (!a.eof())
{
    int blocksize;
    blocksize = a.readBlock (buffer, 1024);
    b.writeBlock (buffer, blocksize);
}
a.close();
b.close();
```

QFileDialog

Definiert in *qfiledialog.h*, abgeleitet von *QDialog*.

Diese Klasse öffnet ein Dialog-Fenster auf dem Bildschirm, das in Abbildung 6.10 zu sehen ist. In diesem Fenster kann der Benutzer eine Datei des Dateisystems auswählen, entweder durch Auswählen eines Verzeichnisses und einer Datei mit der Maus oder durch Eintippen des Dateinamens über die Tastatur.

Die gängigste Art, diese Klasse zu benutzen, besteht darin, eine der statischen Methoden *getOpenFileName*, *getOpenFileNames*, *getSaveFileName* oder *getExistingDirectory* aufzurufen. Diese Methoden erzeugen ein modales Dialogobjekt und kehren erst zum Aufrufer zurück, wenn der Benutzer die Auswahl beendet hat.

KDE-Programme sollten nicht diesen Dialog benutzen, sondern die Dialogklasse *KFile*, um ein einheitliches Look&Feel zu bieten. *KFile* bietet weiterhin die Möglichkeit, auch Dateien aus dem Internet per HTTP- oder FTP-Protokoll auszuwählen.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

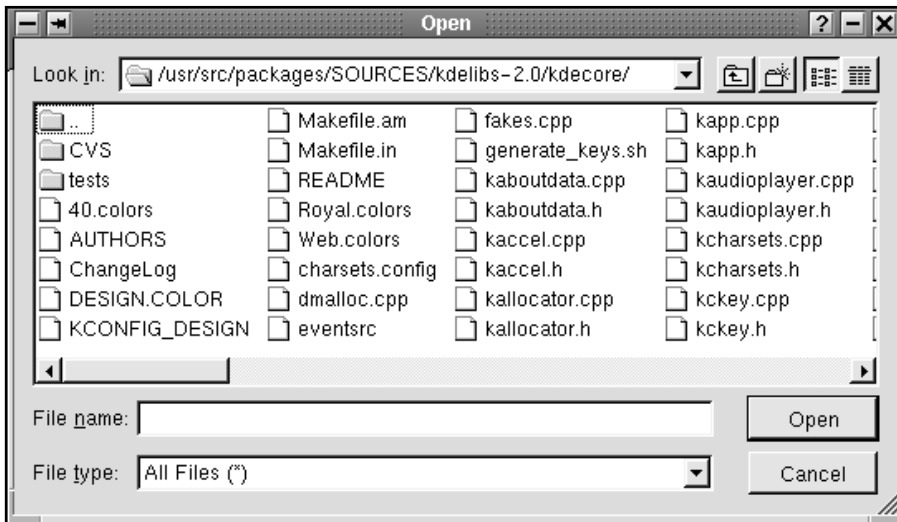


Abbildung 6-10 QFileDialog

QFileIconProvider

Definiert in *qfiledialog.h*, abgeleitet von *QObject*.

Diese Klasse ordnet einer Datei (gegeben durch ein *QFileInfo*-Objekt) ein Icon zu. Ein *QFileIconProvider*-Objekt kann benutzt werden, um in einem *QFileDialog*-Fenster die vor dem Dateinamen angezeigten Icons festzulegen.

QFileInfo

Definiert in *qfileinfo.h*.

Diese Klasse bietet Zugriff auf den Modus und die Eigenschaften einer Datei (z.B. Größe, Zeitpunkt der letzten Veränderung, Zugriffsrechte) sowie Methoden zur Analyse von Dateinamen, wie z.B. für das Aufspalten in Dateiname und Erweiterung. Diese Klasse ist dabei plattformunabhängig, arbeitet also auf allen Betriebssystemen, die Qt unterstützen.

Ein *QFileInfo*-Objekt kann mit einem der Konstruktoren aus einem Dateinamen-String oder aus einem *QFile*-Objekt erzeugt werden.

Weitere Informationen finden Sie in Kapitel 4.18.2, *Dateiinformatoren*.

QFilePreview

Definiert in *qfiledialog.h*.

Diese abstrakte Basisklasse ermöglicht die Erweiterung von *QFileDialog*, so dass beim Auswählen einer Datei eine Vorschau der Datei in einem eigenen Fenster angezeigt wird. Wenn Sie eine solche Vorschau realisieren wollen, müssen Sie eine eigene Klasse von *QWidget* (oder einer Unterklasse) und *QFilePreview* ableiten. Überschreiben Sie die abstrakte Methode *QFilePreview::previewUrl (const QUrl &)*. Diese Methode sollte die übergebene Datei laden und die Vorschau im Widget anzeigen.

QFocusData

Definiert in *qfocusdata.h*.

Diese Klasse wird intern in *QWidget* verwendet. Ein Objekt dieser Klasse verwaltet eine Liste von Widgets, die den Tastaturfokus bekommen können. Die Methode *QWidget::focusData* liefert das Objekt eines Widgets zurück. Mit den Methoden *next* und *prev* kann man sich das Widget angeben lassen, das in der Liste der Widgets als nächstes bzw. vorhergehendes Widget den Tastaturfokus bekommt.

Weitere Informationen zum Tastaturfokus finden Sie in Kapitel 3.2.3, *Die wichtigsten Widget-Eigenschaften*, Abschnitt *Tastaturfokus*. Einige Informationen zum Wechseln des Fokus mit den Tasten  und  +  können Sie in Kapitel 4.4.1 *Event-Methoden* im Abschnitt *keyPressEvent*, *keyReleaseEvent* nachlesen.

QFocusEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen eines Fokus-Events. Dieses Objekt wird an die Methode *QWidget::focusInEvent* und *QWidget::focusOutEvent* übergeben. Die Methode *gotFocus* liefert TRUE, wenn das Widget den Fokus erhalten hat (*focusInEvent*), *lostFocus* liefert TRUE, wenn es den Fokus verloren hat (*focusOutEvent*).

QFont

Definiert in *qfont.h*.

Die *QFont*-Klasse legt den Zeichensatz (Font) fest, der zum Zeichnen von Text benutzt wird. Der Zeichensatz wird dabei durch eine Reihe von Attributen spezifiziert: Schriftfamilie, Schriftgröße, Schriftstärke (normal oder fett) und Schriftwinkel (normal oder kursiv), darstellbare Zeichen, unterstrichen und durchgestrichen. Ein Objekt der Klasse *QFont* kann jede Kombination von Attributen annehmen. Wird mit *QFont* der Font zum Zeichnen festgelegt, so versucht Qt, einen Zeichensatz zu finden, der den geforderten Attributen am ehesten entspricht. Je nachdem, ob eine solche Schriftart installiert ist oder nicht, kann das Ergebnis auf verschiedenen Systemen etwas unterschiedlich ausfallen. Ein *QFont*-

Objekt kann als Argument der Methode *QPainter::setFont* benutzt werden, um Text in einem bestimmten Zeichensatz darzustellen. Mit der Methode *QWidget::setFont* kann man den Standardzeichensatz für ein Widget einstellen. *QApplication::setFont* setzt den Standardzeichensatz für alle neu erzeugten Widgets.

QFontDatabase

Definiert in *qfontdatabase.h*.

Diese Klasse bietet Informationen über alle auf dem System installierten Schriftarten. Beispielsweise liefert die Methode *families* eine Liste (*QStringList*) aller Schriftfamilien zurück. Diese Klasse wird meist in Dialogen zur Auswahl einer Schriftart verwendet (*QFontDialog*, *KFontDialog*), um eine Liste aller Schriftarten zu erstellen.

QFontDialog

Definiert in *qfontdialog.h*, abgeleitet von *QDialog*.

Diese Klasse implementiert ein Dialogfenster für die Auswahl eines Zeichensatzes. In den meisten Fällen werden Sie die statische Methode *QFontDialog::getFont* benutzen. Diese Methode öffnet das Dialogfenster und kehrt erst dann zum Aufrufer zurück, wenn der Anwender den Zeichensatz festgelegt hat. Der Rückgabewert ist der ausgewählte Zeichensatz.

Sie können alternativ für KDE-Programme auch die Klasse *KFontDialog* aus der KDE-Bibliothek *kdeui* benutzen.

QFontInfo

Definiert in *qfontinfo.h*.

Die Klasse *QFontInfo* kann benutzt werden, um Informationen über den genauen Font zu bekommen, der benutzt wird. Falls die Attribute eines in *QFont* spezifizierten Zeichensatzes nicht genau erfüllt werden können (*QFont::exactMatch* liefert FALSE), kann man zu diesem *QFont*-Objekt ein *QFontInfo*-Objekt erzeugen, das die exakten Attribute des verwendeten Bildschirm-Fonts liefert.

QFontMetrics

Definiert in *qfontmetrics.h*.

Mit dieser Klasse kann man zu einem *QFont*-Objekt Informationen über Höhe und Breite der Zeichen ermitteln lassen.

QFrame

Definiert in *qframe.h*, abgeleitet von *QWidget*.

Diese Widget-Klasse stellt ein Rechteck mit einem Rahmen dar. Die Art des Rahmens kann auf vielfältige Weise festgelegt sein. Durch Schattierungseffekte kann

man den Effekt eines hervortretenden bzw. abgesenkten Bereichs erzielen. Wie bei allen Widgets kann man dem Frame-Widget Unter-Widgets zuordnen, die innerhalb des Frames dargestellt werden. So kann man mit einem Frame-Widget andere Widgets wie z. B. Buttons oder Labels optisch zu einer Gruppe zusammenfassen.

QFrame ist auch die Basisklasse für viele andere Qt-Widgets, die einen Rahmen besitzen. Um eine eigene Widget-Klasse zu erzeugen, leiten Sie sie von *QFrame* ab und überschreiben die virtuelle Methode *drawContents*.

Weitere Informationen finden Sie in Kapitel 3.7.7, *Verwaltungselemente*.

QFtp

Definiert in *qftp.h*, abgeleitet von *QNetworkProtocol*, enthalten im Modul *Network*.

Diese Klasse ermöglicht den Zugriff auf Dateien über das FTP-Protokoll. Nähere Informationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*.

QGL

Definiert in *qgl.h*, enthalten im Modul *OpenGL*.

Diese Klasse dient nur als Namensraum und definiert einige Konstanten für das OpenGL-Modul. Siehe auch die Klasse *QGLWidget*.

QGLWidget

Definiert in *qgl.h*, abgeleitet von *QWidget* und *QGL*, enthalten in Modul *OpenGL*.

Diese Klasse ermöglicht ein Einbinden von OpenGL-Fenstern in Qt-Widgets. Auf diese Weise lassen sich 3D-Darstellungen von OpenGL sehr einfach mit einer ansprechenden Oberfläche von Qt ausstatten. Siehe auch Kapitel 4.5, *Flimmerfreie Darstellung*.

QGrid

Definiert in *qgrid.h*, abgeleitet von *QWidget*.

Ein Widget dieser Klasse ordnet die eingefügten Unter-Widgets tabellenartig in Spalten und Zeilen an. Der Platz, den die einzelnen Unter-Widgets dabei zugewiesen bekommen, ergibt sich aus den von ihnen zurückgelieferten Werten in *sizeHint* und *sizePolicy*.

Zwischen den Widgets wird ein fünf Pixel breiter Streifen Platz gelassen. Im Konstruktor legt man die Anzahl der Spalten fest. Jedes Widget, das das *QGrid*-Widget als Vater-Widget zugeordnet bekommt, wird dann der Reihe nach in die Tabellenaufteilung eingefügt. Mit der Methode *skip* kann man einen Tabellenplatz frei lassen.

Genauere Informationen über die Anwendung von *QGrid* finden Sie in Kapitel 3.6.3, *Einfache Layouts mit QHBox, QVBox und QGrid*.

QGridLayout

Definiert in *qlayout.h*, abgeleitet von *QLayout*.

Diese Klasse übernimmt die Verwaltung der Größe und Position von mehreren Unter-Widgets innerhalb eines Widgets. Die Elemente werden dabei tabellenartig angeordnet. Dazu wird im Konstruktor die Anzahl der Zeilen und Spalten angegeben. Mit der Methode *addWidget* kann ein Widget an einer bestimmten Stelle der Tabelle eingefügt werden. Mit der Methode *addMultiCellWidget* kann ein Widget sich auch über mehrere Zeilen und/oder Spalten erstrecken. Die Klasse besitzt weitere Methoden, um die Breite bzw. Höhe der Spalten und Zeilen festzulegen.

Eine sehr detaillierte Beschreibung der Vorgehensweise finden Sie in Kapitel 3.6.4, *Die Layout-Klassen QBoxLayout und QGridLayout*.

QGroupBox

Definiert in *qgroupbox.h*, abgeleitet von *QFrame*.

Ein Widget dieser Klasse zeichnet einen Rahmen mit einer Titelzeile auf den Bildschirm. Es wird meist benutzt, um eine Reihe von Einstellungs-Widgets in einem Fenster unter einem Namen grafisch zu gruppieren.

Weitere Informationen finden Sie in Kapitel 3.7.7, *Verwaltungselemente*.

QGuardedPtr

Definiert in *qguardedptr.h*.

Ein Objekt dieser Template-Klasse kann wie ein Zeiger benutzt werden. Der Datentyp, auf den dieser Zeiger zeigt, kann eine beliebige von *QObject* abgeleitete Klasse sein.

QGuardedPtr prüft nach, ob das Objekt, auf das es zeigt, noch vorhanden ist oder ob es bereits mit *delete* gelöscht wurde. Es benutzt dazu das Signal *destroyed*, das jede *QObject*-Instanz im Destruktor aussendet. So können Sie vermeiden, dass Sie auf ein bereits gelöscht Objekt zugreifen, was einen Laufzeitfehler zur Folge hätte.

QHBox

Definiert in *qhbox.h*, abgeleitet von *QFrame*.

Dieses Widget ordnet alle Unter-Widgets horizontal in einer Reihe von links nach rechts an. Der benötigte Platz wird dabei aus den Werten ermittelt, die die Unter-Widgets in den Methoden *sizeHint* und *sizePolicy* zurückliefern.

Da *QHBoxLayout* von *QFrame* abgeleitet ist, kann das Widget mit einem Rahmen versehen werden. Standardmäßig wird kein Rahmen dargestellt. Zwischen den Unter-Widgets wird ein Streifen in der Breite des Rahmens frei gelassen. Ohne Rahmen wird also kein Platz zwischen den Widgets gelassen.

Weitere Informationen finden Sie in Kapitel 3.6.3, *Einfache Layouts mit QHBoxLayout, QVBoxLayout und QGridLayout*.

QHBoxLayout

Definiert in *qlayout.h*, abgeleitet von *QBoxLayout*.

Diese Klasse übernimmt die Verwaltung der Größe und Position von mehreren Unter-Widgets innerhalb eines Widgets. Die Elemente werden dabei in einer Reihe nebeneinander von links nach rechts angeordnet. Mit der Methode *addWidget* kann ein Widget in das Layout eingefügt werden. Mit *addLayout* kann ein anderes Layout-Objekt eingefügt werden.

QHBoxLayout ist von *QBoxLayout* abgeleitet und unterscheidet sich von der Basis-klasse nur dadurch, dass im Konstruktor keine Angabe über die Ausrichtung des Layouts erfolgt, da implizit die Ausrichtung *LeftToRight* benutzt wird.

Eine sehr detaillierte Beschreibung der Vorgehensweise finden Sie in Kapitel 3.6.4, *Die Layout-Klassen QBoxLayout und QGridLayout*.

QButtonGroup

Definiert in *qbuttongroup.h*, abgeleitet von *QButtonGroup*.

Diese Widget-Klasse ergänzt die Klasse *QButtonGroup* um ein automatisches Layout der eingefügten Unter-Widgets. Sie kombiniert also die Funktionalität von *QButtonGroup* und *QHBoxLayout*.

Die eingefügten Unter-Widgets werden von links nach rechts eingefügt.

QHeader

Definiert in *qheader.h*, abgeleitet von *QWidget*.

Diese Widget-Klasse zeichnet die Kopfzeile einer Tabelle auf den Bildschirm, wie sie von *QListView* benutzt wird. In dieser Kopfzeile sind verschiedene Einträge vorhanden, deren Größe und Reihenfolge mit der Maus verändert werden kann. In der Regel benötigen Sie diese Klasse nie selbst.

QGroupBox

Definiert in *qgroupbox.h*, abgeleitet von *QGroupBox*.

Diese Widget-Klasse ergänzt die Klasse *QGroupBox* um ein automatisches Layout der eingefügten Unter-Widgets. Sie kombiniert also die Funktionalität von *QGroupBox* und *QHBoxLayout*. Alle eingefügten Unter-Widgets der Klasse *QRadioButton*

werden damit automatisch zu sich ausschließenden Widgets, von denen maximal ein Widget im Zustand »an« sein kann.

Die eingefügten Unter-Widgets werden von links nach rechts eingefügt.

QHideEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

Diese Klasse speichert die Daten eines Hide-Events, der vom X-Server geschickt wird, wenn ein Widget versteckt wird. Dieser Event wird ausgelöst, wenn ein Toplevel-Widget minimiert wird. Er kann auch mit der Methode *QWidget::hide* ausgelöst werden. Dieser Event wird an die Event-Methode *QWidget::hideEvent* weitergeleitet.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, Abschnitt *showEvent*, *hideEvent*, *closeEvent*. Allgemeine Informationen finden Sie in Kapitel 3.4, *Hintergrund: Event-Verarbeitung*, sowie im Abschnitt zur Klasse *QEvent*.

QIconDrag

Definiert in *qiconview.h*, abgeleitet von *QDragObject*, enthalten im Modul *Iconview*.

Ein Objekt dieser Klasse wird benutzt, um ein Icon per Drag&Drop zu verschieben. Insbesondere wird es in der Widget-Klasse *QIconView* benutzt. Nähere Informationen finden Sie in Kapitel 4.15.2, *Drag&Drop*, sowie im Abschnitt zur Klasse *QIconView*.

QIconSet

Definiert in *qiconset.h*.

Ein Objekt dieser Klasse kann ein Icon – also ein kleines Bild – in zwei verschiedenen Größen (*Large* und *Small*) und drei Zuständen (*Active*, *Normal* und *Disabled*) speichern. Es reicht dabei in der Regel, ein einzelnes Icon abzulegen. Die anderen fünf Icons werden automatisch bei Bedarf generiert, indem das vorhandene Icon vergrößert oder verkleinert und für den Zustand *Disabled* in Graustufen umgewandelt wird. Falls die Qualität der berechneten Icons nicht ausreichend ist, kann jedes der sechs Icons auch einzeln in das Objekt eingefügt werden. Ein solches *QIconSet*-Objekt kann beispielsweise in der Werkzeugleiste eingesetzt werden.

QImage

Definiert in *qimage.h*.

In einem Objekt dieser Klasse kann ein rechteckiges Teilbild im Hauptspeicher der Applikation abgelegt werden. Die Farbtiefe dieses Bildes kann ein, acht oder 32 Bit pro Pixel betragen. Die Höhe und Breite des Bildes ist auf 32.767 Pixel

beschränkt. Über den so genannten Alpha-Kanal kann jedem Pixel eine Transparenz zugewiesen werden.

Diese Klasse bietet sehr effiziente Zugriffe auf einzelne Pixel, ist jedoch zum Zeichnen von komplexen Grafikprimitiven nur schlecht geeignet. Die Darstellung auf dem Bildschirm ist langsamer als bei *QPixmap*.

Genaue Informationen über die Anwendung von *QImage* finden Sie in Kapitel 4.3.1, *QImage*. Kapitel 4.3.3, *Effizienzbetrachtungen*, gibt einige Hinweise zu den Einsatzgebieten von *QImage* und *QPixmap*. Weitere Informationen über die Anwendung der Transparenz finden Sie in Kapitel 4.3.4, *Transparenz in Teilbildern*.

QImageDrag

Definiert in *qdragobject.h*, abgeleitet von *QDragObject*.

In einem Objekt dieser Klasse kann ein *QImage*-Objekt abgelegt werden, um es per Drag&Drop in ein anderes Fenster zu ziehen oder um es in die Zwischenablage zu kopieren.

Nähere Informationen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag&Drop*.

QImageIO

Definiert in *qimage.h*.

Diese Klasse implementiert das Laden und Speichern von Bildern in verschiedenen Dateiformaten. Über die statische Methode *defineIOHandler* kann man eigene Funktionen hinzufügen, die neue Dateiformate lesen und schreiben können. Diese werden dann automatisch benutzt, sobald eine Bilddatei dieses Formats geöffnet oder gespeichert werden soll.

QInputDialog

Definiert in *qinputdialog.h*, abgeleitet von *QDialog*.

Diese Klasse bietet eine Reihe von statischen Methoden, mit denen einfache Dialoge zur Eingabe von einer einfachen Zeile Text, einer *int*- oder *double*-Zahl oder zur Auswahl aus einer Liste von Optionen erzeugt werden. Für komplexere Dialoge mit mehr Eingaben ist es meist nötig, einen eigenen Dialog zu entwerfen (siehe Kapitel 3.8, *Der Dialogentwurf*). Als Hilfsmittel für die Erstellung eines Dialogs ist das Programm Qt Designer von Trolltech sehr gut geeignet (siehe Kapitel 5.4, *Qt Designer*).

QIntCache

Definiert in *qintcache.h*, abgeleitet von *QGCache*.

Dieses Template implementiert einen Cache-Speicher für einen beliebigen Datentyp. Ein Cache wird verwendet, um bereits geladene Daten abzuspeichern, damit sie bei einem erneuten Zugriff schneller erreicht werden können.

Die Elemente des Cache müssen mit *new* auf dem Heap angelegt worden sein. Das Löschen der Elemente übernimmt das *QCache*-Objekt. Die Schlüssel zum Auffinden eines Cache-Elements sind Zahlen vom Typ *int*. Jedem Element wird ein Kostenwert zugeordnet (meist ist das der verbrauchte Speicherplatz, es kann aber auch eine beliebige andere positive ganze Zahl sein). Sobald die Summe der Kostenwerte einen Maximalwert überschreitet, werden die Elemente, die am längsten nicht benutzt worden sind, aus dem Cache gelöscht. Intern wird für die Suche nach einem Element im Cache ein Hash-Verfahren benutzt, so dass das gesuchte Element in der Regel sofort gefunden wird.

Weitere Informationen über *QIntCache* finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QIntCacheIterator

Definiert in *qintcache.h*, abgeleitet von *QGCacheIterator*.

Diese Klasse realisiert ein Iterator-Objekt für ein *QIntCache*-Objekt, um alle Elemente des Cache der Reihe nach durchzugehen. (Siehe auch *QIntCache* sowie Kapitel 4.7.2, *Container-Klassen*, und Kapitel 4.7.3, *Iterator-Objekte*.)

QIntDict

Definiert in *qintdict.h*, abgeleitet von *QGDict*.

Diese Template-Klasse implementiert ein »Wörterbuch« (engl. Dictionary): Zu einem *Schlüssel* wird sehr effizient der zugehörige *Eintrag* im Wörterbuch gefunden. Die Schlüssel sind dabei Zahlen vom Typ *int*, die Einträge sind Zeiger auf einen beliebigen Typ, der durch die Typangabe im Template festgelegt ist.

Realisiert wird die Suche intern über eine *Hash-Tabelle*. Einem *QIntDict*-Objekt muss beim Erzeugen eine Größe zugewiesen werden. Diese Größe ist jedoch nicht die Maximalanzahl von Einträgen – prinzipiell ist die Anzahl der Einträge unbegrenzt –, sondern die Anzahl der Hash-Positionen. Dieser Wert sollte möglichst größer sein als die Anzahl der zu erwartenden Einträge, da ansonsten die Suche in der Hash-Tabelle ineffizienter werden kann. Weiterhin sollte dieser Wert eine Primzahl sein, da dann die Verteilung der Einträge in der Hash-Tabelle meist günstiger ist.

QIntDict ist eine Qt-Container-Klasse, d.h. alle Einträge sind Zeiger auf einen Datentypen. Mit der Methode *setAutoDelete* (geerbt von *QContainer*) kann man

festlegen, dass diese Einträge von *QIntDict* verwaltet werden und mit *delete* freigegeben werden, sobald sie aus dem Wörterbuch entfernt werden. Eine Ausnahme ist dabei die Methode *take*, die die Elemente in keinem Fall freigibt.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QIntDictIterator

Definiert in *qintdict.h*, abgeleitet von *QGDictIterator*.

Ein Objekt dieser Klasse kann genutzt werden, um alle Einträge eines *QIntDict*-Objekts der Reihe nach anzuschauen, zum Beispiel, um den Inhalt einer Hash-Tabelle in eine Datei zu schreiben.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QIntValidator

Definiert in *qvalidator.h*, abgeleitet von *QValidator*.

Diese Validator-Klasse prüft, ob eine Texteingabe (zum Beispiel in einem *QLineEdit*-Widget) eine korrekte Darstellung einer ganzen Zahl zwischen einer Ober- und einer Untergrenze ist. Die Zahl kann ein Vorzeichen haben, darf jedoch nur in dezimaler Darstellung eingegeben werden.

Dieser Validator bietet keine *fixup*-Methode.

QIODevice

Definiert in *qiodevice.h*.

Diese Klasse ist die abstrakte Basisklasse für ein allgemeines Ein-/Ausgabe-Gerät. Sie enthält virtuelle Methoden zum Öffnen und Schließen des Geräts, zum Lesen und Schreiben von Blöcken und von einzelnen Zeichen und zum Abfragen und Setzen der aktuellen Schreib-/Lese-Position.

Qt enthält bereits drei von *QIODevice* abgeleitete Klassen: *QFile* implementiert den Zugriff auf eine Datei im Dateisystem, *QBuffer* implementiert den Zugriff auf einen Speicherbereich wie auf eine Datei und *QSocket* ermöglicht das Schreiben und Lesen in einen Socket.

Wenn Sie eine eigene Klasse von *QIODevice* ableiten wollen, müssen Sie mindestens die rein virtuellen Methoden *open*, *close*, *flush*, *size*, *readBlock*, *writeBlock*, *getch*, *putch* und *ungetch* überschreiben. In der Regel müssen Sie auch die Methode *at* sowie die Methode *atEnd* überschreiben. Außerdem sollten Sie im Konstruktor die Geräteattribute mit *setFlags* setzen.

Ein Objekt der Klasse *QIODevice* (oder einer abgeleiteten Klasse) kann auch als Quelle für eine asynchrone Datenübertragung dienen. Für weitere Informationen siehe *QIODeviceSource* sowie Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

QIODeviceSource

Definiert in *qasyncio.h*, abgeleitet von *QDataSource*.

Ein Objekt dieser Klasse kann als Quelle für eine asynchrone Datenübertragung mit *QDataPump* dienen. Geben Sie dazu im Konstruktor ein *QIODevice*-Objekt an (in der Regel ein *QFile*-Objekt, aber auch ein *QBuffer*-Objekt oder eine andere, von *QIODevice* abgeleitete Klasse).

Genaue Informationen zur Anwendung finden Sie in Kapitel 4.13.3, *Datenübertragung im Hintergrund mit QDataPump*.

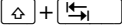
QKeyEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen eines Tastatur-Events. In diesem Objekt wird die Taste gespeichert, die den Event ausgelöst hat, sowie der ASCII-Code des Zeichens, der dadurch erzeugt wurde. Ein Objekt dieser Klasse wird den Methoden *QWidget::keyPressEvent* und *QWidget::keyReleaseEvent* übergeben.

Genauere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *keyPressEvent*, *keyReleaseEvent*.

Besonderheiten:

Die Tasten  und  werden im Allgemeinen nicht an die Methoden *keyPressEvent* und *keyReleaseEvent* weitergereicht, da sie bereits in der Methode *QWidget::event* dazu benutzt werden, den Tastaturfokus auf das nächste bzw. vorhergehende Widget zu übertragen. Wie Sie trotzdem diese Events erhalten können, wird ebenfalls in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *keyPressEvent*, *keyReleaseEvent* beschrieben.

Die Klasse *QKeyEvent* besitzt die beiden Methoden *accept* und *ignore*, mit denen Sie ein Flag im *QKeyEvent*-Objekt setzen bzw. löschen können. Wenn Sie *ignore* aufrufen, so wird nach der Rückkehr aus der Methode *keyPressEvent* (oder *keyReleaseEvent*) der Tastatur-Event an das Vater-Widget weitergeleitet und dort bearbeitet.

QLabel

Definiert in *qlabel.h*, abgeleitet von *QFrame*.

In einem *QLabel*-Widget kann ein Text (*QString*), ein Bild (*QPixmap*) oder ein bewegtes Bild (*QMovie*) auf dem Bildschirm dargestellt werden. Dieses Widget reagiert dabei nicht auf Maus- oder Tastatur-Events. *QLabel* wird in erster Linie dazu benutzt, einen Zustand anzuzeigen oder einen erklärenden Text auszugeben.

QLabel ist von *QFrame* abgeleitet und kann daher auch einen Rahmen um das darzustellende Element anzeigen.

QLabel kann auch Text im RichText-Format (einer Untermenge von HTML) darstellen. Beispiele hierfür finden Sie im Einführungsbeispiel in Kapitel 2.2, *Das erste Qt-Programm*, sowie in der *Übung 2.1* in Kapitel 2.2.5 (siehe Abbildung 6.12).

Weitere Informationen finden Sie in Kapitel 3.7.1, *Statische Elemente*.

Beispiel:

Die folgende Zeile erzeugt ein *QLabel*-Element, wie es in Abbildung 6.11 dargestellt ist.

