

The
Pragmatic
Programmers

The
Pragmatic
Programmers

Add Book

Sichern

Entwickeln mit dem iPhone SDK

Bill Dudney &
Chris Adamson

Übersetzt von Thomas Demmig

Q W E R T Z U I O P
A S D F G H J K L
Y X C V B N M

Return

Leerzeichen

.?123

Deutsche Übersetzung von

O'REILLY®

Was die Leser zu „Entwickeln mit dem iPhone SDK“ sagen

Ich liebe den direkten, nicht krampfhaft spaßigen Ansatz dieses Buchs. Chris und Bill schreiben verständlich, detailliert und gleichzeitig nicht übergenau oder belehrend. Für jemanden wie mich, der gerade erst damit beginnt, sich mit der iPhone-Entwicklung zu beschäftigen, führt an diesem Buch kein Weg vorbei.

Alex Lindsay

Gründer von Pixel Corps

Durch Duneys und Adamsons Buch wird das Erlernen der iPhone-Entwicklung rationeller. Die Themen werden umfassend und gleichzeitig so detailliert behandelt, dass jeder Entwickler sie versteht. Lesen Sie dieses Buch und kommen Sie dann zu unserem nächsten iPhoneDevCamp.

Raven Zachary

Gründer des iPhoneDevCamp

Präsident von Small Society

Wenn Sie die nächste große iPhone-Anwendung bauen wollen, ist dieses Buch der perfekte Ausgangspunkt. Von den ersten Kapiteln mit „Hallo Welt“ bis zu den fortgeschritteneren Themen wie dem Umgang mit Audio erklären Ihnen Dudney und Adamson, was Sie wissen müssen, um eine tolle iPhone-Anwendung zu schreiben.

Michael Sanford

Gründer von FlipSide5, Inc.

Jeder, der die iPhone-Entwicklung erlernen oder seine Kenntnisse verbessern will, sollte dieses Buch lesen. Anfänger werden die leicht verständlichen Codebeispiele von Bill Dudney und Chris Adamson zu schätzen wissen. Erfahrene Entwickler werden detaillierte Beispiele vorfinden, die das iPhone SDK nutzen. Definitiv eine Empfehlung.

Henry Balanon

Gründer und leitender Entwickler von Bickbot.inc

Autor von GigaOms TheAppleBlog.com

Dies ist eine ausgezeichnete Quelle für iPhone-Entwickler – vollständig und umfassend.

Marcus Crafter

iPhone-Entwickler, redartisan.com

Wenn Sie nach einer soliden Grundlage zum Erlernen der iPhone-Entwicklung suchen, empfehle ich dieses Buch unbedingt. Es bietet umfassende Codebeispiele und behandelt alle Themen im iPhone SDK-Umfeld. So können Sie direkt durchstarten.

John Muchow

Gründer von iPhoneDeveloperTips.com

CTO von 3 Sixty Software

Entwickeln mit dem
iPhone SDK

Entwickeln mit dem iPhone SDK

Bill Dudney
Chris Adamson

Deutsche Übersetzung von
Thomas Demmig

O'REILLY®

Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.

Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:

O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

Copyright der deutschen Ausgabe:
© 2010 by O'Reilly Verlag GmbH & Co. KG

Die Originalausgabe erschien 2009 unter dem Titel
iPhone SDK Development, 1st Edition bei Pragmatic Bookshelf, Inc.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte bibliografische Daten
sind im Internet über <http://dnb.d-nb.de> abrufbar.

Übersetzung und deutsche Bearbeitung: Thomas Demmig
Lektorat: Christine Haite, Köln
Korrektur: Sibylle Feldmann, Düsseldorf
Satz: Andreas Franke, SatzWERK, Siegen; www.satz-werk.com
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-951-9

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhaltsverzeichnis

Vorwort	1
1 Einführung	3
1.1 In diesem Buch	5
1.2 Danksagungen	7
2 Hallo iPhone	11
2.1 Starten	11
2.2 Das „Hallo iPhone“-Projekt erstellen	13
2.3 Eine einfache Oberfläche erstellen	16
2.4 Rotieren des Texts – Anpassen des UI	19
2.5 Rotieren des Texts – Anpassen des Codes	20
2.6 Im Code navigieren	21
2.7 Ihre Anwendung auf Ihrem iPhone ausführen	22
3 Grundlagen der iPhone-Entwicklung	25
3.1 Die Softwarearchitektur des iPhone	26
3.2 Code in Objective-C	27
3.3 Grundlegende Klassen von Cocoa Touch	29
3.4 Arbeiten mit Xcode und dem Interface Builder	30
3.5 Anatomie Ihrer iPhone-Anwendung	41
3.6 Verhaltensanpassung durch Delegieren	44
3.7 Speicherverwaltung der Anwendung	47
3.8 Zugriff auf Variablen als Eigenschaften	49
3.9 Kompakt: Was man sich merken sollte	52
4 View-Controller	55
4.1 Implementieren einer Button-Action	55
4.2 Ein Modell erstellen	61

4.3	Den Controller um Outlets und Actions ergänzen	64
4.4	Das UI anpassen	66
4.5	Den Controller implementieren	67
4.6	Den neuen View-Controller erstellen	70
4.7	Erstellen des UI.	74
4.8	Der MovieEditorViewController	78
4.9	Der Edit View-Controller im Interface Builder	80
5	Tabellen-Views	83
5.1	Elemente einer Tabelle	83
5.2	Tabellenbasierte Navigation einrichten	85
5.3	Tabellendaten modellieren.	87
5.4	Tabellenfelder	90
5.5	Tabellen bearbeiten.	94
5.6	In Tabellen navigieren	96
5.7	Eigene Felder in Tabellen-Views	104
5.8	Tabellendaten sortieren.	110
6	Navigation	115
6.1	Navigation in Mail	115
6.2	Der Navigations-Controller	116
6.3	Navigationsbasierte Anwendungen	117
6.4	View-Controller auf den Stack schieben.	121
6.5	Die Navigationsleiste anpassen	122
6.6	View-Controller vom Stack holen.	128
7	Zwischen Bereichen wechseln: Tab-Controller	131
7.1	Wann man Tabs verwendet	131
7.2	Einen Tab-Controller erstellen.	133
7.3	View-Controller in Tab-Controllern	135
7.4	Viele Controller	140
8	Datei-I/O	143
8.1	Das Dateisystem erforschen	144
8.2	Unser Projekt erstellen	147
8.3	Dateiattribute auslesen	154
8.4	Daten aus Dateien lesen	160
8.5	Asynchrones Lesen von Dateien	164
8.6	Dateien und Verzeichnisse erstellen und löschen.	168
8.7	Daten in Dateien schreiben	175
8.8	Eigenschaftslisten und NSCoder	178

9	Einstellungen	181
9.1	Anzeige einer umklappbaren Einstellungs-View in Ihrer Anwendung	181
9.2	Einstellungen verwalten	184
9.3	Einstellungen ändern und aktualisieren	187
9.4	Abstecher: Sekündliches Aktualisieren der Uhr	191
9.5	Die Systemanwendung für Einstellungen nutzen	192
9.6	Einstellungen laden, die in der System-App konfiguriert wurden	200
10	Die Datenbank SQLite	205
10.1	Ihre Datenbank erstellen	206
10.2	Die Beispielanwendung erstellen	209
10.3	Ihre Datenbank auf das iPhone bringen	212
10.4	Ihre Datenbank auf dem iPhone verwenden	215
11	Core Data	225
11.1	Die Konferenzanwendung	226
11.2	Der Core Data-Stack	229
11.3	Aufbau des Core Data-Stack	232
11.4	Modellieren	235
11.5	Track-Tabellen-View	243
11.6	Die Tracks einlesen	247
11.7	Tracks ändern	248
11.8	Navigation	252
12	Verbindung zum Internet	259
12.1	Ein Browser in zehn Minuten mit der UIWebView	259
12.2	Daten über das Netzwerk lesen	265
12.3	HTTP-Authentifizierung	270
12.4	XML von Webservices parsen	277
12.5	Mails aus Ihrer Anwendung schicken	289
13	Peer-to-Peer-Netzwerke	293
13.1	Ad-hoc-Netzwerk-Services mit Bonjour	293
13.2	Bonjour Service Discovery	295
13.3	Game Kit	301
13.4	Ein Spiel mit Bluetooth-Unterstützung aufsetzen	303
13.5	Erstellen eines Peer Pickers	304
13.6	Einen Peer Picker-Delegate bereitstellen	306
13.7	Spiellogik im Netz	308

13.8	Mit der GKSession kommunizieren.	310
13.9	Voice Chat.	316
14	Videos abspielen	319
14.1	Videos abspielen mit dem MPMoviePlayerController. . . .	319
14.2	Nachrichten vom Movie Player empfangen.	324
14.3	Unterstützte Medienformate	326
15	Zugriff auf die iPod-Bibliothek	329
15.1	Den iPod-Player überwachen.	329
15.2	Die iPod-Anwendung steuern	339
15.3	Die iPod-Bibliothek verwenden	341
15.4	In der iPod-Bibliothek stöbern.	352
16	Audiodaten abspielen und aufzeichnen	355
16.1	Einen AVAudioRecorder erstellen.	355
16.2	Unkomprimierte Audioformate	359
16.3	Kodierte Audioformate.	364
16.4	Den AVAudioRecorder verwenden	368
16.5	Aufzeichnungsinformationen anzeigen.	372
16.6	Audiodaten mit dem AVFramework abspielen	377
16.7	Audio-Sessions	382
17	Core Audio	387
17.1	Prozedurale C-APIs verwenden	388
17.2	Systemklänge abspielen	391
17.3	Überblick über Core Audio	397
18	Events, Gesten und Multi-Touch	403
18.1	Event-Modell	403
18.2	Berührungen verfolgen	406
18.3	Tippen.	408
18.4	Multi-Touch-Gesten	409
19	Zeichnen in eigenen Views	415
19.1	Zeichenmodell.	415
19.2	Vektorgrafik	416
19.3	Pfade.	418
19.4	Grafikkontext	423
19.5	Eine neue View anzeigen	425

20 Bilder und Fotos zeichnen	427
20.1 Grundlagen der Bildausgabe	428
20.2 Die Bildausgabe anpassen	430
20.3 Image Picker.	432
20.4 Videos aufnehmen	437
21 Core Animation	439
21.1 Einführung in Core Animation	439
21.2 UIView animieren	440
21.3 Layer	447
21.4 OpenGL ES	449
22 Accelerometer	453
22.1 Ausrichtung des Geräts ermitteln.	454
22.2 Schüttelgesten über die Responder-Kette erhalten	455
22.3 Rohdaten des Accelerometers auslesen.	456
22.4 Accelerometer-Daten filtern	463
23 Adressbuch	471
23.1 Adressbuch-UI	471
23.2 People Picker-Delegate	473
23.3 Den People Picker erzeugen und konfigurieren	475
23.4 Person-Controller	475
23.5 Neue Kontakte hinzufügen	477
24 iPhone Location-API	481
24.1 Wissen, wo man ist.	482
24.2 Positions-Updates.	486
24.3 Kompass	491
25 Map Kit	493
25.1 Contact Mapper	493
25.2 Eine Karte anzeigen	494
25.3 Anmerkungen in einer Karte.	497
25.4 Eine Anmerkung auswählen.	507
26 Anwendungsintegration	511
26.1 Starten anderer Anwendungen.	511
26.2 Selbst integrierbar werden	513

27 Debuggen	519
27.1 Fehler beim Build verstehen und beheben	520
27.2 Fehler beim Importieren/Linken verstehen und beheben	522
27.3 Die iPhone SDK-Dokumentation verwenden	525
27.4 Fehler im Interface Builder verstehen und beheben	528
27.5 Debuggen	530
27.6 Über-freigegebene „Zombie“-Objekten finden	539
28 Performance-Tuning	545
28.1 Performanceanalyse mit Shark	546
28.2 Performanceanalyse mit Instruments	551
28.3 Performanceanalyse mit dem Clang Static Analyzer	557
29 Davor und danach	563
29.1 Richtig beginnen	564
29.2 Abrunden	569
29.3 Weitere Features	571
29.4 Betatesten	572
29.5 Die App in den Store bringen	573
29.6 Für Ihre Anwendung werben	574
Literaturverzeichnis	577
Index	581

Vorwort

Am 9. Januar 2007 um 9:42 Uhr hat sich die Welt für immer verändert. Es wurde uns etwas Besonderes präsentiert, aber nicht jeder von uns wusste, wie besonders es in einer so kurzen Zeit werden würde.

Das Produkt, das wir als das iPhone kennen, ist mehr als ein Spielzeug. Es ist ein Phänomen. Viele von uns kennen die Nachrichten, Aktien-trends und den Terminkalender des neuen Tages schon, bevor sie aufstehen. Und dank des iPhone SDK gibt es mehr als 50.000 andere Dinge, die wir tun können – wo und wann wir wollen. Es ist tatsächlich „Your Life in Your Pocket“.

Als ich im Jahr 2001 bei Apple anfang, gab es keinen App Store, kein iPhone und keinen iPod. Das Mac OS X war erst ein paar Wochen alt. Es ist erstaunlich, zu sehen, wie weit sich all das entwickelt hat. Das iPhone hat seine eigene Ökonomie geschaffen, in der Sie keine Patente oder großen Geldgeber mehr benötigen, um erfolgreich zu sein. Sie brauchen nur noch eine Idee und die Motivation, sie auch umzusetzen.

Als iPhone-Entwickler sind Sie nicht nur Teil dieses Phänomens, sondern auch Mitglied einer Familie passionierter, brillanter Entwickler, Designer und Marketingexperten, die für ihre Arbeit mehr Stolz empfinden als die meisten anderen Menschen, denen Sie bisher begegnet sind. In vielen Fällen geht ihr Engagement auf die ersten Versionen des Mac OS X oder sogar noch auf das klassische Mac OS zurück. Konferenzen mit ihnen sind wie große Familientreffen. Sie teilen ihr Wissen mit „Konkurrenten“, weil sie wissen, dass bessere Anwendungen – egal von wem sie stammen – zu einer besseren Plattform führen. Sie lieben diese

Technologie. Sie leben sie. Sie haben ihre Karriere darauf aufgebaut. Dadurch dass ich seit mehr als acht Jahren mit ihnen zusammenarbeiten konnte, wurde ich dazu inspiriert, meinen Traumjob aufzugeben und mich ihnen anzuschließen.

Diese Kombination aus Technologie und Community hat zu diesen historischen Zeiten geführt, in denen wir jetzt leben. Nur ein Jahr nach der Eröffnung des App Store ist es schwer, sich noch daran zu erinnern, wie das Leben vorher war. Und es wird noch so viel mehr kommen.

Die Plattform wächst weiter, und wir alle lernen immer noch dazu. Ihr Engagement wird ebenso dazu beitragen. Dieses Buch ist Ihr erster Schritt auf dieser besonderen Reise.

Willkommen in unserer Welt. Sie werden sie lieben.

Matt Drance,
11. Juli 2009

Von meinem iPhone gesendet

Kapitel 1

Einführung

Das iPhone hat alles verändert.

Wenn Sie diese Zeilen lesen, wird Ihnen vielleicht gar nicht mehr auffallen, dass sich das neueste und coolste Mobiltelefon eines jeden Herstellers in Aussehen und Bedienung mehr an einem iPhone orientiert als an den früheren, eigenen Modellen. Apples eigene iPhone-Werbespots präsentieren dabei mittlerweile vor allem die Anwendungen, die Sie und andere Entwickler bauen. Die Marketingkampagne baut größtenteils darauf auf, dass das iPhone eine Plattform für tolle Software ist – Software, die Sie bauen.

Man übertreibt kaum, wenn man sagt, dass das iPhone SDK und der App Store die Art und Weise verändert haben, in der wir Software für mobile Geräte entwickeln, veröffentlichen und nutzen – und das bereits in der kurzen Zeit, in der beides verfügbar ist.

Vor dem iPhone war das Schreiben von Software für solche kleinen Geräte gespickt mit Fallen. Viele Entwickler beklagten sich über eine starke Fragmentierung aufgrund der sehr unterschiedlichen Eigenschaften der verschiedenen Geräte (Bildschirmgröße, Farbtiefe, Eingabemöglichkeiten und so weiter) und der unterschiedlichen Implementierungen von vermeintlichen Standardtechnologien. Durch diese Differenzen waren die Entwickler gezwungen, das teure und schwierige Spiel des „Write once, test everywhere“ zu spielen. Und das auch nur dann, wenn sie einen Markt bedienen konnten. In manchen Fällen nutzten die Gerätehersteller und Mobilfunkfirmen die Sicherheitsfeatures des Technologieanbieters, um ihre Geschäftsmodelle abzusichern und nur eigene Anwendungen auf dem Gerät zuzulassen oder Programme von dritter Seite bei grundlegenden Diensten einzuschrän-

ken. Das hat ausgereicht, um die Entwickler zurück an den Desktop fliehen zu lassen – wenn nicht gar zum Server.

Als iPhone SDK-Entwickler müssen Sie sich um eine Fragmentierung keine Gedanken machen. iPhone und iPod touch verhalten sich sehr vorhersagbar, und es gibt nur ein paar Modelle mit konsistenten Features und weitgehend gleichem Verhalten. Das SDK bringt die Tools und Technologien des Mac OS X mit – im jahrelangen Einsatz für ausgezeichnete Desktop-Anwendungen erprobt und verbessert – und stellt die grundlegenden Elemente für iPhone- und iPod touch-Anwendungen bereit. Während andere mobile Plattformen eine Exception auslösen, wenn Sie den Netzwerk-Stack ansprechen, erhalten Sie beim iPhone solch ausnehmend gute Technologien bereitgestellt wie das sich selbst vernetzende Bonjour. Auf anderen Geräten ist das Abspielen von Sounddateien eher eine zufällige Angelegenheit, während Sie beim iPhone eine umfangreiche Unterstützung für alle möglichen Medien erhalten. Und wenn Ihre Anwendung fertig ist, müssen Sie nicht mit einem Anbieter zusammenkommen, der Ihr „Partner“ sein will (und 99 Cent von jedem verdienten Euro nimmt), und sie brauchen auch nicht davon auszugehen, gar nicht erst zum Zuge zu kommen, weil Sie kein Big Player sind, der mit ausreichend Patenten winken kann. Im App Store können Sie Ihre Anwendung jedem iPhone- und iPod touch-Benutzer anbieten. Sie bestimmen den Preis, von dem Apple einen deutlich kleineren (und durchaus vernünftigen) Anteil abbekommt.

Das Ergebnis ist revolutionär. Der App Store wächst derart schnell, dass es sich nicht lohnt, irgendwelche Zahlen zu nennen, weil sie schon veraltet wären, wenn Sie dieses Buch lesen. Stattdessen ist es vielleicht hilfreicher, die grundlegenden Veränderungen zu beobachten, die sich in der Branche durch den Doppelschlag SDK und App Store ergeben haben. Netzanbieter, die bisher ihre Netze für andere gesperrt hatten, beeilen sich nun, ihren eigenen Anwendungsshop zu öffnen, weil sie Angst haben, dass jede coole, neue Anwendung für das iPhone potenziell noch mehr ihrer Kunden weglockt. Einige Mobile-Entwickler arbeiten nur noch für das iPhone. Und da iPhone-Anwender anscheinend bereitwilliger Programme herunterladen und bezahlen, ist es recht abenteuerlich, darauf zu hoffen, auf anderen Plattformen eine Benutzerbasis aufbauen zu können, die dann keine Anwendungen von dritter Seite herunterlädt. In Unternehmen schließlich werden teilweise eigene iPhone-Anwendungen geschrieben und an den Außendienst verteilt.

Wenn Sie dieses Buch lesen, ist die Wahrscheinlichkeit recht hoch, dass auch Sie bei dieser mobilen Revolution mitmachen wollen.

Und wir sind froh, dass Sie das tun, denn wir wollen Ihnen auf dieser Reise unter die Arme greifen – vom Herunterladen des SDK bis zum Veröffentlichen Ihrer ersten Anwendung.

Wir gehen zunächst einmal nur von zwei Dingen aus:

- Sie haben einen Computer, auf dem das aktuelle iPhone SDK laufen kann. Das ist im Moment ein Mac mit einem Intel-Prozessor, auf dem Mac OS X Leopard 10.5.4 oder neuer läuft.¹
- Sie sind mit einer der „Curly Brace“-Programmiersprachen (C, C++, C#, Java, Ruby und so weiter) und objektorientierter Programmierung im Allgemeinen vertraut. Wenn Sie bisher eher mit Skriptsprachen gearbeitet haben (JavaScript, ActionScript, PHP), werden einige der Programmierpraktiken für das iPhone eine Herausforderung sein, aber hoffentlich eine, die nicht unüberwindlich ist.

1.1 In diesem Buch

Wir beginnen Ihre Reise mit einer Vorstellung der Plattform. Es geht los mit einem schnellen Erfolg in Form einer „Hallo Welt“-iPhone-Anwendung. Gestärkt durch diese erste Erfahrung steigen wir dann in ein paar Plattformgrundlagen ein, die Sie im Folgenden brauchen. Wir behandeln das Basiswissen zu den Tools (Xcode, Interface Builder) und zu Objective-C – der Sprache für die Cocoa Touch-Entwicklung.

In den nächsten vier Kapiteln schauen wir uns View Controller und die Views an, die die Basis der meisten iPhone-Anwendungen bilden. Zunächst geht es um die allgemeineren Views und View-Controller, dann kümmern wir uns um die UI-Metaphern, die Bestandteil von so gut wie jeder iPhone-Anwendung sind: Tab-Leisten, Navigationselementen und Tabellen-Views.