```
QLabel *z = new QLabel ("This is a QLabel with\n"  
                        "two lines of text", this);
```

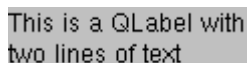


Abbildung 6-11 *QLabel* mit zwei Zeilen Text



Abbildung 6-12 *QLabel* mit aufwendigem RichText-Inhalt (siehe Übung 2.1)

QLayout

Definiert in *qabstractlayout.h*, abgeleitet von *QObject* und *QLayoutItem*.

Diese Basisklasse implementiert eine Reihe von Methoden für die automatische Anordnung von Unter-Widgets in einem Fenster. Von dieser Klasse sind *QBoxLayout* und *QGridLayout* abgeleitet.

Genauere Informationen finden Sie in Kapitel 3.6.4, *Die Layout-Klassen QBoxLayout und QGridLayout*.

QLayoutItem

Definiert in *qabstractlayout.h*.

Diese interne Klasse bildet die Basisklasse für alle Objekte, die ein Element verwalten, das in ein Layout eingefügt werden kann: Widgets (*QWidgetItem*), Layouts (*QLayout*) und Platzhalter (*QSpacerItem*).

QLCDNumber

Definiert in *qlcdnumber*, abgeleitet von *QFrame*.

Ein *QLCDNumber*-Widget stellt eine Zahl in Form einer 7-Segment-Anzeige dar. Die Darstellung kann dezimal, hexadezimal, oktal oder binär erfolgen. Das Widget kann ganze Zahlen (Typ *int*) oder auch reelle Zahlen (Typ *double*) darstellen.

Weitere Informationen finden Sie in Kapitel 3.7.2, *Anzeigeelemente*.

Beispiel:

Die folgende Zeile erzeugt ein *QLCDNumber*-Widget, wie es in Abbildung 6.13 zu sehen ist.

```
QLCDNumber *z = new QLCDNumber (4, this);  
z->display (42);
```



Abbildung 6-13 *QLCDNumber*

QLineEdit

Definiert in *qlineedit.h*, abgeleitet von *QWidget*.

Ein *QLineEdit*-Widget ermöglicht dem Anwender die Eingabe eines Textes. Der eingegebene Text kann mit der Methode *text* abgefragt werden. Das Signal *textChanged* wird jedes Mal aufgerufen, wenn sich der Text ändert, also bei jedem Tastendruck des Anwenders. Das Signal *returnPressed* wird aufgerufen, wenn der Anwender die -Taste betätigt.

Speziell für die Eingabe von Passwörtern kann mit der Methode *setEchoMode* (*Password*) das Widget so eingestellt werden, dass jedes eingegebene Zeichen durch ein Stern (*) dargestellt wird.

Bereiche des Texts können mit der Maus markiert werden. Der Text wird dabei in die Zwischenablage kopiert. Mit der mittleren Maustaste kann Text aus der Zwischenablage in das Widget eingefügt werden.

Mit der Methode *setValidator* kann dem Widget ein Objekt einer von *QValidator* abgeleiteten Klasse zugewiesen werden, das die Eingabe auf zulässige Werte einschränkt. So kann man beispielsweise mit den bereits definierten Klassen *QIntValidator* und *QDoubleValidator* die Eingabe auf korrekte ganze Zahlen bzw. korrekte reelle Zahlen einschränken.

Weitere Informationen finden Sie in Kapitel 3.7.6, *Eingabeelemente*.

Beispiel:

Der folgende Code erzeugt ein *QLineEdit*-Widget, wie es in Abbildung 6.14 zu sehen ist.

```
QLineEdit *z = new QLineEdit (this);  
// Anfangstext setzen  
z->setText ("Burkhard Lehner");
```

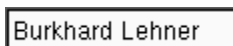
A screenshot of a QLineEdit widget, which is a single-line text input field. It has a thin border and contains the text "Burkhard Lehner" in a standard sans-serif font.

Abbildung 6-14 *QLineEdit*

QList

Definiert in *qlist.h*, abgeleitet von *QGList*.

Die Template-Klasse *QList* <*type*> implementiert eine doppelt verkettete Liste mit Zeigern auf Elemente vom Typ *type*. Sie können Elemente in die Liste einfügen und aus ihr löschen, nach Elementen suchen und die Elemente der Liste schrittweise durchlaufen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QListBox

Definiert in *qlistbox.h*, abgeleitet von *QScrollView*.

In einem Widget der Klasse *QListBox* werden Einträge (Text oder Bilder, auch gemischt) untereinander angezeigt. Mit der Maus oder den Cursor-Tasten und der Leertaste kann ein Eintrag ausgewählt werden. Reicht der Platz des Widgets nicht aus, um alle Einträge darzustellen, wird das Fenster automatisch mit einem Rollbalken versehen.

Durch einen Aufruf der Methode *setMultiSelection (TRUE)* kann das Widget angewiesen werden, auch mehrere Einträge selektieren zu lassen.

Mehr Informationen finden Sie in Kapitel 3.7.5, *Auswahlelemente*.

Beispiel:

Der folgende Code erzeugt ein *QListBox*-Widget, wie es in Abbildung 6.15 zu sehen ist.

```
QListBox *z = new QListBox (this);
z->insertItem ("Germany");
z->insertItem ("Switzerland");
z->insertItem ("United States");
z->insertItem ("Australia");
z->insertItem ("Mexico");
z->insertItem ("China");
```



Abbildung 6-15 *QListBox*

QListBoxItem

Definiert in *qlistbox.h*.

QListBoxItem ist die abstrakte Basisklasse für Elemente, die in ein *QListBox*-Widget eingefügt werden können. In Qt existieren bereits die Unterklassen *QListBoxPixmap* (für Bilder vom Typ *QPixmap*) und *QListBoxText* (für Texte). Sie können auch eigene Klassen von *QListBoxItem* ableiten, um andere Elemente darstellen zu lassen. Dazu müssen Sie in Ihrer abgeleiteten Klasse die Methoden *height*, *width* und *paint* geeignet überschreiben.

QListBoxPixmap

Definiert in *qlistbox.h*, abgeleitet von *QListBoxItem*.

Diese Klasse enthält ein *QPixmap*-Element, das in ein *QListBox*-Widget eingefügt werden kann. Es wird ausschließlich intern in *QListBox* benutzt.

QListBoxText

Definiert in *qlistbox.h*, abgeleitet von *QListBoxItem*.

Diese Klasse enthält einen Text, der in ein *QListBox*-Widget eingefügt werden kann. Sie wird ausschließlich intern in *QListBox* benutzt.

QListIterator

Definiert in *qlist.h*, abgeleitet von *QGListIterator*.

Mit dieser Template-Klasse können Sie ein Iterator-Objekt zu einem *QList*-Objekt erzeugen lassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QListView

Definiert in *qlistview.h*, abgeleitet von *QScrollView*.

Die Widget-Klasse *QListView* ermöglicht die Darstellung einer mehrspaltigen Tabelle oder eines Hierarchiebaums. Dazu fügt man in das *QListView*-Widget Elemente der Klasse *QListViewItem* ein. Diese Elemente werden untereinander dargestellt. Falls der Platz nach unten oder nach rechts nicht ausreicht, werden automatisch Rollbalken hinzugefügt.

Die einzelnen *QListViewItem*-Elemente können mit der Maus oder den Cursor-Tasten und der Leertaste ausgewählt werden. Durch einen Aufruf der Methode *setMultiSelection (TRUE)* können auch mehrere Elemente gleichzeitig ausgewählt sein.

Um eine Hierarchie darzustellen (wie zum Beispiel die Verzeichnisstruktur im Dateimanager *Konqueror*), fügt man in die enthaltenen *QListViewItem*-Objekte weitere *QListViewItem*-Objekte ein. Die so entstehende Hierarchie wird durch eine Einrückung im *QListView*-Widget angezeigt. Teile der Hierarchie können ausgeblendet werden.

Weitere Informationen über den Einsatz von *QListView* finden Sie in Kapitel 3.7.5, *Auswahlelemente*.

Beispiel:

Der folgende Code erzeugt ein *QListView*-Widget, wie es in Abbildung 6.16 zu sehen ist. Die baumartige Hierarchie entsteht durch die Zuordnung der *QListViewItem*-Elemente an das entsprechende Vorgängerelement.

```
QListView *z = new QListView (this);
z->addColumn ("Directory");
z->addColumn ("Used Space");
z->setColumnAlignment (1, AlignRight);

QListViewItem *i1, *i2, *i3, *i4, *i5;
i1 = new QListViewItem (z, "/", "320 M");
i2 = new QListViewItem (i1, "opt", "580 k");
i3 = new QListViewItem (i1, "usr", "12 M");
i4 = new QListViewItem (i2, "kde", "570 k");
i5 = new QListViewItem (i4, "include", "120 k");
i6 = new QListViewItem (i4, "share", "300 k");
```

Directory	Used Space
/	320 M
└─ opt	580 k
└─ kde	570 k
└─ include	120 k
└─ share	300 k
└─ usr	12 M

Abbildung 6-16 *QListView*

QListViewItem

Definiert in *qlistview.h*, abgeleitet von *Qt*.

Ein Objekt dieser Klasse enthält die Informationen einer Zeile eines *QListView*-Widgets. Es kann eigene *QListViewItem*-Elemente zugewiesen bekommen, die in der dargestellten Hierarchie unter diesem Eintrag aufgelistet sind und die ausgeblendet werden können.

Bis zu acht Texteinträge für die Spalten eines *QListViewItem*-Objekts können direkt im Konstruktor festgelegt werden. Wenn Sie mehr Spalten benötigen oder eine flexible Spaltenanzahl benutzen, verwenden Sie stattdessen die Methode *QListViewItem::addColumn*.

Ein Beispiel für die Anwendung von *QListViewItem* finden Sie in Abschnitt *QListView*.

QLocalFs

Definiert in *qlocalfs.h*, abgeleitet von *QNetworkProtocol*.

Diese Klasse ermöglicht den Zugriff auf das lokale Dateisystem. Sie benutzen diese Klasse meist nicht selbst, sondern greifen automatisch auf sie zu, wenn Sie *QUrlOperator* auf eine lokale Datei anwenden. Weitere Informationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*. Wollen Sie direkt auf eine lokale Datei zugreifen, so verwenden Sie besser die Klasse *QFile* (siehe Kapitel 4.18, *Dateizugriffe*).

QMainWindow

Definiert in *qmainwindow.h*, abgeleitet von *QWidget*.

Die Widget-Klasse *QMainWindow* implementiert ein Hauptfenster, also ein Toplevel-Widget, das eine Menüleiste, eine oder mehrere Werkzeugleisten und eine Statuszeile besitzt, sowie ein Widget einer beliebigen Klasse als Anzeigefenster.

Für KDE-Programme sollten Sie stattdessen die Klasse *KMainWindow* benutzen, da sie ein spezielles Look&Feel hat, das für KDE-Applikationen typisch ist. *KMainWindow* ist eine Unterklasse von *QMainWindow*, ergänzt die Klasse nur um einige Methoden.

Weitere Informationen finden Sie in Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*.

QMap

Definiert in *qmap.h*.

Ein Objekt der Template-Klasse *QMap* <*key*, *type*> implementiert – ähnlich wie *QDict* – eine Zuordnungstabelle von Schlüsseln (vom Typ *key*) zu Werten (vom Typ *type*). Im Gegensatz zu *QDict* kann bei *QMap* der Typ des Schlüssels frei bestimmt werden. Außerdem arbeitet *QMap* nicht mit Zeigern auf die Werte, sondern enthält die Werte selbst. Sie werden also in die Datenstruktur kopiert. Voraussetzung dafür ist, dass die Klasse *type* einen Copy-Konstruktor besitzt, der als *public* deklariert ist. Somit sind beispielsweise alle von *QObject* abgeleiteten Klassen als Werte ausgeschlossen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QMapConstIterator

Definiert in *qmap.h*.

Diese Klasse implementiert einen Iterator für ein *QMap*-Objekt. Mit diesem Iterator kann man alle Elemente des *QMap*-Objekts nacheinander durchlaufen. Die Klasse *QMapConstIterator* unterscheidet sich von *QMapIterator* nur dadurch, dass sie die Elemente des *QMap*-Objekts als *const*-Referenzen zurückliefert. Sie können die Elemente also nicht verändern. Sie kann daher auch für ein *QMap*-Objekt angelegt werden, das als *const* definiert ist.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QMapIterator

Definiert in *qmap.h*.

Diese Klasse implementiert einen Iterator für ein *QMap*-Objekt. Mit diesem Iterator kann man alle Elemente des *QMap*-Objekts nacheinander durchlaufen. Dabei erhält man eine Referenz auf jedes Objekt zurück, kann also die Elemente auch verändern.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und Kapitel 4.7.3, *Iterator-Objekte*.

QMenuBar

Definiert in *qmenubar.h*, abgeleitet von *QFrame* und *QMenuData*.

Ein *QMenuBar*-Widget implementiert eine Menüzeile. Die einzelnen Menüpunkte können mit der Maus ausgewählt werden. In der Regel öffnet sich dann ein Popup-Menü mit weiteren Menübefehlen.

Für ein KDE-Programm sollten Sie die Klasse *KMenuBar* benutzen, da sie direkt von der Hauptfensterklasse *KMainWindow* verwaltet werden kann.

Weitere Informationen finden Sie in Kapitel 3.5.9, *Das Hauptfenster für reine Qt-Programme*.

QMenuData

Definiert in *qmenudata.h*.

QMenuData bildet die Basisklasse für Menüs, insbesondere für *QMenuBar* und *QPopupMenu*. Sie enthält Methoden zum Eintragen von Befehlen in das Menü. Sie speichert jeden Befehl unter einer ID ab, die entweder automatisch erzeugt oder selbst festgesetzt werden kann. Zu jedem Befehl kann auch ein Objekt und eine Slot-Methode gespeichert werden, die ausgelöst werden soll, wenn der Befehl aufgerufen wird.

Die Anordnung der Befehle wird erst in den Unterklassen implementiert. Da die Klasse *QMenuData* nicht von *QObject* abgeleitet ist, kann sie das Signal *activated* nicht implementieren. Auch dieses Signal ist erst in den Unterklassen implementiert.

QMessageBox

Definiert in *qmessagebox.h*, abgeleitet von *QDialog*.

Dieses Dialog-Widget stellt ein Fenster mit einem einfachen Text und mit ein bis drei Buttons sowie ein Icon dar. Das häufigste Einsatzgebiet von *QMessageBox* sind einfache Meldungsfenster, die mit OK bestätigt werden müssen, und einfache Fragen an den Anwender, auf die er mit JA oder NEIN (oder eventuell ABBRECHEN) antwortet.

In den meisten Fällen reicht es aus, eine der statischen Methoden *information*, *warning*, *critical* oder *about* zu benutzen. Diese Methoden unterscheiden sich nur durch das Icon, das sie darstellen.

Wenn Sie ein eigenes Icon benutzen wollen, müssen Sie ein Objekt der Klasse *QMessageBox* erzeugen und mit *setIcon* ein *QPixmap*-Objekt als Icon festlegen.

Die Klasse *QMessageBox* ist mächtiger und hat eine einfachere Schnittstelle als die Klasse *KMsgBox*. *QMessageBox* ist daher meist die bessere Wahl.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*.

Beispiel:

Die folgenden Zeilen erzeugen ein *QMessageBox*-Fenster, wie es in Abbildung 6.17 zu sehen ist. Dabei ist YES der Default-Button, wird also durch die Betätigung von  automatisch ausgeführt. CANCEL ist die Aktion, die mit der -Taste ausgeführt wird.

```
int result;  
result = QMessageBox::warning (0, "KAWDraw - Unsaved Data",  
    "The document /home/usr/plt.kawd\n"  
    "contains unsaved data!\n\nDo you want to "  
    "save before closing?",  
    QMessageBox::Yes | QMessageBox::Default,  
    QMessageBox::No,  
    QMessageBox::Cancel | QMessageBox::Escape);  
switch (result)  
{  
    case 0: // save  
    case 1: // no save  
    case 2: // cancel  
}
```



Abbildung 6-17 *QMessageBox*

QMetaObject

Definiert in *qmetaobject.h*.

Ein Objekt dieser Klasse enthält Informationen über die Struktur eines *QObject*-Objekts: seine Signale, Slots, Verbindungen, den Klassennamen, die Vererbungsstruktur und die Properties. Nähere Informationen finden Sie in Kapitel 3.1.4, *Informationen über die Klassenstruktur*.

Für jede von *QObject* abgeleitete Klasse erzeugt der Meta Object Compiler (*moc*) automatisch eine eigene Klasse, die von *QMetaObject* abgeleitet ist.

QMetaProperty

Definiert in *qmetaobject.h*.

Diese Klasse speichert Informationen über eine einzelne Property einer *QObject*-Klasse. Sie wird in *QMetaObject* als Rückgabetyt bei den Methoden verwendet, die Informationen über die Properties liefern. Nähere Informationen zu Properties finden Sie in Kapitel 3.1.4, *Informationen über die Klassenstruktur*.

QMimeSource

Definiert in *qmime.h*.

Diese abstrakte Klasse stellt die Grundlage für ein kodiertes Datenpaket dar, das in verschiedene andere Formate umgewandelt werden kann. Die Formate werden dabei als Mime-Typen in Strings gespeichert. Ihre Hauptanwendungsgebiete sind Drag&Drop und die Zwischenablage.

Wenn Sie eigene Datenpakete in einer Klasse abspeichern wollen, müssen Sie diese Klasse von *QMimeSource* ableiten und die beiden Methoden *format* und *encodedData* überschreiben. Die Methode *format* besitzt einen Parameter vom Typ *int*. Für die verschiedenen Werte von 0, 1, 2 usw. liefert *format* einen String (vom Typ *const char**) zurück, der den Mime-Typ eines Formats angibt, in den das Paket umgewandelt werden kann. Dabei sollten die gängigen Formate vorn stehen, die Formate, in die nur mit Informationsverlust umgewandelt werden kann, hinten. Falls keine weiteren Formate unterstützt werden, muss *format* für den entsprechenden Wert im Parameter einen Null-Zeiger zurückgeben. Die Methode *encodedData* erhält als Parameter den Mime-Typ des gewünschten Formats. Als Rückgabewert muss sie die Daten des Pakets, umgewandelt in das angegebene Format, in einem *QByteArray*-Objekt zurückgeben.

Nähere Informationen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag&Drop*.

QMotifStyle

Definiert in *qmotifstyle.h*, abgeleitet von *QStyle*.

Diese Klasse überschreibt die Methoden von *QStyle* mit Zeichenroutinen, die ein Aussehen der Widgets wie in einer Motif-Umgebung ergeben.

QMouseEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der von der Maus ausgelöst wurde. Ein solches Objekt wird den Methoden *mousePressEvent*, *mouseMoveEvent*, *mouseReleaseEvent* und *mouseDoubleClickEvent* übergeben. Es enthält Informationen über die Position des Maus-Cursors zum

Zeitpunkt des Events, den Status der Maustasten und einiger Tasten auf der Tastatur sowie über die Maustaste, die den Event ausgelöst hat.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, Abschnitt *mousePressEvent*, *mouseReleaseEvent*, *mouseMoveEvent*, *mouseDoubleClickEvent*, *mouseWheelEvent*.

QMoveEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der durch das Verschieben eines Widgets ausgelöst wird. Ein solches Objekt wird der Methode *moveEvent* übergeben. Es enthält Informationen über die alte Position des Widgets.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, Abschnitt *moveEvent*, *resizeEvent*.

QMovie

Definiert in *qmovie.h*.

In einem Objekt dieser Klasse wird ein bewegtes Bild (z.B. GIF oder PNG) abgespeichert. Es liefert die Einzelbilder in einem *QPixmap*-Objekt zurück. Die Klasse *QLabel* kann ebenfalls ein *QMovie*-Objekt anzeigen. Dazu kann beispielsweise folgender Code benutzt werden:

```
QLabel *myLabel = new QLabel (this);  
myLabel->setMovie (QMovie ("themovie.gif"));
```

QMultiLineEdit

Definiert in *qmultilineedit.h*, abgeleitet von *QTableView*.

Diese Widget-Klasse implementiert ein mehrzeiliges Eingabefenster für Text. *QMultiLineEdit* wird zum einen als einfaches Editor-Widget eingesetzt, zum anderen kann es in den Modus *ReadOnly* gesetzt werden, so dass es als Anzeige-Widget für mehrzeiligen Text gut geeignet ist.

QMultiLineEdit fügt automatisch Rollbalken in das Widget ein, wenn der Platz zur Darstellung des Textes nicht ausreicht. Mit der Maus oder der Tastatur (Cursortasten zusammen mit der -Taste) kann man Text markieren, der in die Zwischenablage kopiert wird. Mit der mittleren Maustaste kann man Text aus der Zwischenablage in das Fenster kopieren.

Weitere Informationen finden Sie in Kapitel 3.7.6, *Eingabeelemente*. Diese Klasse wird auch im Beispiel in Kapitel 3.5.6, *Applikation für ein Dokument*, eingesetzt, um einen kleinen Texteditor zu implementieren.

Beispiel:

Der folgende Code erzeugt ein *QMultiLineEdit*-Widget, wie es in Abbildung 6.18 zu sehen ist.