Wir empfehlen, diese Kapitel auf jeden Fall zu lesen – egal was für eine Art iPhone-Anwendung Sie schreiben wollen. Sie können dann den Rest des Buchs der Reihe nach durchlesen oder sich die für Ihre Erfordernisse notwendigen Kapitel herauspicken. Die verbleibenden Kapitel lassen sich grob in sechs Abschnitte unterteilen.

¹ Die Anforderungen des SDK, insbesondere die minimale Betriebssystemversion werden sich sehr wahrscheinlich in Zukunft ändern. Unter <http://developer.apple.com/iphone> finden Sie die aktuellen Spezifikationen.

Wenn Sie auf Ihrem Gerät Daten ablegen müssen, finden Sie die verschiedenen Techniken in vier Kapiteln beschrieben. Wir beginnen mit dem Zugriff auf das Dateisystem und die Optionen zum Speichern und Auslesen von Einstellungen. Dann zeigen wir zwei Möglichkeiten, die eingebaute Datenbank zu nutzen: direkt mit der SQLite-API oder über das objektrelationale Mapping durch das Core Data Framework aus dem iPhone OS 3.0.

Ihre Anwendung erfordert möglicherweise die Kommunikation mit Servern oder anderen Geräten, daher sind die nächsten zwei Kapitel dem Leben im Netzwerk gewidmet. Im ersten verbinden wir uns mit Websites, Webservices und E-Mail-Diensten im öffentlichen Internet, während das zweite die selbstvernetzenden Fähigkeiten von Bonjour und das in 3.0 neue Game Kit Framework behandelt, mit dem sich Geräte mit iPhone OS untereinander verbinden können.

Der nächste Abschnitt besteht aus vier Kapiteln, in denen es um Medien auf Ihrem Gerät geht. Wir beginnen mit dem einfachen Framework für das Betrachten von Videos auf dem iPhone. Dann schauen wir uns das in 3.0 neu hinzugekommene Framework für den Zugriff auf die iPod-Musikbibliothek des Benutzers an. Entwickler, die die Audiofähigkeiten der Plattform ausnutzen wollen, finden ein Kapitel über das Abspielen und Aufzeichnen von Audiodaten mit dem in 3.0 neuen Framework AVFoundation. Wir schließen den Abschnitt mit einem Überblick über das Core Audio Framework, das auf einer tieferen Ebene ansetzt.

Sie werden vermutlich die Art und Weise anpassen wollen, in der Ihre Anwender mit Ihrer Anwendung kommunizieren. Wir kehren also zum Präsentationsbereich zurück und schauen uns Events, Multi-Touch und Gesten an. Das Anzeigen selbst gezeichneter Views ist Thema der nächsten drei Kapitel. Wir besprechen das Zeichnen mit der umfangreichen Core Graphics Library, aber auch das Bearbeiten und Zeichnen von Bildern, die entweder von Ihrer Anwendung bereitgestellt werden oder aus der Bildbibliothek des Benutzers stammen. Dieser Abschnitt endet mit einem Kapitel zu Core Animation, der Technologie, die all den schönen und natürlich anmutenden Animationen zugrunde liegt, die Ihnen überall im iPhone-UI begegnen.

Als Nächstes behandeln wir die spezielleren Technologien, die nicht jede iPhone-Anwendung nutzen wird, aber die Ihr Programm bei einem Einsatz mit Augenmaß zu etwas Besonderem machen werden. Manche Entwickler werden daran interessiert sein, das eingebaute Accelerometer zu verwenden, um die Ausrichtung und Bewegung des Geräts zu

erkennen und es dem Benutzer damit zu ermöglichen, die Anwendung auf eine neue und interessante Art und Weise zu bedienen. Entwickler von Anwendungen zur Kommunikation mit anderen Personen können das Address Book Framework nutzen, um auf die Benutzerkontakte zuzugreifen. Und weil iPhone und iPod touch Geräte sind, die man nicht nur zu Hause liegen hat, gibt es auch zwei dazu passende Kapitel – eines über das Core Location Framework, mit dem man herausfinden kann, wo sich das Gerät befindet, und eines über das in 3.0 neue Map Kit Framework, mit dem Sie Kartendaten in Ihrer Anwendung präsentieren können.

Der letzte Abschnitt bietet schließlich vier Kapitel über die letzten verbleibenden Schritte an, mit denen Ihre Anwendung rund wird. „Application Integration“-APIs ermöglichen es Ihrer Anwendung, andere Anwendungen zu starten und von anderen Anwendungen aufgerufen zu werden. Sie werden lernen, Ihre Applikation noch zu perfektionieren, indem beschrieben wird, wie Sie Ihren Code debuggen und wie Sie die Performance mit den verschiedenen Tools von Xcode verbessern. Ein letztes Kapitel fügt alles zusammen – wie Sie Ihren Entwicklungsprozess optimieren, Ihr Produkt in den App Store bekommen und es dort der Öffentlichkeit präsentieren.

1.2 Danksagungen

Von Chris Adamson

Als ich 2001 Teil der großen Entlassungswelle war, hatte ich mir geschworen, mit der Entwicklung von mobilen Anwendungen fertig zu sein. Ein paar Jahre später, im Endspurt meines zweiten Buchs, schwor ich mir, nie wieder eines zu schreiben. Da ich jetzt also eine Einführung zu einem Buch über die iPhone-Entwicklung schreibe, wäre es vielleicht angebracht, ein paar erklärende Worte fallen zu lassen.

Eigentlich lässt sich alles durch einen Moment im März 2008 erklären. Ich war auf einer Java-Konferenz und lauschte recht skeptisch der Ankündigung zum iPhone SDK. Eigentlich ging keiner davon aus, dass wir mit dem SDK eine ernsthafte Unterstützung erhalten würden. Daher waren alle ausgesprochen überrascht, dass es einen Großteil von Mac OS X Cocoa enthielt, wobei die Oberfläche sorgfältig an ein berührungsempfindliches Handheld-Gerät angepasst wurde. Während ich es bei den durch die Netzwerkanbieter eingeschränkten Java-Mobiltelefonen nie geschafft hatte, auf das Netzwerk zuzugreifen, bot das SDK

hier *echte BSD-Sockets* und als Bonus das selbstvernetzende Bonjour an. Mediendaten, die auf anderen Plattformen häufig nicht ernsthaft eingesetzt werden konnten, wurden hier mit Core Audio unterstützt – durch den gleichen Code wie in professionellen Audioanwendungen auf dem Mac.

Und das alles für ein Telefon? Ich hatte Blut geleckt. Das war der Moment, auf den ich wohl die ganze Zeit gewartet hatte: Ich wusste sofort, dass alles, was ich in den letzten Jahren getan hatte, unwichtig geworden war und dass ich das hier wollte – brauchte –, um weiterzumachen. Ich lud noch in der Nacht das SDK über eine sehr langsame DSL-Leitung herunter, programmierte auf dem Flug nach Hause meine erste „Hello World“-Anwendung und wurde ein paar Monate später für dieses Buch engagiert.

Natürlich muss ich unserem Lektor Daniel Steinberg dafür danken, dass er mich ansprach und davon überzeugte, der richtige Koautor für dieses Buch zu sein. Er hat dieses riesige Projekt über all die Monate des Schreibens und Überarbeitens in der Spur gehalten, auch als wir alles anhielten, weil wir Sorge hatten, dass das Non-Disclosure Agreement von Apple uns davon abhalten würde, das Buch jemals veröffentlichen zu können. Als das NDA dann fiel, schusterten wir unser erstes Beta-Release zusammen. Andy und Dave von Prags sei unser Dank gewiss auch für eine sehr produktives (und ja, ich sage es – *agiles*) System zum Schreiben von Büchern. Bill war ein sehr aktiver und produktiver Koautor, jemand, dessen Stärken genau zu meinen Schwächen passten. Wir beide danken auch unseren vielen technischen Korrektoren und der großen Community, die auf <http://pragprog.com/> auf der Forum- und Fehlerseite zum Buch Tonnen von Rückmeldungen gegeben haben. Und wie immer einen großen Dank an meine Familie (Kelly, Keagan und Quinn), die irgendwie nicht nur den Stress und die Verluste durch diesen Buchmarathon, sondern auch noch durch einen Umzug quer durch den Kontinent überstanden haben – nur ein paar Wochen vor Beta 1.

Die obligatorische Liste der gehörten Musik: Dieses Mal war es Bend Sinister, My Chemical Romance, Rilo Kiley, The Polyphonic Spree, Immaculate Machine, ... And You Will Know Us by the Trail of Dead, The Cribs, Kaki King sowie die Streams und Podcasts von CBC Radio 3.²

2 Aktuelle Informationen finden Sie unter <http://www.last.fm/user/invalidname>.

Von Bill Dudley

Das Schreiben eines Buchs ist eine ziemlich große Sache. Aber natürlich sind es nicht nur die Autoren, die ihr Bestes geben, um für einen Inhalt zu sorgen. Die Lektoren (zumindest die, die meine Texte gelesen haben) arbeiten genauso hart (wenn nicht härter), um das Techigelaber und den passiven Schreibstil in ein akzeptables Englisch umzuformen. Daher bedanke ich mich bei Daniel dafür, hier wieder alles gegeben zu haben.

Das iPhone-Entwicklungsteam hat natürlich auch ein riesiges Dankeschön dafür verdient, Tag und Nacht daran zu arbeiten, etwas herzustellen, das so viel Spaß macht.

Die Korrektoren haben ebenfalls sehr viel Zeit geopfert. Es haben viele geholfen, aber ein paar blieben als besonders hilfreich im Gedächtnis. Daher möchte ich mich (ohne durch die Reihenfolge werten zu wollen) bei den folgenden Leuten bedanken: Tim Isted, Steven Troughton Smith, Patrick Burleson, Jason Hunter, Andy Simmons, Marcus Cramer, Tom Hauburger, David Hodge, David Smith und Dee Wu. Das Buch ist durch eure harte Arbeit noch viel besser geworden!

Ich möchte mich ebenso bei den vielen Menschen bedanken, die im iPhone Studio die richtigen Fragen gestellt haben, so dass ich viele Dinge aus einer anderen Perspektive betrachten konnte. Diese Fragen und meine Reaktionen darauf haben ebenfalls bei der Entstehung dieses Buchs geholfen. Danke sehr!

Auch meine Familie hat mich auf der langen Reise dieses Buchs sehr unterstützt. Vielen Dank dafür, dass ich nachts lange aufbleiben und früh am Morgen arbeiten durfte (und den größten Teil der Zeit dazwischen), um das Buch fertigzustellen. Ich möchte insbesondere meinem ältesten Sohn Andrew danken, der eine große Hilfe darin war, die Dinge zu finden, die ich immer wieder vergessen hatte. Und schließlich möchte ich dem 2000 Jahre alten jüdischen Zimmermann dafür danken, dass mein Leben so viel reicher geworden ist.

Wir sind von dieser Plattform wirklich begeistert und gespannt darauf, was Sie mit Ihrem neu gewonnenen Wissen anstellen werden. Es ist toll, ein iPhone-Entwickler zu sein, und es ist toll, Sie als Teil der Community dabeizuhaben!

Kapitel 2

Hallo iPhone

Das iPhone ist eine erstaunliche Plattform. Es ist angefüllt mit cooler Technologie, die Sie geradezu dazu auffordert, etwas damit zu machen. Der App Store ist voll mit Anwendungen, die Sie inspirieren und motivieren werden. Die Möglichkeiten sind praktisch unbegrenzt.

Lassen Sie uns mit einer einfachen „Hallo Welt“-Anwendung beginnen. Im Rest des Buchs werden Sie viele der Technologien und APIs kennenlernen. Sie werden alles erfahren – von der Positionsbestimmung bis zum Accelerometer, von den View-Controllern bis zu Core Animation. In diesem Kapitel machen wir Sie aber erst einmal mit den Tools und dem grundlegenden Entwicklungszyklus für das iPhone vertraut.

Unsere erste Anwendung ist mit Absicht sehr einfach. Wir erstellen ein Programm, das den statischen Text „Hallo iPhone“ auf dem Bildschirm ausgibt. Sie werden zuerst prüfen, ob Sie alle Entwicklungstools installiert haben, die Sie benötigen. Dann werden Sie zwei davon nutzen, um Ihre Anwendung zu erstellen und auf dem iPhone-Simulator laufen zu lassen. Da dies eine iPhone-Anwendung ist, werden Sie den Text rotieren lassen, wenn das Gerät auf die Seite gedreht wird. Schließlich werden Sie diese Anwendung herunterladen und auf Ihrem iPhone ausführen.

2.1 Starten

Wenn Sie schon Anwendungen für Mac OS X entwickeln, sind Sie mit Xcode, dem Interface Builder und den anderen Entwicklungstools vertraut. Die Anwendungsentwicklung für das iPhone läuft anders ab, weil das Zielgerät ein anderes ist, aber viele der Tools, APIs und Fähigkeiten, die Sie einsetzen, sind die gleichen. Dieses erste Kapitel wird dann für Sie eine einfache Übung sein.

Terminologie

Wenn wir „iPhone“ sagen, meinen wir ein beliebiges Gerät mit iPhone OS. Wenn Sie also einen iPod touch Ihr Eigen nennen, fühlen Sie sich nicht ausgeschlossen – auch Sie sind gemeint. Was wir in dieser Ausgabe allerdings (noch) nicht behandeln, ist die Entwicklung für das neue iPad.

Sollten Sie von einer anderen Plattform zum iPhone kommen – willkommen. Sie haben in der Vergangenheit bestimmt schon eine Integrierte Entwicklungsumgebung – englisch *Integrated Development Environment (IDE)* – verwendet. Für iPhone-Entwickler erfüllt *Xcode* diesen Zweck. *Xcode* bietet alle Features, die Sie brauchen, wie zum Beispiel die Dateiverwaltung, das Kompilieren, Debuggen und die Fehlermeldungen. Wenn Sie sich erst einmal mit *Xcode* vertraut gemacht haben, werden Sie sich darin genauso schnell zurechtfinden wie in der Ihnen geläufigen IDE. Kennen Sie sich mit dem Entwickeln von Programmen noch so gar nicht aus, sind wir sehr zuversichtlich, dass Sie es spannend und interessant finden.

Sie werden während der Entwicklung für Ihr iPhone vor allem drei Tools verwenden: *Xcode*, *Interface Builder* und *Instruments*. Mit *Xcode* verwalten Sie Ihr Projekt und schreiben, starten und debuggen Ihren Code. Mit dem *Interface Builder* erstellen Sie Ihre Benutzerschnittstelle und verbinden sie mit Ihrem Code. *Instruments* hilft Ihnen dabei, Bereiche zu finden, die die Performance ausbremsen. Dieses Buch wird keines dieser Tools bis ins kleinste Detail behandeln, wir werden Ihnen aber dabei helfen, alle drei zu nutzen, und Sie auch auf Features hinweisen, die notwendig sind, um ein bestimmtes Ziel für die zu bauende Anwendung zu erreichen. Wann immer Sie mehr über ein bestimmtes Feature von *Xcode* erfahren wollen, sollten Sie auf die Dokumentation von Apple zurückgreifen. Sie ist ausgezeichnet und in *Xcode* eingebaut.

Lassen Sie uns sicherstellen, dass Sie alle Entwicklungstools installiert haben. Wenn das nicht schon geschehen ist, laden Sie das iPhone SDK von <http://developer.apple.com/iphone> herunter. Sie müssen sich als iPhone Developer anmelden – das ist kostenlos. Um am iPhone Developer Program teilzunehmen, müssen Sie hingegen 99 US-Dollar bezahlen, aber das brauchen Sie, um ein Zertifikat zu erhalten, mit dem Sie Ihre Anwendungen auf Ihrem Gerät laufen lassen können.

Nachdem Sie das Paket heruntergeladen haben, installieren Sie es durch einen Doppelklick auf die .dmg-Datei und dann auf das iPhone SDK-Paket. Folgen Sie den Anweisungen auf dem Bildschirm und wählen Sie den Standardpfad, um die Developer Tools zu installieren. Wenn die Installation abgeschlossen ist, finden Sie Xcode unter `/Developer/Applications/Xcode.app`. Haben Sie ein anderes Ausgangsverzeichnis gewählt, sollte der Pfad dementsprechend sein.

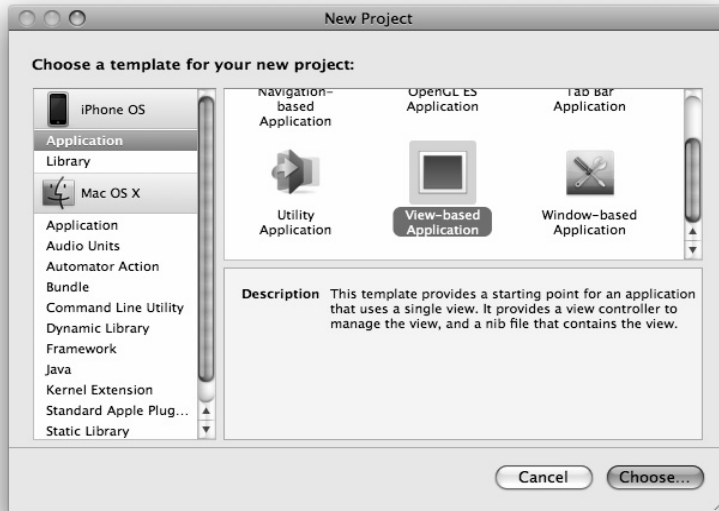
Führen Sie Xcode durch einen Doppelklick aus. Sie werden Xcode, den Interface Builder und Instruments vermutlich praktischerweise im Dock behalten wollen, um leichter darauf zugreifen zu können.

Wenn Sie Xcode aufrufen, sehen Sie den netten Willkommensbildschirm, von dem aus Sie eine ganze Reihe von Informationen erreichen können. Über „Getting started with Xcode“ gelangen Sie zum Bereich der Developer Documentation. Rechts im Kasten findet sich auch ein Link „RSS Feeds“, über den Sie stets informiert werden, welche neuen Beispiele und Dokumentationen Apple herausgebracht hat. Das iPhone Dev Center enthält viele, viele Links auf die unterschiedlichen Bereiche. Lassen Sie diesen Ausgangsbildschirm ruhig bei jedem Start von Xcode öffnen, zumindest bis Sie mit dem Programm ausreichend vertraut sind. Nehmen Sie sich die Zeit, die einzelnen Abschnitte zu durchstöbern, um zu wissen, was sich wo befindet. Das ist der beste Weg, Xcode kennenzulernen.

2.2 Das „Hallo iPhone“-Projekt erstellen

Nachdem Xcode nun bei Ihnen läuft, legen Sie Ihr erstes Projekt an. Wählen Sie `FILE` → `NEW PROJECT` oder verwenden Sie , um den „New Project Wizard“ zu starten. Wenn Sie dort `IPHONE OS` → `APPLICATION` anwählen, erhalten Sie eine Liste mit Projektvorlagen (Templates) für das iPhone.

Wir werden in den Beispielen hier im Buch viele der Vorlagen verwenden. Jetzt wählen Sie bitte die Vorlage „View-based Application“ und klicken auf den Button `CHOOSE`.



Wenn das Fenster für die Ordnerauswahl zu sehen ist, wählen Sie einen aus und geben dem Projekt den Namen Hallo. Xcode nutzt das Template „View-based Application“, um eine Anwendung mit einer einzelnen View zu erstellen. Es werden für uns eine ganze Reihe weiterer Dateien erzeugt, aber die schauen wir uns später an, daher brauchen Sie sich darum zurzeit keine Gedanken zu machen. Sie sollten nun etwas haben, das so aussieht wie das in Abbildung 2.1.

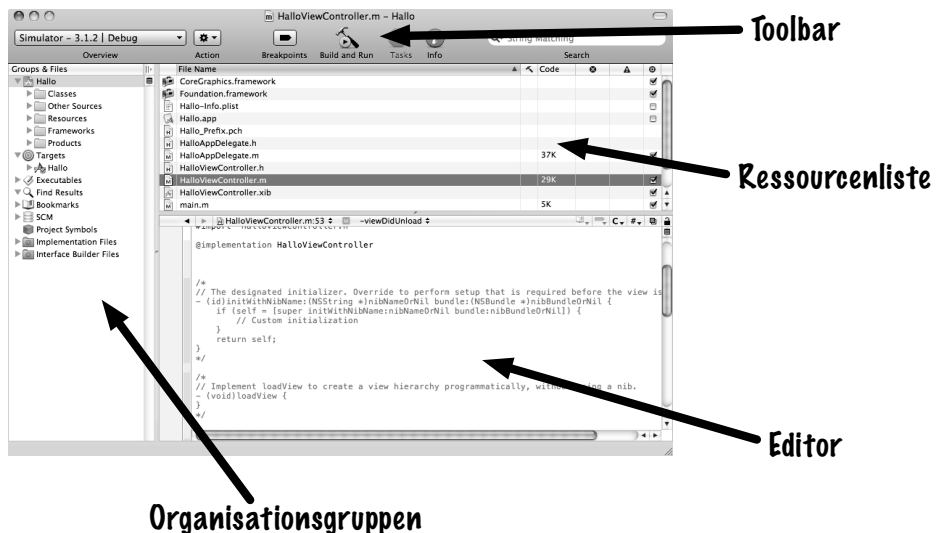


Abbildung 2.1: Hallo iPhone-Projekt

Auf der linken Seite des Fensters sehen Sie eine Liste mit Organisationsgruppen. Jede dieser Gruppen kann von Ihnen mit Leben gefüllt werden, damit Sie nicht im Chaos versinken. Die Gruppen sind nicht implizit mit einer Dateisystemstruktur verbunden, daher können Sie hier Ihren Code und Ihre Ressourcen so organisieren, wie es Ihnen am sinnvollsten erscheint. Wenn Sie gerade beginnen, ist es häufig einfacher, das initiale Setup zu nutzen, aber wenn Sie schon eine Struktur haben, die Ihnen mehr liegt, können Sie sich hier gern austoben.