```
QMultiLineEdit *z = QMultiLineEdit (this);
z->append ("Hello, Peter!\n\nYou don't believe what\n"
          "I saw yesterday!\n\n"
          "The brand new book \n\n\"KDE- "
          "und Qt-Programmierung\"!");
```

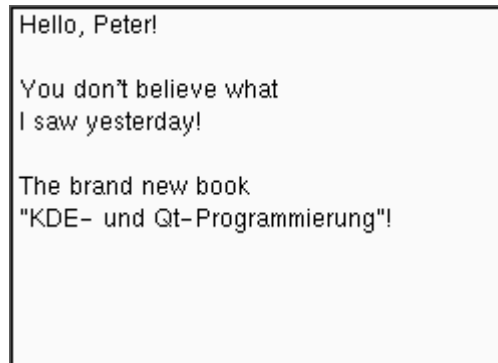


Abbildung 6-18 *QMultiLineEdit*

QMutex

Definiert in *qthread.h*, abgeleitet von *Qt*.

Diese Klasse gehört zur Thread-Unterstützung von Qt und implementiert einen geschützten Bereich (*Mutex*). Um zu gewährleisten, dass kritische Bereiche nur von einem einzigen Thread verarbeitet werden, kann man vor diesem Bereich die Methode *lock* aufrufen. Dieser Bereich ist nun für andere Threads gespermt. Ein Aufruf von *unlock* nach der Beendigung hebt die Sperre wieder auf.

QNetworkOperation

Definiert in *qnetworkprotocol.h*, abgeleitet von *QObject*.

Ein Objekt dieser Klasse repräsentiert eine Dateioperation über das Netzwerk. Es enthält Informationen über den Fortschritt und den Zustand der Operation. Nähere Informationen zu Netzwerkoperationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*.

QNetworkProtocol

Definiert in *qnetworkprotocol.h*, abgeleitet von *QObject*.

Diese abstrakte Basisklasse ist die Grundlage für alle implementierten Protokolle zum Netzwerkdateizugriff. Zur Zeit sind in der Qt-Bibliothek die abgeleiteten

Klassen *QFpt* und *QLocalFs* enthalten; weitere werden bald folgen. Sie können auch eigene Protokolle implementieren, indem Sie eine eigene Klasse von *QNetworkProtocol* ableiten. Nähere Informationen zu Netzwerkoperationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*.

QNPInstance

Definiert in *qnp.h*, abgeleitet von *QObject*.

Dieses Objekt repräsentiert ein Plug-In für den Netscape Navigator. Mit ihm können Sie eigene Plug-Ins schreiben, die Sie einbinden können. Besonders interessant sind Netscape-Plug-Ins auch deswegen, weil auch der KDE-Dateimanager *Konqueror* diese Plug-Ins einbinden kann.

QNPWidget

Definiert in *qnp.h*, abgeleitet von *QWidget*.

Ein Widget dieser Klasse ist ein Plug-In-Fenster für Netscape-Plug-Ins. Sie können es benutzen, um eigene Plug-Ins zu entwerfen (siehe auch die Klasse *QNPInstance*).

QObject

Definiert in *qobject.h*, abgeleitet von *Qt*.

Diese Klasse bildet die Grundlage für alle Widget-Klassen und einige andere Klassen der Qt-Bibliothek. In ihr ist das Signal-Slot-Konzept implementiert. Außerdem kann aus *QObject*-Objekten eine Hierarchie aufgebaut werden, die die Speicherfreigabe für den Programmierer erleichtert.

Eine sehr ausführliche Beschreibung dieser Klasse finden Sie in Kapitel 3.1, *Die Basisklasse – QObject*.

QPaintDevice

Definiert in *qpaintdevice.h*.

Diese abstrakte Basisklasse ist die Grundlage für alle Klassen, auf denen mit einem *QPainter*-Objekt gezeichnet werden kann. In Qt sind bereits die Unterklassen *QWidget*, *QPixmap*, *QPrinter* und *QPicture* von *QPaintDevice* abgeleitet.

Eine genaue Beschreibung dieser Klasse finden Sie in Kapitel 4.2, *Zeichnen von Grafikprimitiven*.

QPaintDeviceMetrics

Definiert in *qpaintdevicemetrics.h*.

Ein Objekt dieser Klasse bietet Informationen über ein *QPaintDevice*-Objekt, wie zum Beispiel über die Höhe und Breite in Pixel oder in Millimeter oder über die Farbtiefe und Anzahl der Farben.

QPainter

Definiert in *qpainter.h*, abgeleitet von *Qt*.

Ein Objekt dieser Klasse kann mit einem *QPaintDevice*-Objekt verbunden werden. *QPainter* enthält eine Reihe von Methoden, um Zeichnungen auf dem *QPaintDevice*-Objekt ausführen zu lassen, zum Beispiel Linien, Rechtecke, Polygone usw. Ein *QPaintDevice*-Objekt kann zu einem bestimmten Zeitpunkt mit maximal einem *QPainter*-Objekt verbunden sein.

Eine genaue Beschreibung der Zeichenbefehle von *QPainter* finden Sie in Kapitel 4.2.1, *QPainter*. Aber auch Kapitel 4.2.2, *Linienarten und Füllmuster*, Kapitel 4.2.3, *Koordinaten-Transformationen*, und Kapitel 4.2.4, *Beschränkungen auf Ausschnitte*, enthalten Informationen über die Eigenschaften von *QPainter*.

QPaintEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der das Neuzeichnen eines Widgets oder von Teilen davon auslösen soll. Ein solcher Event wird vom X-Server verschickt, wenn Teile des Widgets, die vorher verdeckt waren, aufgedeckt werden, oder wenn das Widget zum ersten Mal mit *show* angezeigt wird. Er kann auch durch die Methode *QWidget::update* ausgelöst werden. Das *QPaintEvent*-Objekt enthält Informationen über den Bereich, der neu gezeichnet werden muss.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *paintEvent*.

QPalette

Definiert in *qpalette.h*.

Ein Objekt dieser Klasse enthält drei Objekte der Klasse *QColorGroup*. Es wird benutzt, um die Farbgebung eines Widgets festzulegen. Jedes Widget besitzt ein eigenes Objekt der Klasse *QPalette*.

Weitere Informationen finden Sie in Kapitel 4.1.8, *Die Widget-Farbpalette*.

Besonderheiten:

QPalette arbeitet auf implizit gemeinsamen Daten. Das bedeutet, dass in der Regel alle Widgets eine Referenz auf dieselben Palettendaten benutzen. Nur wenn die Palette eines Widgets geändert wird, erhält es eine unabhängige Palette. Informationen hierzu finden Sie in Kapitel 4.7.1, *Gemeinsame Daten*.

QPen

Definiert in *qpen.h*, abgeleitet von *Qt*.

In einem Objekt dieser Klasse werden die Attribute eines Zeichenstifts für die Ausgabe mit einem *QPainter*-Objekt gespeichert. Ein Zeichenstift besitzt die Attribute Liniendicke, Linienfarbe und Linienart (durchgezogen, gestrichelt, keine Linie).

Weitere Informationen finden Sie in Kapitel 4.2.2, *Linienarten und Füllmuster*.

QPicture

Definiert in *qpicture.h*, abgeleitet von *QPaintDevice*.

In einem *QPicture*-Objekt kann man Zeichenbefehle eines *QPainter*-Objekts abspeichern. Diese Zeichenbefehle können beispielsweise in einer Datei abgelegt und von dort wieder eingelesen werden. Man kann die Zeichenbefehle in der gleichen Reihenfolge in einem anderen *QPainter*-Objekt ausführen lassen.

Weitere Informationen finden Sie in Kapitel 4.2.6, *Unterklassen von QPaintDevice*.

QPixmap

Definiert in *qpixmap.h*, abgeleitet von *QPaintDevice* und *Qt*.

In einem *QPixmap*-Objekt kann ein Teilbild im Speicher des X-Servers abgespeichert werden, ohne es auf den Bildschirm zu zeichnen. Ein *QPixmap*-Objekt wird oft benutzt, um aufwendige Zeichnungen zunächst im Speicher auszuführen und sie dann in einer einzigen Operation auf den Bildschirm zu kopieren.

Detaillierte Informationen über den Einsatz von *QPixmap* finden Sie in Kapitel 4.3, *Teilbilder – QImage und QPixmap*.

QPixmapCache

Definiert in *qpixmapcache.h*.

Diese Klasse implementiert ein globales Cache-Element für *QPixmap*-Objekte. Alle Methoden sind statisch, so dass alle Zugriffe auf den Cache nur auf das globale Objekt erfolgen. Dieses Objekt wird im Konstruktor von *QApplication* erzeugt.

Mit der Methode *insert* kann ein *QPixmap*-Objekt mit einem String als Schlüssel (in der Regel der Dateiname, falls das Bild aus einer Grafikdatei gelesen wurde) abgelegt werden. Mit der Methode *find* kann man ein bereits abgespeichertes Bild zu einem Schlüssel suchen. Auf diese Weise kann man Bilder zwischenspeichern, die oft benutzt werden, so dass sie nicht jedes Mal aus einer Datei gelesen werden müssen.

QPlatinumStyle

Definiert in *qplatinumstyle.h*, abgeleitet von *QWindowsStyle*.

Diese Klasse definiert Zeichenroutinen für die Darstellung von Widgets im Platinum-Stil, der an den Windows-Stil angelehnt ist.

QPNGImagePacker

Definiert in *pngio.h*, abgeleitet von *QPNGWriter*.

Einem Objekt dieser Klasse ordnet man im Konstruktor ein *QIODevice*-Objekt zu (in der Regel eine Datei vom Typ *QFile*). Mit der Methode *packImage* kann dann ein *QImage*-Bild in die Datei im PNG-Format eingefügt werden. Alle eingefügten Bilder werden hintereinander abgelegt. Auf diese Weise kann man bewegte PNG-Dateien erzeugen lassen.

QPoint

Definiert in *qpoint.h*.

In einem Objekt der Klasse *QPoint* kann man die Koordinaten eines Punkts im 2D-Koordinatensystem ablesen.

Die Klasse enthält einige Methoden, um die gespeicherten Koordinaten zu verschieben, zu skalieren oder mit einem anderen Punkt zu addieren (wobei sich jeweils die x- und die y-Koordinaten addieren).

QPoint wird oft auch als Parameter für eine Zeichenroutine in *QPainter* benutzt. Die Übergabe eines *QPointer*-Objekts ist oft übersichtlicher als die Übergabe von zwei *int*-Parametern.

QPointArray

Definiert in *qpointarray.h*, abgeleitet von *QArray* *<QPoint>*.

Diese Klasse definiert ein Array von Punkten im Koordinatensystem. Sie wird oft eingesetzt, um die Eckpunkte eines Polygons festzulegen.

QPointArray besitzt eine Reihe von Methoden, um das Koordinaten-Array mit Werten zu füllen. So kann man die Eckpunkte eines Rechtecks (*QRect*) direkt in ein *QPoint*-Array eintragen lassen oder eine Ellipse, einen Ellipsenabschnitt oder eine Bezierkurve in einen feinen Linienzug von Geradenstücken umwandeln lassen.

QPopupMenu

Definiert in *qpopupmenu.h*, abgeleitet von *QMenuData*.

Ein Popup-Widget implementiert ein Fenster mit einer Liste von Befehlen, die geöffnet werden kann und aus der der Anwender mit der Maus einen Befehl auswählen kann. Popup-Menüs werden am häufigsten für die Unterbefehle der

Menüzeile benutzt. Eine andere Anwendung ist der Einsatz als Kontextmenü, das mit der rechten Maustaste geöffnet werden kann.

Ein Beispiel für den Einsatz dieser Klasse finden Sie in Kapitel 2.3, *Das erste KDE-Programm*.

QPrintDialog

Definiert in *qprintdialog.h*, abgeleitet von *QDialog*.

Diese Klasse implementiert das Dialogfenster aus Abbildung 6.19, in dem der Anwender den zu benutzenden Drucker und einige Druckeroptionen wie das Seitenformat und den Bereich der zu druckenden Seiten einstellen kann.

Am einfachsten benutzt man diese Klasse, indem man die statische Methode *getPrinterSetup* aufruft. Diese erzeugt das Dialogfenster und kehrt erst zum Aufrufer zurück, wenn der Anwender das Fenster schließt.

Diese Klasse wird auch von der Methode *QPrinter::setup* benutzt.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*, sowie in Kapitel 4.17, *Drucken mit Qt*.

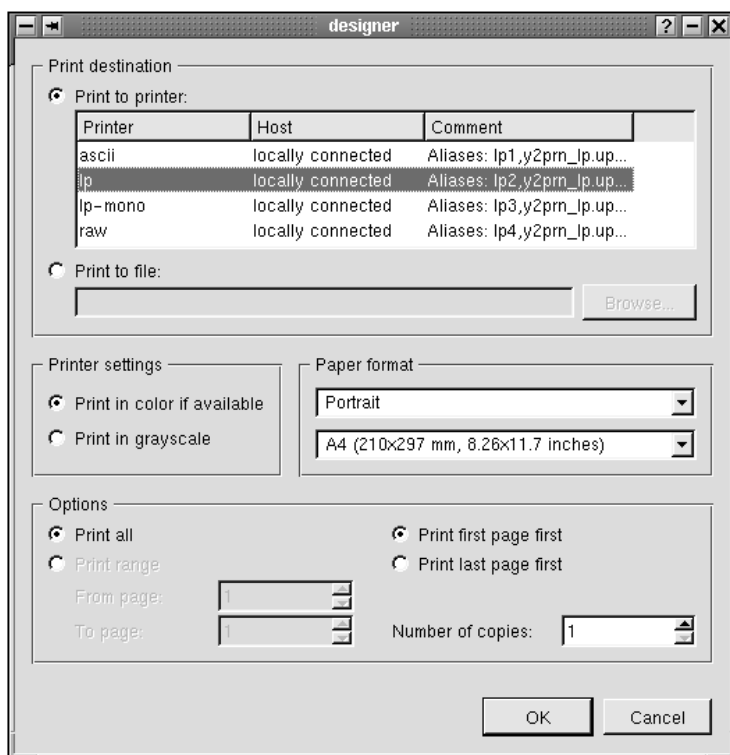


Abbildung 6-19 QPrintDialog

QPrinter

Definiert in *qprinter.h*, abgeleitet von *QPaintDevice*.

Mit einem Objekt dieser Klasse können Sie auf einen Drucker zeichnen. *QPrinter* ist von *QPaintDevice* abgeleitet, daher können Zeichnungen mit der Klasse *QPainter* ausgeführt werden. Die Ausgabe geschieht im Postscript-Format.

Weitere Informationen finden Sie in Kapitel 4.2.6, *Unterklassen von QPaintDevice*, sowie in Kapitel 4.17, *Drucken mit Qt*.

QProgressBar

Definiert in *qprogressbar.h*, abgeleitet von *QFrame*.

Dieses Widget stellt einen Fortschrittsbalken auf dem Bildschirm dar, mit dem man meist anzeigen lässt, wie viel Prozent einer längeren Operation bereits abgearbeitet worden sind.

Weitere Informationen finden Sie in Kapitel 3.7.2, *Anzeigeelemente*.

QProgressDialog

Definiert in *qprogressdialog.h*, abgeleitet von *QSemimodal*.

Diese Widget-Klasse implementiert ein Dialogfenster mit einem Fortschrittsbalken. Sie kann eingesetzt werden, um den Anwender darüber zu informieren, dass eine längere Operation stattfindet, und ihm die Möglichkeit zu geben, die noch benötigte Zeit abzuschätzen. Das Fenster enthält auch einen Button mit der Aufschrift ABBRECHEN.

QProgressDialog ist von *QSemimodal* abgeleitet. Solange das Dialogfenster geöffnet ist, sind die anderen Fenster nicht mehr bedienbar.

Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*, sowie detailliert in Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*.

QPtrDict

Definiert in *qptrdict.h*, abgeleitet von *QDict*.

In dieser Template-Klasse *QPtrDict* <type> wird eine Hash-Tabelle implementiert, die Schlüsseln vom Typ *void** Werte vom Typ *type** zuordnet. Diese Klasse kann zum Beispiel eingesetzt werden, um eine Zuordnung von Widgets auf beliebige andere Werte zu erstellen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie im Abschnitt zur Klasse *QDict* hier im Referenzteil.

QPtrDictIterator

Definiert in *qptrdict.h*, abgeleitet von *QGDictIterator*.

Mit dieser Template-Klasse können Sie ein Iterator-Objekt zu einem *QPtrDict*-Objekt erzeugen lassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, sowie in Kapitel 4.7.3, *Iterator-Objekte*.

QPushButton

Definiert in *qpushbutton.h*, abgeleitet von *QPushButton*.

Diese Widget-Klasse erzeugt eine rechteckige Schaltfläche mit einem Text oder einem Bild (vom Typ *QPixmap*) als Beschriftung. Die Schaltfläche kann mit der Maus angeklickt werden. Beim Drücken der Maustaste wird das Signal *pressed* gesendet, beim Loslassen das Signal *released*. Am häufigsten wird allerdings das Signal *clicked* benutzt, das gesendet wird, wenn der Button zuerst gedrückt und dann die Maustaste innerhalb des Buttons auch losgelassen wird. So kann sich der Anwender nach dem Drücken der Maustaste noch anders entscheiden und die Maus von der Schaltfläche bewegen.

Weitere Informationen finden Sie in Kapitel 3.7.3, *Buttons*.

Beispiel:

Das folgende Programmstück erzeugt ein *QPushButton*-Widget, wie es in Abbildung 6.20 zu sehen ist. Beim Klick auf den Button wird der Slot *action* aufgerufen. Der Button kann auch mit der Tastenkombination **Alt+B** aktiviert werden.

```
QPushButton *z =
    new QPushButton ("This is a &Button", this);
connect (z, SIGNAL (clicked()),
        this, SLOT (action()));
```

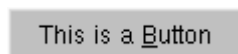


Abbildung 6-20 *QPushButton*

QQueue

Definiert in *qqueue.h*, abgeleitet von *QGList* (nur *private*).

Diese Klasse implementiert die Datenstruktur einer Warteschlange (FIFO-Liste), in die Elemente eingefügt und in der gleichen Reihenfolge auch wieder entfernt werden können.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QRadioButton

Definiert in *qradiobutton.h*, abgeleitet von *QPushButton*.

Das *QRadioButton*-Widget zeichnet einen Knopf auf den Bildschirm, der den Zustand »ein« oder »aus« besitzen kann. Der Anwender kann den Zustand durch Anklicken mit der Maus oder durch Drücken der Leertaste (wenn das *QRadioButton*-Widget den Tastaturfokus hat) umschalten. Rechts neben dem Knopf kann eine Beschriftung (ein Text oder ein Bild) stehen.

QRadioButton wird fast ausschließlich für die Auswahl von einer aus mehreren Optionen eingesetzt. Jede Option wird durch ein *QRadioButton*-Widget dargestellt. Alle Widgets werden in ein *QButtonGroup*-Objekt eingefügt, das gewährleistet, dass genau eine der Optionen eingeschaltet ist.

Wenn Sie dem Anwender eine normale Auswahl einer Option zwischen »ein« und »aus« ermöglichen wollen, so benutzen Sie *QCheckBox*.

Weitere Informationen finden Sie in Kapitel 3.7.1, *Statische Elemente*, Kapitel 3.7.3, *Buttons*, sowie in Kapitel 3.7.5, *Auswahlelemente*. Weitere Hinweise zur Anwendung in Dialogen finden Sie in Kapitel 3.8.4, *Optionsdialoge*.

Beispiel:

Die folgende Zeile erzeugt ein *QRadioButton*-Widget, wie es in Abbildung 6.21 zu sehen ist.

```
QRadioButton *z = new QRadioButton ("Option 1", this);
```

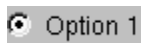


Abbildung 6-21 *QRadioButton*

QRangeControl

Definiert in *qrangecontrol.h*.

Diese Klasse bildet die Basis für alle Einstellungs-Widgets, die einen bestimmten Bereich abdecken sollen, zum Beispiel *QSlider* und *QScrollBar*. In einem Objekt dieser Klasse kann man eine obere und eine untere Grenze für den eingestellten Wert festlegen sowie eine Schrittweite für kleine und eine Schrittweite für große Schritte. *QRangeControl* verwaltet einen ganzzahligen Wert und sorgt dafür, dass er immer zwischen der Ober- und Untergrenze liegt.

Wenn Sie eine eigene Klasse von *QRangeControl* ableiten, müssen Sie die virtuelle Methode *valueChange* überschreiben. Sie wird jedes Mal aufgerufen, wenn sich der verwaltete Wert geändert hat. (Da *QRangeControl* nicht von *QObject* abgeleitet ist, kann es keine Signale schicken. Daher wurde hier der Weg über eine virtuelle Methode gewählt.) Zusätzlich können Sie noch *rangeChange* und *stepChange* überschreiben. Diese virtuellen Methoden werden aufgerufen, wenn sich der zulässige Wertebereich oder die Schrittweiten geändert haben.

QRect

Definiert in *qrect.h*.

In einem Objekt dieser Klasse kann man die genaue Position und Größe eines Rechtecks im 2D-Koordinatensystem abspeichern. Dazu werden die Koordinaten der oberen linken Ecke sowie die Breite und die Höhe abgespeichert. Alle Angaben sind ganze Zahlen.

Ein *QRect*-Objekt kann aus vier ganzen Zahlen (*x*, *y*, *Breite*, *Höhe*), zwei Punkten vom Typ *QPoint* (obere linke und untere rechte Ecke) oder einem Punkt vom Typ *QPoint* und einer Größenangabe vom Typ *QSize* (obere linke Ecke und Größe) erzeugt werden.

QRect bietet eine Vielzahl von Methoden, um Position und Größe abzufragen: Mit *top* und *bottom* kann man die y-Koordinaten des oberen und unteren Rands des Rechtecks ermitteln, mit *left* und *right* die x-Koordinaten des linken und rechten Rands. Mit *topLeft*, *topRight*, *bottomLeft* und *bottomRight* kann man die Koordinaten der Eckpunkte (vom Typ *QPoint*) ermitteln. Mit *center* kann man den Mittelpunkt des Rechtecks berechnen lassen (gerundet auf ganze Zahlen). Mit *contains* kann man testen, ob das Rechteck einen bestimmten Punkt oder ein anderes Rechteck vollständig enthält. Mit *intersect* kann man das Rechteck bestimmen, das durch den Schnitt von zwei Rechtecken entsteht, und mit *unite* das kleinste Rechteck, das beide angegebenen Rechtecke enthält.

Besonderheiten:

Beachten Sie, dass gilt:

```
right() == left() + width() - 1  
bottom() == top() + height() - 1
```

Diese (zunächst etwas ungewöhnliche) Definition ergibt sich aus der Rechnung mit Pixel-Koordinaten: Ein Rechteck, das zehn Pixel breit ist und beim Pixel mit der x-Koordinate 10 beginnt, enthält als letztes Pixel das Pixel mit der x-Koordinate 19 (nicht 20!).

QRegExp

Definiert in *qregexp.h*.

In einem Objekt der Klasse *QRegExp* kann ein regulärer Ausdruck abgelegt werden. Ein regulärer Ausdruck enthält ein Suchmuster für die Suche innerhalb von Strings. *QRegExp* wird beispielsweise bei der Suche in *QString* verwendet. *QRegExp* kann aber auch in den Modus *Wildcard* gesetzt werden. In diesem Fall erkennt *QRegExp* nur die Dateinamen-Wildcards *?* und ***.

Weitere Informationen finden Sie in Kapitel 4.7.4, *Die String-Klassen – QString und QStringRef*.

QRegion

Definiert in *qregion.h*.

In einem Objekt der Klasse *QRegion* wird ein Bereich abgespeichert, der bei der Beschränkung von Zeichenbefehlen auf Ausschnitte verwendet wird. *QRegion* kann dabei sehr komplexe Bereiche enthalten. Ausgehend von den Grundobjekten Rechteck, Ellipse und Polygon, können alle Bereiche erzeugt werden, die sich aus diesen Grundobjekten durch Mengenoperationen herstellen lassen.

Weitere Informationen finden Sie in Kapitel 4.2.4, *Beschränkung auf Ausschnitte*.

QResizeEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der durch eine Größenänderung eines Widgets ausgelöst wird. Ein solches Objekt wird der Methode *resizeEvent* übergeben. Es enthält Informationen über die alte Größe des Widgets.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *moveEvent*, *resizeEvent*.

QScrollBar

Definiert in *qscrollbar.h*, abgeleitet von *QWidget* und *QRangeControl*.

Ein Widget der Klasse *QScrollBar* zeichnet Rollbalken, die aus einem beweglichen Knopf sowie zwei Buttons mit Pfeilen bestehen. *QScrollBar* wird eingesetzt, um einen Ausschnitt aus einem größeren Bereich in einem kleinen Fenster zu zeichnen, unter anderem in den Klassen *QListBox*, *QListView*, *QScrollView* und *QTableView*. Sie können *QScrollBar* auch benutzen, um einen Wert aus einem Wertebereich einzustellen. Für diese Aufgabe ist jedoch *QSlider* besser geeignet.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungselemente*.

QScrollView

Definiert in *qscrollview.h*, abgeleitet von *QFrame*.

Ein Widget der Klasse *QScrollView* stellt ein Fenster zu einem größeren Ausschnitt dar. In diesem Fenster können Unter-Widgets platziert werden. Falls die Unter-Widgets über den Rand des *QScrollView*-Widgets hinausgehen, werden automatisch Rollbalken an das Fenster gezeichnet. Mit diesen kann der Ausschnitt verschoben werden. Neben eingefügten Unter-Widgets kann man auch durch Überschreiben der virtuellen Methode *drawContentsOffset* in das Widget zeichnen lassen.

QSemaphore

Definiert in *qthread.h*, abgeleitet von *Qt*.

Diese Klasse gehört zum Thread-Support von Qt und dient dazu, ähnlich wie *QMutex*, kritische Code-Bereiche zu schützen, so dass sie von maximal n Threads gleichzeitig durchlaufen werden. Alle anderen Threads müssen warten, bis andere Threads den Code-Bereich fertig durchlaufen haben. Die Zahl n wird dabei im Konstruktor der Klasse festgelegt. (Ein *QSemaphore*-Objekt mit $n = 1$ entspricht einem *QMutex*-Objekt.)

QSemimodal

Definiert in *qsemimodal.h*, abgeleitet von *QWidget*.

QSemimodal erzeugt ein Toplevel-Widget, das vom Verhalten einem modalen *QDialog*-Fenster ähnelt. Solange das *QSemimodal*-Fenster geöffnet ist, werden alle Maus- und Tastatur-Events für die anderen Fenster der Applikation ignoriert. Im Unterschied zu *QDialog* wird aber keine eigene Event-Schleife gestartet. Statt mit der Methode *exec* (wie bei *QDialog*) wird ein *QSemimodal*-Fenster einfach mit *show* gestartet. Qt kehrt sofort aus dieser Methode wieder zum Aufrufer zurück (im Gegensatz zu einem modalen *QDialog*-Fenster, bei dem Qt erst nach dem Schließen des Fensters wieder zurückkehrt).

Die bisher einzige Anwendung von *QSemimodal* ist die Klasse *QProgressDialog*. Weitere Informationen finden Sie in Kapitel 3.7.8, *Dialoge*, in Kapitel 3.8.3, *Modale Dialoge*, und in Kapitel 4.13.1, *Event-Verarbeitung während langer Berechnungen*.

QServerSocket

Definiert in *qserversocket.h*, abgeleitet von *QObject*.

Ein *QServerSocket*-Objekt erzeugt einen TCP/IP-Socket eines Servers, der auf eine Verbindung (mit der Unix-Betriebssystem-Funktion *connect*) wartet. Wird eine solche Verbindung hergestellt, so ruft *QServerSocket* die virtuelle Methode *newConnection* auf.

Ausführliche Informationen und ein Beispiel finden Sie in Kapitel 4.19.1, *Socket-Verbindungen*.

QSessionManager

Definiert in *qsessionmanager.h*, abgeleitet von *QObject*.

Ein *QSessionManager*-Objekt enthält Methoden, um beim Herunterfahren des X-Servers allen laufenden Programmen die Möglichkeit zu geben, ihren Zustand zu sichern. Der Konstruktor ist als *private* deklariert, kann also nicht aufgerufen werden. Es gibt pro Applikation nur ein *QSessionManager*-Objekt, das von *QApplication* angelegt wird.

Auch KDE hat ein eigenes Konzept zum Session-Management entwickelt, das einfacher zu benutzen ist, da automatisch eine Konfigurationsdatei zur Verfügung gestellt wird und die Klasse *KMainWindow* bereits viel von der Funktionalität des Session-Managements enthält.

Weitere Informationen finden Sie in Kapitel 4.16.2, *Session-Management in Qt*.

QShared

Definiert in *qshared.h*.

Ein *QShared*-Objekt enthält einen Referenzzähler, der für Datenstrukturen mit (implizit oder explizit) gemeinsam genutzten Daten verwendet wird. Bei solchen Datenstrukturen wird in der Regel ein privater Zeiger auf eine Klasse mit den Daten benutzt, wobei diese Klasse von *QShared* abgeleitet ist.

Weitere Informationen finden Sie in Kapitel 4.7.1, *Gemeinsame Daten*.

QShowEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der beim Anzeigen eines Fensters durch die Methode *show* vom X-Server ausgelöst wird. Ein solches Objekt wird der Methode *showEvent* übergeben.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *showEvent*, *hideEvent*, *closeEvent*.

QSignal

Definiert in *qsignal.h*, abgeleitet von *QObject*.

Mit einem Objekt dieser Klasse können parameterlose Signale an Slots verschickt werden. Diese Klasse wird meist benutzt, wenn man eine Verbindung mit einem parameterlosen Slot herstellen will, ohne eine eigene Klasse von *QObject* abzuleiten. Das kann zwar als schnelle Lösung hilfreich sein, sauberer ist es auf jeden Fall, die eigene Klasse von *QObject* abzuleiten und eine eigene Signalmethode zu definieren.

QSignalMapper

Definiert in *qsignalmapper.h*, abgeleitet von *QObject*.

In einem Objekt dieser Klasse kann man parameterlose Signale von verschiedenen Objekten sammeln und ein Signal aussenden lassen, das als Parameter eine Zahl oder einen String verwendet, der angibt, von welchem Objekt das Signal gesendet wurde.

Für weitere Informationen siehe Kapitel 3.1.2, *Das Signal-Slot-Konzept*, Abschnitt *Beispiel 7: Sammeln mehrerer Signale in einem Slot*.

QSimpleRichText

Definiert in *qsimplerichtext.h*.

Ein Objekt dieser Klasse kann einen Text mit RichText-Tags speichern. Mit der Methode *draw* kann dieser Text mit einem *QPainter*-Objekt, das als Parameter angegeben ist, gezeichnet werden.

Weitere Informationen finden Sie in Kapitel 4.2.1, *QPainter*, im Abschnitt *Methoden zum Zeichnen von Text*.

QSize

Definiert in *qsize.h*.

In einem Objekt der Klasse *QSize* kann eine Größenangabe eines Rechtecks (Breite und Höhe in Pixel) gespeichert werden. Die Angabe von Höhe und Breite wird dabei in je einer Variablen vom Typ *QCOORD* gespeichert.

QSizeGrip

Definiert in *qsizegrip.h*, abgeleitet von *QWidget*.

Ein Widget der Klasse *QSizeGrip* stellt eine Ecke auf dem Bildschirm dar, an der die Größe eines Toplevel-Widgets verändert werden kann. Diese Ecke wird in der Regel unten rechts platziert. Sie ist in der Regel größer als der Fensterrahmen der Dekoration des Window-Managers und ist daher für den Anwender leichter zu handhaben.

Um ein Toplevel-Widget mit dieser Möglichkeit auszustatten, müssen Sie nur ein *QSizeGrip*-Widget mit dem Toplevel-Widget als Vater-Widget erzeugen. Das *QSizeGrip*-Widget müssen Sie in der rechten unteren Ecke platzieren. Klickt der Anwender nun mit der Maus in diese Ecke und zieht sie, ändert sich automatisch die Größe des Toplevel-Widgets entsprechend.

QSizePolicy

Definiert in *qsizepolicy.h*.

In einem Objekt dieser Klasse kann die Angabe eines Widgets über seinen Platzbedarf abgespeichert werden. Diese Klasse wird als Rückgabewert der Methode *QWidget::sizePolicy* benutzt. Die Klasse enthält je eine Angabe für die Breite und für die Höhe. In ihr wird festgelegt, wie der Wert, den die Methode *QWidget::sizeHint* zurückliefert, interpretiert werden muss.

Weitere Informationen finden Sie in Kapitel 3.6.5, *Platzbedarfsberechnung für Widgets*.

QSlider

Definiert in *qslider.h*, abgeleitet von *QWidget* und *QRangeControl*.

Mit dieser Widget-Klasse wird ein Schieberegler implementiert, mit dem der Anwender einen Wert aus einem Wertebereich einstellen kann.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungselemente*.

Beispiel:

Die folgende Zeile erzeugt ein *QSlider*-Widget, wie es in Abbildung 6.22 zu sehen ist.

```
// Wertebereich von 0 bis 200, kleine Schrittweite 10,  
// große Schrittweite 60  
QSlider *z = new QSlider (0, 200, 10, 60,  
                        QSlider::Horizontal, this);
```



Abbildung 6-22 *QSlider*

QSocket

Definiert in *qsocket.h*, abgeleitet von *QIODevice* und *QObject*.

Diese Klasse stellt eine Socket-Verbindung mit einem anderen Rechner her. Sie ist von *QIODevice* abgeleitet, daher sind auch die Stream-Klassen *QTextStream* und *QDataStream* auf den Socket anwendbar. Ausführliche Informationen und ein Beispiel finden Sie in Kapitel 4.19.1, *Socket-Verbindungen*.

QSocketDevice

Definiert in *qsocketdevice.h*, abgeleitet von *QIODevice*.

Auch diese Klasse repräsentiert eine Socket-Verbindung, aber auf einer hardware-nahen Ebene. Sie haben so eine bessere Kontrolle über die Verbindung, müssen aber auch vieles von Hand aufrufen, was *QSocket* automatisch für Sie übernimmt. Weitere Informationen finden Sie in Kapitel 4.19.1, *Socket-Verbindungen*.

QSocketNotifier

Definiert in *qsocketnotifier.h*, abgeleitet von *QObject*.

Mit einem Objekt dieser Klasse kann man ein Socket (aber auch eine Datei oder eine Pipe) auf neue Daten testen lassen, ohne dabei den Prozess zu blockieren.

Detaillierte Informationen finden Sie in Kapitel 4.13.2, *Blockierungsfreie Ein- und Ausgabe*.

QSortedList

Definiert in *qsortedlist.h*, abgeleitet von *QList*.

Diese Template-Klasse *QSortedList* <type> implementiert eine doppelt verkettete Liste eines beliebigen Zeigertyps. Sie arbeitet exakt wie die Klasse *QList* <type> (von der sie abgeleitet ist), überschreibt jedoch die virtuelle Vergleichsfunktion *compareItems*. Dazu werden der ==-Operator und der <-Operator der Klasse *type* benutzt.

Die Liste wird automatisch aufsteigend sortiert, wenn alle Elemente mit der Methode *inSort* eingefügt wurden. Wurden die Elemente mit *append* oder *insert* eingefügt, ist eine Sortierung nicht gewährleistet.

QSound

Definiert in *qsound.h*, abgeleitet von *QObject*.

Diese einfache Klasse kann Dateien über die Soundkarte ausgeben. Am einfachsten benutzen Sie dazu die statische Methode *play*. Ihr übergeben Sie den Dateinamen der Datei, die abgespielt werden soll. Eine weitere Kontrolle über die Ausgabe (anhalten, fortführen, Lautstärke einstellen) bietet diese Klasse nicht. Sie benutzt das *Network Audio System* (NAS), das installiert sein muss. Auf KDE-Systemen sollten Sie stattdessen die KDE-Klassen benutzen (*KAudioPlayer*), da die Chance größer ist, dass die entsprechenden Treiber installiert sind. Weitere Informationen finden Sie in Kapitel 4.14, *Audio-Ausgabe*.

QSpacerItem

Definiert in *qabstractlayout.h*, abgeleitet von *QLayoutItem*.

Diese interne Klasse repräsentiert freien Platz in einer Layout-Liste (der zum Beispiel mit *addSpacing* erzeugt wird).

QSpinBox

Definiert in *qspinbox.h*, abgeleitet von *QFrame* und *QRangeControl*.

In einem *QSpinBox*-Widget kann der Anwender mit zwei kleinen Pfeil-Buttons am rechten Rand den angezeigten Wert vergrößern bzw. verkleinern. Alternativ kann der Anwender den Wert auch über die Tastatur eingeben.

Weitere Informationen finden Sie in Kapitel 3.7.4, *Einstellungs-elemente*.

Beispiel:

Die folgende Zeile erzeugt ein *QSpinBox*-Widget, wie es in Abbildung 6.23 zu sehen ist.

```
// Wertebereich 20 bis 60
QSpinBox *z = new QSpinBox (20, 60, 1, this);
// Anfangswert 38
```

```
z.setValue (38);
// Suffix festlegen
z.setSuffix ("%");
```

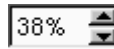


Abbildung 6-23 QSpinBox

QSplitter

Definiert in *qsplitter.h*, abgeleitet von *QWidget*.

Ein *QSplitter*-Widget ordnet die enthaltenen Unter-Widgets nebeneinander (Orientierung *Horizontal*) oder untereinander (Orientierung *Vertical*) an. Dabei wird zwischen den Unter-Widgets eine Linie gezeichnet, die mit der Maus verschoben werden kann. So kann die Größe der einzelnen Unter-Widgets verändert werden.

Weitere Informationen finden Sie in Kapitel 3.7.7, *Verwaltungselemente*.

Beispiel:

Der folgende Code erzeugt ein *QSplitter*-Widget wie in Abbildung 6.24.

```
QSplitter *z = new QSplitter (QSplitter::Vertical, this);
QMultiLineEdit *m1, *m2;
m1 = new QMultiLineEdit (z); // oberes Widget
m2 = new QMultiLineEdit (z); // unteres Widget
```

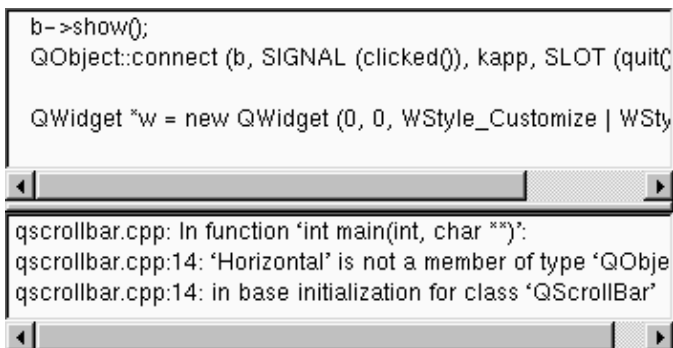


Abbildung 6-24 QSplitter

QSplitter kann beliebig viele Teilfenster verwalten. Geben Sie das *QSplitter*-Widget dazu einfach als Vater-Widget für alle Teilfenster an. Innerhalb eines *QSplitter*-Widgets sind alle Teilfenster entweder nebeneinander (horizontal) oder untereinander (vertikal) angeordnet. Sie können aber komplexere Aufteilungen erzeugen, indem Sie als Teilfenster in einem *QSplitter*-Widget selbst wiederum ein anderes *QSplitter*-Widget benutzen.

QStack

Definiert in *qstack.h*, abgeleitet von *QGList* (nur *private*).

Ein Objekt dieser Template-Klasse implementiert einen Kellerspeicher (Stack, LIFO-Struktur). Die Elemente, die auf einem Kellerspeicher abgelegt werden, können in der umgekehrten Reihenfolge wieder aus ihm entfernt werden.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QStatusBar

Definiert in *qstatusbar.h*, abgeleitet von *QWidget*.

Ein Widget dieser Klasse implementiert eine Statuszeile, die in der Regel am unteren Rand eines Hauptfensters dargestellt wird. *QStatusBar* wird in einem Hauptfenster der Klasse *QMainWindow* benutzt.

Wenn Sie ein KDE-Programm entwickeln und die Hauptfensterklasse *KMainWindow* benutzen, sollten Sie statt *QStatusBar* besser *KStatusBar* benutzen, da *KMainWindow* bereits Methoden enthält, um eine Statuszeile vom Typ *KStatusBar* einzufügen.

Weitere Informationen finden Sie in Kapitel 3.5.5, *Die Statuszeile*.

QStoredDrag

Definiert in *qdrag.h*, abgeleitet von *QDragObject*.

In einem Objekt dieser Klasse können Sie einen Datenblock für Drag&Drop sowie für die Zwischenablage benutzen, der nur in einem Mime-Format verfügbar sein muss. Sie können den Datenblock der Klasse *QByteArray* zuweisen. So ersparen Sie sich das Ableiten einer eigenen Klasse von *QDragObject*. Wenn Sie dagegen ein Objekt erstellen wollen, das in verschiedene Formate umgewandelt werden kann, müssen Sie eine eigene Klasse ableiten.

Weitere Informationen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag&Drop*.

QStrList

Definiert in *qstrlist.h*, abgeleitet von *QStrList*.

Ein Objekt dieser Klasse kann eine Liste von Strings des Typs *char** speichern. Diese Klasse verhält sich dabei weitgehend wie die Klasse *QStrList*, von der sie abgeleitet ist. In ihr ist allerdings die virtuelle Methode *compareItems* so überschrieben, dass beim Vergleich von zwei Elementen nicht zwischen Groß- und Kleinschreibung unterschieden wird. Diese Methode wird beim Einfügen in die Liste mit *inSort* sowie beim Suchen benutzt.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QString

Definiert in *qstring.h*.

In einem Objekt dieser Klasse kann ein String im Unicode-Format abgespeichert werden. Jedes Zeichen belegt dabei zwei Byte. *QString* enthält eine Reihe von Methoden und Operatoren, um Strings aneinander zu hängen, Teilstrings zu bilden, den Text in Groß- oder Kleinbuchstaben zu verwandeln, nach Teiltexten zu suchen usw.

Ebenso enthält *QString* einige Konstruktoren, um aus einem Text vom Typ *char** oder vom Typ *QCString* einen Text im Unicode-Format zu erzeugen. Somit kann eine String-Konstante an jeder Stelle eingesetzt werden, an der ein *QString*-Parameter erwartet wird. Mit der Methode *ascii* kann der Text eines *QString*-Objekts in ASCII umgewandelt werden. Für diese Umwandlung gibt es auch den Cast-Operator von *QString* nach *const char**. So kann *QString* auch überall dort benutzt werden, wo ein String vom Typ *const char** erwartet wird.

Benutzen Sie *QString* für alle Texte, die auf dem Bildschirm angezeigt werden sollen, damit auch ausländische Sonderzeichen problemlos dargestellt werden.

Weitere Informationen finden Sie in Kapitel 4.7.4, *Die String-Klassen – QString und QCString*.

QStringList

Definiert in *qstringlist.h*, abgeleitet von *QValueList <QString>*.

In einem Objekt dieser Klasse kann eine doppelt verkettete Liste von *QString*-Objekten gespeichert werden. Im Gegensatz zu *QStrList* und *QStrIList* werden in dieser Liste Strings vom Typ *QString* (also mit voller Unicode-Unterstützung) und nicht vom Typ *char** gespeichert.

QStrList

Definiert in *qstrlist.h*, abgeleitet von *QList <char*>*.

In einem Objekt dieser Klasse kann eine doppelt verkettete Liste von Strings im ASCII-Format (*char**) abgespeichert werden. Die Methode *compareItems* ist so überladen, dass die Elemente der Liste beim Einfügen mit *inSort* sowie beim Suchen mit *strcmp* verglichen werden.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QStyle

Definiert in *qstyle.h*, abgeleitet von *QObject*.

Diese abstrakte Klasse definiert eine Reihe von Methoden zum Zeichnen von typischen Widget-Elementen, die beim Zeichnen von Widgets benutzt werden können. Durch Ableiten einer eigenen Klasse kann man diese Methoden über-

schreiben und so ein anderes Look&Feel für die Widgets einer Applikation erzeugen. Sie setzen ein neues *QStyle*-Objekt mit der Methode *QApplication::setStyle*.

In Qt sind bereits einige Klassen von *QStyleObject* abgeleitet: *QMotifStyle*, *QWindowsStyle*, *QCDEStyle* sowie *QPlatinumStyle*.

QStyleSheet

Definiert in *qstylesheet.h*, abgeleitet von *QObject*.

In einem *QStyleSheet*-Objekt werden Informationen gespeichert, wie die Tags eines RichText-Strings zu interpretieren sind. Durch Ableiten einer eigenen Klasse können Sie die Tags anders interpretieren.

Qt

Definiert in *qnamespace.h*.

Die Klasse *Qt* enthält eine Reihe von Aufzählungstypen und Konstanten, die in Qt häufig benutzt werden. In den Qt-Versionen vor 2.0 waren sie global definiert, wodurch es zu einer »Verschmutzung« des Namensraums kam: In Ihrem Programm, insbesondere aber auch in allen eingebundenen Header-Dateien, konnten diese Bezeichner nicht mehr verwendet werden. Um dieses Problem zu lösen, wurde die Klasse *Qt* eingeführt. *Qt* ist Basisklasse von *QObject*. Innerhalb der Methoden einer von *QObject* abgeleiteten Klasse können die Aufzählungstypen und Konstanten so benutzt werden, als seien sie global definiert. Außerhalb muss durch Voranstellen von *Qt::* diese Klasse spezifiziert werden.

Weitere Informationen zur Portierung alter Programme finden Sie in Kapitel 4.22.4, *Aufzählungstypen und Konstanten zusammengefasst in der Klasse Qt*.

QTabBar

Definiert in *qtabbar.h*, abgeleitet von *QWidget*.

Ein *QTabBar*-Widget erzeugt eine Zeile mit Karteikartenreitern (Tab), aus denen der Anwender einen Reiter auswählen kann. Das Hauptanwendungsgebiet von *QTabBar* ist die Klasse *QTabDialog*, in der mehrere Widgets in einem Dialogfenster dargestellt werden können, von denen eine Seite über den zugehörigen Karteikartenreiter angezeigt wird, während die anderen versteckt sind.

QTabDialog

Definiert in *qtabdialog.h*, abgeleitet von *QDialog*.

Diese Klasse implementiert ein Dialog-Fenster, das mehrere Seiten enthalten kann, aus denen der Anwender über eine Karteikartenreiter-Leiste (*QTabBar*) eine Seite auswählen kann. Diese Seite wird angezeigt. Die einzelnen Seiten sind Wid-

gets, von denen genau ein Widget mit *show* dargestellt wird, während die anderen mit *hide* versteckt werden.

Besonders häufig wird *QTabDialog* eingesetzt, wenn der Anwender so viele Optionen auswählen kann, dass sie nicht mehr auf eine Seite passen. Die Optionen werden dann in sinnvolle Gruppen unterteilt und einzelnen Seiten zugeordnet.

QTabDialog enthält am unteren Rand des Fensters zwei bis fünf Buttons, deren Beschriftung man festlegen kann.

QTabDialog besteht hauptsächlich aus einem *QTabWidget*-Objekt und den Buttons, die in einem modalen Dialogfenster angeordnet werden. Wenn Sie keine Buttons benötigen – weil Sie beispielsweise die Seiten innerhalb eines anderen Widgets darstellen wollen –, benutzen Sie die Klasse *QTabWidget*.

Weitere Informationen finden Sie in Kapitel 3.8.4, *Optionsdialoge*.

Beispiel:

Der folgende Code erzeugt ein *QTabDialog*-Fenster, wie es in Abbildung 6.25 zu sehen ist.

```
// Seite 1 erzeugen
QWidget *a = new QWidget ();
QCheckBox *c1 = new QCheckBox ("Option 1", a);
QCheckBox *c2 = new QCheckBox ("Option 2", a);

// Layout für Seite 1
QVBoxLayout *l = new QVBoxLayout (a, 10, 10);
l->addWidget (c1);
l->addWidget (c2);
l->addStretch (1);
l->activate();

// analog Widget b und c für Seite 2 und 3
....

// Dialogfenster erzeugen
QTabDialog *z = new QTabDialog ();
// alle drei Seiten einfügen
z->addTab (a, "Page 1");
z->addTab (b, "Page 2");
z->addTab (c, "Page 3");
z->show ();
```

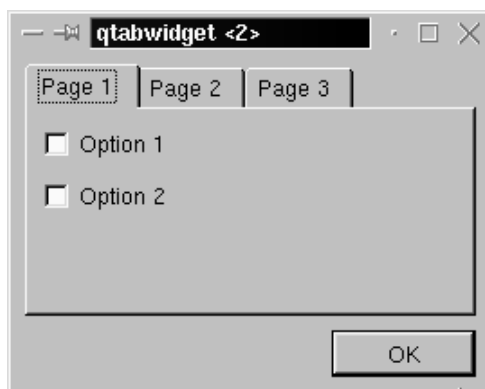


Abbildung 6-25 QTabDialog

QTable

Definiert in *qtable.h*, abgeleitet von *QScrollView*, enthalten im Modul *Table*.

Diese relativ neue Widget-Klasse bietet ähnlich wie *QListView* eine Tabelle mit Spalten und Zeilen. Allerdings kann man die einzelnen Einträge innerhalb der Tabelle editieren. Es ergibt sich so ein Widget, das man aus Tabellenkalkulationsprogrammen kennt.

QTableView

Definiert in *qtableview.h*, abgeleitet von *QFrame*.

Diese Klasse ist eine abstrakte Basisklasse für GUI-Elemente, die Informationen in einer Tabellenform darstellen sollen. Die Anzahl der Spalten und Zeilen ist flexibel. Die Breite und Höhe der Zellen in der Tabelle kann für alle Zellen konstant sein oder für jede Zeile und Spalte flexibel. Das Widget kann automatisch mit Rollbalken ausgestattet werden, wenn der Inhalt für das Widget zu groß ist.

Wenn Sie eine eigene Klasse von *QTableView* ableiten wollen, müssen Sie die virtuelle Methode *paintCell* überschreiben. In ihr implementieren Sie die Zeichenoperationen, die den Inhalt einer Zelle zeichnen. Das übergebene *QPainter*-Objekt ist bereits so eingestellt, dass die Zeichenoperationen korrekt verschoben und abgeschnitten werden.

Ein Beispiel für die Definition einer eigenen *QTableView*-Klasse finden Sie in Kapitel 4.4.3, *Beispiel: Dame-Brett*.

QTabWidget

Definiert in *qtabwidget.h*, abgeleitet von *QWidget*.

In ein *QTabWidget*-Objekt können mehrere Seiten in Form eines *QWidget*-Objekts eingefügt werden, von denen eine Seite aus der Karteikartenreiter-Liste

ausgewählt werden kann. Nur diese Seite wird angezeigt, die anderen Widgets werden mit *hide* versteckt.

QTabWidget besteht aus einem *QTabBar*-Widget und einem *QWidgetStack*-Widget, die untereinander angeordnet werden.

QTabWidget erzeugt ein Widget, das ähnlich wie *QTabDialog* funktioniert, das jedoch kein Dialog-Fenster ist und keine Buttons enthält.

QTextBrowser

Definiert in *qtextbrowser.h*, abgeleitet von *QTextView*.

Dieses Widget kann Texte im RichText-Format – ein Format, das ähnlich wie HTML aufgebaut ist – anzeigen und dabei auch Links verfolgen. Um einen vollständigen Browser daraus zu erstellen, müssen Sie die Methode *setSource* überschreiben, die den Dokumentnamen zum Beispiel in einen Dateinamen umsetzt und diese Datei einliest. Sie können diese Klasse beispielsweise benutzen, um eine Online-Hilfe zu erstellen. In KDE-Programmen sollten Sie allerdings die KDE-typische Online-Hilfe wählen (siehe Kapitel 4.11.3, *Anwenderhandbuch*).

QTextDrag

Definiert in *qdragobject.h*, abgeleitet von *QDragObject*.

In einem Objekt dieser Klasse kann ein Text abgelegt werden, um ihn per Drag&Drop in ein anderes Fenster zu ziehen oder um ihn in die Zwischenablage zu kopieren.

Nähere Informationen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag&Drop*.

QTextEdit

Definiert in *qtextview.h*, abgeleitet von *QTextView*.

Diese Widget-Klasse ist noch in der Entwicklung. Sie soll ein Editieren eines RichText-Dokuments ermöglichen.

QTextStream

Definiert in *qtextstream.h*, abgeleitet von *QTextStream*.

Ein Objekt dieser Klasse implementiert ein *QTextStream*-Objekt, das nur gelesen werden kann.

QTextOStream

Definiert in *qtextstream.h*, abgeleitet von *QTextStream*.

Ein Objekt dieser Klasse implementiert ein *QTextStream*-Objekt, in das nur geschrieben werden kann.

QTextStream

Definiert in *qtextstream.h*.

Mit einem Objekt dieser Klasse kann man einen Stream erzeugen, der ganz ähnlich benutzt werden kann wie die C++-Streams, die zum Beispiel in *cin*, *cout* und *cerr* benutzt werden. Alle gängigen Datentypen können mit dem <<-Operator in den Stream geschrieben und mit dem >>-Operator aus dem Stream gelesen werden.

Zwei Besonderheiten zeichnen die Klasse *QTextStream* aus: Sie kann nicht nur auf Dateien arbeiten, sondern wird einem *QIODevice* zugeordnet, das zum Beispiel ein *QFile*-Objekt, aber auch ein *QBuffer*-Objekt sein kann. *QTextStream* kann den Text im ASCII-Format oder im Unicode-Format abspeichern. (Die entstehende Datei lässt sich für Unicode-Dateien allerdings dann nicht mit einem normalen Texteditor lesen oder verändern.)

Weitere Informationen finden Sie in Kapitel 4.18.4, *Stream-basierte Ein-/Ausgabe*.

QTextView

Definiert in *qtextview.h*, abgeleitet von *QScrollView*.

In einem *QTextView*-Widget kann ein Text im RichText-Format angezeigt werden. Das RichText-Format benutzt Tags – ähnlich wie HTML –, um den Zeichensatz, die Farbe und die Ausrichtung des Texts festzulegen sowie Links auf andere Seiten anzugeben. Ebenso können Abbildungen in den Text eingebunden werden.

Diese Klasse kann ein RichText-Dokument nur anzeigen. Angeklickte Links werden nicht verfolgt. Dafür sollten Sie *QTextBrowser* benutzen.

QThread

Definiert in *qthread.h*, abgeleitet von *Qt*.

Diese Klasse implementiert plattformunabhängig eine Realisierung von Threads. Um einen eigenen Thread zu erstellen, leiten Sie eine eigene Unterklasse von *QThread* ab und überschreiben die Methode *run*, so dass sie den gewünschten Code ausführt. Um einen solchen Thread nun im Verlauf des Programms zu starten, erzeugen Sie ein Objekt der selbst definierten Klasse und rufen die Methode *start* auf. Der Thread wird dann parallel zur Ausführung des normalen Programms den Code in *run* ausführen. Sobald der Thread die Methode *run* beendet hat, hält er an.

Um kritische Bereiche Ihres Programms zu schützen, die nicht von mehreren Threads gleichzeitig durchlaufen werden dürfen, können Sie die Klasse *QSemaphore* und *QMutex* benutzen.

Weitere Informationen zu Threads finden Sie in Kapitel 4.13.4, *Mehrere Prozesse und Threads*.

QTime

Definiert in *qdatetime.h*.

Ein Objekt dieser Klasse kann eine Uhrzeit speichern und damit einige Berechnungen anstellen. Mit der statischen Methode *currentTime* kann außerdem die aktuelle Uhrzeit aus der Echtzeituhr des Rechners ermittelt werden.

Weitere Informationen finden Sie in Kapitel 4.12.1, Die Klassen *QDate*, *QTime* und *QDateTime*.

QTimer

Definiert in *qtimer.h*, abgeleitet von *QObject*.

Ein Objekt dieser Klasse realisiert einen Qt-Timer, der nach Ablauf eines festgelegten Zeitintervalls (in Millisekunden) ein Signal aktiviert. Der Timer kann einmalige oder regelmäßige Signale schicken.

Weitere Informationen finden Sie in Kapitel 4.12.3, Die Klasse *QTimer*.

QTimerEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der durch einen Timer ausgelöst wird, der mit der Methode *QObject::startTimer* gestartet wurde. Ein solches Objekt wird der Methode *timerEvent* übergeben. Es enthält die Nummer des Timers, der den Event ausgelöst hat.

Weitere Informationen finden Sie in Kapitel 4.12.2, Nutzung von Timer-Events.

QToolBar

Definiert in *qtoolbar.h*, abgeleitet von *QWidget*.

Diese Widget-Klasse implementiert eine Werkzeugleiste für ein *QMainWindow*-Fenster. Die Buttons der Werkzeugleiste können dabei ein Icon, einen Text oder beides besitzen.

Wenn Sie als Hauptfensterklasse *KMainWindow* benutzen, so können Sie *QToolBar* nicht verwenden. Benutzen Sie stattdessen *KToolBar*.

QToolButton

Definiert in *qtoolbutton.h*, abgeleitet von *QPushButton*.

Ein Widget dieser Klasse implementiert einen Button für eine Werkzeugleiste der Klasse *QToolBar*.

QToolTip

Definiert in *qtooltip.h*, abgeleitet von *Qt*.

Mit dieser Klasse lässt sich einem Widget ein kurzer Text zuordnen. Wenn die Maus für eine kurze Zeit auf dem Widget verweilt, öffnet sich ein kleines Fenster, das diesen Text anzeigt.

Weitere Informationen finden Sie in Kapitel 4.11.1, *Tooltips*.

QToolTipGroup

Definiert in *qtooltip.h*, abgeleitet von *QObject*.

Ein Objekt dieser Klasse kann Tooltip-Aktionen von mehreren Widgets sammeln und etwas längere Texte, die den Widgets zugeordnet sind, über ein Signal verschicken. Diese Texte werden häufig in der Statuszeile angezeigt.

Weitere Informationen finden Sie in Kapitel 4.11.1, *Tooltips*.

QTranslator

Definiert in *qtranslator.h*, abgeleitet von *QObject*.

Ein Objekt dieser Klasse übernimmt die Umsetzung von Texten in eine andere Landessprache. In ein *QTranslator*-Objekt können Sie eine Übersetzungsdatei laden. Anschließend installieren Sie dieses Objekt im *QApplication*-Objekt und können danach mit der Methode *QObject::tr* auf die Übersetzungen zugreifen.

Weitere Informationen finden Sie in Kapitel 4.9.2, *Qt-Übersetzungen – Die Methode QObject::tr*.

QUriDrag

Definiert in *qdragobject.h*, abgeleitet von *QDragObject*.

In einem Objekt dieser Klasse kann eine Liste von von URL-Referenzen gespeichert werden, um sie per Drag&Drop in ein anderes Fenster zu ziehen oder um sie in die Zwischenablage zu kopieren.

Nähere Informationen finden Sie in Kapitel 4.15, *Die Zwischenablage und Drag&Drop*.

QUrl

Definiert in *qurl.h*.

Diese Klasse speichert einen Pfad zu einer Datei im URL-Format. Diese Datei kann dabei auch über ein Netzwerk (z.B. über ein HTTP- oder FTP-Protokoll) angesprochen werden. Die Klasse enthält weiterhin viele Methoden, um die angegebene URL zu analysieren oder zu manipulieren. Die KDE-Klassen zum netzwerktransparenten Zugriff verwenden die Klasse *KURL*, die *QUrl* sehr ähnlich ist.

Weitere Informationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*.

QUrlOperator

Definiert in *qurloperator.h*, abgeleitet von *QObject* und *QUrl*.

Diese Klasse enthält viele Methoden zum netzwerktransparenten Zugriff auf Dateien, also zum Laden, Speichern, Umbenennen und Kopieren. Sie benutzen diese Klasse, indem Sie ein Objekt dieser Klasse bilden und die gewünschte Operationsmethode aufrufen. Als Parameter müssen Sie die Dateinamen (als URL) übergeben.

Weitere Informationen finden Sie in Kapitel 4.19.3, *Netzwerktransparenter Dateizugriff in Qt*.

QValidator

Definiert in *qvalidator.h*, abgeleitet von *QObject*.

Diese Klasse implementiert eine abstrakte Basisklasse für Objekte, die zur Kontrolle einer Eingabe benutzt werden können. Man kann beispielsweise einem *QLineEdit*-Objekt ein *QValidator*-Objekt zuordnen, das darauf achtet, dass die Eingabe bestimmte Bedingungen erfüllt.

Qt enthält bereits zwei von *QValidator* abgeleitete Klassen: *QIntValidator* und *QDoubleValidator*. Diese beiden Klassen kontrollieren eine Eingabe daraufhin, ob sie eine gültige ganze Zahl bzw. eine gültige reelle Zahl in vorgegebenen Grenzen ist.

Wenn Sie eine eigene Klasse ableiten wollen, müssen Sie die Methode *validate* überschreiben. In *validate* testen Sie, ob der übergebene String den Bedingungen genügt. Der Rückgabewert der Methode ist entweder *Valid* (Eingabe ist gültig), *Invalid* (Eingabe ist ungültig) oder *Acceptable* (Eingabe ist zur Zeit ungültig, kann aber durch eine weitere Eingabe gültig werden).

Zusätzlich können Sie die Methode *fixup* überschreiben. In ihr können Sie den übergebenen String so verändern, dass daraus eine gültige Eingabe entsteht.

QValueList

Definiert in *qvaluelist.h*.

Ein Objekt der Template-Klasse *QValueList* *<type>* speichert eine doppelt verketete Liste von Einträgen des Typs *type*. Im Unterschied zu *QList* *<type>* werden hier die Elemente selbst (indem sie mit dem Copy-Konstruktor kopiert werden) und keine Referenzen gespeichert. Daher ist es nicht möglich, Klassen zu speichern, deren Copy-Konstruktor nicht *public* ist, zum Beispiel alle von *QObject* abgeleiteten Klassen.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QValueListConstIterator

Definiert in *qvaluelist.h*.

Ein Objekt der Template-Klasse *QValueListConstIterator<type>* implementiert einen Iterator für ein *QValueList<type>*-Objekt. Die Elemente der Liste werden als konstante Referenzen zurückgegeben, sie können also nicht verändert werden. Daher kann ein *QValueListConstIterator*-Objekt auch zu einem als *const* deklarierten *QValueList*-Objekt erzeugt werden.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QValueListIterator

Definiert in *qvaluelist.h*.

Ein Objekt der Template-Klasse *QValueListIterator<type>* implementiert einen Iterator für ein *QValueList<type>*-Objekt. Die Elemente der Liste werden als Referenzen zurückgegeben, sie können also verändert werden.

Weitere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*, und in Kapitel 4.7.3, *Iterator-Objekte*.

QVariant

Definiert in *qvariant.h*.

Ein Objekt dieser Klasse kann Daten der verschiedensten Datentypen speichern. Es hat also ähnliche Möglichkeiten wie ein *union*-Konstrukt der Programmiersprache C, ist dabei aber typsicher. Neben einfachen Typen kann es auch Listen oder Zuordnungstabellen speichern, die ihrerseits wieder *QVariant*-Objekte enthalten.

Genaue Informationen über die Klasse finden Sie in Kapitel 4.7.5, *Flexibler Datentyp – QVariant*.

QVBox

Definiert in *qvbox.h*, abgeleitet von *QHBox*.

Dieses Widget ordnet alle Unter-Widgets untereinander von oben nach unten an. Der benötigte Platz wird dabei aus den Werten ermittelt, die die Unter-Widgets in den Methoden *sizeHint* und *sizePolicy* zurückliefern.

Da *QVBox* von *QHBox* und das wiederum von *QFrame* abgeleitet ist, kann das Widget mit einem Rahmen versehen werden. Standardmäßig wird kein Rahmen dargestellt. Zwischen den Unter-Widgets wird ein Streifen in der Breite des Rahmens freigelassen. Ohne Rahmen wird also kein Platz zwischen den Widgets gelassen.

Weitere Informationen finden Sie in Kapitel 3.6.3, *Einfache Layouts mit QHBoxLayout, QVBoxLayout und QGrid*.

QVBoxLayout

Definiert in *qlayout.h*, abgeleitet von *QBoxLayout*.

Diese Klasse übernimmt die Verwaltung der Größe und Position von mehreren Unter-Widgets innerhalb eines Widgets. Die Elemente werden dabei in einer Spalte untereinander von oben nach unten angeordnet. Mit der Methode *addWidget* kann ein Widget in das Layout eingefügt werden. Mit *addLayout* kann ein anderes Layout-Objekt eingefügt werden.

QVBoxLayout ist von *QBoxLayout* abgeleitet und unterscheidet sich von der Basis-Klasse nur dadurch, dass im Konstruktor keine Angabe über die Ausrichtung des Layouts angegeben wird, da implizit die Ausrichtung *TopToBottom* benutzt wird.

Eine sehr detaillierte Beschreibung der Vorgehensweise finden Sie in Kapitel 3.6.4, *Die Layout-Klassen QHBoxLayout und QGridLayout*.

QVButtonGroup

Definiert in *qvbuttongroup.h*, abgeleitet von *QButtonGroup*.

Diese Widget-Klasse ergänzt die Klasse *QButtonGroup* um ein automatisches Layout der eingefügten Unter-Widgets. Sie kombiniert also die Funktionalität von *QButtonGroup* und *QVBox*.

Die eingefügten Unter-Widgets werden von oben nach unten eingefügt.

QVector

Definiert in *qvector.h*, abgeleitet von *QGVector*.

Diese Template-Klasse ist eine Container-Klasse, die intern ein dynamisches Array mit Zeigern auf die abgelegten Datenelemente enthält. Nähere Informationen finden Sie in Kapitel 4.7.2, *Container-Klassen*.

QVGroupBox

Definiert in *qvgroupbox.h*, abgeleitet von *QGroupBox*.

Diese Widget-Klasse ergänzt die Klasse *QGroupBox* um ein automatisches Layout der eingefügten Unter-Widgets. Sie kombiniert also die Funktionalität von *QGroupBox* und *QVBox*. Alle eingefügten Unter-Widgets der Klasse *QRadioButton* werden damit automatisch zu sich ausschließenden Widgets, von denen maximal ein Widget im Zustand »an« sein kann.

Die eingefügten Unter-Widgets werden von oben nach unten eingefügt.

QWaitCondition

Definiert in *qthread.h*, abgeleitet von *Qt*.

Diese Klasse des Thread-Pakets von Qt ermöglicht die Koordination zwischen verschiedenen Threads.

QWhatsThis

Definiert in *qwhatsthis.h*, abgeleitet von *Qt*.

Mit dieser Klasse kann eine Online-Hilfe ähnlich wie mit *QToolTip* implementiert werden. Bei *QWhatsThis* sind die Erklärungstexte aber im Allgemeinen länger. Ein *QWhatsThis*-Fenster wird nur dann geöffnet, wenn man zunächst einen speziellen Button der Applikation anklickt und dann das Fenster, zu dem man weitere Informationen benötigt.

Detaillierte Informationen finden Sie in Kapitel 4.11.2, *What's-This-Hilfe*.

QWheelEvent

Definiert in *qevent.h*, abgeleitet von *QEvent*.

In einem Objekt dieser Klasse speichert Qt die Informationen zu einem Event, der vom Rad einer Maus ausgelöst wurde. Dazu muss das Rad (das in der Regel in die mittlere Maustaste integriert ist) vom X-Server unterstützt werden. Ein solches Objekt wird der Methode *wheelEvent* übergeben. Es enthält Informationen über den Winkel, um den das Rad gedreht wurde, über die Position des Maus-Cursors zum Zeitpunkt des Events sowie über den Status der Maustasten und einiger Tasten auf der Tastatur.

Weitere Informationen finden Sie in Kapitel 4.4.1, *Event-Methoden*, im Abschnitt *mousePressEvent*, *mouseReleaseEvent*, *mouseMoveEvent*, *mouseDoubleClickEvent*, *mouseWheelEvent*.

QWidget

Definiert in *qwidget.h*, abgeleitet von *QObject*.

Diese Klasse ist die Basisklasse für alle Fenster und GUI-Elemente, die Qt darstellt. Ein *QWidget*-Objekt kann Unter-Widgets enthalten, die innerhalb des Widgets dargestellt werden. *QWidget* kann benutzt werden, um ein Fenster für mehrere Unter-Widgets zu implementieren. Dabei ordnet man die Unter-Widgets meist mit einem *QLayout*-Objekt an.

Sehr detaillierte Informationen über die Anwendung und die Eigenschaften von *QWidget*-Objekten finden Sie in Kapitel 3.2, *Die Fensterklasse – QWidget*.

QWidgetItem

Definiert in *qabstractlayout.h*, abgeleitet von *QLayoutItem*.

Diese interne Klasse repräsentiert ein Widget in einer Layout-Liste (das zum Beispiel mit *addWidget* eingefügt wurde).

QWidgetStack

Definiert in *qwidgetstack.h*, abgeleitet von *QFrame*.

In dieses Widget kann man mit der Methode *addWidget* eine Reihe von Unter-Widgets eintragen. Von diesen Widgets wird immer genau eines dargestellt; alle anderen sind mit *hide* versteckt. Auf diese Weise kann man mehrere Seiten in einem Widget unterbringen, von denen nur eine sichtbar ist. *QWidgetStack* wird unter anderem in *QTabWidget* genutzt, bei dem die anzuzeigende Seite über eine Leiste von Karteikartenreitern ausgewählt werden kann.

Weitere Informationen finden Sie in Kapitel 3.7.7, *Verwaltungselemente*.

QWindowsStyle

Definiert in *qwindowsstyle.h*, abgeleitet von *QStyle*.

Diese Klasse überschreibt die Methoden von *QStyle* mit Zeichenroutinen, die ein Aussehen der Widgets wie in einer Microsoft Windows-Umgebung ergeben.

QWizard

Definiert in *qwizard.h*, abgeleitet von *QDialog*.

Ein Dialogfenster der Klasse *QWizard* kann mehrere Seiten enthalten, die in einem *QWidgetStack*-Objekt gespeichert werden. Der Zugriff auf die einzelnen Seiten erfolgt dabei über die zwei Buttons *PREVIOUS* und *NEXT* am unteren Fens-terrand. Dieser Dialog wird meist eingesetzt, um die Optionen für eine Installa-tion vom Anwender abzufragen, wenn die Optionen voneinander abhängen. Durch diese Vorgehensweise ist gewährleistet, dass der Anwender zunächst alle Seiten durchlaufen hat, bevor er den Dialog abschließen kann.

QWMatrix

Definiert in *qwmatrix.h*.

In einem Objekt dieser Klasse kann eine Koordinatentransformation gespeichert werden, wie sie in *QPainter* benutzt werden kann. Die Transformation wird dabei in einer 3x3-Matrix aus *double*-Werten gespeichert. Nur sechs der neun Werte sind allerdings variabel, die anderen drei sind fest.

Mit den Methoden *shear*, *scale*, *translate* und *rotate* kann die Transformation ver-ändert werden. Durch die Multiplikation von zwei *QWMatrix*-Objekten kann man die Transformationen kombinieren. Mit der Methode *map* können einzelne oder mehrere Punkte transformiert werden.

In einem *QPainter*-Objekt wird die Transformation in einem *QWMatrix*-Objekt abgelegt. Dieses Objekt können Sie auslesen oder setzen.

Weitere Informationen finden Sie in Kapitel 4.2.3, *Koordinaten-Transformationen*, Abschnitt *World-Transformationen*.

QWorkspace

Definiert in *qworkspace.h*, abgeleitet von *QWidget*.

Diese Klasse ermöglicht es, mehrere Unter-Widgets innerhalb des *QWorkspace*-Widgets wie einzelne Fenster darzustellen. Die Unter-Widgets erhalten eine eigene Fensterdekoration und können verschoben und in der Größe geändert werden. Diese Klasse wird üblicherweise zur Realisierung eines Multi Document Interface (MDI) benutzt. Ein Beispiel für die Anwendung finden Sie in Kapitel 3.5.7, *Applikation für mehrere Dokumente*.

QXMLReader

Definiert in *qxml*, enthalten im Modul *XML*.

Diese abstrakte Klasse stellt das Grundgerüst für einen XML-Parser nach SAX2-Norm dar. Um selbst einen Parser für XML-Dateien zu erstellen, können Sie meist auch die einfachere Klasse *QXMLSimpleReader* benutzen. Im Gegensatz zur Klasse *QDomDocument* wird bei diesem Ansatz die XML-Datei nicht komplett eingelesen, sondern sie wird schrittweise gelesen. Beim Auffinden eines Tags wird eine entsprechende Methode aufgerufen, die so überschrieben sein muss, dass sie die gewünschte Reaktion ausführt. Bei diesem Ansatz müssen Sie eine neue Parser-Klasse erstellen. *QDomDocument* dagegen erstellt eine interne Datenstruktur, die Sie durchlaufen können.

A Lösungen zu den Übungsaufgaben

A.1 Lösungen zu Kapitel 2.2

Übung 2.1

Ersetzen Sie einfach in der Zeile