Am oberen Rand des Fensters befindet sich die Toolbar. Dieser konfigurierbare Bereich dient dazu, Befehle abzulegen, die Sie häufig nutzen, so dass Sie sie schnell aufrufen können. Wir haben hier das Auswahlménü OVERVIEW, mit dem wir zwischen dem iPhone-Simulator und dem eigentlichen Gerät als Ziel für die zu bauende Anwendung wechseln können. Wenn Sie dieses Auswahlménü nicht sehen, sollten Sie es hinzufügen (bei manchen Entwicklern ist es von Anfang an zu sehen, dann brauchen Sie hier nichts zu tun).

Um Ihre Toolbar anzupassen, wählen Sie VIEW → CUSTOMIZE TOOLBAR, oder Sie klicken mit gedrückter -Taste in den Hintergrund der Toolbar und wählen im Menü CUSTOMIZE TOOLBAR aus. Wählen Sie das Element OVERVIEW aus und ziehen Sie es auf die Toolbar. Dabei wandelt sich der Cursor in einen grünen Kreis mit einem + darin. Lassen Sie das OVERVIEW-Element auf der linken Seite der Toolbar fallen.

Auf der rechten Seite unterhalb der Toolbar befindet sich die Datei- und Ressourcenliste. Diese Liste zeigt Dateien an, die Teil der links in der Liste „Groups & Files“ ausgewählten Gruppe sind. Da aktuell das Projekt markiert ist, werden alle Dateien im Projekt angezeigt. Klicken Sie auf eine Datei in dieser Liste, wird sie direkt darunter im Editor-Bereich angezeigt. Die Datei- und Ressourcenliste orientiert sich an GROUPS & FILES – wählen Sie dort eine andere Gruppe aus, ändert sich auch die Liste und zeigt nur die Dateien an, die Teil dieser Gruppe sind.

Der Editor-Bereich ist der Bereich, in dem Sie einen Großteil Ihrer Arbeitszeit verbringen werden. Hier haben Sie Zugriff auf die meisten der codeorientierten Features (Vervollständigung, Refaktorisieren und so weiter).

Nachdem wir Xcode nun im Schnelldurchlauf kennengelernt haben, wollen wir unsere neue Anwendung starten, um zu sehen, was wir erhalten. Klicken Sie auf den Button BUILD AND RUN (oder drücken Sie -) , Xcode kümmert sich dann um den Rest. iPhone-Anwendungen sind wie Mac-Anwendungen spezielle Verzeichnisse, auch als

Anwendungs-Bundles (App Bundle) bezeichnet. Während des Build-Prozesses erstellt Xcode ein Anwendungs-Bundle mit Ihrem kompilierten Code; Ihre Ressourcen werden in diesem Bundle an die entsprechenden Stellen kopiert.

Nachdem Xcode nun Ihre Anwendung gebaut und im Simulator installiert hat, wird sie dort auch aufgerufen. Dieser erste Lauf wird zu einem einfachen hellgrauen Hintergrund führen (eine tolle Taschenlampen-App, wenn auch nicht ganz so hell wie mit einem weißen Hintergrund). Mehr ist noch nicht zu sehen, aber wir haben unsere Vorlage ja auch noch nicht angepasst. Lassen Sie uns also ein paar Änderungen an der Oberfläche vornehmen.

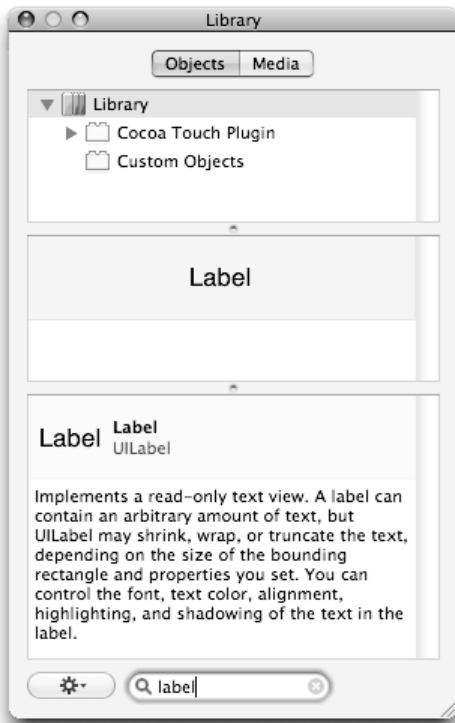
2.3 Eine einfache Oberfläche erstellen

Jetzt ist der richtige Zeitpunkt für den Interface Builder gekommen. Es ist eines der wichtigeren Werkzeuge, die Sie beim Erstellen von iPhone-Anwendungen verwenden werden, und das wichtigste Tool für das Layout und die Konfiguration der Benutzerschnittstelle (User Interface, UI).

Bisher *tut* die Anwendung noch nichts, daher müssen wir auch keinen Code schreiben. Wir werden die Tools zum Erstellen der Oberfläche nutzen. Der Interface Builder (IB) ist, wie der Name schon nahelegt, dafür verantwortlich, die Benutzeroberfläche zu erstellen, über die unsere Anwender interagieren werden. Der IB kennt die UIKit-Objekte, aus denen die Oberfläche erstellt wird, und die Human Interface Guidelines (HIG), die uns dabei helfen sollen, konsistente UIs zu erzeugen. Das ist ein sehr mächtiges Tool, wenn Sie sich erst einmal damit vertraut gemacht haben. Wie bei Xcode kann dieses Buch bei Weitem nicht alle Feinheiten des IB vorstellen, aber mithilfe der Erfahrungen, die Sie beim Erstellen der Beispiele in diesem Buch machen, werden Sie genug lernen, um selbst einzusteigen und sich mit dem Tool zu beschäftigen.

Lassen Sie uns nun die Oberfläche für Ihre erste Anwendung erstellen. Öffnen Sie `HalloViewController.xib`, indem Sie die Datei in Xcode doppelt anklicken (in der Gruppe „Ressources“). Wir wollen ein Label hinzufügen, das den Text „Hallo iPhone“ enthält. Dazu öffnen Sie die Library mit -- (oder `TOOLS → LIBRARY`).

In der Library finden Sie alle Interface- und Controller-Objekte, die Sie in Ihrer `.nib`-Datei (siehe Kasten) instantiieren können. Jetzt suchen wir nach einem Label, also tippen Sie am einfachsten in das Filter-Feld unten im Fenster den Text `label` ein.



Ziehen Sie das Label auf die View und verschieben Sie es, bis es an der Hilfslinie an der linken Seite einrastet.

Wenn das Label jetzt nicht ausgewählt ist, markieren Sie es. Die kleinen blauen Kreise an den Ecken des Labels ermöglichen es Ihnen, die Größe zu verändern. Vergrößern Sie es so weit, dass es von der linken bis zur rechten Hilfslinie reicht, damit es die gesamte Breite einnimmt – abzüglich des Rahmens, der von den Richtlinien empfohlen wird. Verpassen Sie ihm auch eine Höhe von 80 (Sie können dazu die Maus nutzen oder einfach per `⌘-3` den Size Inspector anwählen). Nun ändern Sie mit `⌘-T` die Schriftart und wählen eine Schriftgröße von 48 Punkt. Klicken Sie das Label doppelt an, um den Text zu bearbeiten. Ändern Sie ihn in Hallo iPhone. Sie können auch die Farbe des Texts anpassen und einen Schatten oder andere Attribute hinzufügen, und zwar im Attributes Inspector (`⌘-1`). Dort können Sie ebenfalls dafür sorgen, dass der Text zentriert erscheint.

So sollte es dann aussehen, wenn der Text zentriert, rot und mit einem Schatten versehen ist:



Jetzt können wir die Anwendung ausführen. Speichern Sie Ihre Änderungen im Interface Builder. Wechseln Sie zurück zu Xcode, klicken Sie auf den Button BUILD AND RUN und bewundern Sie Ihr Meisterwerk. Sobald der Simulator gestartet ist, sollten Sie den Text des Labels in der Mitte der View auf dem grauen Hintergrund sehen. Wählen Sie im Simulator das Menüelement HARDWARE → LINKS DREHEN oder HARDWARE → RECHTS DREHEN. Der Simulator dreht das Gerät in 90-Grad-Schritten nach links oder rechts, und der Text dreht sich mit. Wir werden noch entsprechende Anpassungen vornehmen, damit der Text beim Drehen des Geräts lesbar bleibt.



Joe fragt...

Was hat es mit XIB und NIB auf sich?

Der Interface Builder kennt zwei Dateitypen, `.xib` und `.nib`. Die Erweiterung `.xib` ist einfach die XML-Version der `.nib`-Datei. Da der `.nib`-Dateityp schon seit Mac OS X 10.0 (und sogar schon vorher in NeXTStep) existiert und `.xib` recht neu ist, reden viele immer noch von `.nib`-Dateien. Wir werden das hier im Buch auch tun.

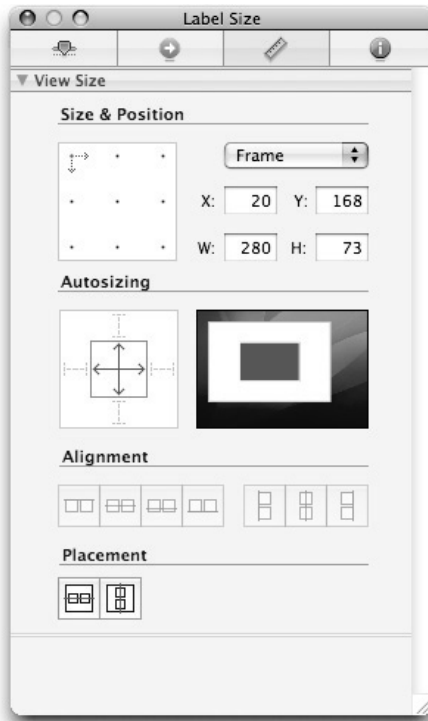
2.4 Rotieren des Texts – Anpassen des UI

Sie müssen zwei Schritte durchführen, um korrekt darauf vorbereitet zu sein, wenn das Gerät gedreht wird. Zum einen müssen Sie ein paar Parameter auf der `UIView` im Interface Builder setzen, und zum anderen müssen Sie mit Xcode eine Codezeile in der Klasse `HalloViewController` ändern. In diesem Abschnitt werden wir das UI-Anpassungen vornehmen, im nächsten dann die Änderungen am Code.

Wählen Sie das Label aus und nutzen Sie den Size Inspector (-) unter dem Autosizing-Abschnitt, um die Konfiguration von Struts und Springs anzupassen. Wir wollen, dass das Label beim Rotieren zentriert bleibt. Daher schalten wir die Struts für alle vier Kanten aus und aktivieren die Springs für beide Richtungen.

Die Konfiguration von Struts und Springs ermöglicht uns, bei Views festzulegen, wie sich die View verhalten soll, wenn die eigene Größe oder die der umgebenden View verändert wird. Struts (Stangen) befinden sich an den äußeren Rändern der View. Durch sie kann man festlegen, dass der Abstand zwischen der View und den Rändern der umgebenden View (dieser Container wird als *Superview* bezeichnet) konstant bleibt. Durch Springs (Federn) können wir festlegen, dass die View in Bezug auf Höhe und Breite wachsen oder schrumpfen kann. Wenn das Gerät gedreht wird, ändert sich die Größe der Superview des Labels und damit auch durch die Einstellungen im Size Inspector die Größe des Labels selbst. Sie müssen die Ausrichtung des Textfelds ebenfalls als zentriert festlegen.

So sieht die Konfiguration von Struts und Springs aus, wenn wir mit unseren Anpassungen fertig sind:



Die Änderungen am UI haben wir jetzt abgeschlossen. Lassen Sie uns zu Xcode zurückkehren und die notwendigen Änderungen am Code vornehmen, so dass sich die Anwendung korrekt verhält, wenn das Gerät gedreht wird.

2.5 Rotieren des Texts – Anpassen des Codes

In Xcode wählen Sie die Gruppe Classes aus und in der dann angezeigten Dateiliste rechts oben die Datei `HalloViewController.m`. Xcode wird die Datei im Editor unter der Liste öffnen. Suchen Sie nach der Methode `shouldAutorotateToInterfaceOrientation:` (entfernen Sie die Kommentarteichen, sollte sie noch auskommentiert sein). Hier ändern Sie den Code, um Ihre Anwendung auf das Drehen des Geräts reagieren zu lassen.

Das iPhone wird Ihre Anwendung durch das Senden von `shouldAutorotateToInterfaceOrientation:` fragen, ob sie sich drehen will (okay, technisch gesehen ist das nicht ganz korrekt, aber hier soll es als Erklärung ausreichen – Sie werden in Kapitel 4, *View-Controller*, auf Seite 55

noch mehr darüber lernen). Sie reagieren auf diese Anfrage, indem Sie entweder YES oder NO zurückgeben. Liefern Sie YES, wird das iPhone Ihre Anwendung mitrotieren, bei NO bleibt Ihre Anwendung stehen.

So sieht der Code für diese Methode aus:

HalloiPhone/Hallo/Classes/HalloViewController.m

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}
```

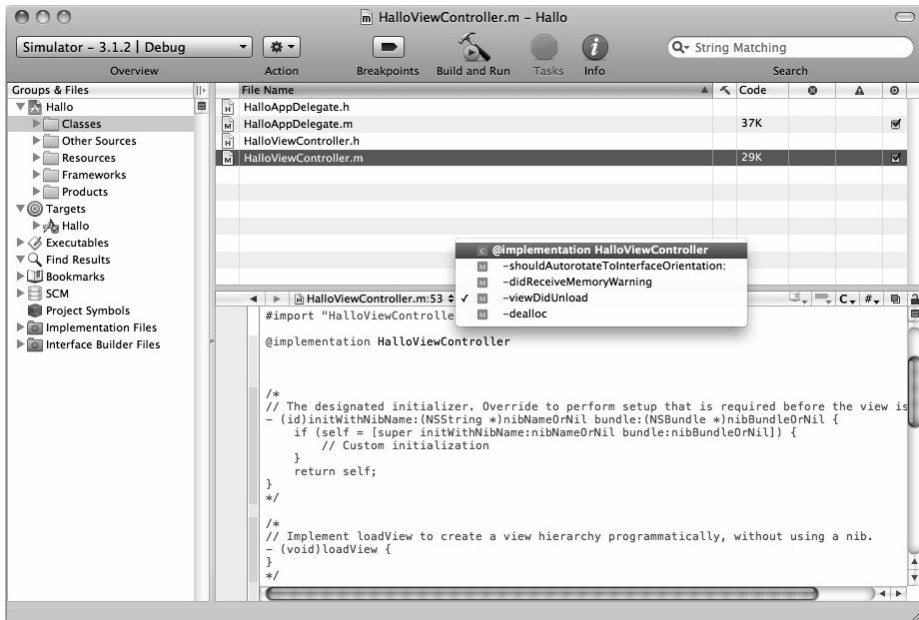
Da Sie bei dieser Anwendung alle Richtungen unterstützen wollen, geben Sie einfach YES zurück. Wir haben jetzt das UI und den Code angepasst, nun können wir die Anwendung erneut starten. Klicken Sie in Xcode also wieder auf den Button BUILD AND RUN.

Nachdem die Anwendung gestartet wurde, können Sie den Simulator mit - und - dazu bringen, das iPhone zu drehen. Das Label bleibt zentriert und ändert seine Ausrichtung zusammen mit dem Gerät.

2.6 Im Code navigieren

In diesem Beispiel gibt es nicht viel Code. Wenn Ihre Projekte größer und Ihre Klassen länger werden, werden Sie in Xcode schneller in Ihrem Quellcode navigieren wollen. Haben Sie eine Datei aus der Liste ausgewählt, wird sie für Sie im Editorbereich geöffnet. Der Bereich besitzt eine ganze Reihe von Buttons, mit deren Hilfe Sie im Code bequem navigieren können. So können Sie auf das Symbol-Auswahlmenü klicken, um direkt zu einem beliebigen Symbol zu springen.

Ein einfacher Klick auf die Symbolliste zeigt alle Symbole so an, wie sie in der Datei gefunden wurden. Wenn es einfacher ist, aus einer alphabetisch sortierten Liste auszuwählen, können Sie die Options-Taste gedrückt halten und dann die Symbolliste anklicken – Xcode wird die Symbole nun in alphabetischer Reihenfolge anzeigen.



2.7 Ihre Anwendung auf Ihrem iPhone ausführen

Im Simulator kann man wunderbar experimentieren, aber es ist immer noch am besten, den Code auf Ihr Gerät zu übertragen und zu sehen, wie er auf Ihre Aktionen reagiert.

Leider werden Sie Ihre Anwendungen erst dann auf Ihr iPhone übermitteln können, wenn Sie Ihre Developer Certificates von Apple erhalten haben. Sind diese bereits installiert, können Sie die Beschreibung im Rest des Kapitels nutzen, um Ihre Anwendung auf Ihr Gerät zu bekommen.

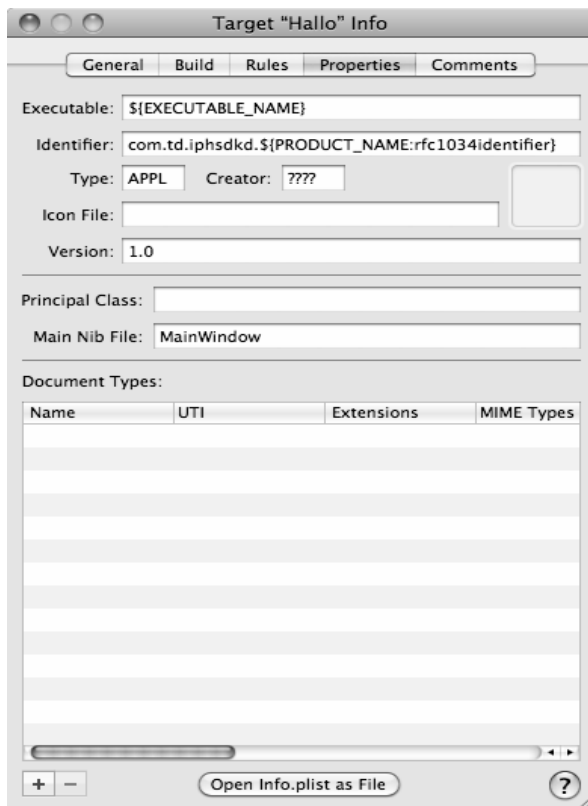
Sind die Zertifikate noch nicht installiert, aber Sie wollen gleich loslegen, surfen Sie zum *iPhone Developer Program* [App08c], wo Sie sich für das Programm anmelden können. Sind Sie dort eingetragen, können Sie zu *Obtaining Your iPhone Development Certificate* [App08f] wechseln, um Informationen darüber zu erhalten, wie Sie Ihre Zertifikate auf Ihre Geräte bekommen und wie Sie sie verwenden, um Ihren Code in Xcode zu signieren. Wenn Sie Ihre Zertifikate noch nicht haben, sollten Sie gleich zum nächsten Kapitel weiterspringen, weil der Rest dieses Kapitels dann nicht sehr interessant sein dürfte.

Um eine Anwendung auf Ihr iPhone zu übertragen, müssen Sie an Ihrem Projekt in Xcode nur sehr wenig ändern. Zunächst müssen Sie den korrekten „Application Identifier“ in Ihren Projekteigenschaften festlegen. Wählen Sie das Target HALLO aus, drücken Sie - (oder

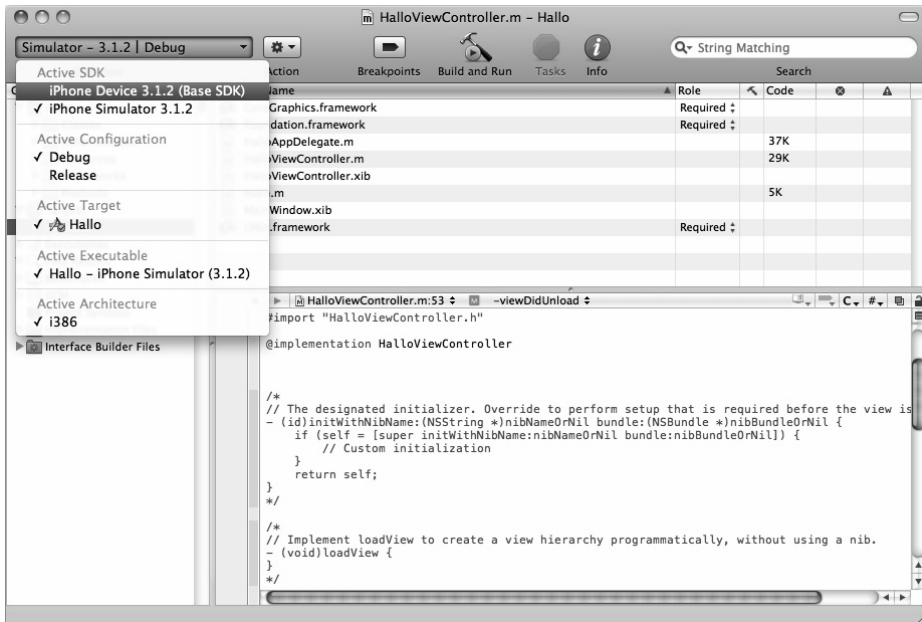
klicken Sie mit der rechten Maustaste und wählen Sie GET INFO) und wählen Sie dann die Registerkarte PROPERTIES.

Der Identifier, den Sie verwenden, muss zu einem der „Provisioning Profiles“ passen, das Sie für Ihre iPhone Developer-Zertifikate eingerichtet haben. Wir haben ein eigenes Provisioning Profile für die Beispiele in diesem Buch erstellt und empfehlen Ihnen, ebenso vorzugehen. Wenn Sie kein spezielles Profil nutzen wollen, können Sie den Identifier auch einfach auf ein bestehendes Profil setzen.

Auf der nächsten Seite zeigen wir die Einstellungen, die wir für dieses Beispiel genutzt haben.



Jetzt müssen Sie noch dafür sorgen, dass die Anwendung nicht mehr auf dem Simulator läuft, sondern auf dem Gerät selbst. Öffnen Sie das Pop-up OVERVIEW. Im Bereich ACTIVE SDK wählen Sie DEVICE – IPHONE OS X (wobei X die auf dem Gerät installierte Version ist).



Wenn Sie diese Einträge nicht sehen, haben Sie vermutlich ein Problem mit Ihren Entwicklerzertifikaten. Melden Sie sich an der Portalseite für iPhone-Entwickler an und lesen Sie sich die Anweisungen in *Obtaining Your iPhone Development Certificate* [App08f] nochmals genau durch, um herauszufinden, ob Sie alles richtig gemacht haben.

Nachdem Sie das Gerät ausgewählt haben, klicken Sie erneut auf den Button BUILD AND RUN. Dieses Mal braucht Xcode ein paar Sekunden, da der gesamte Code neu kompiliert werden muss, damit er auf dem Prozessor des iPhone statt auf Ihrem Mac läuft. Nach dem Kompilieren, Packen und Signieren Ihrer Anwendung wird diese durch Xcode auf Ihrem Gerät installiert und gestartet. Nach dem Start können Sie versuchen, das Gerät zu drehen, um zu sehen, ob sich der Text „Hallo iPhone“ mitdreht.

Glückwunsch! Sie haben Ihre erste eigene iPhone-Anwendung gebaut und ausgeführt. Dazu haben Sie ein bisschen Xcode und IB kennengelernt und wissen nun, wie diese beiden Tools genutzt werden, um den Code und die Benutzerschnittstelle für Ihre Anwendungen zu erstellen.

Kapitel 3

Grundlagen der iPhone-Entwicklung

Nachdem Sie jetzt die Gelegenheit hatten, mit den iPhone SDK-Tools eine einfache Anwendung zu erstellen, wollen wir doch nochmal einen Schritt zurückgehen und verstehen, was wir da getan haben. In diesem Kapitel schauen wir uns den Aufbau des Anwendungs-Stacks im iPhone an, lernen, wie die Tools mit dieser Architektur zusammenarbeiten, und lernen ein paar der Konzepte kennen, die Sie in so gut wie jeder nicht trivialen iPhone-Anwendung beherrschen müssen.

Im vorigen Kapitel haben wir den Code eigentlich kaum angefasst – jetzt werden wir dafür umso tiefer einsteigen. iPhone-Anwendungen sind größtenteils in *Objective-C* geschrieben, einer Reihe objektorientierter Erweiterungen für die klassische Sprache C.

Wir haben in diesem Buch das Ziel, Entwickler mit Erfahrung in einer der diversen auf C basierenden „Curly-Braced“-Sprachen in die Programmierung mit Objective-C einzuführen. Wenn Sie schon mit C++, C# oder Java gearbeitet haben, werden Ihnen viele der Konzepte von Objective-C vertraut sein, auch wenn die Syntax anders aussieht. Haben Sie bisher eher Skriptsprachen wie Ruby, Python oder JavaScript genutzt, gehen wir davon aus, dass Ihnen die Konzepte der Objektorientierung vertraut sind. Sie werden aber nicht unbedingt alle subtilen C-Eigenheiten kennen, insbesondere Zeiger und die Speicherverwaltung. In dem Fall müssen Sie ein wenig mehr Aufwand für die Einarbeitung berücksichtigen.

Wir werden die Grundlagen von Objective-C behandeln, die Sie bei Ihrer tagtäglichen Arbeit benötigen. Wollen Sie tiefer einsteigen, werfen Sie mal einen Blick auf Apples *The Objective-C 2.0 Programming Language* [App09e], Bills Screencast „Coding in Objective-C 2.0“ unter [http:// www.pragprog.com/screencasts](http://www.pragprog.com/screencasts) oder in ein Buch zu Cocoa für den Mac, wie zum Beispiel Daniel Steinbergs *Cocoa Programming: A Quick-Start Guide for Developers* [Ste09].

3.1 Die Softwarearchitektur des iPhone

Der Software-Stack des iPhone ist in mehrere Ebenen unterteilt, wobei Ihre Anwendung auf der obersten Abstraktionsebene und die zentralen Systemdienste auf der untersten Ebene liegen. Von oben nach unten lässt sich der Stack wie folgt zusammenfassen:

- *Ihre Anwendung.*
- *Cocoa Touch* – ein Framework für das Entwickeln von Touchscreen-Anwendungen: UI-Elemente, Event Dispatching, Applikationskontrolle und so weiter. Dazu gehören auch Objektwrapper für wichtige Datentypen (Strings, Collections).
- *Media* – Grafik, Animation, Sound, Video.
- *Core Services* – Collections, Strings, Erkennen der Ausrichtung, SQLite-Datenbank, Adressbuch, Netzwerkunterstützung und so weiter.
- *Core OS Layer* – Unix-Services, Ein-/Ausgabe, Threads, BSD-Sockets, Energieverwaltung und mehr.

Sehr wahrscheinlich werden Sie hauptsächlich mit den GUI-Frameworks und der von Cocoa Touch bereitgestellten OO-Abstraktion arbeiten, daher bilden diese Ebenen auch den Schwerpunkt in diesem Buch (wobei wir bei bestimmten Themen auch tiefer einsteigen). Ein Großteil der Klassen von Cocoa Touch ist dazu gedacht, direkt in Ihrem Code aufgerufen zu werden. Sie können von diesen Klassen Unterklassen bilden, um Funktionalität hinzuzufügen, aber Sie werden merken, dass Sie das deutlich seltener tun müssen als in anderen Sprachen.

3.2 Code in Objective-C

Die Cocoa Touch-Frameworks, die Sie verwenden werden, sind in Objective-C geschrieben und werden auch daraus aufgerufen. Objective-C ist eine Obermenge des ursprünglichen C. Daher können Sie die Syntax von C und Objective-C in Ihrem Anwendungscode beliebig mischen.

Klassen werden in Objective-C erstellt, indem man eine *Header-Datei* und eine *Implementierungsdatei* erzeugt. Diese beiden Dateien haben den gleichen Namen, aber unterschiedliche Erweiterungen. Um eine Klasse `Person` anzulegen, würden Sie eine Datei `Person.h` und eine Datei `Person.m` erstellen. In der Header-Datei stehen die öffentlichen Teile Ihrer Klasse: die Namen und Typen Ihrer Instanzvariablen und die Signaturen Ihrer Methoden, in denen der Rückgabewert und die Parameter beschrieben sind. In der Implementierungsdatei werden diese Methoden dann mit Leben (also Code) gefüllt. Hier deklariert man auch lokale Variablen, die nur für die Implementierung notwendig sind und nicht nach außen dringen sollen. Wie bei C müssen Sie ebenfalls angeben, wo der Compiler den anderen Code findet, den Sie verwenden, indem die Header-Dateien von Klassen **importiert** werden, auf die Sie sich in Ihren Headern oder Ihrer Implementierung beziehen.

In Objective-C sind Methodenaufrufe eigentlich *Nachrichten* (Messages), die von einer kleinen und schnellen Laufzeitumgebung an die Objekte im Speicher verteilt werden. Das ist ein kleiner, aber wichtiger Unterschied, der ein erstaunlich dynamisches Verhalten für eine eigentlich ziemlich formale Sprache ermöglicht. Wenn Sie zum Beispiel eine Methode für eine Objektreferenz aufrufen, die `nil` ist, gibt es keinen Absturz oder Fehler (wie die `NullPointerException` bei Java). Die Laufzeitumgebung bemerkt, dass das Schicken einer Nachricht an ein nicht vorhandenes Objekt sinnlos wäre, und tut einfach nichts.

Methodenaufrufe in Objective-C sind normalerweise in eckige Klammern eingeschlossen und beginnen mit dem Objekt, auf das die Parameternamen und -werte folgen. Wenn Sie zum Beispiel eine `NSString`-Instanz namens `myString` haben, erreichen Sie das dritte Zeichen über einen Methodenaufruf wie den folgenden:

```
myChar = [myString characterAtIndex: 3];
```



Joe fragt...

Muss ich C lernen, um iPhone-Anwendungen zu schreiben?

Nun, wir werden Ihnen natürlich nicht raten, dieses Buch beiseite zu legen und sich durch den „K&R“* zu kämpfen, aber wenn Sie noch nicht mit C vertraut sind, werden Sie doch einiges lernen müssen.

Ein paar iPhone-APIs, die nicht bei Apple entwickelt wurden, nutzen typische C-Praktiken. Dazu gehören OpenGL, OpenAL und SQLite. Es gibt auch Apple-APIs, die in purem C aufgerufen werden (Apple nutzt häufig den Begriff „prozedurales C“, um einen Unterschied zu Objective-C kenntlich zu machen) und die Apple-spezifische Entwurfsmuster nutzen, um sich wie OO anzufühlen. Dazu gehören die Core Foundation, Core Audio und Quartz/Core Graphics, die in Abschnitt 17.1, *Prozedurale C-APIs verwenden*, auf Seite 388 vorgestellt werden.

Aber selbst in den rein auf Objective-C basierenden Cocoa-APIs werden Sie die C-Syntax zum Beispiel für die Ablaufsteuerung nutzen und auch andere C-typische Vorgehensweisen finden – das Verwenden von Zeigern für Ihre Objective-C-Variablen, die Angabe ihrer Adressen (mithilfe des Operators &) als Parameter, um Rückgabewerte von Methodenparametern zu erhalten, Argumentlisten mit variabler Länge und so weiter.

Wir haben C vor langer Zeit erlernt, aber wir verstehen, wie schwierig es ist, von modernen Sprachen zum funktionalen Stil und zur Speicherverwaltung von C zurückzukehren. Wir wollen versuchen, Sie im Folgenden vor besonderen Verhaltensweisen von C zu warnen.

* Das ist die Kurzform für das viel zitierte Buch „Programmieren in C“ von Brian Kernighan und Dennis Ritchie [KR98].

Der größte Unterschied zu anderen Sprachen ist – abgesehen von den eckigen Klammern –, dass die Parameter *immer* über ihren Namen referenziert werden. Diese werden als **Schlüsselwort** bezeichnet und sind vom Wert durch einen Doppelpunkt getrennt. Das bedeutet mehr Aufwand bei der Eingabe, aber der Code lässt sich dadurch viel besser lesen – im Vergleich zu anderen Sprachen im C-Stil, bei denen Sie ein halbes Dutzend Parameter in Ihren Klammern stehen haben können, ohne direkt zu sehen, was jeder davon zu bedeuten hat.

3.3 Grundlegende Klassen von Cocoa Touch

Nachdem Sie nun wissen, dass ein Großteil Ihrer Programmierung in Objective-C vorgenommen werden wird, wollen wir uns einen groben Überblick über die Klassen verschaffen, die vom iPhone SDK bereitgestellt werden.

Das Anwendungs-Framework Cocoa Touch, das schon in Abschnitt 3.1, Die Softwarearchitektur des iPhone, auf Seite 26 erwähnt wurde, enthält die meisten der Klassen, die Sie zum Entwickeln Ihrer ersten Anwendungen nutzen werden. Der Begriff kommt von *Cocoa*, den objektorientierten Frameworks, die für die Programmierung unter Mac OS X entwickelt wurden (und auch schon für NextStep), und von GUI-Klassen, die extra für die Verwendung auf einem mobilen Gerät mit einem Touchscreen gedacht sind (daher das „Touch“).

Das *Foundation*-Framework von Cocoa enthält grundlegende Datenklassen, Tools und einige wichtige Programmierkonventionen, die durch Objective-C allein nicht ausgedrückt werden können, wie zum Beispiel Techniken für die Speicherverwaltung (die wir in Abschnitt 3.7, *Speicherverwaltung der Anwendung* auf Seite 47 noch näher beleuchten werden). Nahezu alle Cocoa-Klassen erben von einer Basisklasse `NSObject`, die in Foundation definiert ist.

Der vielleicht wichtigste Bereich in Foundation sind die Klassen zur Datenverwaltung, die anstelle der prozeduralen Äquivalente in C überall in Cocoa verwendet werden. So wird zum Beispiel der klassische C-String, das Null-terminierte **char**-Array, so gut wie nie in Cocoa genutzt. Stattdessen verwenden Sie `NSString`, das nicht nur die Zeichendaten repräsentiert, sondern auch seine Kodierung. Es gibt Unicode-Unterstützung (und die Kodierungen UTF-8 und UTF-16), wodurch es `NSString` einfach macht, Text in einem beliebigen der unzähligen Zeichensätze auf dem iPhone darzustellen und zu verarbeiten.

Cocoa enthält auch eine umfangreiche Sammlung von *Collection*-Klassen, durch die man sich den Zugriff auf die meisten C-Arrays (oder liebevoll selbst gebaute Collections wie verkettete Listen und Hash-tabellen) sparen kann. Es werden drei Klassen für das Ablegen von Cocoa-Objekten in Collections genutzt: `NSArray` für geordnete Objekt-Collections, `NSSet` für ungeordnete Collections und `NSDictionary` für die Verbindung von Schlüsselobjekten mit Wertobjekten. Diese drei Collections sind *immutable* (unveränderlich) – einmal initialisiert, kön-

nen sie nicht geändert werden. Wollen Sie Inhalte hinzufügen, löschen oder anders bearbeiten, greifen Sie auf die veränderbaren Unterklassen `NSMutableArray`, `NSMutableSet` und `NSMutableDictionary` zurück.

Die Collections können nur Objekte vom Typ `NSObject` speichern. Haben Sie C-Primitive, können Sie sie in Cocoa mithilfe der Wrapperklassen `NSData`, `NSMutableData` (für einen Byte-Puffer) und `NSNumber` (für die skalaren numerischen Typen wie zum Beispiel `int`, `float` oder `bool`) speichern.

Cocoa enthält ein paar weitere, spezielle Datenklassen, so zum Beispiel `NSURL` für URLs (einschließlich der URLs im `file://`-Stil, die Elemente auf dem lokalen Dateisystem repräsentieren, wenn Sie dafür häufig auch `NSString`-Pfade nutzen) und Klassen für Datums- und Zeitwerte, wie `NSDate` und `NSTimeZone`.

Der „Touch“-Teil von Cocoa Touch wird zum großen Teil durch das `UIKit` Framework repräsentiert, das ebenfalls standardmäßig in jede iPhone-Anwendung importiert wird. Dieses Framework bietet ein Darstellungsmodell, Event-Handling, Starten und Beenden der Anwendung und andere Dinge, die für eine per Gesten zu bedienende Applikation notwendig sind. Sie werden damit hauptsächlich über die verschiedenen Benutzerschnittstellenkomponenten arbeiten, die es anbietet: `UIButton`, `UITextView`, `UITableView` und so weiter.

Durch die Datentypen in Foundation und die UI-Komponenten in `UIKit` haben Sie mit Cocoa Touch eine wunderbare Grundlage, auf der Sie Ihre Anwendung aufbauen können.¹

3.4 Arbeiten mit Xcode und dem Interface Builder

Nun denken Sie vielleicht: „Hey, ich habe doch im vorigen Abschnitt fast überhaupt keinen Code geschrieben, um die Anwendung zu erstellen. Warum reden wir jetzt hier plötzlich nur noch davon?“ Das stimmt wohl. Wir haben Ihnen Xcode und den Interface Builder als die wichtigsten Tools schon vorgestellt, daher wollen wir uns nun anschauen, welche Rolle sie spielen. Dazu werfen wir einen Blick auf den Inhalt des Projekts `Hallo.xcodeproj`, das sie in Kapitel 2, *Hallo iPhone*, auf Seite 11 angelegt haben. Wenn Sie dieses Projekt in Xcode geöffnet haben, sollten Sie ein Fenster vor sich sehen, das links die Liste „Groups & Files“

¹ Denken Sie daran, dass Sie über diese Klassen jederzeit mehr erfahren können, wenn Sie auf das Hilfe-Menü von Xcode zugreifen. In Abschnitt 27.3, *Die iPhone SDK-Dokumentation verwenden*, auf Seite 525 erhalten Sie einen Überblick über die Dokumentation in Xcode.

mit dem Inhalt von Hallo als eine Reihe von Ordnern anzeigt. Klicken Sie auf Hallo, sehen Sie im rechten Bereich alle Dateien des Projekts. Sie können aber auch jeden Ordner einzeln anklicken, um die Teile des Projekts nach Typ geordnet zu betrachten:

- *Classes* – Klassendateien sind die C- und Objective-C-Quelldateien, in denen Sie die Funktionalität Ihrer Anwendung bereitstellen. Das Template, mit dem Sie Hallo erstellt haben, hat zwei Klassen erzeugt: `HalloAppDelegate` ist für die Behandlung von Events zuständig, die sich um den Lebenszyklus der Anwendung drehen (wie das Starten und Beenden). `HalloViewController` kümmert sich darum, was Sie auf dem Bildschirm sehen.
- *Other sources* – Hier finden sich Quelldateien, die automatisch generiert wurden und für die Sie nicht verantwortlich oder die nicht interessant sind. `HalloPrex.pch` ist die Datei mit den vorkompilierten Headern (*Precompiled Headers*), die von Xcode erstellt wurde, um das Verarbeiten der Header-Dateien beim Bauen zu beschleunigen. `main.m` ist die Implementierung der Funktion `main()`, die das System aufruft, um Ihre Anwendung zu starten. Sie können hier auch Quellcode von Fremdbibliotheken oder prozedurale C-Quelldateien ablegen (.c-Dateien), die keine „Klassen“ sind.
- *Ressourcen* – Ressourcen sind Dateien, die keinen Code enthalten, aber trotzdem von Ihrer Anwendung zur Laufzeit benötigt werden. Dazu können Grafik- oder Sounddateien gehören, Wörterbücher für die Lokalisierung und so weiter. In diesem Projekt enthält die Datei `Info.plist`² die grundlegenden Einstellungen für die Anwendung, wie zum Beispiel das Symbol und den Namen. Die Ressourcen enthalten auch die .nib-Dateien mit den „gefriergetrockneten“ GUI-Komponenten, aus denen Ihre Benutzerschnittstelle aufgebaut wird. Als Sie im vorigen Kapitel den Interface Builder verwendeten, wurde damit die Datei `HalloViewController.xib` bearbeitet.

² Seit dem iPhone SDK 3.0 stellt Xcode vor den Dateinamen den Projektnamen, wie zum Beispiel `HalloInfo.plist`. Wir werden weiterhin den Begriff `Info.plist` als generischen Verweis auf die Datei verwenden, die für Ihre Anwendungseinstellungen erzeugt wurde.

- *Frameworks* – Hier finden sich die Frameworks, die Ihre Anwendung nutzt. Standardmäßig verknüpft das Xcode-Template hier die Frameworks Core Graphics und UIKit, so dass Sie auf die verschiedenen GUI-Klassen und auf Foundation für die häufig verwendeten Klassen wie Strings, Collections, URLs und so weiter zugreifen können. Wenn wir über die grundlegenden Funktionalitäten hinaus sind, werden wir in späteren Kapiteln noch mehr Frameworks hinzufügen.
- *Products* – In diesem Ordner finden Sie die Dateien, die beim Build-Prozess erstellt werden. In unserem Fall ist das die iPhone-Anwendung `Hallo.app`.

Xcode ist wirklich die Grundlage für all Ihre Anwendungsprojekte. Auch wenn Sie andere Tools nutzen werden, um die GUI-Elemente in der `.nib`-Datei festzulegen (den Interface Builder), Bilder zu erstellen (beliebige Grafikanwendungen wie Photoshop oder Pixelmator) und die Performance zu messen (Instruments), werden all diese Bestandteile der Anwendung in Xcode zusammengeführt und gebaut. Üblicherweise bearbeiten Sie in Xcode ebenfalls den Quelltext, auch wenn Sie eine externe Applikation dafür definieren können (vielleicht BBEdit, TextMate oder sogar Emacs), sollte Ihnen das lieber sein.

Eine interaktive Anwendung entwerfen

Unsere erste Anwendung hat „Hallo iPhone“ ausgegeben, daher wollen wir ihr den Gefallen tun und dem Anwender Hallo sagen. Dafür muss er nach seinem Namen gefragt werden.

Schließen Sie das Projekt Hallo, indem Sie das Projektfenster schließen (sofern es noch offen ist) und mit -- ein neues Projekt öffnen. Im Fenster „New Project“ wählen Sie wieder den Eintrag „View-based Application“. Wenn Sie nach dem Projektnamen gefragt werden, geben Sie zum Beispiel `HalloUser` ein.

Wie im Beispiel des vorigen Kapitels erzeugt das Projekt-Template zwei Klassen für uns (`HalloUserAppDelegate` und `HalloUserController`) sowie ein paar weitere Klassen, die wir im nächsten Abschnitt erläutern werden. Hier müssen wir nur wissen, dass wir in einer Single-View-Anwendung etwas auf die Oberfläche bekommen, indem wir mit der Klasse `HalloUserController` und dem dazugehörigen `HalloUserController.xib` arbeiten, das das gefriergetrocknete GUI für diese

Klasse enthält. Im vorigen Kapitel haben wir nur die `.nib`-Datei bearbeitet, aber dieses Mal werden wir auch die Klasse anpassen, um eine gewisse Interaktivität zu erreichen.

IBOutlets und IBActions deklarieren

In unserer neuen Anwendung kann der Benutzer seinen Namen in ein Textfeld eingeben. Ist er fertig, werden wir die Ausgabe durch den Namen verändern. Geben Sie also Quinn ein, wird sich das Label in „Hallo Quinn“ verändern. Dazu nutzen wir ein Label für die Ausgabe, ein Textfeld für die Benutzereingabe, einen Button, mit dem die Eingabe abgeschlossen wird, und eine Methode, die das Label anpasst, wenn der Button angetippt wird.

Damit das funktioniert, müssen wir Beziehungen zwischen der Anwendungslogik und den GUI-Objekten im Nib herstellen. Dazu nutzen wir spezielle Schlüsselwörter, wenn wir Variablen deklarieren, die sich auf Objekte im Nib beziehen, oder bei Methoden, die durch Events in der Benutzerschnittstelle gerufen werden, wie zum Beispiel durch das Antippen von Buttons. Eine Referenz im Code auf ein Objekt im Nib wird als *Outlet* bezeichnet und durch das Schlüsselwort **IBOutlet** gekennzeichnet. Genauso wird eine Methode, die durch Objekte im Nib aufrufbar sein soll, als *Action* bezeichnet und durch das Schlüsselwort **IBAction** markiert.

Im Allgemeinen deklarieren Sie Outlets und Actions im Code so, dass der IB auf sie zugreifen kann. Bearbeiten Sie Ihre Header-Datei `Hallo-UIViewController.h`, so dass sie wie folgt aussieht:

HalloiPhone/HalloUser/Classes/HalloUserController.h