```
QLabel *l = new QLabel ("<H1>Hallo, Welt!</H1>", 0);
```

das Wort »Welt« durch Ihren Vornamen, und schon spricht das Programm Sie persönlich an.

Für längere Texte empfiehlt es sich, der Übersichtlichkeit halber den Text in mehrere Zeilen aufzuteilen. Dazu beendet man die Zeile mit den Anführungszeichen und beginnt die nächste wiederum mit Anführungszeichen. Der C++-Compiler fasst dann diese Teiltex te zu einem einzigen String zusammen. Die Länge dieses Strings ist zunächst nicht begrenzt, obwohl einige Compiler häufig maximal 32.768 oder 65.536 Zeichen zulassen.

Die folgende Zeile erzeugt ein Fenster wie in Abbildung A.1. Es enthält eine Aufzählung, eine kleine Tabelle sowie verschiedene Textformatierungen.

```
QLabel *l = new QLabel  
( "<QT>"  
  "<H1><CENTER>Qt-RichText</CENTER></H1>"  
  "<H3>Eine Aufzählung:</H3>"  
  "<OL><LI>Eins</LI><LI>Zwei</LI><LI>Drei</LI></OL>"  
  "<H3>Eine Tabelle:</H3>"  
  "<TABLE><TR BGCOLOR=#d0d0ff>"  
  "<TH>Mai</TH><TH>Juni</TH><TH>Juli</TH>"  
  "</TR><TR BGCOLOR=#ffd0d0>"  
  "<TD>DM 237,-</TD><TD>DM 113,-</TD><TD>DM 55,50</TD>"  
  "</TR></TABLE>"  
  "<H3>Verschiedene Formatierungen:</H3>"  
  "<B>fett</B>, <U>unterstrichen</U>, <I>kursiv</I>"  
  "</QT>", 0);
```

Wie Sie sehen, heißt das umschließende Tag-Paar nicht `<HTML>...</HTML>`, sondern `<QT>...</QT>`. Ansonsten sind die Ähnlichkeiten zu HTML aber sehr groß. Auch das Einbinden von Bildern per `<IMG>`-Tag und Links mit dem `<A>`-Tag sind im Qt-RichText möglich, allerdings funktionieren sie bei diesem extrem einfachen Anzeigeelement *QLabel* nicht. Die mächtigeren Klassen, wie *QTextViewer* und *QTextBrowser*, können auch das.



Abbildung A-1 Komplexere Textdarstellung

Übung 2.2

Beim Starten hat das Fenster eine Größe, die gerade den vollständigen Text darstellt. Diese Fenstergröße kann aber mit der Maus verändert werden. Sie können das Fenster auch so klein ziehen, dass nicht mehr der gesamte Text dargestellt wird. Je nach Betriebssystem und Window-Manager ist die absolute Minimalgröße für ein Fenster unterschiedlich: Microsoft Windows hat eine Mindestbreite von ca. 100 Pixeln, während die Höhe bis auf 0 Pixel reduziert werden kann. Der KDE Window-Manager *kwin* hat als Minimalbreite und -höhe ca. 20 Pixel. Eine Maximalgröße für das Fenster gibt es nicht.

Bei der Größenänderung bleibt der Text am linken Rand des Fensters vertikal zentriert. Der Text im Fenster wird automatisch umbrochen, wenn das Fenster zu schmal wird. Der Umbruch findet dabei nur in Leerzeichen statt, nicht innerhalb der Wörter.

Übung 2.3

Ohne Tags wird der Text klein dargestellt, in der Standardtextgröße. Es findet auch kein automatischer Zeilenumbruch mehr statt. Wenn Sie nun mehrzeiligen Text erzeugen wollen, fügen Sie Zeilenvorschubzeichen (, \n') an den Umbruchstellen ein.

Durch Einfügen des Befehls *setFont* wird eine neue Schriftart gewählt, hier mit Schriftgröße 48. Sie können hier auch andere Schriftarten verwenden, die auf Ihrem System installiert sind. Durch Einfügen des Befehls *setAlignment* wird der Text zentriert statt linksbündig innerhalb des Fensters dargestellt.

Die Reihenfolge dieser beiden Zeilen hat keine Auswirkung auf die Darstellung. Beide Zeilen müssen allerdings nach der Zeile zur Erzeugung des Fensters stehen, da vorher die Variable *l* noch gar nicht bekannt ist und das *QLabel*-Objekt noch nicht existiert. Steht der Befehl *setFont* hinter der Befehl *show*, so ist die anfängliche Fenstergröße falsch: Sie wird beim Aufruf von *show* festgelegt. Wird anschließend die Schriftgröße verändert, stimmt der berechnete Wert nicht mehr.

Fügt man wieder Tags ein, so hat die Wahl der Schriftart keine Wirkung mehr: In RichText-Darstellungen werden nur die in den Tags festgelegten Schriftspezifikationen benutzt. Der Befehl *setAlignment* (*Qt::AlignCenter*) bewirkt aber, dass der ganze Textblock innerhalb des Fensters zentriert wird. Werden mehrere Textzeilen dargestellt, so bilden sie aber weiterhin einen einheitlichen linken Rand.

Übung 2.4

Wenn Sie den Befehl *app.setMainWidget(l)*; entfernen, wird das Programm nicht mehr beendet, wenn Sie das Fenster schließen. Wenn Sie das Programm von der Kommandozeile aus gestartet haben, so merken Sie das daran, dass die Eingabe im Terminal weiterhin blockiert ist.

Um das Programm nun zu beenden, müssen Sie den Prozess gezielt abbrechen. Unter Linux oder einem anderen Unix-Betriebssystem lassen Sie sich dazu mit dem Programm *ps* alle Prozesse anzeigen und beenden dann mit dem Programm *kill* den unnützen Prozess. Unter Microsoft Windows benutzen Sie zum Beenden den Task-Manager.

Es gibt eine ganze Reihe von Möglichkeiten, wie man das Programm beendet. Kapitel 3.3.1, *Qt-Applikationen*, geht im Abschnitt *Beenden des Programms* auf einige dieser Möglichkeiten ein.

Übung 2.5

Für zwei Fenster erzeugen Sie einfach zwei *QLabel*-Objekte:

```
#include <qapplication.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l1 = new QLabel ("<H1>Fenster 1!</H1>", 0);
    l1->show();
    QLabel *l2 = new QLabel ("<H1>Fenster 2!</H1>", 0);
    l2->show();
    app.setMainWidget (l1);
    return app.exec();
}
```

Für beide müssen Sie die *show*-Methode aufrufen, damit sie auch angezeigt werden. Die Header-Datei *qlabel.h* muss natürlich nur einmal eingebunden werden.

Das Programm wird nun beendet, nachdem das Fenster mit dem Inhalt »Fenster 1!« geschlossen wurde. Dadurch wird auch automatisch Fenster 2 geschlossen. Schließen Sie dagegen Fenster 2 zuerst, hat das noch keine Auswirkung.

Nur ein Fenster kann Hauptfenster sein. Wenn Sie versuchen, durch zwei Aufrufe der Methode *setMainWidget* beide Fenster zum Hauptfenster zu erklären, zählt der letzte Aufruf.

Ersetzen Sie die Zeile

```
app.setMainWidget (l1);
```

durch die Zeile:

```
QObject::connect (&app, SIGNAL (lastWindowClosed()),
                  &app, SLOT (quit()));
```

Nun ist keines der beiden Fenster mehr Hauptfenster, aber das Programm wird automatisch beendet, sobald alle Fenster geschlossen sind. Die Fenster sind nun also völlig gleichberechtigt. Diese Zeile brauchen Sie noch nicht zu verstehen: Sie benutzt das Signal-Slot-Konzept, das ausführlich in Kapitel 3.1.2 besprochen wird.

Für zehn Fenster ist es natürlich einfacher, eine Schleife zu benutzen:

```
#include <qapplication.h>
#include <qlabel.h>

int main (int argc, char **argv)
{
    QApplication app (argc, argv);
    QLabel *l;
    for (int i = 0; i < 10; i++)
    {
        l = new QLabel ("<H1>Ein Fenster</H1>", 0);
        l->show();
    }
    QObject::connect (&app, SIGNAL (lastWindowClosed()),
                    &app, SLOT (quit()));
    return app.exec();
}
```

Die Variable *l* wird hier einfach zehnmal mit einem neuen Objekt belegt. Da alle Fenster im Destruktor des *QApplication*-Objekts automatisch gelöscht werden, brauchen wir uns die Fenster nicht zu merken. Wollten wir sie von Hand löschen, so müssten die Zeiger der erzeugten *QLabel*-Objekte in einem Array oder einer Liste abgespeichert werden.

In unserem Fall haben alle Fenster den gleichen Textinhalt. Mit etwas mehr Aufwand kann man natürlich jedes Fenster auch mit einem individuellen Text versehen.

A.2 Lösungen zu Kapitel 3.1

Übung 3.1

Das Programm kann beispielsweise so aussehen:

```
#include <qobject.h>
#include <qobjectlist.h>
#include <qapplication.h>
#include <iostream.h>

QObject *topParent (QObject *obj)
{
    // Solange noch ein Vorgesetzter vorhanden ist,
    // in der Hierarchie hochlaufen
    while (obj->parent())
        obj = obj->parent();

    // Ergebnis zurückgeben
    return obj;
}

void printDirectChildren (QObject *obj)
{
    // Nur wenn Objekt existiert
    // und dessen Kinder-Liste ebenfalls
    if (obj && obj->children())
    {
        // Kopie der Kinder-Liste anlegen
        QObjectList list = *(obj->children());

        // Kinder-Liste durchlaufen
        QObject *aktuell;
        for (aktuell = list.first(); aktuell;
            aktuell = list.next())
        {
            // Name ausgeben
            cout << aktuell->name() << endl;
        }
    }
}

void printAllChildren (QObject *obj)
{

```

```
// Nur wenn Objekt existiert
// und dessen Kinder-Liste ebenfalls
if (obj && obj->children())
{
    // Kopie der Kinder-Liste anlegen
    QList<QObject> list = *(obj->children());

    // Liste durchlaufen
    QObject *aktuell;
    for (aktuell = list.first(); aktuell;
         aktuell = list.next())
    {
        // ein Kind ausgeben
        cout << aktuell->name() << endl;
        // und rekursiv dessen Kinder ausgeben
        printAllChildren (aktuell);
    }
}

int main (int argc, char **argv)
{
    // Als Erstes immer ein QApplication-Objekt anlegen
    QApplication app (argc, argv);

    // Firmenhierarchie anlegen
    QObject *obj1 =
        new QObject (0, "Direktor Herr Dr. Klöbner");
    QObject *obj2 =
        new QObject (obj1, "Chefsekretärin Frau Hansen");
    QObject *obj3 =
        new QObject (obj2, "Sekretärin Frau Peters");
    QObject *obj4 =
        new QObject (obj2, "Sekretärin Frau Dann");
    QObject *obj5 =
        new QObject (obj1, "Abteilungsleiterin Frau Kurz");
    QObject *obj6 =
        new QObject (obj5, "Chefentwickler Herr Hobel");
    QObject *obj7 =
        new QObject (obj6, "Ingenieur Herr Maren");
    QObject *obj8 =
        new QObject (obj5, "Vorarbeiter Herr Meier");
    QObject *obj9 =
        new QObject (obj8, "Arbeiter Herr Völkner");
    QObject *obj10 =
        new QObject (obj8, "Auszubildende Frau Dorn");
```

```

// Testweise die ganze Hierarchie ausgeben
cout << "Die Firma :" << endl;
obj1->dumpObjectTree();
cout << endl;

// Aufgabe a)
cout << "obj6 hat den Namen:" << endl;
cout << obj6->name() << endl << endl;

// Aufgabe b)
cout << "Der Vorgesetzte von obj8 ist:" << endl;
cout << obj8->parent()->name() << endl << endl;

// Aufgabe c)
cout << "Der höchste Chef von obj4 ist:" << endl;
cout << topParent (obj4)->name() << endl << endl;

// Aufgabe d)
cout << "Ausgabe aller direkten Untergebenen "
      "von obj5:" << endl;
printDirectChildren (obj5);
cout << endl;

// Aufgabe e)
cout << "Ausgabe ALLER Untergebenen von obj1:" << endl;
printAllChildren (obj1);
cout << endl;

// Aufgabe f)
cout << "Neuer Stand der Firma nach "
      "\"delete obj6\":" << endl;
delete obj6;
obj1->dumpObjectTree();
}

```

Zu Frage f)

Am einfachsten mit:

```
delete obj6;
```

Dadurch wird *obj7* automatisch ebenfalls gelöscht, da es Kind von *obj6* ist.

Auch Folgendes ist möglich:

```
delete obj7;
delete obj6;
```

Dann wird *obj7* zunächst gelöscht und aus der Liste der Children von *obj6* entfernt, anschließend wird *obj6* gelöscht.

Folgende Lösung funktioniert dagegen *nicht*:

```
delete obj6;
delete obj7;
```

Das produziert einen Laufzeitfehler, da bei der Ausführung von *delete obj7* die Instanz bereits gelöscht ist.

Zu Frage g)

Nein, da die Hierarchie nachträglich nicht mehr umgestellt werden kann. Die einzige Möglichkeit ist, zunächst *obj2*, *obj3* und *obj4* zu löschen und anschließend Frau Peters und Frau Dann wieder zu erzeugen.

Übung 3.2

Bei einem weiteren Aufruf des *connect*-Befehls wird eine zweite Verbindung parallel zur ersten aufgebaut. Eine Betätigung des Buttons ruft damit zweimal den Slot *countUp* auf, der Zähler wird also jedes Mal um 2 erhöht.

Beim Aufruf von *disconnect* werden anschließend beide Verbindungen wieder aufgelöst, da sie beide auf die Angaben passen.

Übung 3.3

Der *connect*-Befehl verbindet das Signal *send* mit sich selbst. Verschickt man nun ein Signal, so aktiviert es sich selbst, aktiviert sich ein weiteres Mal und so weiter. Das Programm wird also in einer Endlosschleife gefangen, bis es durch die Methodenaufrufe zu einem Stack-Überlauf kommt.

Übung 3.4

Betrachten wir die Beispielverbindung zwischen Objekt *o1* mit Signal *s* und Objekt *o2* mit Slot *r*, beide ohne Parameter. Jede der folgenden Zeilen bewirkt die Auflösung der Verbindung. Welche Variante Sie wählen, bleibt Ihnen selbst überlassen.

- statische *disconnect*-Variante:

```
QObject::disconnect (&o1, SIGNAL (s()), &o2, SLOT (r()));
QObject::disconnect (&o1, 0, &o2, SLOT (r()));
QObject::disconnect (&o1, SIGNAL (s()), 0, SLOT (r()));
QObject::disconnect (&o1, SIGNAL (s()), &o2, 0);
QObject::disconnect (&o1, 0, 0, SLOT (r()));
QObject::disconnect (&o1, 0, &o2, 0);
QObject::disconnect (&o1, SIGNAL (s()), 0, 0);
QObject::disconnect (&o1, 0, 0, 0);
```

- nicht-statische Variante 1:

```
o1.disconnect (SIGNAL (s()), &o2, SLOT (r()));
o1.disconnect (SIGNAL (s()), 0, SLOT (r()));
```

```

o1.disconnect (0, &o2, SLOT (r()));
o1.disconnect (0, 0, SLOT (r()));
o1.disconnect (SIGNAL (s()), &o2, 0);
o1.disconnect (SIGNAL (s()), &o2);
o1.disconnect (SIGNAL (s()), 0, 0);
o1.disconnect (SIGNAL (s()), 0);
o1.disconnect (SIGNAL (s()));
o1.disconnect (0, &o2, 0);
o1.disconnect (0, &o2);
o1.disconnect (0, 0, 0);
o1.disconnect (0, 0);
o1.disconnect (0);
o1.disconnect ();

```

- nicht-statische Variante 2:

```

o1.disconnect (&o2, SLOT (r()));
o1.disconnect (&o2, 0);
o1.disconnect (&o2);

```

Übung 3.5

Es ist grundsätzlich in C++ nicht verboten, eine Methode als *inline* zu deklarieren, auch wenn man die Adresse der Methode ermittelt, wie es zum Beispiel im Signal/Slot-Konzept nötig ist. In diesem Fall wird zusätzlich eine Nicht-*inline*-Variante der Methode kompiliert, man erreicht also keinen Geschwindigkeitsgewinn. Wird der Slot allerdings nicht nur durch Signale, sondern auch direkt aufgerufen, kann die Deklaration als *inline* einen Vorteil bringen.

Übung 3.6

```
object.isA ("QFrame")
```

liefert FALSE, da *object* eine Instanz der Klasse *QObject* ist, nicht der Klasse *QFrame*

```
timer.isA ("QTimer")
```

liefert TRUE

```
frame.isA ("QWidget")
```

liefert FALSE, da *frame* eine Instanz der Klasse *QFrame* ist, nicht der Klasse *QWidget*

```
widget.isA ("QWidget")
```

liefert FALSE, da die Groß-/Kleinschreibung relevant ist

```
frame.inherits ("QWidget")
```

liefert TRUE, da *QFrame* von *QWidget* abgeleitet ist

```
frame.inherits ("QObject")
```

liefert TRUE, da *QFrame* von *QWidget* abgeleitet ist, das seinerseits von *QObject* abgeleitet ist

```
widget.isherits ("QFrame")
```

liefert FALSE, da *QWidget* nicht Unterklasse von *QFrame* ist, sondern umgekehrt

```
timer.inherits ("QTimer")
```

liefert TRUE nach der Definition von *inherits*

```
timer.inherits ("QObject")
```

liefert TRUE und sollte für alle Klassen TRUE liefern, da die Methode *inherits* nur dann existiert, wenn die Klasse von *QObject* abgeleitet ist

```
timer.isWidgetType()
```

liefert FALSE, da *QTimer* nicht von *QWidget*, sondern nur von *QObject* abgeleitet ist

```
frame.isWidgetType()
```

liefert TRUE, da *QFrame* von *QWidget* abgeleitet ist

A.3 Lösungen zu Kapitel 3.6

Übung 3.7

Das erste Fenster spaltet sich in einen linken und einen rechten Teil, wobei der rechte nochmals aus drei übereinander liegenden Teilen besteht. Der Code dazu sieht so aus:

```
QHBoxLayout *top = new QHBoxLayout (0);

QMultiLineEdit *ml = new QMultiLineEdit (top);
QVBoxLayout *buttons = new QVBoxLayout (top);

QPushButton *b1 = new QPushButton ("Add new... ", buttons);
QPushButton *b2 = new QPushButton ("Compress", buttons);
QPushButton *b3 = new QPushButton ("Delete", buttons);
```

Das zweite Fenster hat noch eine Hierarchiestufe mehr: Es ist in einen oberen und einen unteren Teil untergliedert, der untere wiederum in einen linken und

einen rechten. Der rechte Teil unten spaltet sich wiederum in drei übereinander liegende Teile auf. Der Code dazu sieht wie folgt aus:

```
QVBox *top = new QVBox (0);

QMultiLineEdit *m1l = new QMultiLineEdit (top);
QHBox *part = new QHBox (top);

QMultiLineEdit *m1 = new QMultiLineEdit (part);
QVBox *buttons = new QVBox (part);

QPushButton *b1 = new QPushButton ("Add new... ", buttons);
QPushButton *b2 = new QPushButton ("Compress", buttons);
QPushButton *b3 = new QPushButton ("Delete", buttons);
```

Übung 3.8

Wir benutzen hier ein *QGrid*-Objekt mit drei Spalten. Mit der Methode *skip* kann man ein Element im Gitter frei lassen. Wir benutzen daher *skip* unmittelbar vor dem Einfügen der »0«. Nach der »0« könnte man noch einmal *skip* aufrufen, das ist aber nicht nötig, da wir keine weiteren Widgets einfügen wollen.

```
QGrid *top = new QGrid (3);
QPushButton *b1 = new QPushButton ("1", top);
QPushButton *b2 = new QPushButton ("2", top);
QPushButton *b3 = new QPushButton ("3", top);
QPushButton *b4 = new QPushButton ("4", top);
QPushButton *b5 = new QPushButton ("5", top);
QPushButton *b6 = new QPushButton ("6", top);
QPushButton *b7 = new QPushButton ("7", top);
QPushButton *b8 = new QPushButton ("8", top);
QPushButton *b9 = new QPushButton ("9", top);
top->skip();
QPushButton *b0 = new QPushButton ("0", top);
```

Übung 3.9

Auch hier verwenden wir *QGrid*, wiederum mit drei Spalten. Hier müssen wir *skip* allerdings etwas häufiger anwenden.

```
QGrid *top = new QGrid (3);
top->skip();
QPushButton *b1 = new QPushButton (top);
b1->setPixmap (QPixmap ("up.png"));
top->skip();
QPushButton *b2 = new QPushButton (top);
b2->setPixmap (QPixmap ("back.png"));
top->skip();
QPushButton *b3 = new QPushButton (top);
b3->setPixmap (QPixmap ("forward.png"));
```

```
top->skip();
QPushButton *b4 = new QPushButton (top);
b4->setPixmap (QPixmap ("down.png"));
```

Um die Bilder einzubinden, benötigen Sie entsprechende Grafikdateien. Diese können Sie selbst mit einem Programm wie *KIconEdit* erstellen, oder Sie benutzen die bereits vorgefertigten Icons des KDE-Systems. In unserem Fall können Sie beispielsweise die Dateien *up.png*, *down.png*, *back.png* und *forward.png* nutzen. Diese befinden sich im Verzeichnis *\$KDEDIR/share/icons/locolor/16x16/actions*. In anderen Unterverzeichnissen finden Sie die gleichen Dateien, allerdings mit einer anderen Auflösung oder Farbanzahl. Kopieren Sie diese Dateien in das aktuelle Verzeichnis.

Übung 3.10

Statt eines *QVBox*-Objekts kann man auch ein *QGrid*-Objekt mit nur einer Spalte benutzen:

```
QGrid *vbox = new QGrid (1, parent, name);
```

Statt eines *QHBox*-Objekts benutzt man ein *QGrid*-Objekt mit einer Zeile:

```
QGrid *hbox = new QGrid (1, QGrid::Vertical, parent, name);
```

Übung 3.11

- a) Das folgende Listing wird als Grundlage für die Teilaufgaben b) bis i) benutzt, indem nur der Teil für das Einfügen in das Layout-Objekt neu geschrieben wird.