```

1  #import <UIKit/UIKit.h>
2
3  @interface HalloUserController : UIViewController {
4
5      IBOutlet UILabel *helloLabel;
6      IBOutlet UITextField *nameField;
7
8  }
9
10 -(IBAction) sayHello: (id) sender;
11
12 @end

```

Mit diesem Code werden zwei Outlets und eine Action erzeugt. Da es sich bei den Outlets um Variablen handelt und sie daher Teil der Klassenstruktur sind, gehören sie in den Block der **@interface**-Deklaration, der

in geschweiften Klammern steht. Das erste Outlet in Zeile 5 deklariert einen Zeiger auf ein UILabel namens `helloLabel`, wobei das zusätzliche **IBOutlet** unseren Wunsch signalisiert, diese Referenz mit einem Objekt zu verbinden, das im Interface Builder erstellt wurde. Durch diese Verbindung werden wir dazu in der Lage sein, das Label im Interface Builder zu bearbeiten, es aber auch zur Laufzeit durch Code anzupassen, um den „benannten“ Text auszugeben. Entsprechend deklarieren wir in Zeile 6 ein UITextField als IBOutlet, um dessen Wert im Code auslesen zu können. Schließlich deklarieren wir in Zeile 10 eine Methode, die sich um das Antippen des Buttons kümmert, wobei wir sie als Action erstellen (beachten Sie, dass wir kein Outlet für den Button benötigen, wenn wir ihn nicht direkt verändern wollen, beispielsweise seine Größe oder seine Beschriftung durch Code anpassen wollen).

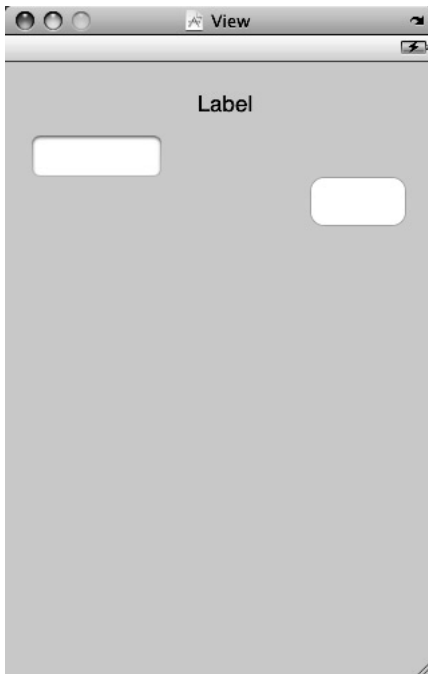
Wie Sie sehen können, ist die Syntax der Methodendeklaration in Objective-C recht ungewöhnlich. Sie beginnt mit einem einzelnen Zeichen (-), um eine Instanzmethode zu deklarieren – sie wird also für eine bestimmte Instanz der Klasse aufgerufen. Mit + deklarieren Sie eine Klassenmethode, die unabhängig von einer Instanz ist. Danach folgt in Klammern der Rückgabotyp. (IBAction) entspricht (void) insoweit, dass es keinen Wert zurückgibt, aber durch das Schlüsselwort **IBAction** wird angezeigt, dass wir diese Methode mit GUI-Events im Interface Builder verbinden wollen. Schließlich folgen die Argumente als Paare aus Parametertyp und -name. In diesem Fall ist der Typ (id) ein Zeiger auf ein beliebiges Objekt. Das Muster (id) sender wird von allen IBAction-Deklarationen genutzt, damit die aufgerufene Methode weiß, welches Objekt (der sender) sie gerufen hat.

Die Oberfläche im IB erstellen

Jetzt können Sie das GUI im Interface Builder erstellen und die Komponenten mit Ihrem Code verknüpfen. Speichern Sie aber zunächst die Header-Datei, wenn Sie das noch nicht getan haben.

Als nächstes klicken Sie im Projektfenster von Xcode doppelt auf `HalloUserController.xib`, um diese Datei im IB zu öffnen. Wie im vorigen Kapitel wird der IB ein Dokumentenfenster mit drei Objekten zeigen: zwei Proxyobjekte (File's Owner und First Responder, auf die wir später noch eingehen) und View. Das Vorschaufenster der View ist eventuell schon offen, ansonsten klicken Sie doppelt auf View, um es zu öffnen. Ihre View besteht zunächst nur aus einem leeren Bereich, der genau der Größe des iPhone-Bildschirms entspricht. Ziehen Sie aus der Library drei Objekte darauf: ein Label, ein Textfeld und einen Button.

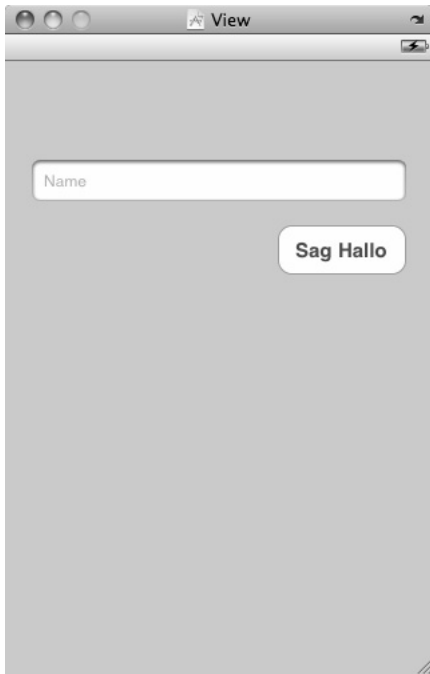
Ihre Vorschau sollte nun in etwa so aussehen:



Ganz offensichtlich könnte man an dieser Benutzeroberfläche noch ein bisschen arbeiten. Lassen Sie uns die drei Komponenten wie folgt anpassen:

- Markieren Sie das Label, um dessen Handles sichtbar zu machen. Ziehen Sie nun den rechten und linken Rand so weit nach außen, bis die gestrichelten blauen Ränder erscheinen. Damit wird das Label die gesamte Breite des Bildschirms einnehmen, falls der Anwender einen wirklich langen Namen besitzt. Um den Text zu zentrieren, öffnen Sie den Attributes Inspector ( - ) und klicken im Bereich ALIGNMENT auf das Symbol zum Zentrieren des Texts (also das mittlere).
- Wählen Sie das Textfeld aus und ziehen Sie es ebenfalls bis zu den Randlinien. Im Attributes Inspector können Sie den Placeholder „Name“ angeben, der in hellgrauer Schrift dem Anwender mitteilt, was er mit dem Feld machen soll (hier also zum Beispiel, den Namen einzugeben).
- Klicken Sie schließlich doppelt auf den Button oder ändern Sie sein Attribut „Title“, um den Text auf „Sag Hallo“ zu ändern .

So sieht nun die vollständige Benutzeroberfläche aus. Aus Gründen der Einfachheit wollen wir uns in diesem Beispiel nicht darum kümmern, die Komponenten auf ein Rotieren oder eine Größenanpassung reagieren zu lassen.



Nachdem wir nun unsere Benutzerschnittstelle eingerichtet haben, können wir sie mit unserem Code verbinden. Dazu wechseln wir zurück zum Dokumentenfenster von `HalloUserController.xib` und seinen drei Symbolen. Jetzt werden wir mit „File’s Owner“ arbeiten. Dabei handelt es sich nicht um ein echtes Objekt, sondern um einen Stellvertreter für ein Objekt, das mit dem „Besitzer“ des Nib verbunden ist, wenn das Nib geladen wird.

Xcode hat schon die Klasse des File’s Owner auf `HalloUserController` gesetzt (was Sie über den Identity Inspector per - herausfinden können). Das ist die Grundlage dafür, unseren Code mit der Oberfläche verbinden zu können. Das Objekt, das diese .nib-Datei und die angepasste View „besitzt“, ist der `HalloUserController`, für den wir durch das Bearbeiten der Header-Datei Actions und Outlets erzeugt haben. Somit weiß der IB schon, dass der File’s Owner diese Outlets und Actions zur Laufzeit besitzt.

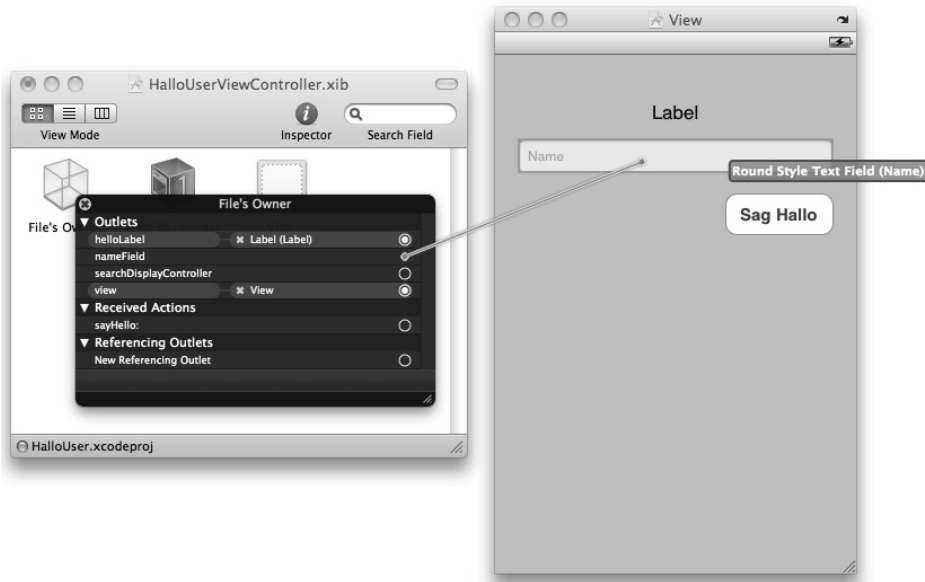
Da wir diese **IBOutlet**s und **IBAction**s bereits gepflegt haben, sehen Sie schon `helloLabel` und `nameField`, wenn Sie mit der rechten Maustaste (oder mit `Ctrl`) auf FILE'S OWNER klicken.³



Jetzt werden wir diese Verbindungen auch endlich herstellen. Klicken Sie im grauen Heads-Up-Display (HUD) mit der Liste der Outlets auf den Kreis neben `helloLabel` und ziehen Sie das Label in das Vorschau-fenster. Wenn Sie mit der Maus über das Label ziehen, wird der IB einen Rahmen darum anzeigen und ein Fenster öffnen, in dem Ihre vorgeschlagene Verbindung als Label (Label) angezeigt wird. Damit sehen Sie, dass Sie gerade dabei sind, sich mit einem Label (genauer, einem UILabel) zu verbinden, dessen Inhalt den Text „Label“ enthält. Lassen Sie die Maustaste los, um die Verbindung herzustellen. Der Name der verbundenen Komponente wird dann im HUD-Fenster erscheinen.

³ Sie können die Outlets und die Actions eines Objekts auch im Connections Inspector (`⌘-2`) sehen.

Das Gleiche machen Sie nun mit dem Outlet `nameField`. Die Drag-Action sehen Sie im folgenden Bild:



Sie können auf ähnliche Art und Weise für die Action eine Verbindung in Gegenrichtung herstellen – vom Widget zum File's Owner. Klicken Sie ihn mit der rechten Maustaste an, um eine Liste der Events zu erhalten,⁴ und ziehen Sie dann eine Verbindung vom Event „Touch Up Inside“ zum File's Owner. Wenn Sie die Maus loslassen, erscheint ein kleines HUD-Fenster mit den deklarierten Actions, mit denen Sie sich verbinden können. Da Sie nur eine **IBAction** deklariert haben, können Sie hier nur `sayHello:` wählen.

Jetzt sind die Verbindungen im Interface Builder hergestellt. Damit das UI etwas aufgeräumter ist, sollten Sie vielleicht dem Label einen leeren String zuweisen, damit sich die Anwendung beim Starten nicht mit „Label“ meldet. Dann sind Sie im IB fertig. Speichern Sie Ihre Änderungen und schließen Sie den IB.

⁴ Auch hier können Sie die Liste der Verbindungen über den Connections Inspector einsehen. Als Abkürzung können Sie bei gedrückter `[Ctrl]`-Taste von einem Button auf ein Empfängerobjekt ziehen, um das Event „Touch Up Inside“ zu verbinden, ohne überhaupt die Liste der Events anzuzeigen.

Die Action implementieren

Sie haben nun ein GUI mit Verbindungen von den per IB erzeugten Widgets zu Instanzvariablen in Ihrer Klasse. Wenn Sie die Anwendung jetzt starten, führt ein Tippen auf den Button „Sag Hallo“ zu einem Aufruf von `sayHello:`. Das Problem ist nur, dass Sie diese Methode noch nicht geschrieben haben, daher wird die Anwendung mit dem Fehler „unrecognized selector“ abbrechen. Darum wollen wir uns jetzt kümmern.

Die Implementierungen Ihrer Methode gehören in die Implementierungsdatei der Klasse. Diese besitzt die Dateierendung `.m`. Öffnen Sie im Projektfenster von Xcode die Datei `HalloUserController.m`. Die Methode kann an einer beliebigen Stelle zwischen den Zeilen mit den Schlüsselwörtern `@implementation` und `@end` stehen. Geben Sie die folgende Methodenimplementierung in der Zeile ein, die auf `@implementation HalloUserController` folgt:

`HalloiPhone/HalloUser/Classes/HalloUserController.m`

```

1  -(void) sayHello: (id) sender {
2      NSString *userName = nameField.text;
3      NSString *helloMessage =
4          [[NSString alloc] initWithFormat: @"Hallo %" , userName];
5      helloLabel.text = helloMessage;
6      [helloMessage release];
7      nameField.text = NULL;
8  }
```

- In der zweiten Zeile referenzieren wir die Instanzvariable `nameField` und lesen den eingetippten Text aus. Da `UILabel` den Text als eine *Eigenschaft (Property)* besitzt, lesen und schreiben wir ihn über den Punktoperator. Wir werden weiter unten in diesem Kapitel noch mehr über Eigenschaften erzählen.
- Als Nächstes erstellen wir eine Nachricht, indem wir in Zeile 3 einen neuen `NSString` anlegen. Wie üblich bei Cocoa-Objekten, die durch Code erzeugt werden, allozieren wir zunächst mit `alloc` den Speicher für das Objekt, um danach den Inhalt zu initialisieren. Um die String-Ersetzung anzulegen, können wir einen Format-String verwenden und die Benutzereingabe einbauen. Mit dem führenden `@` können wir schnell einen statischen `NSString` mit dem Inhalt `Hallo %@` allozieren. Das `%@` ist ein *Format-Spezifizierer*, durch den wir eine String-Repräsentation eines beliebigen Cocoa-Objekts einfügen können. Hier greifen wir auf den Wert von `userName` zurück.
- Den so formatierten String setzen wir in Zeile 5 für das Label. Dazu holen wir uns die Instanzvariable `helloLabel` und setzen deren Eigenschaft `text`.

- Schließlich müssen wir noch ein bisschen aufräumen. Wir haben in Zeile 3 für einen String Speicher alloziert. Da wir diesen String nicht länger benötigen, müssen wir ihn wieder freigeben, um auch den Speicher dafür nicht weiter zu reservieren. In Zeile 6 schicken wir dazu die Nachricht `release` an den String. Technisch gesehen, wird damit der Speicher nicht freigegeben, wir sagen nur, dass wir nicht länger am Objekt interessiert sind. Wenn also niemand anderes mehr Interesse an dem String hat, wird der Speicher freigegeben (das geschieht hier nicht, da das `UILabel` auch noch Bedarf an diesem Objekt hat, weil wir es als Text für das Label genutzt haben). Wir werden das Speicher-verwaltungssystem in diesem Kapitel noch genauer behandeln.

Unsere Anwendung kann jetzt gestartet werden. Klicken Sie auf **BUILD AND RUN**, um sie im Simulator auszuführen. In der Anwendung können Sie nun in das Textfeld klicken, Ihren Namen eingeben und dann auf „Sag Hallo“ tippen, um Ihren Namen aufzurufen:



3.5 Anatomie Ihrer iPhone-Anwendung

Bisher liefen die Beschreibungen ein bisschen in der Art „Tippen Sie dies hier, tun Sie das dort, kümmern Sie sich nicht um die kleinen Heinzelmänner“ ab. Also wollen wir nun einmal einen Schritt zurücktreten und uns anschauen, wie Ihre Anwendung gestartet wird und wie all die Teile miteinander interagieren. Wenn das detaillierter wird, als Sie es sich wünschen, brauchen Sie sich keine Gedanken darum zu machen – wir besprechen es hier, um den Ablauf einer iPhone-Anwendung zu entzaubern und zu zeigen, warum das Projekt-Template so strukturiert ist, wie es ist.

Wie schon in Abschnitt 3.4, *Arbeiten mit Xcode und dem Interface Builder*, auf Seite 30 beschrieben, enthält die Datei `main.m` die Implementierung der Funktion `main()`, die das System aufruft, um Ihre Anwendung zu starten. Wenn Sie möchten, machen Sie diese Datei einmal auf – Sie werden nur vier Zeilen vorfinden:

HalloiPhone/HalloUser/main.m

```

1  int main(int argc, char *argv[]) {
2
3      NSAutoreleasePool pool = [[NSAutoreleasePool alloc] init];
4      int retVal = UIApplicationMain(argc, argv, nil, nil);
5      [pool release];
6      return retVal;
7  }
```

Die Signatur in Zeile 1 ist eine typische `main()`-Funktion aus C, die dazu gedacht ist, von der Befehlszeile mit einer Reihe von Argumenten aufgerufen zu werden. In der dritten Zeile wird ein *Autorelease Pool* zur Speicherverwaltung eingerichtet (den wir weiter unten in Abschnitt 3.7, *Speicherverwaltung der Anwendung*, auf Seite 47 noch erläutern werden), und in Zeile 5 wird er wieder freigegeben, wenn die Anwendung endet.

In Zeile 4 rufen wir die Funktion `UIApplicationMain` auf, um die Haupt-Eventschleife zu starten und die Anwendung hochzubringen. Die ersten beiden Argumente enthalten die Optionen von der Befehlszeile (wenn es welche gibt), während das dritte und vierte Argument die Main-Klasse der Anwendung und ihren *Anwendungs-Delegate* definieren – eine Klasse in Ihrem Projekt, die sich um die Events zum Starten und Beenden der Anwendung kümmert. Sind diese Werte `nil`, wie es hier der Fall ist, geht UIKit davon aus, dass es die Anwendung aus einer `.nib`-Datei laden muss.

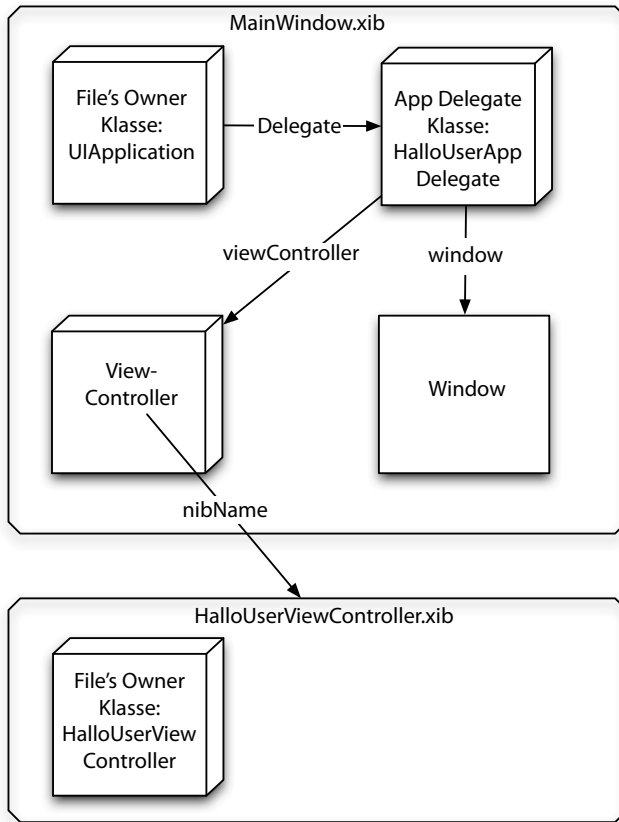
Die Datei `Info.plist` enthält das Haupt-Nib-Bundle für die Anwendung, normalerweise `MainWindow.xib`. Daher geht UIKit davon aus, dass es den App-Delegate in diesem Nib findet. Klicken Sie doppelt auf `MainWindow.xib`, um es im Interface Builder zu öffnen.

Sie sollten ein Fenster wie das folgende sehen:



Das Nib enthält neben den schon erwähnten Proxyobjekten File's Owner und First Responder ein App-Delegate-Symbol (HelloUser App Delegate), einen View-Controller (HelloUser View-Controller) und ein Fenster (Window).

Öffnen Sie den Connections Inspector (-) für das Anwendungs-Delegate-Objekt, damit Sie sehen, wie alles zusammengehört. Jede iPhone-Anwendung muss ein einzelnes UIWindow-Objekt besitzen. Der Anwendungs-Delegate hat eine Verbindung zum Fensterobjekt des Nib. Zudem besitzt er eine Verbindung zu einem View-Controller-Objekt, bei dem es sich um eine Instanz der Klasse `HelloUserViewController` handelt, die wir weiter oben im Kapitel angepasst haben. Wie Sie schon gesehen haben, stellt der View-Controller die Logik zu einer UI-View bereit, nämlich der, die wir in der Datei `HelloUserViewController.xib` bearbeitet haben. Wenn Sie sich den View-Controller mit dem Attributes Inspector anschauen, werden Sie tatsächlich sehen, wo die Referenz auf diese .nib-Datei zu finden ist. Klicken Sie doppelt auf den View-Controller, lesen Sie im Vorschauenfenster der View „Loaded from HelloUserViewController“.



Fassen wir zusammen: Das System ruft die Funktion `main()` auf, die wiederum `UIApplicationMain()` startet. Dort wird mit `Info.plist` in die Haupt-.nib-Datei geschaut, in der ein Anwendungs-Delegate (verbunden mit dem einen Fenster der Anwendung) und ein View-Controller, der aus einer anderen .nib-Datei geladen wird, beschrieben sind. Die erste Möglichkeit, eigenen Code einzubringen, ergibt sich mit den Methoden beim Starten des App-Delegate (wie zum Beispiel `applicationDidFinishLaunching:`) und dem View-Controller, der geladen wird (der Callbacks wie `initWithCoder:` enthält, wenn er aus der .nib-Datei geladen wird, und `viewDidLoad`, wenn die View geladen ist). Aber in unserer Anwendung müssen wir nichts tun, bis der Button angetippt wurde. Sie werden vermutlich nie in `main.m` oder `MainWindow.xib` werkeln müssen, da die Callbacks beim Starten der Anwendung bessere Stellen bieten, an denen man Code einfügen kann. Aber Sie wissen jetzt, wie all die Dinge in Ihrem Xcode-Projekt zusammenarbeiten, um Ihre Anwendung auf den Bildschirm zu bekommen.

3.6 Verhaltensanpassung durch Delegieren

Der Anwendungs-Delegate ist ein Beispiel für eines der wichtigsten Entwurfsmuster in Cocoa: *Delegieren*.

Die Idee des Delegierens ist, dass ein Objekt ein einzelnes *Delegate*-Objekt enthält, das es bei bestimmten Events aufruft. Aus Sicht des Delegate ist das wie ein Callback oder eine Benachrichtigung:⁵ „Sag Bescheid, wenn etwas passiert.“ Aus Sicht des delegierenden Objekts geht es eher darum, Verantwortung abzugeben: „Ich weiß nicht, was getan werden muss (falls überhaupt), wenn dieses Event auftritt – kümmer dich also besser darum.“

Beim Anwendungs-Delegate erhält das Objekt, das UIApplicationMain als Anwendungs-Delegate deklariert, Callbacks für verschiedene Events, die die gesamte Anwendung betreffen: wenn sie vom Home-Bildschirm aufgerufen wird, wenn sie über eine URL von einer anderen Anwendung gestartet wird (siehe Kapitel 26, *Anwendungsintegration*, Seite 511), wenn sie vor zu wenig verfügbarem Speicher gewarnt wird und so weiter. Das ist ein Beispiel für einen formalen Delegate, der durch ein Objective-C-Protokoll UIApplicationDelegate definiert wurde. Dieses Protokoll besitzt wie eine normale Cocoa-Klasse eine eigene Dokumentationsseite, aber statt zu beschreiben, wie die vorhandenen Klassenmethoden arbeiten, steht hier, wann die Delegate-Methoden aufgerufen werden und was Sie bei einer Implementierung dieser Methoden tun sollten oder sogar müssen. Eine Klasse, die tatsächlich ein Anwendungs-Delegate werden soll, muss in ihrer Header-Datei deklarieren, dass sie das Protokoll implementiert. Zudem müssen auch alle Delegate-Methoden, die nicht als „optional“ gekennzeichnet sind, implementiert werden.⁶

Es gibt ein paar weitere Delegate-Protokolle, die wir in HalloUser nutzen könnten, damit sich die Anwendung mehr wie eine typische iPhone-Anwendung verhält. So haben Sie vielleicht beim Ausführen der Anwendung bemerkt, dass die virtuelle Tastatur nach dem Antippen des Buttons SAG HALLO nicht verschwunden ist und die Return-Taste keine Auswirkungen zeigt. Das sind für iPhone-Anwendungen immer wieder vorkommende Aufgaben. Bei Letzterer müssen wir für das Textfeld ein Delegate bereitstellen.

⁵ Cocoa enthält auch ein *Benachrichtigungssystem*, das mit null oder mehr Zuhörern funktioniert. Design und Anwendung unterscheiden sich aber grundlegend vom Delegieren, und beide Systeme lassen sich nicht einfach austauschen.

⁶ Für Java-Programmierer sind Protokolle in Objective-C wie Java-**Interfaces**, allerdings mit der zusätzlichen Möglichkeit, optionale Methoden zu besitzen.

Um die Tastatur wieder loszuwerden, müssen Sie dem Textfeld mitteilen, dass es seine Rolle als „First Responder“ aufzugeben hat – also die Komponente, die die Benutzereingabe als Erstes erhält. Das lässt sich nach dem Antippen des Buttons ganz leicht erreichen, indem man am Ende von `sayHello:` noch folgende Zeile hinzufügt:

```
[nameField resignFirstResponder];
```

Aber was sollen wir tun, wenn die Return-Taste gedrückt wurde? Das ist kein Button aus unseren GUI-Komponenten – die Tastatur wird automatisch durch das Textfeld bereitgestellt –, daher können wir auch nicht einfach eine Event-Handler-Methode damit verknüpfen. Aber wenn Sie sich die Dokumentation⁷ für `UITextField` anschauen, werden Sie feststellen, dass die Klasse eine Eigenschaft `delegate` besitzt, die durch das Protokoll `UITextFieldDelegate` definiert ist. Dabei handelt es sich um eine definierte Gruppe zusammengehörender Methoden. Schauen Sie sich dieses Protokoll an, sehen Sie, dass es eine ganze Reihe von Methoden gibt, die den Delegate auf Events aufmerksam machen, die mit dem Textfeld zu tun haben. Eine davon ist `textFieldShouldReturn` – es scheint, als könnten wir genau diese Methode brauchen.

Sie legen fest, dass ein Objekt ein paar oder alle Delegate-Methoden implementiert, indem Sie den Delegate-Namen im **@interface**-Header in den spitzen Klammern nach dem Namen der Klasse angeben, die diese Klasse erweitert.⁸ Für unsere Zwecke ist `HalloUserController` eine gute Wahl, da diese Klasse schon über das Textfeld Bescheid weiß. Deklarieren Sie daher das Implementieren des Delegate-Protokolls, indem Sie die Header-Datei `HalloUserController.h` bearbeiten und das Protokoll in spitzen Klammern nach der Klassendeklaration hinzufügen, zum Beispiel so:

```
HalloiPhone/HalloUser/Classes/HalloUserController.h
```

```
@interface HalloUserController : UIViewController <UITextFieldDelegate> {
```

Jetzt müssen Sie den View-Controller als Delegate für das Textfeld festlegen. Das können Sie im Code machen, aber da wir den Rest unseres GUI im Interface Builder definiert haben, ist es sinnvoll, den Delegate ebenfalls mit dem IB anzugeben. Klicken Sie doppelt auf `HalloUserController.xib`, um die Datei im IB zu öffnen. Mit einem

⁷ In Kapitel 27, *Debuggen*, auf Seite 519 gibt es einen Abschnitt über das Finden von Klassendokumentationen in Xcode.

⁸ In manchen einfachen Fällen gibt es kein formales Delegate-Protokoll, und der Delegate implementiert einfach ein paar oder alle Methoden, die in der Dokumentation aufgeführt sind.

Rechtsklick auf das Textfeld oder durch die Nutzung des Connection Inspector für das Feld sehen Sie, dass es eine leere Verbindung namens `delegate` gibt. Ziehen Sie den kreisförmigen Verbindungspunkt auf File's Owner (das den `HalloUserController` repräsentiert, weil dies das dem Nib zugewiesene Objekt ist), wodurch Sie den `HalloUserController` als Delegate für das Textfeld deklarieren. Denken Sie daran, Ihre Änderungen zu speichern, bevor Sie den IB verlassen.

Indem Sie den View-Controller als Textfeld-Delegate gesetzt haben, erhält er durch Events ausgelöste Callbacks vom Textfeld. Jetzt müssen Sie nur noch für die für Sie interessanten Callbacks Implementierungen bereitstellen.

Im Folgenden finden Sie eine einfache Methode, um die Tastatur verschwinden zu lassen. Sie kann irgendwo in `HalloUserController.m` zwischen `@implementation` und `@end` stehen:

```
HalloiPhone/HalloUser/Classes/HalloUserController.m
```

```
-(BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];
    return YES;
}
```

Die Signatur dieser Methode muss genau zur Signatur passen, die durch das Delegate-Protokoll definiert wurde, daher ist es nicht die schlechteste Idee, sie aus der Dokumentation oder der Header-Datei herauszukopieren.

Wie Sie sehen können, wird der Methode ein `textField` übergeben (bei dem es sich vermutlich um `nameField` handelt, da unsere Anwendung nur ein Textfeld besitzt), an das wir einfach die Nachricht `resignFirstResponder` senden. Damit wird das Textfeld angewiesen, nicht länger Texteingaben zu empfangen (zumindest, bis es nicht wieder ange-tippt wird), womit es dafür sorgt, dass die virtuelle Tastatur verschwindet. Dann geben wir `YES` zurück, weil das Protokoll uns dazu auffordert, wenn wir möchten, dass der Button sein Standardverhalten für ein Antippen implementieren soll.

Klicken Sie in Xcode auf BUILD AND RUN, geben Sie Text ein und tippen Sie dann auf den Return-Button. Die Tastatur sollte verschwinden (wenn nicht, prüfen Sie, ob Sie den Delegate im IB mit File's Owner verbunden haben und Ihre Methodensignatur genau der aus dem Codebeispiel entspricht).

Das Hallo-Label wird nicht angepasst, da sich der Code dazu in `sayHello:` befindet, das nur durch ein Antippen des Buttons `SAG HALLO` aufgerufen wird. In dem nicht ungewöhnlichen Fall, dass der gleiche Code sowohl durch ein Tippen auf einen Button als auch durch die Return-Taste ausgeführt werden soll, stecken Sie die gemeinsam genutzte Funktionalität normalerweise in eine Hilfsmethode, die dann von beiden Event-Handlern aufgerufen wird.

Vielleicht ist Ihnen positiv aufgefallen, dass Sie bei Protokollen nur so viel Funktionalität implementieren müssen, wie Sie benötigen. Alle Methoden in `UITextFieldDelegate` sind optional, daher können Sie alle Events ignorieren, an denen Sie kein Interesse haben. Das ist nicht immer so, da manche Delegate-Protokolle Methoden deklarieren, die von Ihnen implementiert werden müssen, damit der Compiler nicht meckert. Aber trotzdem ist es ein praktisches Muster und eines, das Ihnen überall in Cocoa begegnen wird.

3.7 Speicherverwaltung der Anwendung

Eine weiterer beachtenswerter Aspekt von iPhone-Anwendungen – einer, der vielleicht nicht so erquicklich ist wie das Delegieren – ist der Umgang mit dem Arbeitsspeicher. Wenn Sie mit modernen Sprachen gearbeitet haben, die eine Garbage Collection besitzen (zum Beispiel Java, C#, Ruby und sogar Objective-C auf dem Desktop), haben Sie sich vermutlich wenig Gedanken um die Speicherverwaltung gemacht, da die Objekte, die Sie nicht mehr benötigten, automatisch abgeräumt wurden, ohne dass Sie etwas dafür tun mussten. In iPhone-Anwendungen ist nun das manuelle Verwalten des Speichers einer der schwierigsten Aspekte, dem sich Entwickler gegenübersehen. Passen Sie da nicht genau auf, kann Ihre Anwendung schnell Speicherlecks schaffen, wodurch sie letztendlich riskiert, vom System beendet zu werden.

Konzeptionell ist die Speicherverwaltung in iPhone-Anwendungen gar nicht so schwer zu begreifen, es ist eher die Disziplin beim Freigeben von Speicher, die man sich angewöhnen muss. Cocoa nutzt ein recht leicht zu verstehendes System, bei dem die Referenzen gezählt werden. Alle Objekte werden mit einem *Referenzzähler* von 1 alloziert. Dieser Referenzzähler kann mit zwei Instanzmethoden beeinflusst werden: `retain` erhöht den Referenzzähler um 1, während er durch `release` verringert wird. Erreicht der Zähler die 0, kann das Objekt freigegeben werden.

Bei diesem „Erhöhen um 1“ kann folgende Richtschnur zur Speicherverwaltung angelegt werden: Wenn Sie an einem Objekt interessiert

sind, das Sie aus einem Methodenaufruf erhalten, erhöhen Sie den Zähler. Haben Sie kein weiteres Interesse mehr, verringern Sie ihn.

Es gibt eine *grundlegende Regel* für die Speicherverwaltung in Cocoa: Sie besitzen jedes Objekt, das Sie durch eine Methode erzeugen, in der die Wörter `alloc`, `new` oder `copy` vorkommen. Und jedes Objekt, das Sie besitzen, müssen Sie irgendwann `per release` freigeben.

Die andere Seite der Medaille ist, dass Sie kein Objekt besitzen, das Sie auf einem anderen Weg erhalten haben, wie zum Beispiel als Rückgabewert einer Methode. Sie geben diese Objekte nicht `per release` frei, weil nicht Sie dafür verantwortlich sind.

Vielleicht wollen Sie aber solch ein Objekt besitzen können. In diesem Fall können Sie den Referenzzähler des Objekts `per retain` erhöhen. So werden Sie ebenfalls Besitzer und müssen das Objekt irgendwann später wieder `per release` freigeben.

Eine interessante Variante ist die Idee von Objekten, die `per autorelease` freigegeben werden. Wenn Sie einem Objekt die Nachricht `autorelease` schicken, fügen Sie es damit einem Pool von Objekten hinzu, denen irgendwann in der Zukunft, im Allgemeinen bei der Rückkehr des Aufrufstapels (Call Stack) des Event Dispatcher, ein `release` geschickt wird. Mit `autorelease` erhält der Aufrufende eine Chance, das zurückgegebene Objekt in Besitz zu nehmen. Tut er das nicht, wird es dann doch freigegeben.

Sie erinnern sich vielleicht daran, dass wir schon ein bisschen Speicherverwaltung betrieben haben – in der Implementierung von `sayHello`: in Abschnitt 3.4, *Arbeiten mit Xcode und dem Interface Builder*, auf Seite 30. Wir haben `per alloc` einen `NSString` erzeugt und ihn `per release` wieder freigegeben, als wir ihn nicht mehr brauchten.

Manche Klassen stellen sehr praktische Konstruktoren bereit, durch die Sie sich einen Schritt sparen können. Statt der üblichen Kombination aus `alloc` und `initWithXXX` können Sie manchmal eine Klassenmethode verwenden, die ein Objekt unter `Autorelease-Verwaltung` erstellt. So bietet zum Beispiel `NSString` eine Klassenmethode `stringWithFormat:` an, die genau so funktioniert wie die Kombination aus `alloc` und `initWithFormat:`, nur dass der zurückgegebene String unter `Autorelease-Verwaltung` steht. Sie brauchen (und sollten) ihn also nicht selbst explizit `per release` freigeben.

In unserer Anwendung haben wir ein paar kleine Speicherlecks. Wir deklarieren per `retain` Eigenschaften für Objekte, die niemals freigegeben werden. Klar, das sind per IB erstellte Objekte, die erst freigegeben werden, wenn die gesamte Anwendung beendet wird. Aber wir sollten uns ab jetzt immer ein bestimmtes Verhalten zur guten Gewohnheit werden lassen: Wenn Ihr Objekt beendet wird, sollte es alle Objekte freigeben, auf die es Referenzen hält. Sie wissen, wann Ihr Objekt freigegeben wird, denn dann wird seine Methode `dealloc` aufgerufen. Daher sollten Sie in `HalloUserController.m` die Objekte freigeben, auf die Sie Instanzvariablenreferenzen halten:

HalloiPhone/HalloUser/Classes/HalloUserController.m

```
- (void)dealloc {
    [helloLabel release];
    [nameField release];
    [super dealloc];
}
```

3.8 Zugriff auf Variablen als Eigenschaften

Eine Ergänzung in Objective-C, die Ihnen überall in den iPhone-APIs begegnet, ist die *Eigenschaft* (Property). Wir haben Eigenschaften schon genutzt, als wir den `text` eines Textfelds ausgelesen und eine andere Eigenschaft `text` für das Label gesetzt haben.

Konzeptionell gesehen, sind Eigenschaften einfach Instanzvariablen mit einer Namenskonvention für Getter- und Setter-Methoden. So hat zum Beispiel die Klasse `UILabel` eine Eigenschaft `text`, die Sie mit einer Methode `setText` setzen und über `text` auslesen können. Normalerweise nutzen Sie den Punktoperator für Zuweisungen, daher schreiben Sie statt `[myLabel setText: myString]` eher `myLabel.text = myString`, und statt `myString = [myLabel text]`⁹ verwenden Sie `myString = myLabel.text`.

Tatsächlich gehen Eigenschaften aber über eine reine Namenskonvention hinaus, da Sie damit das Speicher- und Threading-Verhalten der Eigenschaft deklarieren und optional dieses Verhalten auch durch den Compiler erstellen lassen können.

Sie definieren eine Eigenschaft über eine Anweisung in der Header-Datei:

```
@property (attributes) proptype proptype;
```

⁹ Beachten Sie, dass in Cocoa die Getter kein `get` im Methodennamen nutzen.

In der Implementierungsdatei stellen Sie dann entweder Implementierungen der Getter- und Setter-Methoden bereit, oder Sie lassen sie durch den Compiler für Sie *synthetisieren*:

```
@synthesize propname;
```

Das Schöne an Eigenschaften ist, dass Sie Attribute nutzen können, um Informationen über die Eigenschaften anzugeben, die Sie allein mit Instanzvariablen oder Getter- und Setter-Methoden nicht umsetzen können. Die wichtigsten Attribute beschreiben drei Bereiche:

- *Veränderbarkeit*: Eine Eigenschaft kann durch **readwrite** (den Standardwert) lesbar und veränderbar oder durch **readonly** nur lesbar sein.
- *Speicherverwaltung*: Bei Settern ist das Standardverhalten ein einfaches **assign**, das für primitive Typen ausreichend ist. Für Eigenschaften, die Objekte sind, können Sie mit **retain** eine retain-Nachricht an das zu setzende Objekt schicken (und damit sicherstellen, dass Ihr Objekt der Eigentümer davon ist), während gleichzeitig per **release** der vorherige Wert freigegeben wird. Die dritte Option **copy** sorgt durch ein Kopieren für den Eigenschaftswert, wodurch Sie ihn ebenfalls besitzen.
- *Thread-Handling*: Standardmäßig sind die synthetisierten Getter und Setter Thread-sicher. Da die meisten iPhone-Anwendungen nur einen Thread nutzen und ihre Arbeit in der Hauptschleife von UIKit erledigen, können Sie Ihre Eigenschaften als **nonatomic** deklarieren. Sie sind dadurch nicht mehr Thread-sicher, dafür aber schneller.

Wie deklarieren wir nun die Eigenschaften für `helloLabel` und `nameField`. In `HalloUserController.h` fügen Sie nach der Deklaration der Methode `sayHello`: (irgendwo außerhalb der geschweiften Klammern des **@interface**-Blocks und vor dem **@end**) die Deklarationen für nicht atomare, haltende Eigenschaften für die beiden Instanzvariablen ein, die wir schon deklariert haben.