```
// Vater-Widget erzeugen
QWidget *top = new QWidget ();

// Zwei Unter-Widgets erzeugen
QPushButton *b1 = new QPushButton ("B1", top);
QPushButton *b2 = new QPushButton ("B2", top);

// Layout-Objekt erzeugen (zu top)
QHBoxLayout *l = new QHBoxLayout (top);

// Widgets einfügen
l->addWidget (b1);
l->addWidget (b2);

// Toplevel-Widget anzeigen
top->show();
```

- b) Hier gibt es zwei Lösungen. Man kann den Parameter *autoBorder* im Konstruktor von *QHBoxLayout* nutzen, um Platz zwischen den Widgets zu schaffen:

```
QHBoxLayout *l = new QHBoxLayout (top, 0, 20);
```

Oder man fügt den Zwischenraum mit *addSpacing* von Hand ein:

```
l->addWidget (b1);  
l->addSpacing (20);  
l->addWidget (b2);
```

- c) Button 2 erhält allen überschüssigen Platz, wenn er einen Streckungsfaktor größer als 0 erhält:

```
l->addWidget (b1);  
l->addWidget (b2, 1);
```

- d) Der überschüssige Platz kann frei bleiben, indem man *addStretch* benutzt:

```
l->addStretch (1);  
l->addWidget (b1);  
l->addWidget (b2);
```

- e) Die Buttons bleiben am Rand und minimal, wenn man *addStretch* in der Mitte benutzt:

```
l->addWidget (b1);  
l->addStretch (1);  
l->addWidget (b2);
```

- f) Der freie Platz wird gleichmäßig auf die Buttons und den Zwischenraum verteilt, wenn alle drei den gleichen Streckungsfaktor besitzen:

```
l->addWidget (b1, 1);  
l->addStretch (1);  
l->addWidget (b2, 1);
```

Beachten Sie, dass die Buttons größer sind als der Zwischenraum, da sie *zusätzlich* zu ihrer minimalen Breite ein Drittel des Freiraums bekommen.

- g) Die Buttons werden folgendermaßen in der Mitte zentriert:

```
l->addStretch (1);  
l->addWidget (b1);  
l->addWidget (b2);  
l->addStretch (1);
```

- h) Die Buttons stehen in ästhetisch günstigem Abstand, wenn der Freiraum gleichmäßig zwischen ihnen verteilt wird und ihre eigene Breite sich dem anpasst:

```
l->addStretch (1);  
l->addWidget (b1, 1);  
l->addStretch (1);  
l->addWidget (b2, 1);  
l->addStretch (1);
```

- i) Ein typisches Beispiel für die Buttons unter einem Dialogfenster: Am linken Rand ein Button für die Kontexthilfe, am rechten Rand die Buttons zum Akzeptieren, Anwenden oder Abbrechen.

```

QWidget *top = new QWidget ();
// Hier noch Elemente, die im Fenster angezeigt werden
// sollen
// ...

// Dann die Buttons für das untere Ende
// Auch Unter-Widgets von top
QPushButton *b1 = new QPushButton ("Help", top);
QPushButton *b2 = new QPushButton ("Ok", top);
QPushButton *b3 = new QPushButton ("Apply", top);
QPushButton *b4 = new QPushButton ("Cancel", top);

// Haupt-Layout-Objekt erzeugen (zu top)
QVBoxLayout *topLayout = new QVBoxLayout (top);

// Widgets des Dialogfensters einfügen
// ...

// Layout-Objekt für die Buttons erzeugen
// Kleiner Zwischenraum zwischen den Buttons
QHBoxLayout *buttons = new QHBoxLayout (5);

// In das Haupt-Layout-Objekt einfügen
topLayout->addLayout (buttons);

// Buttons einfügen
buttons->addWidget (b1);
buttons->addStretch (1);
buttons->addWidget (b2);
buttons->addWidget (b3);
buttons->addWidget (b4);

```

Übung 3.12

Das Fenster besteht aus drei nebeneinander liegenden Teilen. Der linke und der rechte Teil enthalten jeweils ein *QMultiLineEdit*-Objekt, der mittlere Teil enthält zwei untereinander liegende Buttons. Alle vier Widgets sind also direkte Unter-Widgets des Vater-Widgets. Das Listing sieht etwa wie folgt aus:

```

QWidget *top = new QWidget ();
// Dann die Buttons für das untere Ende
// Auch Unter-Widgets von top
QMultiLineEdit *m1 = new QMultiLineEdit (top);
QMultiLineEdit *m2 = new QMultiLineEdit (top);
QPushButton *b1 = new QPushButton ("Add >>", top);
QPushButton *b2 = new QPushButton ("<< Remove", top);

```

```
// Haupt-Layout-Objekt erzeugen (zu top)
// Abstand zum Rand und zwischen den Elementen 10 Pixel
QHBoxLayout *topLayout = new QHBoxLayout (top, 10);
// Button-Layout-Objekt erzeugen (Unter-Layout)
// Übernimmt Abstand zwischen den Elementen von topLayout
QVBoxLayout *buttons = new QVBoxLayout ();

// Zuerst topLayout füllen
topLayout->addWidget (m1);
topLayout->addLayout (buttons);
topLayout->addWidget (m2);

// Dann buttons füllen
buttons->addStretch (1);
buttons->addWidget (b1);
buttons->addWidget (b2);
buttons->addStretch (1);
```

Übung 3.13

a) Das folgende Listing dient wieder als Grundlage für die Aufgabenteile b) bis d)

```
// Vater-Widget erzeugen
QWidget *top = new QWidget ();

// View Unter-Widgets erzeugen
QPushButton *b1 = new QPushButton ("Button 1", top);
QPushButton *b2 = new QPushButton ("Second Button", top);
QPushButton *b3 = new QPushButton ("3", top);
QPushButton *b4 = new QPushButton ("Number 4", top);

// Layout-Objekt erzeugen (zu top)
QGridLayout *l = new QGridLayout (top, 2, 2);

// Widgets an festgelegten Stellen einfügen
l->addWidget (b1, 0, 0);
l->addWidget (b2, 0, 1);
l->addWidget (b3, 1, 0);
l->addWidget (b4, 1, 1);

// Toplevel-Widget anzeigen
top->show();
```

Beachten Sie, dass die beiden Spalten nicht gleich breit sind. Der Grund dafür ist der Button »Second Button«. Er hat bereits eine so große Mindestbreite, dass die Mindestbreite der rechten Spalte größer ist als die Mindestbreite der linken Spalte. Der *verbleibende* Platz wird dann genau zur Hälfte auf die linke und die rechte Spalte verteilt.

- b) Die rechte Spalte übernimmt den vollständigen Restplatz, wenn man den Streckfaktor auf einen Wert größer als 0 setzt (unmittelbar vor `l->activate()`):

```
l->setColStretch (1, 1);
```

- c) Damit sich die beiden Spalten an den linken und rechten Rand drängen, fügen wir eine leere Spalte in der Mitte ein, der wir den Streckungsfaktor 1 geben. Diese Spalte erhält dann den gesamten restlichen Platz, da die beiden anderen Spalten den Faktor 0 haben:

```
QGridLayout *l = new QGridLayout (top, 2, 3);
l->addWidget (b1, 0, 0);
l->addWidget (b2, 0, 2);
l->addWidget (b3, 1, 0);
l->addWidget (b4, 1, 2);
l->setColStretch (1, 1);
```

- d) Damit wir in der Mitte einen kleinen, festen Zwischenraum bekommen, fügen wir eine leere Spalte ein, der wir mit `addColSpacing` eine Mindestbreite geben. Damit aber die beiden Button-Spalten sich den restlichen Platz teilen und die mittlere Spalte dabei leer ausgeht, müssen die Button-Spalten einen Streckungsfaktor größer als 0 bekommen:

```
QGridLayout *l = new QGridLayout (top, 2, 3);
l->addWidget (b1, 0, 0);
l->addWidget (b2, 0, 2);
l->addWidget (b3, 1, 0);
l->addWidget (b4, 1, 2);
l->addColSpacing (1, 10);
l->setColStretch (0, 1);
l->setColStretch (2, 1);
```

Übung 3.14

Nehmen wir an, das Vater-Widget ist in der Variablen *top* abgelegt und die Buttons (alle direkte Unter-Widgets von *top*) in den Variablen *b1* bis *b9*, *b0* und *bkomma*. *b0* muss dann zwei Tabellenplätze belegen, nämlich von Zeile 3 bis Zeile 4 (also nur eine Zeile) und von Spalte 0 bis Spalte 1 (also zwei Spalten). Das Listing sieht dann so aus:

```
QGridLayout *l = new QGridLayout (top, 4, 3);
l->addWidget (b1, 0, 0);
l->addWidget (b2, 0, 1);
l->addWidget (b3, 0, 2);
l->addWidget (b4, 1, 0);
l->addWidget (b5, 1, 1);
l->addWidget (b6, 1, 2);
l->addWidget (b7, 2, 0);
```

```

l->addWidget (b8, 2, 1);
l->addWidget (b9, 2, 2);
l->addMultiCellWidget (b0, 3, 3, 0, 1);
l->addWidget (bkomma, 3, 2);

```

Übung 3.15

Für die Lösung mit *QBoxLayout* brauchen wir zwei Layout-Elemente:

```

QVBoxLayout *topLayout = new QVBoxLayout (top);
QHBoxLayout *firstRow = new QHBoxLayout ();

topLayout->addLayout (firstRow);
topLayout->addWidget (b3);

firstRow->addWidget (b1);
firstRow->addWidget (b2);

```

Die Lösung mit *QGridLayout* ist kürzer, da wir nur ein Layout-Objekt benötigen. Der untere Button ist dann ein Widget, das zwei nebeneinander liegende Tabellenplätze belegt, von Zeile 1 bis Zeile 1 und von Spalte 0 bis Spalte 1:

```

QGridLayout *topLayout = new QGridLayout (top, 2, 2);
l->addWidget (b1, 0, 0);
l->addWidget (b2, 0, 1);
l->addMultiCellWidget (b3, 1, 1, 0, 1);

```

A.4 Lösungen zu Kapitel 4.1

Übung 4.1

Bis auf die Farbe Violett sind bei allen Farben Sättigung und Helligkeit auf Maximalwerte gesetzt. Violett muss dagegen etwas dunkler gehalten werden, da es sonst zu hell ist und eher der Farbe Lila oder Magenta ähnelt. Die HSV- und RGB-Werte sind in Tabelle A.1 aufgelistet.

Farbe	HSV-Werte	RGB-Werte
Rot	(0, 255, 255)	(255, 0, 0)
Orange	(30, 255, 255)	(255, 128, 0)
Gelb	(60, 255, 255)	(255, 255, 0)
Grün	(120, 255, 255)	(0, 255, 0)
Blau	(240, 255, 255)	(0, 0, 255)
Violett	(280, 255, 170)	(115, 0, 168)

Tabelle A-1 HSV- und RGB-Werte für die Regenbogenfarben

Übung 4.2

Die RGB-Werte (255, 255, 128) beschreiben eine Farbe mit vollem Rot- und Grünanteil sowie einem halben Blauanteil. Der Farbton ist also Gelb, durch den Blauanteil jedoch deutlich in Richtung Weiß verschoben. Es ergibt sich also ein helles, weißliches Pastellgelb. Hinter den RGB-Werten (80, 80, 80) verbirgt sich ein Grauton, da alle drei Farbanteile gleich sind. Da 80 nur etwa ein Drittel des möglichen Bereichs bis 255 ist, handelt es sich hier um ein dunkles Grau. Mit den RGB-Werten (180, 220, 180) wird wieder eine insgesamt helle Farbe festgelegt, da alle drei Farbanteile hoch sind. Da der Grünanteil größer ist als die Anteile von Rot und Blau, ergibt sich ein blasses Pastellgrün.

Die Werte (30, 220, 100) im HSV-Modell beschreiben einen Orange-Ton, was Sie am Wert für den Farbton (Hue) von 30° erkennen können, der zwischen Rot (0°) und Gelb (60°) liegt. Die Farbsättigung ist mit 220 recht hoch, was zu einem kräftigen Ton führt. Die Helligkeit ist mit 100 aber eher gering, so dass sich insgesamt ein Braunton ergibt. Die Farbe, die im HSV-Modell durch die Werte (330, 200, 80) festgelegt ist, liegt im Farbton mit 330° zwischen Magenta (300°) und Rot (0° = 360°). Die Sättigung ist mit 200 recht hoch, die Helligkeit mit 80 aber eher niedrig. Es ergibt sich somit ein dunkles, kräftiges Lila. Mit den Werten (0, 80, 255) wird eine Farbe beschrieben, die vom Farbton her Rot ist (0°), allerdings ist es bei einer geringen Sättigung von nur 80 eher blass. Durch die hohe Helligkeit von 255 wird die Farbe weißlich. Insgesamt ergibt sich ein helles Rosa.

Übung 4.3

Solange noch kein Objekt der Klasse *QApplication* erzeugt wurde, gibt es noch keine Verbindung zum X-Server. Farben können daher noch nicht allokiert werden. Die internen Umrechnungsroutinen der Klasse *QColor* funktionieren jedoch unabhängig davon; unser Programm braucht also kein *QApplication*-Objekt.

```
#include <qcolor.h>
#include <stdio.h>

void writeColor (const QColor &c) {
    printf ("RGB: (%d, %d, %d)\t", c.red (), c.green (),
                                                c.blue ());

    int h, s, v;
    c.hsv (&h, &s, &v);
    printf ("HSV: (%d, %d, %d)\n", h, s, v);
}

int main () {
    writeColor (QColor (255, 255, 128));
    writeColor (QColor (80, 80, 80));
    writeColor (QColor (180, 220, 180));
    writeColor (QColor (30, 220, 100, QColor::Hsv));
}
```



```

        writeColor (QColor (140, 75, 255, QColor::Hsv));
        writeColor (QColor (338, 135, 100, QColor::Hsv));
        return 0;
    }

```

Das Programm erzeugt die folgende Ausgabe:

```

RGB: (255, 255, 128)HSV: (60, 127, 255)
RGB: (80, 80, 80)   HSV: (-1, 0, 80)
RGB: (180, 220, 180)HSV: (120, 46, 220)
RGB: (100, 57, 14)  HSV: (30, 219, 100)
RGB: (80, 17, 49)   HSV: (330, 201, 80)
RGB: (255, 175, 175)HSV: (0, 80, 255)

```

Übung 4.4

Wir erzeugen die Buttons der Reihe nach und legen sie in einem *QVBox*-Widget ab.

```

QVBox *w = new QVBox (0);
QColor backColor =
    QApplication::palette().active().background();
for (int i = 0; i < 6; i++)
{
    QColor c;
    QString bezeichnung;
    switch (i)
    {
        case 0:
            c.setHsv (0, 255, 255);   bezeichnung = "Rot"; break;
        case 1:
            c.setHsv (30, 255, 255);  bezeichnung = "Orange"; break;
        case 2:
            c.setHsv (60, 255, 255);  bezeichnung = "Gelb"; break;
        case 3:
            c.setHsv (120, 255, 255); bezeichnung = "Grün"; break;
        case 4:
            c.setHsv (240, 255, 255); bezeichnung = "Blau"; break;
        case 5:
            c.setHsv (280, 255, 170); bezeichnung = "Violett"; break;
    }
    QPushButton *button = new QPushButton (bezeichnung, w);
    QPalette pal (c, backColor);
    button->setPalette (pal);
}
w->show();

```

Übung 4.5

Unsere neue Klasse *QLEDNumber* ist von *QLCDNumber* abgeleitet. Sie erbt die gesamte Funktionalität. Der einzige Unterschied ist, dass sie im Konstruktor die Widget-Palette ändert.

```
class QLEDNumber : public QLCDNumber
{
    Q_OBJECT
public:
    QLEDNumber (QWidget *parent = 0, const char *name = 0)
        : QLCDNumber (parent, name)
    {
        // Modus auf "Flat" setzen, also keine Schattierung am Rand
        setSegmentStyle (Flat);

        // Palette setzen: Background auf Schwarz, Foreground auf Rot
        QPalette p = QApplication::palette();
        // Setzt alle drei Gruppen (Active, Inactive und Disabled)
        p.setColor (QColorGroup::Background, black);
        p.setColor (QColorGroup::Foreground, red);
        setPalette (p);
    }
};
```

A.5 Lösungen zu Kapitel 4.2

Übung 4.6

Um das Dreieck in der Hintergrundfarbe im weißen Kreis zu zeichnen, könnte man zunächst einen Kreis zeichnen und anschließend ein graues Dreieck (als Polygon mit drei Eckpunkten) dort hinein zeichnen. Dabei muss man jedoch zum einen die korrekte Füllfarbe für das Dreieck setzen. Zum anderen muss man die drei Eckpunkte des Dreiecks korrekt berechnen, was insbesondere bei den beiden unteren Punkten nicht so leicht ist, wenn sie wirklich genau auf der Kreislinie liegen sollen.

Stattdessen zeichnen wir drei Kreisabschnitte, einen von 90° (oben) bis 210° (links unten), einen von 210° bis 330° (rechts unten) und einen dritten von 330° bis 90°. Zusammen ergeben die Außenlinien der drei Kreisabschnitte einen vollen Kreis, und die inneren Linien ergeben das (nicht ausgefüllte) Dreieck. Der Code sieht in diesem Fall so aus:

```
QBrush brush (white);
p.setBrush (brush);
p.drawChord (60, 10, 80, 80, 90 * 16, 120 * 16);
p.drawChord (60, 10, 80, 80, 210 * 16, 120 * 16);
p.drawChord (60, 10, 80, 80, 330 * 16, 120 * 16);
```

Für das Yin-Yang-Symbol benötigen wir eine Reihe von halben Kreisen, die mit *drawChord* gezeichnet werden. Der Anfangswinkel ist im Fall eines Halbkreises mit Rundung nach links 90°, bei der Rundung nach rechts 270°. Als Zeichenstift setzen wir zunächst *NoPen*, damit die Halbkreise ohne Rand gezeichnet werden. Zum Schluss zeichnen wir um das Symbol noch einen Kreis ohne Füllung (*NoBrush*) und mit der Liniendicke 2.

```
p.setPen (NoPen);
p.setBrush (black);
p.drawChord (50, 0, 100, 100, 90 * 16, 180 * 16);
p.setBrush (white);
p.drawChord (50, 0, 100, 100, 270 * 16, 180 * 16);
p.drawChord (75, 50, 50, 50, 90 * 16, 180 * 16);
p.setBrush (black);
p.drawChord (70, 0, 50, 50, 270 * 16, 180 * 16);
p.drawEllipse (92, 67, 16, 16);
p.setBrush (white);
p.drawEllipse (92, 17, 16, 16);
p.setBrush (NoBrush);
p.setPen (QPen (black, 2));
p.drawEllipse (50, 0, 100, 100);
```

Um den Button mit Schattierungseffekten und runden Ecken zu zeichnen, benutzen wir hier zunächst für die eigentliche Button-Fläche die Methode *drawRoundRect*. Alternativ hätte man auch ein Rechteck und zwei Halbkreise zeichnen können.

Die Schattierungseffekte am Rand zeichnen wir mit Kreisabschnitten, die wir mit *drawArc* erzeugen. Die obere, weiße Kante besteht dabei aus einem achteil Kreis rechts oben, einer geraden Linie am oberen Rand und daran anschließend einem Dreiachteilkreis oben links. Die untere, dunkelgraue Kante besteht aus einem achteil Kreis unten links, einer geraden Linie unten und einem Dreiachteilkreis unten rechts. Alternativ hätte man auch hier die Methode *drawRoundRect* benutzen können, die man zweimal aufruft, einmal mit einem weißen und einmal mit einem dunkelgrauen Zeichenstift. Durch die Wahl eines geeigneten Zeichenbereichs mit *setClipRegion* hätte man dabei gewährleisten können, dass nur der gewünschte Teil der Außenlinie gezeichnet wird.

Anschließend wird noch die Beschriftung in den Button gezeichnet. Dazu setzen wir den Zeichensatz mit *setFont* und schreiben den Text zentriert (*AlignCenter*) und mit Unterstreichung (*ShowPrefix*).

```
p.setBrush (lightGray);
p.setPen (NoPen);
p.drawRoundRect (20, 20, 160, 60, 37, 99);
p.setPen (QPen (white, 2));
p.drawArc (120, 20, 60, 60, 45 * 16, 45 * 16);
```

```

p.drawLine (150, 20, 50, 20);
p.drawArc (20, 20, 60, 60, 90 * 16, 135 * 16);
p.setPen (QPen (gray, 2));
p.drawArc (20, 20, 60, 60, 225 * 16, 45 * 16);
p.drawLine (50, 80, 150, 80);
p.drawArc (120, 20, 60, 60, 270 * 16, 135 * 16);
p.setPen (black);
p.setFont (QFont ("helvetica", 34));
p.drawText (20, 20, 160, 60,
            AlignCenter | ShowPrefix, "&Ende");

```

A.6 Lösungen zu Kapitel 4.12

Übung 4.7

Die hier beschriebene Klasse *AlarmTimer* hat zwei Konstruktoren: Der erste Konstruktor nimmt die Weckzeit als *QDateTime*-Objekt entgegen, der zweite als *QTime*-Objekt.

Für den Fall, dass beim Erzeugen eines *AlarmTimer*-Objekts der Weckzeitpunkt bereits abgelaufen ist, wird ein Timer mit Zeitintervall 0 gestartet (siehe die *setTimerAppropriate*-Methode). Damit wird das Signal *wakeUp* erst gesendet, wenn *timerEvent* aufgerufen wird, also erst zu Beginn der Haupt-Event-Schleife. So hat das Objekt, das das *AlarmTimer*-Objekt erzeugt hat, noch ausreichend Gelegenheit, das Signal *wakeUp* mit einem Slot zu verbinden.

Unsere *AlarmTimer*-Klasse arbeitet für beliebige Zeitspannen korrekt (auch für mehr als 68 Jahre!) gibt den Alarm auf die Millisekunde genau aus (vorausgesetzt, die Rechneruhr ist entsprechend genau und ein anderer Prozess blockiert nicht gerade den Rechner). Es wird für lange Zeitspannen nur jede Minute ein Timer-Event geschickt, die benötigte CPU-Zeit ist also äußerst gering. Diese Klasse besitzt also eine extrem breite Anwendungspalette.

```

class AlarmTimer: public QObject
{
    Q_OBJECT
public:
    AlarmTimer (QDateTime wakeup, QObject *parent = 0,
                const char *name = 0);
    AlarmTimer (QTime wakeup, QObject *parent = 0,
                const char *name = 0);
    ~AlarmTimer () {}

    bool timeIsOver ()
    {return expire <= QDateTime::currentDateTime ();}

signals:

```

```

    void wakeUp ();

protected:
    void timerEvent (QTimerEvent *);

private:
    void setTimerAppropriate ();
    QDateTime expire;
};

AlarmTimer::AlarmTimer (QDateTime wakeup, QObject *parent,
                        const char *name):
    QObject (parent, name)
{
    // Alarmzeitpunkt speichern
    expire = wakeup;
    // Timer starten
    setTimerAppropriate ();
}

// Dieser Konstruktor nimmt eine Uhrzeit (QTime) statt
// eines Zeitpunkts (QDateTime)
AlarmTimer::AlarmTimer (QTime wakeup, QObject *parent,
                        const char *name):
    QObject (parent, name)
{
    if (QTime::currentTime () <= wakeup)
        expire = QDateTime (QDate::currentDate (), wakeup);
    else
        // Falls die angegebene Uhrzeit bereits vorbei ist,
        // wähle morgen.
        expire = QDateTime (QDate::currentDate ().addDays (1),
                            wakeup);
    setTimerAppropriate ();
}

void AlarmTimer::timerEvent (QTimerEvent *) {
    // Da es nur einen aktiven Timer gibt, brauchen wir
    // die ID des Timers nicht und lassen daher den
    // Parameter QTimerEvent aus.

    killTimers ();
    QDateTime now = QDateTime::currentDateTime();
    if (timeIsOver())
        // Falls Uhrzeit erreicht, Signal senden
        emit wakeUp ();
    else
        // sonst nächsten Timer setzen

```

```

        setTimerAppropriate ();
    }

void AlarmTimer::setTimerAppropriate () {
    // Setzt den Timer so, dass er zum Zeitpunkt des Weckens
    // aktiviert wird, spätestens jedoch nach einer Minute.
    QDateTime now = QDateTime::currentDateTime ();
    int restTime;

    // Wir testen zunächst, ob sich das Datum zwischen dem
    // aktuellen Zeitpunkt und dem Weckzeitpunkt um
    // mindestens zwei Tage unterscheidet. Dann sind es
    // noch mindestens 24 Stunden bis zum Ablauf des Timers.
    // Wir testen zunächst das Datum, da bei einer
    // Zeitspanne von mehr als 68 Jahren secsTo einen
    // Überlauf hat und somit nicht zuverlässig ist.
    if (now.daysTo (expire) >= 2 || // mehr als 24 Stunden
        now.secsTo (expire) > 60) // mehr als eine Minute
        restTime = 60 * 1000;      // dann nächster Test
                                   // in einer Minute
    else {
        int restTime = now.secsTo (expire) * 1000 +
                       expire.msecs () -
                       now.msecs ();
        // Falls bereits abgelaufen, Zeitintervall 0
        if (restTime < 0) restTime = 0;
    }
    startTimer (restTime);
}

```

Übung 4.8

Unser Testprogramm setzt pro Testzyklus zwei Timer. Der eine läuft eine Sekunde lang, der andere beim ersten Zyklus 0 Millisekunden, beim zweiten Zyklus eine Millisekunde usw. Die verstrichene Zeit wird gemessen, bis entweder der erste Timer abgelaufen ist (also maximal eine Sekunde) oder der zweite Timer 100 Events erzeugt hat. Nach jedem Zyklus wird ausgegeben, wie viele Events der zweite Timer erzeugt hat und wie viel Zeit dabei verstrichen ist sowie die durchschnittliche Zeit für einen Event des zweiten Timers. So kann man sich ein sehr genaues Bild verschaffen, ob die Timer in Qt wirklich exakt funktionieren.

Das Programm ist ein Beispiel für die Benutzung von zwei Timer-Events. Dazu werden die IDs der beiden Timer in den Variablen *maxTimeID* (für den ersten Timer, der eine Sekunde lang läuft) bzw. *timerID* (für den zweiten Timer) gespeichert. In der Methode *timerEvent* wird dann getestet, welcher Timer den Event verursacht hat.

```

#include <qapplication.h>
#include <qdatetime.h>

class TestTimerResolution: public QObject {
    Q_OBJECT

public:
    TestTimerResolution (QObject *parent=0,
                        const char* name=0):
        QObject (parent, name) {}
    ~TestTimerResolution () {}
    void start ();

signals:
    void done ();

protected:
    void timerEvent (QTimerEvent *event);

private:
    void runTest ();
    int timerID, maxTimeID;
    QTime stopWatch;
    int counter, current;
    bool maxTimeExpired;
};

void TestTimerResolution::start () {
    current = 0;
    runTest ();
}

void TestTimerResolution::timerEvent (QTimerEvent *event) {
    if (event->timerId () == timerID) {
        // Test-Timer: Anzahl der Events zählen
        counter++;
        if (counter >= 100 || maxTimeExpired) {
            killTimers ();
            int time = stopWatch.elapsed ();
            debug ("%4d ms Timer: %4d Events, %4d ms => "
                    "%.1f ms pro Event",
                    current, counter, time,
                    ((double) time) / counter);
            current++;
            if (current == 40)
                emit done ();
            else
                runTest ();
        }
    }
}

```

```
} else if (event->timerId () == maxTimeID)
    maxTimeExpired = TRUE;
}

void TestTimerResolution::runTest () {
    maxTimeExpired = FALSE;
    counter = 0;
    timerID = startTimer (current);
    maxTimeID = startTimer (1000);
    stopWatch.start ();
}

int main (int argc, char **argv) {
    QApplication a (argc, argv);
    TestTimerResolution *tt = new TestTimerResolution (&a);
    QObject::connect (tt, SIGNAL (done ()),
                     &a, SLOT (quit ()));
    tt->start ();
    return a.exec ();
}
```

Hier eine typische Ausgabe eines Linux-Systems:

```
0 ms Timer: 100 Events, 11 ms => 0.1 ms pro Event
1 ms Timer: 52 Events, 1034 ms => 19.9 ms pro Event
2 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
3 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
4 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
5 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
6 ms Timer: 51 Events, 1019 ms => 20.0 ms pro Event
7 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
8 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
9 ms Timer: 51 Events, 1020 ms => 20.0 ms pro Event
10 ms Timer: 60 Events, 1018 ms => 17.0 ms pro Event
11 ms Timer: 34 Events, 1020 ms => 30.0 ms pro Event
12 ms Timer: 34 Events, 1020 ms => 30.0 ms pro Event
13 ms Timer: 34 Events, 1019 ms => 30.0 ms pro Event
14 ms Timer: 34 Events, 1020 ms => 30.0 ms pro Event
15 ms Timer: 42 Events, 1040 ms => 24.8 ms pro Event
16 ms Timer: 41 Events, 1020 ms => 24.9 ms pro Event
17 ms Timer: 44 Events, 1029 ms => 23.4 ms pro Event
18 ms Timer: 47 Events, 1030 ms => 21.9 ms pro Event
19 ms Timer: 48 Events, 1020 ms => 21.2 ms pro Event
20 ms Timer: 51 Events, 1040 ms => 20.4 ms pro Event
21 ms Timer: 48 Events, 1020 ms => 21.2 ms pro Event
22 ms Timer: 46 Events, 1030 ms => 22.4 ms pro Event
23 ms Timer: 44 Events, 1020 ms => 23.2 ms pro Event
24 ms Timer: 42 Events, 1020 ms => 24.3 ms pro Event
25 ms Timer: 41 Events, 1045 ms => 25.5 ms pro Event
```


26 ms Timer:	39 Events, 1020 ms => 26.2 ms pro Event
27 ms Timer:	38 Events, 1040 ms => 27.4 ms pro Event
28 ms Timer:	36 Events, 1020 ms => 28.3 ms pro Event
29 ms Timer:	35 Events, 1020 ms => 29.1 ms pro Event
30 ms Timer:	34 Events, 1040 ms => 30.6 ms pro Event
31 ms Timer:	33 Events, 1039 ms => 31.5 ms pro Event
32 ms Timer:	32 Events, 1040 ms => 32.5 ms pro Event
33 ms Timer:	31 Events, 1040 ms => 33.5 ms pro Event
34 ms Timer:	30 Events, 1040 ms => 34.7 ms pro Event
35 ms Timer:	29 Events, 1020 ms => 35.2 ms pro Event
36 ms Timer:	28 Events, 1020 ms => 36.4 ms pro Event
37 ms Timer:	28 Events, 1049 ms => 37.5 ms pro Event
38 ms Timer:	27 Events, 1039 ms => 38.5 ms pro Event
39 ms Timer:	26 Events, 1020 ms => 39.2 ms pro Event

Man sieht anhand der Ergebnisse, dass ein Timer mit einer Zeitspanne von unter 20 ms auf diesem System nicht ausreichend schnell bearbeitet wird. Der Grund dafür ist das, nachdem das Testprogramm die Kontrolle abgegeben hat, zunächst einen anderen Prozess arbeiten lässt. Frühestens 20 ms später erhält unser Testprogramm die Kontrolle zurück. Größere Intervalllängen sind kein Problem. Sie werden sehr exakt ausgeführt.

Ein Timerintervall von 0 wird ebenfalls problemlos ausgeführt. In diesem Fall gibt das Testprogramm die Kontrolle gar nicht an das Betriebssystem zurück, sondern führt direkt den nächsten Timer-Event aus. In diesem System ist es also nur eingeschränkt möglich, kurze Timer im Bereich zwischen 1 und 20 ms exakt zu benutzen.

Das Messergebnis hängt natürlich auch stark von der Auslastung des Systems ab. Wenn viele andere Prozesse ebenfalls die CPU beanspruchen, nehmen die Ungenauigkeiten der Timer in Qt weiter zu.

Übung 4.9

Diese Implementierung benutzt jeweils einen SingleShot-Timer. Nach jeder abgelaufenen Sekunde wird die Zeit bis zur nächsten vollen Sekunde neu berechnet. So erreichen wir eine exakte Synchronisierung auf den Sekundenbeginn. Ein regelmäßiger Timer, der jede Sekunde einen Event erzeugt, müsste zunächst synchronisiert werden. Außerdem könnte er im Laufe der Zeit womöglich etwas abweichen, zum Beispiel weil ein anderer Prozess den Rechner für einige Sekunden blockiert hatte. Unsere Lösung mit dem SingleShot-Timer ist deutlich stabiler. Sie benötigt auch nur sehr wenig Systemressourcen, da der Timer nur einmal pro Sekunde aufgerufen wird.