```
HalloiPhone/HalloUser/Classes/HalloUserController.h
```

```
@property (nonatomic, retain) UILabel *helloLabel;  
@property (nonatomic, retain) UITextField *nameField;
```

Nachdem wir nun diese Eigenschaften erstellt haben, benötigen wir in `HalloUserController.m` nur noch eine zusätzliche Zeile direkt nach **@implementation**.

Eigenschaften: Alt und neu

Zum Zeitpunkt der Entstehung dieses Buchs gibt es bei der Unterstützung von Eigenschaften einen Unterschied zwischen dem iPhone-Simulator und dem echten Gerät, da auf dem Gerät die „moderne“ Objective-C-Laufzeitumgebung arbeitet, auf dem Simulator aber nicht. Hauptunterschied ist, dass Sie bei der modernen Laufzeitumgebung eine Eigenschaft nicht explizit durch eine Instanzvariable ergänzen müssen. Es reicht, nur die **@property**-Anweisung zu schreiben – die Objective-C-Laufzeitumgebung erledigt dann schon den Rest.

Aber wenn Sie das tun, werden Sie feststellen, dass sich Ihr Code nicht kompilieren lässt, wenn das aktuelle SDK für den Simulator eingestellt ist. In diesem Buch werden wir alles so programmieren, dass der Simulator keine Probleme hat, indem wir alle Eigenschaften explizit durch Instanzvariablen ergänzen.

Wenn Sie eine Eigenschaft anders benennen wollen als die dazugehörige Instanzvariable, können Sie in der **@synthesize**-Anweisung ein **=** verwenden. Haben Sie zum Beispiel die Instanzvariable `myTheProp` für die Eigenschaft `theProp`, würden Sie `@synthesize theProp = myTheProp;` in Ihrer `.m`-Datei nutzen. Beachten Sie auch, dass Sie **IBOutlet** sowohl für die Eigenschaft wie auch für die Instanzvariable deklarieren können. Wir ziehen Ersteres vor.

`HalloiPhone/HalloUser/Classes/HalloUserController.m`

```
@synthesize helloLabel, nameField;
```

Damit kann unsere Klasse (und alle anderen, die uns aufrufen) auf diese Eigenschaften über den Punktoperator zugreifen. Das ist richtig praktisch, wenn Sie sie verketteten müssen – zum Beispiel über `halloUserController.nameField.text`.

Die UIKit-Klassen verwenden sehr häufig Eigenschaften, wie Sie vielleicht schon bei der Verwendung der Eigenschaft `text` in den Klassen `UILabel` und `UITextField` bemerkt haben. Und wenn Sie zustimmen, dass die Eigenschaften den Code ein bisschen lesbarer gestalten – neben ihrer Angabe von Veränderbarkeit, Speicherverwaltung und Threading-Verhalten –, hoffen wir, dass Sie sie regelmäßig für Getter und Setter verwenden, so wie wir es im Rest des Buchs auch tun.

3.9 Kompakt: Was man sich merken sollte

Wir haben in diesem Kapitel ein recht großes Gebiet behandelt – ganz ohne Interaktivität begonnen und zum Schluss ein paar fortgeschrittene Fähigkeiten in Bezug auf Klassen-Member genutzt.

Statt einfach noch einmal eine Zusammenfassung zu geben, schließen wir dieses Kapitel mit einer Liste von Dingen, die man auf jeden Fall im Hinterkopf behalten sollte, weil sie in so gut wie jeder iPhone-Anwendung gebraucht werden.

- Auch wenn andere Klassen und Nibs als Teil der Anwendungsstartsequenz eher geladen werden, greifen Sie im Allgemeinen auf `applicationDidFinishLaunching:` Ihres Anwendungs-Delegate sowie auf `initWithCoder:` und `viewDidLoad` der View-Controller zurück, um beim Starten der Anwendung eigenen Code unterzubringen.
- Instanzvariablen in Objective-C gehören in die Header-Datei und dort in die **@interface**-Deklaration. Dabei wird immer der Typ benötigt. Handelt es sich um Objekte, brauchen wir auch das Zeigersymbol (*).
- Methodendeklarationen gehören in Objective-C in die Header-Datei außerhalb des **@interface**-Blocks in der Form: `-(returntype) methodName: (parameter1type) parameter1 parameter2Name: (parameter2type) parameter2 ... ;`. Alternativ können Sie eine Methodendeklaration (oder nur ihre Implementierung) in der `.m`-Datei unterbringen, wenn andere Klassen nicht darauf zugreifen sollen.
- Objekte werden durch Code (üblicherweise mit `alloc` und `init`) oder mit dem Interface Builder erzeugt. Versuchen Sie nicht, das gleiche Objekt auf beiden Wegen anzulegen ... Sie werden damit zwei Objekte erzeugen.
- Um mit Objekten zu arbeiten, die im Interface Builder erstellt wurden, deklarieren Sie Instanzvariablen als **IBOutlet**s und Methoden zum Event-Handling als solche, die **IBAction** zurückgeben. Dann stellen Sie in der `.nib`-Datei mit dem IB die Verbindungen her. Denken Sie daran, die Datei im IB zu sichern, bevor Sie das Projekt in Xcode bauen.

- Um ein Delegate-Protokoll zu implementieren, tragen Sie den Protokollnamen in spitzen Klammern in die **@interface**-Anweisung Ihrer Header-Datei ein – nach dem Klassennamen, von dem Sie ableiten. In der Implementierungsdatei (.m) implementieren Sie dann alle erforderlichen Methoden und all die, an denen Sie interessiert sind. Schließlich machen Sie Ihr Objekt zum Delegate, indem Sie entweder ein `delegate`-Outlet im IB verbinden oder indem Sie eine Eigenschaft im Code setzen (zum Beispiel `textField.delegate = self;`).¹⁰
- Deklarieren Sie Eigenschaften durch eine **@property**-Deklaration in der Header-Datei und nutzen Sie danach **@synthesize** in der Implementierung, damit der Compiler die Getter und Setter für Sie erzeugt.
- Ergänzen Sie die bereitgestellte Implementierung von `dealloc`, um per `release` beliebige Instanzvariablen freizugeben, auf die Ihr Objekt eventuell Anspruch erhebt. Ihr Code sollte `dealloc` niemals direkt aufrufen, stattdessen wird diese Methode automatisch ausgeführt, wenn ein Objekt von der Laufzeitumgebung von Objective-C freigegeben wird.
- Schließlich sollten Sie immer an die grundlegende Speicherregel denken: Sie besitzen ein Objekt, das Sie durch `alloc`, `new` oder `copy` erzeugt haben. Irgendwann müssen Sie es per `release` (oder `autorelease`) wieder freigeben. Objekte, die Sie auf anderen Wegen erhalten haben, besitzen Sie nicht und müssen sie auch nicht wieder freigeben, sofern Sie nicht durch den Aufruf von `retain` ein Besitzer geworden sind.

¹⁰ Wir haben `self` noch nicht erwähnt, aber Sie haben vielleicht schon geschlossen, dass damit ein Objekt auf sich selbst verweisen kann – wie `self` in Ruby oder `this` in Java.

Kapitel 4

View-Controller

Jedes Fenster Ihrer iPhone-Anwendung wird durch einen *View-Controller* (VC) betreut, der dafür verantwortlich ist, dass die View angezeigt wird und auf alle Aktivitäten reagiert, die der Anwender darauf vornehmen kann. Ihr View-Controller ist das C in MVC (Model - View - Controller). Er muss sowohl mit der View als auch mit dem Modell in Ihrer Anwendung kommunizieren. Wenn Ihnen das MVC-Entwurfsmuster nicht vertraut ist, finden Sie weitere Informationen darüber in der Apple-Dokumentation im *Cocoa Fundamentals Guide: Cocoa Design Patterns* [App06a].

Wir werden mit einem View-Controller beginnen, der auf einen angetippten Button reagiert, indem er einen Log-Eintrag erzeugt. Wir haben also zunächst kein Modell und verwenden den View-Controller nur dazu, mit der View zu interagieren. Als Nächstes werden wir dem View-Controller mehr Verantwortung übertragen, indem wir eine Modellklasse *Movie* bauen, eine Instanz davon erzeugen und deren Daten dann auf dem Bildschirm anzeigen. In dieser Phase werden Sie den View-Controller als Vermittler zwischen Modell und View kennenlernen. Im dritten Teil des Beispiels werden wir einen zweiten View-Controller hinzufügen und zeigen, wie die Kontrolle zwischen beiden übergeben wird.

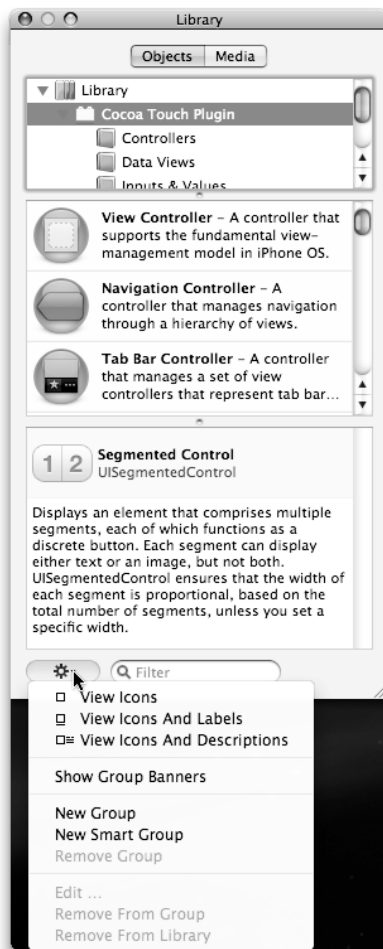
4.1 Implementieren einer Button-Action

Lassen Sie uns einen einfachen View-Controller ein wenig genauer anschauen. Erstellen Sie in Xcode ein neues Projekt, wobei Sie wieder das Template für die View-based Application nutzen. Wie Sie vielleicht noch wissen, erreichen Sie dies über NEW PROJECT des FILE-Menüs oder über --. Geben Sie dem Projekt den Namen *Movie*.¹ Nachdem Xcode Ihr neues Projekt angelegt hat, öffnen Sie die Datei *MovieView-*

Controller.xib im Interface Builder, indem Sie sie doppelt anklicken. Die Datei sollte vertraut aussehen. Abgesehen vom Namen ist es die gleiche Datei, mit der wir in Abschnitt 2.2, *Das „Hallo iPhone“-Projekt erstellen*, auf Seite 13 begonnen haben.

Wir werden dieser ersten View einen Button hinzufügen, der dann in unserem View-Controller zum Ausführen einer Action-Methode führt. Der erste Schritt ist also, der View den Button hinzuzufügen. Sie finden die Buttons in der Library. Um diese zu öffnen, wählen Sie **TOOLS** → **LIBRARY** oder drücken  -  - .

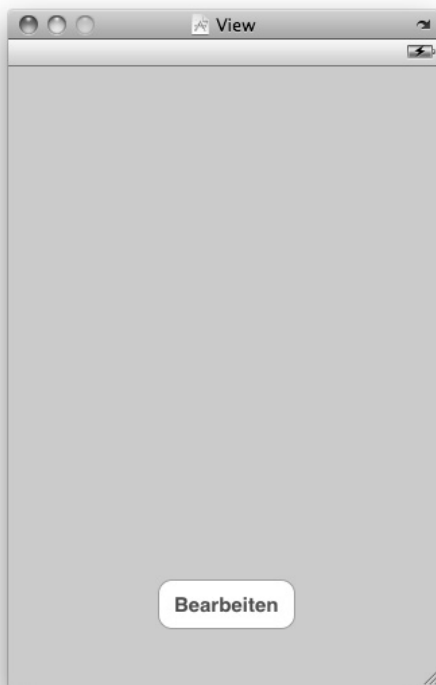
Sie sollten nun etwas sehen, das folgendem Screenshot ähnelt:



1 Auch wenn dieses erste Beispiel nur sehr wenig mit Filmen zu tun hat, werden wir es im gesamten Kapitel verwenden. Der Grund für die Benennung wird klar, wenn wir damit beginnen, den zweiten View-Controller zu bauen.

Es gibt drei Wege, die Liste der Elemente anzeigen zu lassen. Sie können sie einfach als Symbole betrachten, was die meisten erfahreneren UIKit-Entwickler bevorzugen, da die Symbole kleiner sind und sie sie schon kennen. Sie können die Symbole auch zusammen mit einem Label anzeigen lassen, was den Wiedererkennungswert erhöht. Oder Sie entscheiden sich für das Symbol zusammen mit einer Beschreibung. Wir empfehlen Ihnen, mit Letzterem zu beginnen, da die Beschreibungen Ihnen dabei helfen können, zu verstehen, was jedes Widget macht und wie man es verwendet. Der vorherige Screenshot zeigt die geöffnete Library mit dem ausgeklappten Auswahlmenü.

Entscheiden Sie sich für einen Rounded Rect Button, also einen rechteckigen Button mit abgerundeten Ecken, und ziehen Sie ihn auf die View. Legen Sie ihn in der Nähe des unteren Endes ab und verpassen Sie ihm die Beschriftung „Bearbeiten“, indem Sie doppelt auf die Mitte des Buttons klicken und dann Bearbeiten eingeben. Sie sollten jetzt eine View haben, die mehr oder weniger wie die folgende aussieht:



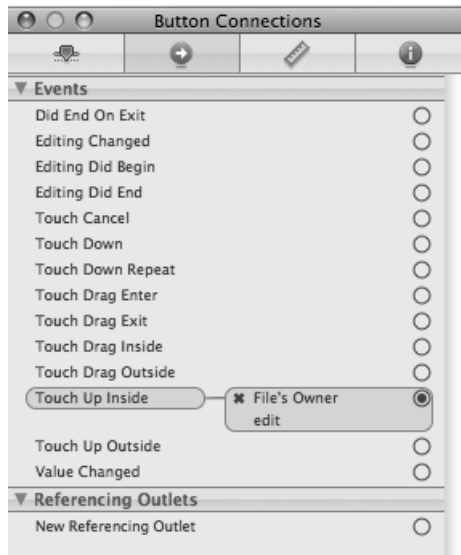
Der Button sollte natürlich auch etwas tun, wenn der Benutzer ihn antippt. Wie Sie in Abschnitt 3.4, *Arbeiten mit Xcode und dem Interface Builder*, auf Seite 30 gesehen haben, sind drei Schritte nötig, um das zu bewerkstelligen. In Xcode deklarieren Sie eine Methode in der Header-Datei von `MovieViewController`. Diese Methode erhält als Rückgabewert den Typ `IBAction`. Dann wechseln Sie wieder zum Interface Builder und verbinden den Button mit dieser Methode. Schließlich kehren Sie zurück zu Xcode, um die Methode zu implementieren.

Öffnen Sie in Xcode die Datei `MovieViewController.h` und ergänzen Sie die folgende Zeile direkt vor der Compiler-Direktive `@end`:

```
ViewControllers/Movie01/Classes/MovieViewController.h
```

```
- (IBAction)edit;
```

Speichern Sie Ihre Arbeit in Xcode und wechseln Sie wieder zum IB, um den Button mit dieser Aktion zu verbinden. Wählen Sie dort den Button aus und öffnen Sie den Connections Inspector (-). Ziehen Sie vom kleinen Kreis rechts von Touch Up Inside zum File's Owner und dort im Pop-up-Fenster zur Aktion `edit`. Der Connection Inspector des Buttons sollte nun so aussehen:



Herzlichen Glückwunsch, Sie haben gerade eine Verbindung im Interface Builder erstellt. Wir werden noch viele weitere solcher Verbindungen herstellen. Damit haben Sie ein Paar aus Ziel und Action hergestellt (mehr zur Kombination Ziel/Action finden Sie im Kasten auf Seite 60). Das Ziel ist der File's Owner und die Action die Methode `edit`. Durch das Herstellen dieses Paares wird dafür gesorgt, dass der Button die Methode `edit` aufruft, wenn er angeklickt wird. Wir wissen, dass das immer noch ein bisschen wie Magie aussieht, aber wir werden das Geheimnis etwas lüften, wenn wir mehr Beispiele erstellen und mehr Verbindungen schaffen.

Mit dem IB sind wir nun fürs Erste fertig. Speichern Sie also Ihre Arbeit, beenden Sie den IB und wechseln Sie wieder zu Xcode. Jetzt müssen wir die Methode in der Datei `MovieViewController.m` implementieren. Öffnen Sie die Datei in Xcode und fügen Sie den folgenden Code irgendwo zwischen den Compiler-Direktiven `@implementation` und `@end` ein:

ViewControllers/Movie01/Classes/MovieViewController.m

```
- (IBAction)edit {
    NSLog(@"Methode edit aufgerufen" );
}
```

Dieser Code wird aufgrund der Verbindung, die wir im IB erstellt haben, aufgerufen, wenn der Button aktiviert wird. Er protokolliert jedes Mal, wenn der Benutzer den Finger vom Button nimmt, die Meldung „Methode edit aufgerufen“ an der Konsole.

Lassen Sie uns ausprobieren, was wir geschafft haben. Speichern Sie Ihre Änderungen und klicken Sie dann in Xcode auf den Button BUILD AND RUN, um die Anwendung auszuführen. Wenn Sie auf den Button tippen, sollten Sie die Protokollmeldung in der Konsole sehen ( -  - ). Der View-Controller reagiert auf einen Benutzer, der mit den Steuerelementen auf dem Bildschirm interagiert. Wenn wir in Kapitel 5, *Tabellen Views*, auf Seite 83 zu den komplexeren Interaktionen kommen, wie dem Hinzufügen und Löschen von Einträgen in der Tabellen-View, wird die prinzipielle Interaktion gleich sein. Der Benutzer wird eine View auf dem Bildschirm antippen oder dorthin ziehen, und im View-Controller wird dann eine durch diese Aktion ausgelöste Methode aufgerufen, die dementsprechend reagiert.

Ziel/Action-Paradigma

Der IB nutzt das Ziel/Action-Paradigma sehr häufig, um es uns leicht zu machen, Elemente der Benutzerschnittstelle mit Methoden in unserem Code zu verbinden. Das Ziel ist das Objekt, dem das Oberflächenelement (zum Beispiel ein Button) eine Nachricht schicken wird. Die Action ist die zu sendende Nachricht. Wir sagen also, dass dem Ziel die Action-Nachricht geschickt wird, wenn das Steuerelement aktiviert wird.

Das UIControl definiert diesen Mechanismus und ermöglicht uns, mit einer von drei Methodensignaturen als Reaktion auf Events diverse Ziel/Action-Kombinationen herzustellen. Damit wird auf die Multi-Touch-Natur des iPhones eingegangen. Diese drei Signaturen sind:

- `-(IBAction)action`
- `-(IBAction)action:(id)sender`
- `-(IBAction)action:(id)sender forEvent:(UIEvent *)event`

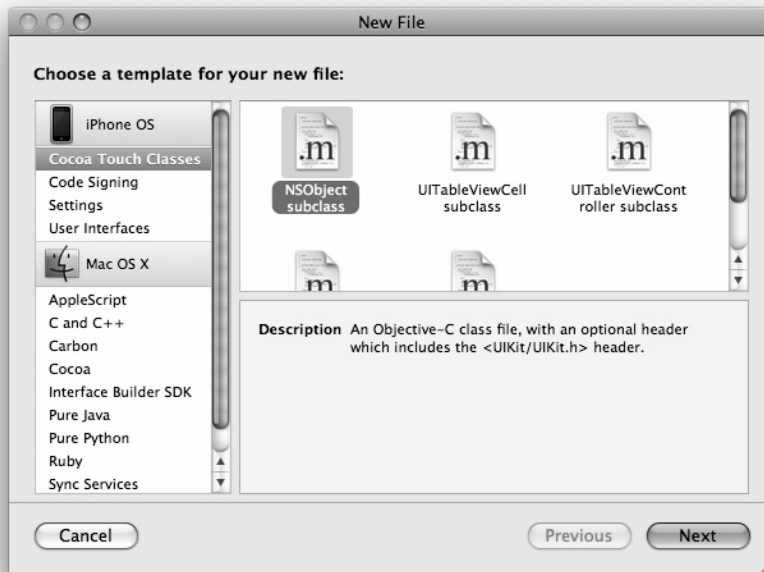
Die erste Option ist die, die wir auch für unsere `edit`-Methode genutzt haben, da wir dort keinerlei Informationen vom Button benötigen, damit die Funktion korrekt arbeitet. Bei anderen Steuerelementen (wie zum Beispiel einem Schieberegler) wollen Sie vielleicht auch auf das Objekt zugreifen können, mit dem der Benutzer gerade arbeitet (das Argument `sender`), damit Sie den aktuellen Wert in Ihrem Code verwenden können. Wenn Sie schließlich das `event` brauchen, weil Sie zum Beispiel ein Multi-Touch-Event erwarten, können Sie die dritte Methodenversion nutzen, um auch auf dieses zusätzliche Objekt zugreifen zu können.

Unabhängig vom gewählten Vorgehen erhalten Sie durch das Ziel/Action-Paradigma enorm viel Flexibilität bei der Interaktion von UI-Elementen mit Ihrem Code.

4.2 Ein Modell erstellen

Wir haben schon das Model-View-Controller-Muster erwähnt, aber noch kein Modell gebaut, um damit zu arbeiten. Ein echtes Modell kapselt die Grundlagen Ihrer Anwendung. In iTunes bilden Klassen wie Song oder Podcast und die Aktionen, die sie unterstützen, wie zum Beispiel play, das Modell. In iPhoto sind es Klassen wie Photo oder Album. Um eine ernsthafte Anwendung zu bauen, brauchen Sie ein reales Modell. Das Modell muss nicht komplex sein, aber wir benötigen schon mehr als nur einen String. Wir werden eine Klasse Movie erstellen, die unser Modell darstellen soll. Dann werden wir eine Instanz von Movie erzeugen, die die Daten enthält, und das bestehende UI so anpassen, dass es diese Daten anzeigt. Der View-Controller wird dabei das Bindeglied sein, das sowohl mit dem Modell als auch mit der View verbunden ist.

Lassen Sie uns damit beginnen, die Klasse Movie zu erstellen. Wählen Sie in Xcode die Gruppe Classes und klicken Sie sie dann entweder mit der rechten Maustaste oder bei gedrückter `[Ctrl]`-Taste an. Wählen Sie nun aus dem Popup-Menü `ADD → NEW`. Im sich öffnenden Dialogfenster wählen Sie `COCOA TOUCH CLASSES → OBJECTIVE-C CLASS`. Im Auswahlmenü `SUBCLASS OF` muss „NSObject“ stehen. Klicken Sie nun auf den Button `NEXT`.



Geben Sie der Klasse den Namen `Movie` und stellen Sie sicher, dass die beiden Checkboxen markiert sind, um die `.h`-Datei zu erstellen und die Datei dem Ziel hinzuzufügen.

Nachdem wir nun unsere Klasse `Model` definiert haben, wollen wir sie mit ein bisschen Leben füllen, indem wir ihr drei Eigenschaften verpassen: einen `NSString` mit dem Namen `title`, der den Titel des Films enthält, eine `NSNumber` namens `boxOfficeGross` mit den Besuchszahlen im Kino und einen `NSString` mit dem Namen `summary`, der eine Zusammenfassung des Inhalts aufnehmen kann. Wir werden in den geschweiften Klammern in unserer Header-Datei drei Variablen deklarieren, die zu diesen Eigenschaften gehören, und für jede Eigenschaft `@property` nutzen, so dass der Compiler die entsprechenden Getter- und Setter-Methoden für uns erzeugen kann.

Zudem wollen wir eine eigene `init`-Methode für unsere Klasse haben, damit wir die drei Eigenschaften beim Erzeugen eines neuen Films initialisieren können. Ergänzen Sie also die Klasse um die Methode `initWithTitle:boxOfficeGross:summary:`. Ihre Header-Datei sollte jetzt so aussehen:

ViewControllers/Movie02/Classes/Movie.h