```
// Zeigt in einem Fenster die verbleibenden Sekunden
// bis zum Jahr 2100.
// Dazu wird in jeder Sekunde die Zeit bis zum Beginn
```

```
// der nächsten Sekunde ermittelt und ein
// SingleShot-Timer benutzt.

#include <qapplication.h>
#include <qpushbutton.h>
#include <qlabel.h>
#include <qdatetime.h>
#include <qtimer.h>
#include <qlayout.h>

class CounterWidget : public QWidget {
    Q_OBJECT

public:
    CounterWidget (QWidget *parent=0, const char *name=0);
    ~CounterWidget () {}

private slots:
    void updateTime ();

private:
    QLabel *secondsLabel;
};

CounterWidget::CounterWidget (QWidget *parent,
                              const char *name):
    QWidget (parent, name)
{
    QVBoxLayout *layout = new QVBoxLayout (this, 10);

    secondsLabel = new QLabel (this);
    secondsLabel->setAlignment (AlignCenter);
    updateTime ();
    layout->addWidget (secondsLabel);

    QPushButton *endeButton =
        new QPushButton ("Beenden", this);
    connect (endeButton, SIGNAL (clicked ()),
            qApp, SLOT (quit ()));
    layout->addWidget (endeButton);
}

// Die Slot-Methode updateTime wird vom Timer
// aufgerufen. Sie berechnet die verbleibende Zeit
// bis zum Jahr 2100 und setzt den nächsten Timer so,
// dass er gerade zum Beginn der nächsten Sekunde wirksam
// wird.
void CounterWidget::updateTime () {
    QDateTime now = QDateTime::currentDateTime ();
```

```
// berechnet die Zeit bis zum Anfang
// der nächsten vollen Sekunde
int msecUntilNextSec = 1000 - now.time ().msec ();

// berechnet die Anzahl der Sekunden
// bis zum 1. Januar 2100
int restSeconds =
    now.secsTo (QDateTime (QDate (2100, 1, 1)));

// Falls wir uns unmittelbar vor dem Beginn einer
// neuen Sekunde befinden (weniger als 100 ms),
// so warten wir auf den Beginn der übernächsten
// neuen Sekunde. Dadurch erreichen wir eine
// gewisse Toleranz, so dass der Timer-Event
// auch im synchronisierten Zustand etwas
// früher auftreten darf.
if (msecUntilNextSec < 100) {
    msecUntilNextSec += 1000;
    restSeconds--;
}

// Starten des SingleShot-Timers
// auf den Beginn der nächsten Sekunde
QTimer::singleShot (msecUntilNextSec, this,
                    SLOT (updateTime ()));

// Ausgeben der verbleibenden Sekunden auf den Bildschirm.
QString labelText;
labelText.sprintf ("Noch %d Sekunden\n"
                  "bis zum Jahr 2100", restSeconds);
secondsLabel->setText (labelText);
}

int main (int argc, char **argv) {
    QApplication app (argc, argv);
    CounterWidget mainWidget;
    app.setMainWidget (&mainWidget);
    mainWidget.show ();
    return app.exec ();
}
```

Übung 4.10

Wir werden hier zwei Varianten vorstellen, die beide eine Schaltfläche des Typs *QPushButton* um die Funktionalität eines schneller werdenden Auto-Repeats erweitern. Die erste Variante definiert eine neue Klasse, die von *QPushButton* abgeleitet ist. Sie übernimmt dabei selbst die Abfrage der Maus-Events, da sonst *QPushButton* zusätzliche Signale senden würde. Um die Auto-Repeat-Funktion zu

realisieren, benutzt die Klasse zwei *QTimer*-Objekte. Ein Objekt hat ein Intervall von einer Sekunde und verkürzt bei jedem Auslösen das Intervall des anderen Timers. Dieser zweite Timer bewirkt ein Senden der Signale, wenn er ausgelöst wird.

```
class AutorepeatButton: public QPushButton {
    Q_OBJECT

public:
    AutorepeatButton (QWidget *parent = 0,
                     const char *name = 0);
    AutorepeatButton (const char* text, QWidget *parent = 0,
                     const char *name = 0);
    ~AutorepeatButton () {}

protected:
    void mousePressEvent (QMouseEvent *ev);
    void mouseReleaseEvent (QMouseEvent *ev);
    void mouseMoveEvent (QMouseEvent *ev);

private slots:
    void emitNextClicked ();
    void speedup ();

private:
    bool holding, inside;
    int repeatStatus;
    QTimer *clickTimer, *speedupTimer;
};

AutorepeatButton::AutorepeatButton (QWidget *parent,
                                     const char *name)
    : QPushButton (parent, name)
{
    holding = FALSE;
    inside = FALSE;
    clickTimer = new QTimer (this);
    connect (clickTimer, SIGNAL (timeout ()),
            SLOT (emitNextClicked ()));
    speedupTimer = new QTimer (this);
    connect (speedupTimer, SIGNAL (timeout ()),
            SLOT (speedup ()));
}

AutorepeatButton::AutorepeatButton (const char *text,
                                     QWidget *parent, const char *name)
    : QPushButton (text, parent, name)
{
```

```
    holding = FALSE;
    inside = FALSE;
    clickTimer = new QTimer (this);
    connect (clickTimer, SIGNAL (timeout ()),
            SLOT (emitNextClicked ()));
    speedupTimer = new QTimer (this);
    connect (speedupTimer, SIGNAL (timeout ()),
            SLOT (speedup ()));
}

void AutorepeatButton::mousePressEvent (QMouseEvent *ev)
{
    // Nur auf die linke Maustaste innerhalb
    // des Buttons reagieren
    if (ev->button () != LeftButton ||
        !hitButton (ev->pos ()))
        return;

    setDown (TRUE); // Button ist gedrueckt
    holding = TRUE;
    inside = TRUE;
    repeatStatus = 0;
    speedupTimer->start (1000); // Jede Sekunde
    // Im Gegensatz zu QPushButton senden wir hier bereits
    // einmal clicked ganz am Anfang.
    emit clicked ();
    emit pressed ();
}

void AutorepeatButton::mouseReleaseEvent (QMouseEvent *ev)
{
    // Falls Button nicht gedrückt oder nicht
    // die linke Maustaste losgelassen, ignorieren
    if (!holding || ev->button () != LeftButton) return;

    setDown (FALSE);
    holding = FALSE;
    inside = FALSE;
    speedupTimer->stop ();
    clickTimer->stop ();
    emit released ();
}

void AutorepeatButton::mouseMoveEvent (QMouseEvent *ev)
{
    bool nowInside = hitButton (ev->pos ());
    if (!holding || nowInside == inside)
        return;
```

```

inside = nowInside;
if (inside) {
    setDown (TRUE);
    speedupTimer->start (1000);
    repeatStatus = 0;
} else {
    setDown (FALSE);
    speedupTimer->stop ();
    clickTimer->stop ();
}
}

void AutorepeatButton::emitNextClicked () {
    // Signale senden, als wäre der Button kurz
    // losgelassen worden.
    emit released ();
    emit clicked ();
    emit pressed ();
}

void AutorepeatButton::speedup () {
    repeatStatus++;
    switch (repeatStatus) {
        case 1: // 200 ms
            clickTimer->changeInterval (200);
            break;
        case 2: // 50 ms
            clickTimer->changeInterval (50);
            break;
        case 3: // So schnell wie möglich
            //(danach kann der speedupTimer
            // deaktiviert werden)
            clickTimer->changeInterval (0);
            speedupTimer->stop ();
    }
}

```

Die zweite Variante benutzt eine zusätzliche Klasse, *AutorepeatObject*, abgeleitet von *QObject*, die man einem *QPushButton* zuordnet. Von diesem liest sie die Signale *pressed* und *released* und ermittelt daraus, wann und wie lange der Button heruntergedrückt ist. Diese Klasse kann jedem Button zugewiesen werden, der von *QButton* abgeleitet ist, also nicht nur *QPushButton*. Auch in dieser Variante werden zwei Timer der Klasse *QTimer* benutzt. Ein Timer löst das Signal *repeatClicked* aus, der andere verringert das Zeitintervall des ersten.

Um diese Klasse zu benutzen, erzeugt man zunächst einen normalen Button. Anschließend erzeugt man ein *AutorepeatObject*-Objekt, bei dem man als Vater-

objekt das Button-Objekt einträgt. (So wird das *AutorepeatObject*-Objekt auch automatisch gelöscht, wenn der Button gelöscht wird.) Man verbindet dann das Signal *repeatClick* mit einem Slot, der die gewünschte Aktion ausführt.

```
class AutorepeatObject: public QObject {
    Q_OBJECT
public:
    AutorepeatObject (QPushButton *parent,
                      const char *name = 0);
    ~AutorepeatObject () {}

signals:
    void repeatClicked ();

private slots:
    void speedup ();
    void buttonPressed ();
    void buttonReleased ();

private:
    int repeatStatus;
    QTimer *clickTimer, *speedupTimer;
};

AutorepeatObject::AutorepeatObject (QPushButton *parent,
                                     const char *name)
    : QObject (parent, name)
{
    if (!parent) {
        debug ("AutorepeatObject needs a QPushButton parent!");
        return;
    }
    connect (parent, SIGNAL (pressed ()),
             SLOT (buttonPressed ()));
    connect (parent, SIGNAL (released ()),
             SLOT (buttonReleased ()));

    clickTimer = new QTimer (this);
    // Wir verbinden das Signal des clickTimer
    // direkt mit dem Signal repeatClicked. So
    // benötigen wir keinen zusätzlichen Slot.
    connect (clickTimer, SIGNAL (timeout ()),
             SIGNAL (repeatClicked ()));
    speedupTimer = new QTimer (this);
    connect (speedupTimer, SIGNAL (timeout ()),
             SLOT (speedup ()));
}
```

```
void AutorepeatObject::buttonPressed () {
    repeatStatus = 0;
    speedupTimer->start (1000); // Jede Sekunde
    emit repeatClicked ();
}

void AutorepeatObject::buttonReleased () {
    speedupTimer->stop ();
    clickTimer->stop ();
}

void AutorepeatObject::speedup () {
    repeatStatus++;
    switch (repeatStatus) {
        case 1: // 200 ms
            clickTimer->changeInterval (200);
            break;
        case 2: // 50 ms
            clickTimer->changeInterval (50);
            break;
        case 3: // So schnell wie möglich
            // (danach kann der speedupTimer
            // deaktiviert werden)
            clickTimer->changeInterval (0);
            speedupTimer->stop ();
    }
}
```

Übung 4.11

Unsere neue Timer-Klasse trägt den Namen *SafeTimer*. Sie erhält die gleichen Methoden wie die Klasse *QTimer*, bis auf zwei Ausnahmen: Die Möglichkeit, einen Singleshot-Timer zu starten, ist in *SafeTimer* nicht implementiert, und das Signal *timeout* hat einen zusätzlichen Parameter vom Typ *int*. In diesem Parameter liefert die Klasse zurück, wie viele »verschlafene« Aktivierungen seit dem letzten Aufruf verstrichen sind. So kann der Empfänger des Signals selbst entscheiden, ob er in einer einfachen Schleife jeden Timer-Event nachholt, entsprechend viele Schritte auf einmal durchführt oder die zusätzlichen Events einfach ignoriert. Eine mögliche Anwendung ist die Anzeige von Videodaten. Damit die Bilder mit einer festen Wiederholungsrate gezeigt werden, verbindet man einen Timer mit dem Slot, der das nächste Bild anzeigt. Wurden nun einige Aufrufe »verschlafen«, so kann dieser Slot einfach eine entsprechende Anzahl von Bildern überspringen. Somit bleiben die Bilder zeitsynchron.

In der Implementierung wird in einem *QTime*-Objekt gespeichert, wann die nächste Aktivierung fällig wäre. Aus der aktuellen Zeit wird dann die Zeitspanne bis zur nächsten Aktivierung berechnet und ein Singleshot-Timer gestartet.

Sobald dieser abgelaufen ist, wird die aktuelle Zeit mit der Zeit verglichen, zu der der Timer hätte ausgelöst werden sollen. Kam der Timer zu spät, wird aus der Differenz die Anzahl der verpassten Events errechnet.

```
class SafeTimer : public QObject
{
    Q_OBJECT
public:
    SafeTimer (QObject *parent=0, const char *name=0)
        : QObject (parent, name), running (FALSE) {}
    ~SafeTimer () {}

    bool isActive () {return running;}
    void start (int msec);
    void stop ();
    void changeIntervall (int msec) {start (msec);}
signals:
    void timeout (int number);
private slots:
    void timerSlot();
private:
    void setNextTimer();
    int interval;
    bool running;
    QTime nextEvent;
    QTimer singleShotTimer;
}

SafeTimer::SafeTimer (QObject *parent, const char *name)
    : QObject (parent, name), running (FALSE)
{
    connect (singleShotTimer, SIGNAL (timeout ()),
            SLOT (timerSlot()));
}

void SafeTimer::start (int msec)
{
    if (running)
        stop();
    if (msec < 0)
        msec = 0;
    interval = msec;
    nextEvent = QTime::currentTime().addMSecs (interval);
    running = TRUE;
    setNextTimer();
}

void SafeTimer::stop ()
```

```
{
    if (!running)
        return;
    running = FALSE;
    // Falls noch Timeraufrufe verschlafen wurden,
    // diese nachholen.
    if (interval > 0 &&
        nextEvent <= QTime::currentTime())
        timerSlot();
    singeShot.stop();
}

void SafeTimer::timerSlot ()
{
    if (!running) return;
    int lostTime = nextEvent.msecsTo (QTime::currentTime());
    // Überlauf bei Mitternacht, bei mehr als einer Sekunde
    // zu früh
    if (lostTime < -1000)
        lostTime += 86400000; // ms pro Tag
    if (interval <= 0)
    {
        // Maximaler Timer, schickt ein Signal
        emit timeout (1);
        nextEvent = QTime::currentTime();
    }
    else
    {
        // Normaler Timer, verschlafene Events errechnen
        int events = lostTime / interval + 1;
        nextEvent.addMsecs (events * interval);
        emit timeout (events);
    }
    setNextTimer();
}

void SafeTimer::setNextTimer()
{
    int ms = QTime::curentTime ().msecsTo (nextEvent);
    if (ms < 0)
        ms = 0;
    singeShotTimer.start (ms, TRUE);
}
```

A.7 Lösungen zu Kapitel 4.12

Übung 4.12

Für unser Programm benutzen wir eine selbst geschriebene Klasse *ShowSocket* – abgeleitet von *QMultiLineEdit* –, die an einem Socket (oder einem anderen Dateideskriptor) horcht und die eintreffenden Daten als ASCII-Zeichen in einem Fenster darstellt. Diese Klasse verwenden wir, um die Daten vom Dateideskriptor 0 (*stdin*) anzuzeigen.

```
#include <kapp.h>
#include <qsocketnotifier.h>
#include <qmultilinedit.h>

#include <unistd.h>

class ShowSocket : public QMultiLineEdit
{
    Q_OBJECT
public:
    ShowSocket (int sock, QObject *parent = 0,
                const char *name = 0);
    ~ShowSocket () {}

protected slots:
    void receive (int sock);
private:
    QSocketNotifier notify;
};

ShowSocket::ShowSocket (int sock, QObject *parent,
                        const char *name)
    : QMultiLineEdit (parent, name),
      notify (sock, QSocketNotifier::Read)
{
    setReadOnly (TRUE); // nur anzeigen
    connect (&notify, SIGNAL (activated (int)),
            this, SLOT (receive (int)));
}

void ShowSocket::receive (int sock)
{
    char buffer [1024];
    int length;
    // Block von max. 1024 Zeichen einlesen
    length = read (sock, buffer, 1024);
    // Falls Stream beendet (EOF), nicht weiter
    // auf Daten warten
```

```
    if (length <= 0)
        notify.setEnabled (FALSE);
    else
        append (QString (buffer, length));
}

int main (int argc, char **argv)
{
    KApplication app (argc, argv);
    ShowSocket *widget = new ShowSocket (0);
    widget->show();
    QObject::connect (&app, SIGNAL (lastWindowClosed ()),
                     &app, SLOT (quit()));
    app.exec ();
}
```

B Die Lizenzen von Qt und KDE

In diesem Anhang finden Sie die vier wichtigsten Lizenzen für die Benutzung von Qt und KDE im Originalwortlaut. Für die kostenlose Nutzung der Qt Free Edition gilt die *Qt Free Edition License* (für Versionen vor Qt 2.0) bzw. die *Qt Public License* (QPL, für Versionen ab Qt 2.0). Für die KDE-Programme (wie zum Beispiel für den *kfm* oder den *kwm*) gelten die Lizenzbedingungen der *General Public License* (GPL), für die KDE-Bibliotheken die Bedingungen der *Library General Public License* (LGPL).

Nähere Erläuterungen zu den Lizenzen finden Sie in Kapitel 1.5, *Lizenzbestimmungen*.

B.1 Qt Free Edition License für Versionen vor Qt 2.0

QT FREE EDITION LICENSE

Copyright (C) 1992-1998 Troll Tech AS. All rights reserved.

This is the license for Qt Free Edition version 1.42; it covers private use, use of third-party application programs based on Qt, and development of free software for the free software community.

COPYRIGHT AND RESTRICTIONS

The Qt toolkit is a product of Troll Tech AS. The Qt Free Edition is limited to use with the X Window System.

You may copy this version of the Qt Free Edition provided that the entire archive is distributed unchanged and as a whole, including this notice.

You may use this version of the Qt Free Edition to compile, link and run application programs legally developed by third parties.

You may use the Qt Free Edition to create application programs provided that:

- You accept this license.
- Your software does not require modifications to Qt Free Edition.
- You satisfy ONE of the following three requirements

EITHER

Users of your software can freely obtain source code for the software, freely modify the source code (possibly with restrictions on copyright notices, attributions and legal responsibility), and freely redistribute original or modified versions of the software.

OR

Your software is distributed under the GNU GENERAL PUBLIC LICENSE, version 2 or later, as defined by the Free Software Foundation.

OR

Your software is distributed under the GNU LIBRARY GENERAL PUBLIC LICENSE, version 2 or later, as defined by the Free Software Foundation.

If you are paid to develop something with Qt Free Edition or it is a part of your job the following conditions also apply:

- Your software must not require libraries, programs, data or documentation that are not available outside your organization in order to compile or use.
- If and when your organization starts using the software, you must notify Troll Tech AS of the following:
- Your organization's name and purpose.
- The software's name and purpose.
- The software's license.
- That your organization considers the software to be free software.

You may also use the Qt Free Edition to create reusable components (such as libraries) provided that you accept the terms above, and in addition that:

- Your components' license includes the following text:
[Your package] requires the Qt library, which is copyright Troll Tech AS. Freely distributable programs may generally use Qt Free Edition free of charge, see [README.QT] for details.
- README.QT is distributed along with your components.
- Qt Free Edition is not distributed as an integral part of your components.

LIMITATIONS OF LIABILITY

Troll Tech AS makes no obligation under this license to support or upgrade Qt Free Edition, or assist in the use of Qt Free Edition.

In no event shall Troll Tech AS be liable for any lost revenue or profits or other direct, indirect, special, incidental or consequential damages, even if Troll Tech has been advised of the possibility of such damages.

QT FREE EDITION IS PROVIDED AS IS WITH NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

B.2 QPL – Qt Public License – ab Qt 2.0

THE Q PUBLIC LICENSE

version 1.0

Copyright (C) 1999 Troll Tech AS, Norway.

Everyone is permitted to copy and distribute this license document.

The intent of this license is to establish freedom to share and change the software regulated by this license under the open source model.

This license applies to any software containing a notice placed by the copyright holder saying that it may be distributed under the terms of the Q Public License version 1.0. Such software is herein referred to as the Software. This license covers modification and distribution of the Software, use of third-party application programs based on the Software, and development of free software which uses the Software.

Granted Rights

1. You are granted the non-exclusive rights set forth in this license provided you agree to and comply with any and all conditions in this license. Whole or partial distribution of the Software, or software items that link with the Software, in any form signifies acceptance of this license.
2. You may copy and distribute the Software in unmodified form provided that the entire package, including – but not restricted to – copyright, trademark notices and disclaimers, as released by the initial developer of the Software, is distributed.
3. You may make modifications to the Software and distribute your modifications, in a form that is separate from the Software, such as patches. The following restrictions apply to modifications:
 - Modifications must not alter or remove any copyright notices in the Software.
 - When modifications to the Software are released under this license, a non-exclusive royalty-free right is granted to the initial developer of the Software to distribute your modification in future versions of the Software provided such versions remain available under these terms in addition to any other license(s) of the initial developer.
4. You may distribute machine-executable forms of the Software or machine-executable forms of modified versions of the Software, provided that you meet these restrictions:

- You must include this license document in the distribution.
 - You must ensure that all recipients of the machine-executable forms are also able to receive the complete machine-readable source code to the distributed Software, including all modifications, without any charge beyond the costs of data transfer, and place prominent notices in the distribution explaining this.
 - You must ensure that all modifications included in the machine-executable forms are available under the terms of this license.
5. You may use the original or modified versions of the Software to compile, link and run application programs legally developed by you or by others.
 6. You may develop application programs, reusable components and other software items that link with the original or modified versions of the Software. These items, when distributed, are subject to the following requirements:
 - You must ensure that all recipients of machine-executable forms of these items are also able to receive and use the complete machine-readable source code to the items without any charge beyond the costs of data transfer.
 - You must explicitly license all recipients of your items to use and re-distribute original and modified versions of the items in both machine-executable and source code forms. The recipients must be able to do so without any charges whatsoever, and they must be able to re-distribute to anyone they choose.
 - If the items are not available to the general public, and the initial developer of the Software requests a copy of the items, then you must supply one.

Limitations of Liability

In no event shall the initial developers or copyright holders be liable for any damages whatsoever, including – but not restricted to – lost revenue or profits or other direct, indirect, special, incidental or consequential damages, even if they have been advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

No Warranty

The Software and this license document are provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Choice of Law

This license is governed by the Laws of Norway. Disputes shall be settled by Oslo City Court.

B.3 GPL – General Public License

NOTE! The GPL below is copyrighted by the Free Software Foundation, but the instance of code that it refers to (the kde programs) are copyrighted by the authors who actually wrote it.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.

675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you just show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is

modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The »Program«, below, refers to any such program or work, and a »work based on the Program« means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term »modification«.) Each licensee is addressed as »you«.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifi-

cations or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

- 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
- 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
- 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions.

You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and »any later ver-

sion«, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM »AS IS« WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

B.4 LGPL – Library General Public License

NOTE! The LGPL below is copyrighted by the Free Software Foundation, but the instance of code that it refers to (the kde libraries) are copyrighted by the authors who actually wrote it.

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1991 Free Software Foundation, Inc.

59 Temple Place – Suite 330

Boston, MA 02111-1307, USA.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the library GPL. It is numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can

relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a »work based on the library« and a »work that uses the library«. The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

GNU LIBRARY GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called »this License«). Each licensee is addressed as »you«.

A »library« means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The »Library«, below, refers to any such software library or work which has been distributed under these terms. A »work based on the Library« means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term »modification«.)

»Source code« for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
 - a) The modified work must itself be a software library.
 - b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
 - c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
 - d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful. (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must

alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a »work that uses the Library«. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a »work that uses the Library« with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a »work that uses the library«. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a »work that uses the Library« uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether

it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a »work that uses the Library« with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable »work that uses the Library«, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the »work that uses the Library« must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form)

with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:
 - a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
 - b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and »any later version«, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY »AS IS« WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

C Die KDE-Standardfarbpalette

KDE benutzt in all seinen Applikationen eine *Standardfarbpalette* mit 42 Farben. Wenn Sie für Ihr Programm eigene Icons entwerfen, sollten Sie möglichst nur Farben aus dieser Palette benutzen, da dann auf X-Servern mit einem Grafikmodus von 256 Farben noch viele Farbeinträge für andere Programme freibleiben. Tabelle C.1 enthält die Farbinformationen über die Paletteneinträge.

Rot	Grün	Blau	Farbname	Farbe
48	48	48	#303030	Grau (18%)
88	88	88	#585858	Grau (34%)
128	128	128	#808080	Grau (50%)
160	160	160	#a0a0a0	Grau (63%)
195	195	195	#c3c3c3	Grau (76%)
220	220	220	#dcdcdc	Grau (86%)
64	0	0	#400000	Dunkelrot (25%)
0	64	0	#004000	Dunkelgrün (25%)
0	0	64	#000040	Dunkelblau (25%)
64	64	0	#404000	Dunkelbraun (25%)
0	64	64	#004040	Dunkelcyan (25%)
64	0	64	#400040	Dunkelviolett (25%)
128	0	0	#800000	Rot (50%)
0	128	0	#008000	Grün (50%)
0	0	128	#000080	Blau (50%)
128	128	0	#808000	Braun (50%)
0	128	128	#008080	Cyan (50%)
128	0	128	#800080	Magenta (50%)
192	0	0	#c00000	Rot (75%)
0	192	0	#00c000	Grün (75%)
0	0	192	#0000c0	Blau (75%)
192	192	0	#c0c000	Beige (75%)
0	192	192	#00c0c0	Cyan (75%)
192	0	192	#c000c0	Magenta (75%)
255	0	0	#ff0000	Hellrot (100%)
0	255	0	#00ff00	Hellgrün (100%)
0	0	255	#0000ff	Hellblau (100%)

Tabelle C-1 Die KDE-Farbpalette

Rot	Grün	Blau	Farbname	Farbe
255	255	0	#ffff00	Gelb (100%)
0	255	255	#00ffff	Hellcyan (100%)
255	0	255	#ff00ff	Hellmagenta (100%)
255	192	192	#ffc0c0	Pastellrot
192	255	192	#c0ffc0	Pastellgrün
192	192	255	#c0c0ff	Pastellblau
255	255	192	#ffffc0	Blassgelb
192	255	255	#c0ffff	Blasscyan
255	192	255	#ffc0ff	Lila
255	128	0	#ff8000	Orange
192	88	0	#c05800	Ocker
255	168	88	#ffa858	Blassocker
255	220	168	#ffdca8	Blassorange
255	255	255	#ffffff	Weiß
0	0	0	#000000	Schwarz

Tabelle C-1 Die KDE-Farbpalette (Forts.)

Stichwortverzeichnis

Symbols

\$DISPLAY 91

\$KDEDIR 14, 27, 35

\$QTDIR 14, 27

A

A4 514

abort 514

Accelerator 126

accept 357, 360

actionCollection 129, 146

activated 481

add 456

addCmdLineOptions 104

addColSpacing 722

addLayout 207, 720

addMultiCellWidget 210, 723

addSpacing 208, 719

addStretch 719

addWidget 152, 719

adjustSize 85

Aktionen 126

AlarmTimer 469, 728

AlignBottom 306

AlignCenter 306, 727

AlignHCenter 306

AlignLeft 306

AlignRight 306

AlignTop 306

AlignVCenter 306

Allocation Context 280

allowsErrorInteraction 512

allowsInteraction 512

Alpha-Kanal 328, 343

AltButton 352

Anzeigebereich 115

append 418

applnk 35

arg 445

argc 22

argv 22

ASCII 423

ascii 357

ASYNc 566

atomare Datei 525

attach 561

Auflösung von Timern 470

Ausschnitte 320

autoBorder 718

AutoColor 336

autoconf 598

autoDelete 399, 401

automake 598

AutoRepeatButton 470

AvoidDither 337

B

Background 284, 377

BarIcon 590

Base 284, 377

Baumstruktur 46

BDialogPattern 312

beep 495

begin 291

BevelJoin 311

BigEndian 328

binärkompatibel 581

bitBlt 324, 334

bitOrder 328

Block 492

Blockierung 472

blockierungsfrei 471

blockSignals 56

Borland 19

boundingRect 307

brect 307

BrightText 284

Button 284

button 351

ButtonText 284

C

C++ 2, 8

C++-Compiler 13

Cache 413

call 564

Callback 48

canBeRestored 511

cancel 512

capStyle 311

Cascade 183

CDE 1

CharacterSource 489

childEvent 348

children 48

childrenRect 85

className 72
classNameOfToplevel 511
clear 57, 499
clicked 57
Clipboard 497, 498
Clipping 290, 293, 320
close 83
closeEvent 191, 360
closeStdin 492
-cmap 274, 276
Color Allocation 274
 ReadOnly 274
 ReadWrite 274
Color Cube 275
Color Lookup Table 283, 331
color0 341
color1 341
colorGroup 88, 349
ColorOnly 336
commitData 512
config 452
config.h 602
configure 599
configureToolbar 148
connect 52, 714
Container-Klasse 399
contains 406, 419
containsRef 406
ControlButton 352
convertFromImage 336
convertToImage 338
copy 499
CopyROP 335
Counter 61
createGUI 144, 161
createHeuristicMask 343
CrossPattern 312
current 406
currentDate 461
currentDateTime 461
currentTime 461
CustomColor 275
CustomPattern 311
CuttomPattern 312

D

Dame-Brett 367
dark 279, 285
darkBlue 278
darkCyan 278
darkGreen 278

darkMagenta 278
darkRed 278
darkYellow 278
DashDotDotLine 310
DashDotLine 310
DashLine 310
dataChanged 188, 499
Dateinformationen 518
Datencontainer 399
Datenstrukturen 396
Datum 460
Datumsformat 447
daysTo 461
DCOP 553
dcop 555
DCOPClient 561
dcopClient 561
dcopidl 565
dcopidl2cpp 565
DCOPObject 561
DCOP-Signal 571
decomposition 427
defaultDepth 334
defineIOHandler 332
Dekorationsflags 84
Dekorationsflags *siehe* WFlags
deleteEntry 453
deleteGroup 453
delta 355
DensexPattern 312
dequeue 408
desktop 35, 449
Desktop Communications Protocol *siehe*
 DCOP
destroyAllocContext 280
destroyed 56
detach 398
DiagCrossPattern 312
Dialog 240, 248
Dialogfenster 66
DiffuseAlphaDither 337
DiffuseDither 337
DirectColor 273
disconnect 53, 714
-display 91
DNDListBox 506
Document 186
Document-View-Konzept 186
Dokument 153
done 241
DontCare 492

- DontClip 306
- DotLine 310
- download 540
- doxygen 382
- drag 503
- Drag&Drop 359, 497, 500
- dragCopy 503
- dragEnterEvent 359, 501
- dragLeaveEvent 359
- dragMove 503
- dragMoveEvent 359, 502
- DraughtBoard 369
- drawArc 300, 727
- drawChord 303, 726
- drawContentsOffset 376
- drawEllipse 291, 302, 727
- drawImage 331
- drawImageToPainter 346
- drawImageTransparent 345
- drawLine 298
- drawLineSegments 299
- drawPie 304
- drawPixmap 336
- drawPoint 297, 334
- drawPoints 297
- drawPolygon 304
- drawPolyLine 299
- drawQuadBezier 300, 323
- drawRect 301
- drawRoundRect 302, 727
- drawText 305, 728
- dropEvent 359, 501
- dumpObjectInfo 73
- dumpObjectTree 73
- dynamisches Array 400
- E
- egcs 13
- eindimensionale Funktion 361
- elapsed 463
- emit 53
- enableRewind 487
- enableSessionManagement 508
- encodedData 499
- end 291
- enqueue 408
- enterAllocContext 280
- enterEvent 359
- Enterprise-Edition 10
- enterWhatsThisMode 457
- eor 321
- eraseRect 296
- Even-Odd-Algorithmus 305, 321
- Event 44, 112, 348
- eventFilter 113
- exec 92, 241
- execv 492
- exitStatus 492
- Expanding 212
- ExpandTaps 306
- explizit gemeinsame Daten 397
- ExpMask 211
- F
- Factory 577
- Farben 273
- Farbkreis 278
- Farbpsychologie 281
- Farbtabelle 328
- Farbtiefe 328
- Farbton 278, 724
- Farbwürfel 275
 - siehe Color Cube
- FDesigner 5
- FDialogPattern 312
- FIFO-Speicher 401
- fileName 332
- fill 402
- fillRect 296
- find 402, 406, 419
- findNext 406
- findNextRef 406
- findRef 406
- findRev 419
- findtr 444
- Fixed 212
- FlatCap 311
- Flimmern 376
- flush 294
- fnctl 479
- focusInEvent 358
- focusNextPrevChild 358
- focusOutEvent 358
- focusPolicy 358
- Foreground 284
- fork 489
- format 499
- frameGeometry 84
- Free Software Foundation 11
- FreeBCC 19
- fromUtf8 430
- FTFont 440

FTP 163, 244
FTP-Server 8
Füllmuster 309, 312

G

gcc 13
gemeinsame Daten 397
General Public License 749
geometry 85
get 546
getHelpMenu 459
getOpenURL 162
getPrinterSetup 515
GhostScript 513
GIF 223, 332
globalPos 351
gmo 442
gmsgfmt 38
GNOME 5
GrayText 306
Gruppe 449
GTK 5
GUI-Elemente 219
GUIStyle 586

H

hasKey 453
hasViewXForm 316
Haupt-Event-Schleife 472
Hauptfenster 26, 115
head 408
height 314
heightMM 324
Helligkeit 278
--help 98
hide 82
hideEvent 360
Highlight 285
HighlightedText 284
Hilfsmittel 595
HorPattern 312
HSV-Modell 278, 723
HTML 41, 458
HTTP 163, 244
Hue 278

I

i18n 36, 101, 441, 590
I18N_NOOP 102, 443
IANA *siehe* Internet Assigned Numbers
Authority

ICON 590
Icon 126, 590
ignore 357, 360
IgnoreEndian 328
image 332
implizit gemeinsame Daten 397
inFont 440
inherits 73, 715
inline 715
insertButton 456
insertItem 151
installTranslator 444
internal 307
Internationalisierung 441
Internet Assigned Numbers Authority 433
intersect 321
invokeHTMLHelp 459
isA 72, 715
isChecked 133
isEmpty 408
isEnabled 83
ISO 8859 433
isRestored 509, 513
isVisible 83
isWidgetType 73, 716
Iterator 414

J

Job 543
joinStyle 311

K

K Desktop Environment 1, 745
K_DCOP 566
KAboutData 100
KAction 122, 126, 615
KActionMenu 132, 135, 616
KActionSeparator 132, 135
Käsekästchen 368
kappgen 601, 603
KApplication 27, 31, 47, 91, 95, 451, 616
kapptemplate 600, 603
KAudio 495
KAudioPlayer 616
KAWidget 372
KBabel 39
KCmdLineArgs 30, 99, 103, 589
KColorButton 226, 616
KColorCells 234, 617
KColorCombo 234, 618
KColorDialog 226, 245, 618

- KConfig 452, 618
- KConfigBase 449, 619
- KDatePicker 223, 236
- KDateTable 223
- kdcop 555
- kdDebug 591
- KDE 1
- KDE Control Center 282
- KDE Input/Output *siehe* KIO
- KDE *siehe* K Desktop Environment
- KDEBUG 591
- kdDebug 591
- kdecore 27, 615
- KDE-Farbpalette 765
- kdehelp 39
- kdError 591
- kdesdk 600
- kdeui 27, 615
- KDevelop 13, 186, 612
- KDE-Verzeichnisstruktur 35
 - siehe* Verzeichnisstruktur
- kdFatal 591
- KDoc 382, 395
- kdWarning 591
- KEditToolBar 144, 147
- Kellerspeicher 401
- kexample 600
- key 356
- Key_A 356
- Key_Down 356
- Key_F1 356
- Key_Left 356
- Key_Right 356
- Key_Up 356
- keyPressEvent 355
- keyReleaseEvent 355
- kfile 615
- KFileDialog 162, 243, 619
- KFM 592
- kfm 1, 14, 615
- KFontAction 132, 136
- KFontDialog 245, 620
- KFontSizeAction 132, 137
- KGlobal 452, 589
- KGradientSelector 230, 621
- KHelloWorld 26
- KHSSelector 231
- Kicker 35
- KIconButton 226
- KIconDialog 226, 248
- killTimer 465
- killTimers 465
- KImageIO 340
- KIO 163, 537
- Klassenstruktur 72
- KLed 224, 621
- KLibLoader 577
- KListAction 132, 136
- KLocale 441, 447, 621
- klocale 590
- KMainWindow 31, 117, 509, 591, 621
- KMenuBar 117, 142, 622
- KMenuEdit 36
- KMessageBox 120, 161, 242, 622
- KMiniEdit 153, 155, 172, 176
- Knapsack-Problem 472
- Komponente 575
- Konfigurationsdatei 96, 159, 165, 448
- Konqueror 576
- Koordinaten-Transformation 314
- KParts 575
- KPopupMenu 235
- KProcess 491, 622
- KProgress 224
- KRadioAction 132, 134, 623
- KRecentFileAction 175
- KRecentFilesAction 132, 137, 165, 623
- KRun 493
- KSaveFile 526
- KSelectAction 132, 135, 623
- KSeparator 221
- KServerSocket 480, 623
- KSimpleConfig 453, 624
- KSocket 480, 529, 624
- KStatusBar 117, 624
- KStdAction 138, 153, 159, 191
- KTempFile 525
- KTMainWindow 591, 625
- KToggleAction 132, 625
- KToolBar 117, 142, 625
- KTopWidget 591
- KUniqueApplication 106, 554, 625
- KURL 539, 625
- Kurzbefehl 126
- KValueSelector 230
- kwm 1, 14
- KXYSelector 231, 626
- L**
- Landessprache 26, 36
- Landscape 514
- lastWindowClosed 95

LaTeX 459
Latin-1 169, 423
Layout-Klassen 204
Lazy Allocation 277
LD_LIBRARY_PATH 19
ldconfig 19
ldd 18
leaveAllocContext 280
leaveEvent 359
left 418
LeftButton 352
Letter 514
LGPL 11
LGPL *siehe* Library General Public License
libkfile 168
libkio 168
Library General Public License 11, 755
LIFO-Speicher 401
Light 285
light 279
lineare Liste 400
lineTo 298
Linienart 309
Linux 2
LittleEndian 328
Lizenz 3, 9
load 332, 444
Logo 280
Logout 507
lower 80, 83, 418, 427
lpr 514
LZW 332

M

Mailing-Liste 6
main 22, 93
Makefile 72, 599
Makefile.am 600
Makefile.in 599
ManyColor 275, 288
mapFromGlobal 86
mapFromParent 86
mapToGlobal 86
mapToParent 86
Maximum 212
MayGrow 211
MayShrink 211
MDI *siehe* Multi Document Interface
Mehrfachvererbung 70
mehrsprachig 441
memberList 173, 176

menuBar 142
Menüleiste 26, 115, 126
mergetr 444
message 150
messages.po 37
Meta Object Compiler 44
MetaObject 72
MIB-Nummer 434
Microsoft Windows 1, 2, 9
Microsoft Windows *siehe* Windows
Mid 285
MidButton 352
Midlight 285
Mime-Typ 498
Minimum 212
MinimumExpanding 212
minimumSizeHint 211
MINSIZE 586
MiterJoin 311
moc 44, 49, 51, 63, 71, 120, 582, 596
moc *siehe* Meta Object Compiler
modal 241, 249, 255
modeless *siehe* nicht-modal
MonoOnly 336
Motif 5
mouseDoubleClickEvent 351
mouseMoveEvent 351
mousePressEvent 351
mouseReleaseEvent 351
move 84, 359
moveEvent 359
moveTo 298
msecsTo 461
msg2qm 444
msgfmt 38
msgid 38, 442
msgstr 38, 442
Multi Document Interface 171, 176, 190

N

name 45, 48
Namenskonvention 35
-ncols 276
NetAccess 540
Netzwerk 526
newInstance 108
newPage 514
nicht-modal 241, 249
No Dead Key 357
NoBackground 377
NoBrush 301, 312, 727

NoButton 352
NoFocus 358
nonmodal *siehe* nicht-modal
NoPen 301, 310, 727
NormalColor 275
NotifyOnExit 492

O

oldPos 359
oldSize 359
Online-Handbuch 39, 455
Online-Hilfe 26, 39
OpaqueMode 310, 313
Optionsdialog 263
OrderedAlphaDither 337
OrderedDither 337

P

paintCell 369
paintEvent 86, 189, 292, 295, 324, 348
palette 88
parent 45, 48
Part 577
PartManager 581
paste 499
Pipe 479
pixel 329
pixelIndex 329
PNG 128, 223
po 442
pop 408
Portrait 514
pos 84, 351
PreferDither 337
Preferred 212
prepend 418
pressed 56
Primzahl 412
processEvents 243, 473
processExited 492
Professional-Edition 10
progen 598
Property 73
Prozess 489
PseudoColor 274
push 408
put 546

Q

Q_OBJECT 61, 70
Q_PROPERTY 73

QAccel 626
QAction 191, 627
QActionGroup 627
qAlpha 345
QApplication 21, 22, 47, 91, 92, 275, 627
QArray 400, 402, 628
QAsciiCache 413, 628
QAsciiCacheIterator 628
QAsciiDict 400, 409, 411, 628
QAsciiDictIterator 629
QAsyncIO 629
QBitArray 403, 629
QBitmap 290, 336, 629
QBitVal 630
qBlue 329
QBoxLayout 204, 630
QBrush 277, 290, 309, 311, 631
QBuffer 403, 516, 631
QButton 347, 631
QButtonGroup 222, 227, 263, 632
QByteArray 401, 403, 632
QCache 413, 632
QCacheIterator 633
QCanvas 381, 633
QCanvasItem 633
QCanvasView 634
QCDEStyle 634
QChar 417, 426, 634
QCharRef 634
QCheckBox 224, 226, 227, 634
QCheckBoxItem 635
QChildEvent 635
QClipboard 161, 167, 498, 636
QCloseEvent 636
QCollection 636
QColor 276, 311, 637, 724
QColorDialog 245, 637
QColorDrag 637
QColorGroup 283, 349, 637
QComboBox 152, 232, 638
QCommonStyle 639
QConnection 639
QConstString 639
QCString 401, 417, 583, 640
QCursor 88, 641
QCustomEvent 642
QDataPump 483, 642
QDataSink 483, 643
QDataSource 483, 643
QDataStream 521, 522, 643
QDate 460, 645

- QDateTime 460, 645, 728
- QDial 230, 645
- QDialog 47, 241, 249, 255, 645
- QDict 400, 409, 646
- QDictIterator 647
- QDir 519, 647
- QDns 528, 647
- QDomDocument 647
- QDoubleValidator 238, 648
- QDragEnterEvent 648
- QDragLeaveEvent 648
- QDragMoveEvent 648
- QDragObject 503, 649
- QDropEvent 649
- QDropSite 501, 649
- QEvent 112, 348, 650
- QFile 164, 516, 650
- QFileDialog 243, 651
- QFileIconProvider 652
- QFileInfo 518, 652
- QFilePreview 653
- QFocusData 653
- QFocusEvent 358, 653
- QFont 653
- QFontDatabase 654
- QFontDialog 245, 654
- QFontInfo 654
- QFontMetrics 654
- QFrame 199, 221, 249, 263, 654
- QFtp 552, 655
- QGL 655
- QGLWidget 655
- qGreen 329
- QGrid 199, 203, 215, 239, 655, 717
- QGridLayout 204, 210, 656, 723
- QGroupBox 221, 263, 656
- QGuardedPtr 656
- QHBox 199, 214, 239, 656, 716
- QHBoxLayout 208, 216, 657, 718
- QHButtonGroup 657
- QHeader 657
- QHGroupBox 657
- QHideEvent 658
- QIconDrag 658
- QIconSet 191, 658
- QIconView 236
- QImage 327, 658
- QImageDrag 499, 503, 659
- QImageIO 332, 659
- QInputDialog 659
- QIntCache 413, 660
- QIntCachIterator 660
- QIntDict 400, 409, 411, 660
- QIntDictIterator 414, 661
- QIntValidator 238, 661
- QIODevice 483, 516, 661
- QIODeviceSource 483, 662
- QKeyEvent 662
- QLabel 22, 57, 121, 152, 222, 662, 707
- QLayout 664
- QLayoutItem 664
- QLCDNumber 225, 289, 664
- QLEDNumber 289
- QLineEdit 237, 664
- QList 69, 400, 404, 665
- QListBox 58, 232, 503, 665
- QListBoxItem 232, 666
- QListBoxPixmap 666
- QListBoxText 666
- QListIterator 667
- QListView 123, 235, 667
- QListViewItem 668
- QLocalFS 552
- QLocalFs 668
- QMainWindow 191, 668
- QMap 400, 409, 454, 669
- QMapConstIterator 669
- QMapIterator 669
- QMenuBar 191, 670
- QMenuData 143, 670
- QMessageBox 242, 255, 670
- QMetaObject 671
- QMetaProperty 672
- QMimeSource 499, 672
- QMotifStyle 672
- QMouseEvent 351, 672
- QMoveEvent 359, 673
- QMovie 223, 483, 673
- QMultiLineEdit 81, 123, 153, 176, 238, 250, 347, 673, 743
- QMutex 493, 674
- qnamespace.h 356, 584
- QNetworkOperation 550, 674
- QNetworkProtocol 552, 674
- QNPInstance 675
- QNPWidget 675
- QObject 44, 112, 406, 464, 675
- QPaintDevice 290, 309, 320, 323, 675
- QPaintDeviceMetrics 324, 675
- QPainter 277, 290, 293, 320, 676
- QPaintEvent 676
- QPalette 283, 676

- QPen 277, 290, 309, 677
QPicture 290, 323, 325, 677
QPixmap 191, 223, 290, 293, 323, 325, 327, 333, 379, 677
QPixmapCache 677
QPlatinumStyle 678
QPNGImagePacker 678
QPoint 84, 296, 678
QPointArray 297, 678
QPopupMenu 143, 234, 678
QPrintDialog 245, 513, 679
QPrinter 245, 290, 323, 325, 513, 680
QProgressBar 152, 224, 680
QProgressDialog 243, 475, 680
QPtrCache 413
QPtrDict 400, 409, 411, 680
QPtrDictIterator 681
QPushButton 56, 81, 152, 225, 250, 681, 735
QPushbutton 21
QQueue 401, 408, 681
QRadioButton 227, 232, 682
QRangeControl 228, 682
QRect 84, 296, 683
qRed 329
QRegExp 418, 683
QRegion 321, 684
QResizeEvent 684
QRgb 329
qRgb 329
qRgba 344
QScrollBar 229, 684
QScrollView 347, 375, 684
QSemaphore 493, 685
QSemimodal 241, 685
QServerSocket 529, 533, 685
QSessionManager 512, 685
QShared 686
QShowEvent 686
QSignal 686
QSignalMapper 69, 686
QSimpleRichText 308
QSimpleRichText 687
QSize 84, 211, 687
QSizeGrip 687
QSizePolicy 211, 687
QSlider 229, 688
QSocketNotifier 688
QSocket 516, 529, 688
QSocketDevice 516, 528, 688
QSocketNotifier 479
QSortedList 689
QSound 689
QSpacerItem 689
QSpinBox 228, 689
QSplitter 123, 190, 239, 690
QStack 401, 408, 691
QStatusBar 191, 691
QStoredDrag 499, 507, 691
QStrList 691
QString 59, 398, 417, 423, 582, 692
QStringList 408, 453, 692
QStrList 408, 450, 692
QStyle 349, 586, 692
QStyleSheet 693
Qt 1, 584, 693, 745
Qt 2.0 14, 581
Qt Designer 608
Qt Free Edition 10
Qt Free Edition License 745
Qt Public License 747
QTabBar 693
QTabDialog 267, 693
QTable 695
QTableView 347, 367, 369, 695
QTabWidget 240, 267, 695
QTextBrowser 192, 308, 696, 707
QTextCodec 434
QTextDecoder 437
QTextDrag 499, 503, 696
QTextEdit 238, 696
QTextEncoder 437
QTextInputStream 696
QTextOutputStream 696
QTextStream 439, 521, 697
QTextView 308, 697
QTextViewer 707
QThread 493, 697
QTime 460, 698, 728, 740
QTimer 468, 477, 698, 736, 738, 740
QTimerEvent 465, 698
QToolBar 191, 698
QToolButton 226, 457, 698
QToolTip 456, 699
QToolTipGroup 456, 699
QTranslator 444, 699
queryClose 156, 161, 179, 185, 191
questionYesNo 120
quit 92, 475
QUriDrag 699
QUrl 539, 549, 699
QURLDrag 499, 503

QUrlOperator 550, 700
 QValidator 237, 700
 QValueList 400, 404, 700
 QValueListConstIterator 701
 QValueListIterator 701
 QVariant 420, 701
 QVBox 123, 199, 214, 239, 701, 716
 QVBoxLayout 208, 217, 702
 QVButtonGroup 702
 QVector 400, 404, 702
 QVGroupBox 702
 QWaitCondition 493, 703
 QWhatsThis 457, 703
 QWheelEvent 355, 703
 QWidget 44, 66, 76, 112, 219, 249, 290,
 293, 323, 324, 347, 703
 QWidgetItem 704
 QWidgetStack 240, 269, 704
 QWindowsStyle 704
 QWizard 704
 QWMatrix 319, 704
 QWorkspace 176, 190, 705
 QXMLReader 705

R

raise 80, 83
 Read 482
 readBoolEntry 450
 readColorEntry 450
 readDoubleNumEntry 450
 readEntry 450
 readFontEntry 450
 readListEntry 450
 readLongNumEntry 450
 readNumEntry 450
 ReadOnlyPart 577
 readPointEntry 450
 readProperties 509
 readRectEntry 450
 readSizeEntry 450
 readUnsignedLongNumEntry 450
 readUnsignedNumEntry 450
 ReadWritePart 577
 readyToSend 486
 receivedStderr 492
 receivedStdout 492
 released 57
 remove 406, 412, 418, 456
 removeWidget 153
 repaint 86, 351
 reparent 84

replace 412, 418, 419
 resetXForm 319
 resize 84, 359, 402
 resizeEvent 359
 restart 463
 RestartAnyway 512
 RestartIfRunning 512
 RestartImmediately 512
 RestartNever 512
 RESTORE 510
 restoreOverrideCursor 471
 rewind 487
 rewindable 487
 RGB-Modell 276, 279, 723
 RichText 22, 223, 242, 308, 707
 right 418
 RightButton 352
 rotate 317
 RoundCap 311
 RoundJoin 311

S

Sättigung 278, 724
 SafeTimer 740
 Saturation 278
 save 332
 saveProperties 509
 saveState 512
 saveYourself 508
 scale 317
 scanLine 329
 Schach 368
 Schiffeversenken 368
 SDI *siehe* Single Document Interface
 secsTo 461
 select 480
 send 562
 sender 54, 68
 sendEvent 115
 sendTo 486
 Separator 147
 Session 507
 sessionConfig 453
 Session-Management 507
 setAlphaBuffer 343
 setBackgroundColor 87, 310
 setBackgroundMode 87, 310, 313, 377
 setBackgroundPixmap 87, 377
 setBrush 293, 296
 setBuddy 222
 setCaption 90, 167

- setCentralWidget 122
- setClipRect 315, 320, 379
- setClipRegion 315, 321, 378, 727
- setColor 330, 344
- setColorMode 514
- setColorSpec 275, 288
- setColStretch 722
- setCursor 88
- setData 499
- setDoubleClickInterval 352
- setEditable 136
- setEnabled 83, 129, 264, 349, 482
- setEventFilter 113
- setExclusiveGroup 134
- setFixedSize 85, 586
- setFocus 89
- setFocusPolicy 89
- setFont 88, 727
- setFromTo 515
- setGeometry 84, 359
- setHsv 279
- setImage 332, 498
- setLazyAlloc 277
- setMainWidget 23, 78, 95
- setMask 336, 341, 342
- setMaximumSize 85
- setMinimumSize 85, 586
- setMouseTracking 351
- setNum 60, 419
- setOutputFileName 514
- setOverrideCursor 471
- setPageOrder 515
- setPageSize 514
- setPalette 286
- setPen 277, 293, 334
- setPixel 329
- setPixmap 498, 503
- setPrinterName 514
- setPrintProgram 514
- setProgress 476
- setRestartHint 512
- setSizePolicy 211
- setTabArray 306, 308
- setTabOrder 89
- setTabStops 306, 308
- setText 58, 498
- setToggleAction 191
- setUpdatesEnabled 87
- setViewport 315
- setViewXForm 316
- setWindow 314
- setWorldMatrix 319
- setWorldXForm 319
- SGML 458
- sgml2html 459
- Shadow 285
- shared data 397
- Shared Memory 340
- shear 318
- ShiftButton 352
- show 77, 82
- showEvent 360
- ShowPrefix 306, 727
- ShowSocket 743
- SIGNAL 52
- Signal 49
- Signal-Slot-Konzept 48
- simplifyWhiteSpace 418
- Singe-Shot-Timer 733
- Single Document Interface 171, 172, 189
- SingleLine 306
- singleShot 468
- Single-Shot-Timer 740
- size 84
- sizeHint 85, 211, 347, 586
- sizePolicy 85, 211, 347, 585
- sleep 464
- SLOT 52
- Slot 49
- Socket 479, 527
- SocketSink 489
- SolidLine 310
- SolidPattern 312
- Source-kompatibel 581
- sprintf 420
- SquareCap 311
- Standard Template Library 396
- Standardaktionen 138
- Standardfarbpalette 765
- start 463, 468
- Startmenü 35
- startTimer 465
- state 352
- stateAfter 352
- static 281
- statusBar 150
- Statuszeile 115, 149, 456
- stderr 479, 492
- stdin 479, 492, 494
- stdout 479, 492
- STL 396
- STL *siehe* Standard Template Library

stop 468
Stream 521
strech factor 209
Streckungsfaktor 209
stripWhiteSpace 418
Stub-Objekte 562
subtract 321

T

tabArray 308
tabStops 308
take 406
Tastaturfokus 88
TCP/IP 4
temporäre Datei 525
Text 284
Texteditor 153
Theme 282
Thread 489
ThresholdAlphaDither 337
ThresholdDither 337
Tic Tac Toe 368
Tile 183
timeout 468
Timer-Event 463
timerEvent 348, 465, 728
timerId 465
tmake 595
toggled 57
toolBar 142
Tooltip 455
top 408
Toplevel-Widget 77
toString 461
tr 444
TransferJob 546
Transformation 314
translate 316, 590
TransparentMode 310, 313
Transparenz 341
Trolltech 2
TrueColor 274, 328
type 348

U

UConfig 454
UFT-16 431
Uhrzeit 460
Unicode 39, 169, 417, 423, 444, 522, 582
UnicodeNetworkOrder 438
UnicodeReverse 438

UnicodeUTF8 522
Uniform Resource Locator *siehe* URL
unite 321
Unix 2
Unter-Widget 78
update 86
upload 540
upper 418, 427
URL 538
UTF-16 438
UTF-8 39, 169, 430, 438

V

Value 278
Vater-Widget 76
VerPattern 312
Verzeichnis 519
Verzeichnisstruktur 35
Vier Gewinnt 368
View 186
viewport 316
viewportMouseEvent 375
viewportMouseEvent 376
viewportMouseEvent 376
View-Transformation 314
Visual 273
-visual 276

W

Währungsformat 447
Warteschlange 401
wasCanceled 476
WAV-Datei 495
Werkzeugleiste 115, 126
WFlags 84, 90
What's-This-Hilfe 455
whatsThisButton 457
wheelEvent 351
Whitespace 418
Widget 76
Widget-Palette 87, 282
width 314
widthMM 324
Winding-Algorithmus 305, 321
window 316
windowList 184
Windows 1
WordBreak 306
worldMatrix 319
World-Transformation 316
Write 482

writeEntry 450
writeStdin 492
wroteStdin 492
WStyle_Customize 342
WStyle_NoBorder 342
WStyle-Konstanten 90

X

X11 273
XFlush 490
XForms 5
XFree86 4, 13
xgettext 36, 102, 441
XLib 3
XML 117, 143, 145

XPM 128, 332
X-Server 2, 4, 9, 323, 327, 333
XSync 490
X-Terminal 4

Y

Yin-Yang 326, 727

Z

Zahlenformat 447
Zeigercontainer 399
Zuordnungstabelle 400, 408
Zwischenablage 161, 167, 497
Zwischenablage *siehe* Clipboard