```
@interface Movie : NSObject {
    NSString *title;
    NSNumber *boxOfficeGross;
    NSString *summary;
}

-(id)initWithTitle:(NSString *)newTitle
    boxOfficeGross:(NSNumber *)newBoxOfficeGross
    summary:(NSString *)newSummary;

@property(n nonatomic, copy) NSString *title;
@property(n nonatomic, copy) NSNumber *boxOfficeGross;
@property(n nonatomic, copy) NSString *summary;

@end
```

Nachdem wir mit dem Interface fertig sind, müssen wir noch die Implementierung vervollständigen, indem wir `@synthesize`-Anweisungen hinzufügen, um die Getter- und Setter-Methoden für die Eigenschaften zu erzeugen. Wir müssen zudem die eigene `init`-Methode implementieren und unseren Speicher wieder aufräumen.

ViewControllers/Movie02/Classes/Movie.m

```

1  @implementation Movie
2
3  @synthesize title;
4  @synthesize boxOfficeGross;
5  @synthesize summary;
6
7  -(id)initWithTitle:(NSString *)newTitle
8      boxOfficeGross:(NSNumber *)newBoxOfficeGross
9      summary:(NSString *)newSummary {
10     self = [super init];
11     if(nil != self) {
12         self.title = newTitle;
13         self.boxOfficeGross = newBoxOfficeGross;
14         self.summary = newSummary;
15     }
16     return self;
17 }
18
19 -(void) dealloc {
20     self.title = nil;
21     self.boxOfficeGross = nil;
22     self.summary = nil;
23     [super dealloc];
24 }
25
26 @end

```

In dieser Implementierung finden sich ein paar neue Konzepte. Zuerst wollen wir uns über die Implementierung der eigenen `init`-Methode unterhalten. Wir haben in Kapitel 3, *Grundlagen der iPhone-Entwicklung*, auf Seite 25 schon die Anwendung eigener `init`-Methoden kennengelernt, aber dies ist die erste Implementierung, die wir selbst erstellen. Der Code ist nicht komplex, aber es gibt ein paar Punkte, die wir ansprechen sollten. In Zeile 10 haben wir `self` den Ausdruck `[super init]` zugewiesen. Weist man `self` den Wert zu, den die Superklasse aus der `init`-Methode zurückgibt, hat diese damit die Möglichkeit, ein anderes Objekt zurückzuliefern. Dieser etwas merkwürdig klingende Fall tritt in manchen Frameworks durchaus ab und zu auf (vor allem bei Core Data). Daher ist es wichtig, diese Zeile in Ihre eigenen `init`-Methoden mit aufzunehmen. Dann haben wir alle Eigenschaften auf die Werte gesetzt, die unserer `init`-Methode übergeben wurden. Schließlich geben wir in der letzten Zeile `self` zurück.

Die Methode `dealloc` ab Zeile 19 kümmert sich darum, den Speicher freizugeben, der von diesen Objekten genutzt wird.² Da die Eigenschaften als

2 Eine erste Einführung in die Speicherverwaltung haben wir in Abschnitt 3.7, *Speicherverwaltung der Anwendung*, auf Seite 47 gegeben.

copy markiert sind, müssen alle in der Methode `dealloc` wieder freigegeben werden. Dazu setzen wir die Eigenschaften auf `nil`. Wie Sie in Abschnitt 3.8, *Zugriff auf Variablen als Eigenschaften*, auf Seite 49 erfahren haben, gibt eine Eigenschaft, die ihren Objektwert kopiert oder in Besitz nimmt, erst den alten Wert frei, bevor sie den neuen zuweist. Daher setzen wir in diesem Fall die Eigenschaft auf `nil`, wodurch das bestehende Objekt erst freigegeben und dann die Instanzvariable auf `nil` gesetzt wird.

Fantastisch – Sie haben gerade Ihre erste Modellklasse erstellt. Wir werden eine Instanz dieser Klasse nutzen, um die Daten unserer Anwendung im Rest des Kapitels zu verwalten. Aber zuerst sollten Sie Ihre Arbeit sichern. Wir müssen nun unser bestehendes UI anpassen, um eine Instanz von `Movie` anzuzeigen.

4.3 Den Controller um Outlets und Actions ergänzen

Wenn Sie mit der Entwicklung in Cocoa noch nicht so vertraut sind, wird die folgende Vorgehensweise ein wenig ungewohnt sein. Bei der Entwicklung mit Cocoa Touch wechselt man immer wieder zwischen dem IB und Xcode hin und her. Das liegt vor allem an der Natur des View-Controller. Er steht mit einem Bein in der Benutzeroberfläche und mit dem anderen im Code. Da wir die Oberfläche im IB und den Code in Xcode bearbeiten, ist es nachvollziehbar, dass wir zwischen den beiden Tools wechseln. Wir versprechen, dass sich eine eventuell vorhandene Unsicherheit schnell legen wird. Bald wird es Ihnen ganz natürlich vorkommen und auch sinnvoll sein.

Da wir schon in Xcode sind, wollen wir damit beginnen, unseren View-Controller um die neuen Eigenschaften zu ergänzen, die wir noch brauchen. Öffnen Sie die Datei `MovieViewController.h` und fügen Sie vier Eigenschaften und die zugehörigen Instanzvariablen hinzu. Dabei gehen Sie genauso vor wie bei der `Movie`-Klasse. Dieses Mal ergänzen Sie eine Eigenschaft für das `movie`-Objekt, das unser Modell sein wird. Dazu kommen drei Outlets – eines für jedes Datenelement im `movie`-Objekt, das wir anzeigen wollen. Achten Sie darauf, dass bei jeder Eigenschaft das Wort `IBOutlet` zu finden ist.

ViewControllers/Movie02/Classes/MovieViewController.h

```

1  #import <UIKit/UIKit.h>
2
3  ► @class Movie;
4
5  @interface MovieViewController : UIViewController {
6  ►     Movie *movie;

```

```

► UILabel *titleLabel;
► UILabel *boxOfficeGrossLabel;
► UILabel *summaryLabel;
10 }
11
► @property(n nonatomic, retain) Movie *movie;
► @property(n nonatomic, retain) IBOutlet UILabel *titleLabel;
► @property(n nonatomic, retain) IBOutlet UILabel *boxOfficeGrossLabel;
► @property(n nonatomic, retain) IBOutlet UILabel *summaryLabel;
16
17 -(IBAction)edit;
18 @end

```

Die Compiler-Direktive `@class` in Zeile 3 ist eine *Vorwärts-Deklaration* (Forward Declaration), durch die dem Objective-C-Compiler mitgeteilt wird, dass er noch eine Klasse mit dem Namen `Movie` finden wird und dass er keine Fehler oder Warnungen ausgeben soll, wenn er sie nicht findet. Solche Vorwärts-Deklarationen sind in Header-Dateien gang und gäbe, um keine Include-Zirkelverweise zu erzeugen (bei denen ein Header einen anderen aufnimmt, der wiederum den ersten inkludiert). Wir verwenden Vorwärts-Deklarationen, weil der Compiler bei solchen Include-Zirkelverweisen nur sehr ärmliche Fehlermeldungen liefert. Daher ist es besser, sich an die Vorwärts-Deklarationen zu gewöhnen, statt sich immer wieder mit komischen Warnungen herumschlagen zu müssen.

Bevor wir Xcode verlassen, wollen wir die `@synthesize`-Aufrufe und die `import`-Anweisung in die Implementierungsdatei aufnehmen. Öffnen Sie `MovieViewController.m` und fügen Sie die `@synthesize`-Anweisungen für jede Eigenschaft ein. Wir bringen diese Anweisungen immer am Anfang des `@implementation`-Blocks unter.

ViewControllers/Movie02/Classes/MovieViewController.m

```

@synthesize titleLabel;
@synthesize boxOfficeGrossLabel;
@synthesize summaryLabel;
@synthesize movie;

```

Zudem müssen Sie für jede `@class`-Vorwärts-Deklaration so gut wie immer die Header-Datei für die entsprechende Klasse in der Implementierungsdatei importieren – hier also `Movie.h`:

ViewControllers/Movie02/Classes/MovieViewController.m

```

► #import "MovieViewController.h"
► #import "Movie.h"

```

Lassen Sie uns nun unsere Aufmerksamkeit auf die GUI-Seite der Anwendung richten, vergessen Sie vorher aber nicht, alles zu sichern, bevor Sie zum Interface Builder wechseln.

4.4 Das UI anpassen

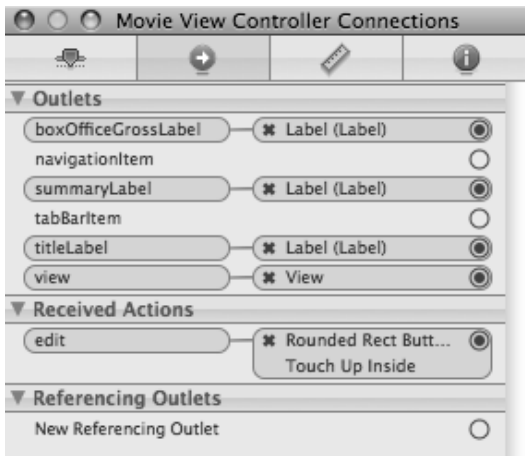
Nachdem wir nun mit unserer Header-Datei fertig sind, wollen wir wieder in den Interface Builder wechseln und `MovieViewController.xib` öffnen. Ziehen Sie drei `UILabel` auf die View, ordnen Sie sie untereinander am oberen Rand an und ziehen Sie sie über die ganze Breite. Die blauen Linien helfen Ihnen beim Ausrichten, so dass sie auch wirklich untereinanderstehen. Nachdem Sie die Labels wie gewünscht angeordnet haben, sollte Ihre View in etwa so aussehen wie die auf dem Screenshot auf der nächsten Seite.

Sobald die Labels platziert sind, verbinden Sie die VC-Outlets damit. Wir müssen diese Verbindungen herstellen, damit unser View-Controller die Labels kennt und die Daten aus dem `movie`-Objekt darüber ausgeben kann.

Verbinden Sie die Outlets mit den entsprechenden Labels. Das können Sie auf zwei verschiedenen Wegen erreichen. Zum einen können Sie bei gedrückter `Ctrl`-Taste die Quelle auf das Zielobjekt ziehen. Wenn Sie die Maus loslassen, wird die Liste gültiger Outlets angezeigt, und Sie entscheiden sich durch einen Klick für einen von ihnen. Das ist alles – sobald Sie das Outlet angeklickt haben, wird die Verbindung hergestellt. Zum anderen können Sie aber auch den Connections Inspector verwenden, wie wir es in Kapitel 3, *Grundlagen der iPhone-Entwicklung*, auf Seite 25 beschrieben haben.



Nachdem Ihre Verbindungen vollständig sind, wählen Sie File's Owner und öffnen den Connections Inspector mit -. Er sollte so aussehen:



Jetzt können wir endlich alle diese Vorbereitungsschritte nutzen und die Informationen für einen bestimmten Film anzeigen.

4.5 Den Controller implementieren

Passen wir nun die Implementierung so an, dass die Daten von movie auf dem Bildschirm angezeigt werden. Als Erstes werden wir das movie-Objekt erstellen. Dabei überschreiben wir die Methode `viewDidLoad` im `MovieViewController`, um eine neue Instanz zu erzeugen, wenn die View, die vom `MovieViewController` kontrolliert wird, geladen ist. Dies ist der Code:

ViewControllers/Movie02/Classes/MovieViewController.m

```

1 -(void)viewDidLoad {
2     [super viewDidLoad];
3     Movie *newMovie = [[[Movie alloc]
4         initWithTitle:@"Iron Man"
5         boxOfficeGross:[NSNumber numberWithInt:650000000.00]
6         summary:@"Cooler Kerl baut coole Rüstung" ] autorelease];
7     self.movie = newMovie;
8 }

```

Der Code ist recht klar. Es wird eine neue Instanz von `Movie` alloziert, die Eigenschaft `movie` wird auf das neue `Movie`-Objekt gesetzt, und dann wird es freigegeben. Interessant daran ist, warum wir `viewDidLoad` nutzen. Um das zu verstehen, müssen wir uns ein bisschen darüber

unterhalten, wie View-Controller arbeiten. Beginnen wir mit der Methode `applicationDidFinishLaunching:` der Klasse `MovieAppDelegate`. Diese Methode wird vom `UIApplication`-Objekt aufgerufen, wenn die Startsequenz erfolgreich abgeschlossen ist. Der Anwendungs-Delegate hat zwei Outlets, die in der `.nib`-Datei `MainWindow.xib` verbunden sind: `window` und `viewController`. Die Eigenschaft `window` ist mit dem Hauptfenster unserer Anwendung verbunden, die Eigenschaft `viewController` mit dem `Movie-View-Controller`. Dies ist der Code für `applicationDidFinishLaunching::`

ViewControllers/Movie02/Classes/MovieAppDelegate.m

```
-(void)applicationDidFinishLaunching:(UIApplication *)application {
    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}
```

Der Anwendungs-Delegate bittet das `window`, die View des `viewController` als Subview hinzuzufügen. Das ist eine recht einfache Codezeile, oder? Dennoch geschieht hinter den Kulissen dabei eine ganze Menge. Wenn ein View-Controller nach seiner View gefragt wird, prüft er zunächst, ob er schon eine hat. Wenn ja, gibt er einfach die schon geladene View zurück. Wenn nicht, ruft er die Methode `loadView` auf. Dieser Prozess arbeitet mehr oder weniger folgende Schritte ab:

- Ist ein Dateiname für eine `.nib`-Datei angegeben (normalerweise im IB definiert, aber auch per Code möglich)?
- Wenn ja, lade die `.nib`-Datei, wobei der View-Controller als File's Owner mitgegeben wird.
- Nachdem die `.nib`-Datei geladen wurde, stelle sicher, dass die View zugewiesen ist. Wenn nicht, wirf eine Exception und gib eine Fehlermeldung aus.

Sollte Ihr View-Controller keinen Dateinamen für eine `.nib`-Datei enthalten, müssen Sie die View manuell in `loadView` aufbauen. Glücklicherweise können wir den Dateinamen im IB definieren, und das manuelle Aufbauen einer View in der Methode `loadView` ist selten bis niemals notwendig.³

³ Wenn Sie sich mit den Nib-Einstellungen des `MovieViewController` vertraut machen wollen, öffnen Sie die Datei `MainWindow.xib` und schauen sich per [⌘]-[1] die Details von `MovieViewController` an. Abschnitt 3.5, *Anatomie Ihrer iPhone-Anwendung*, auf Seite 41 geht genauer darauf ein.

Nachdem die Methode `loadView` beendet ist, wird die Methode `viewDidLoad` aufgerufen. Dies ist genau die richtige Stelle für uns, die Initialisierung durchzuführen (zum Beispiel unser `Movie`-Objekt zu erzeugen), da die Methode nur aufgerufen wird, wenn die View geladen ist. Wenn unsere View aufgrund von Speicherwarnungen wieder freigegeben und auf `nil` gesetzt ist, können wir auch das Modellobjekt auf `nil` setzen. Da wir wissen, dass sich der gesamte Prozess wiederholt, einschließlich des Aufrufs von `viewDidLoad`, wenn die View erneut benötigt wird, haben wir auch dann ein `Movie`-Objekt.

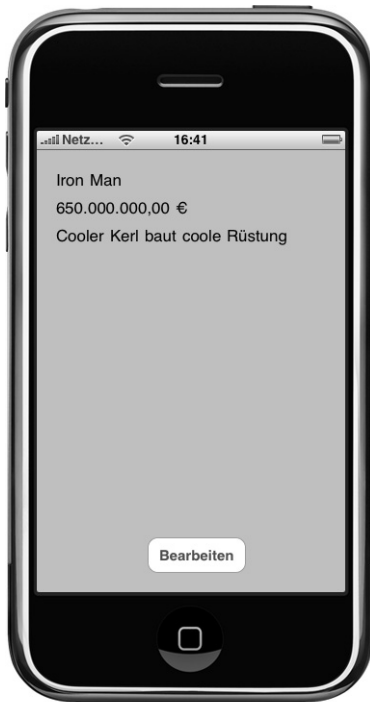
Der nächste Schritt beim Aufbau unseres `Movie-View-Controllers` ist das Implementieren der Methode `viewWillAppear:`. Diese Methode wird jedes Mal aufgerufen, wenn die View sichtbar werden soll. Erinnern Sie sie, dass eine View während der Laufzeit einer Anwendung viele Male angezeigt und wieder vom Bildschirm genommen werden kann. Denken Sie zum Beispiel an die Kontakte-Anwendung – die Liste all Ihrer Kontakte wird immer dann angezeigt, wenn Sie einen einzelnen Kontakt bearbeitet oder auch nur betrachtet haben. Die Methode `viewWillAppear:` wird jedes Mal aufgerufen. Sie ist also der ideale Platz dafür, die Werte zu setzen, die auf der View sichtbar sein sollen. Dies ist der Code für unseren View-Controller:

ViewControllers/Movie02/Classes/MovieViewController.m

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.titleLabel.text = self.movie.title;
    NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
    [formatter setNumberStyle:NSNumberFormatterCurrencyStyle];
    self.boxOfficeGrossLabel.text =
    [formatter stringFromNumber:self.movie.boxOfficeGross];
    [formatter release];
    self.summaryLabel.text = self.movie.summary;
}
```

Mit BUILD AND RUN starten Sie Ihre Anwendung. Der Titel und die weiteren Daten des Films sollten in den Textfeldern erscheinen, wobei das Feld für den Umsatz ein Dollarzeichen enthält.

Dieser Code überträgt die Daten für die `Movie`-Objekte in die Textfelder. Eine neue Klasse in diesem Code ist `NSNumberFormatter`. Wir nutzen solcher Formatter, um zwischen Zahlen und Strings konvertieren zu können.



Herzlichen Glückwunsch! Sie haben erfolgreich einen vollständigen View-Controller und das Modell gebaut, das von ihm betreut wird. Als Nächstes werden Sie dem Anwender erlauben, das Movie-Objekt in einer neuen View zu bearbeiten. Sie werden in Xcode einen weiteren View-Controller erstellen, seine View im IB einrichten und dann die beiden View-Controller miteinander verbinden.

4.6 Den neuen View-Controller erstellen

Lassen Sie uns nun einen zweiten View-Controller erstellen, den wir zum Verwalten der modalen View nutzen. Wählen Sie in Xcode die Gruppe Classes, klicken Sie bei gedrückter `Ctrl`-Taste auf die Gruppe und wählen Sie aus dem Pop-up-Menü `ADD > NEW`. Im nun erscheinenden Wizard wählen Sie in der Gruppe `IPHONE OS > COCOA TOUCH CLASSES` den Eintrag „`UIViewController subclass`“. Achten Sie darauf, dass die Checkbox „With XIB for user interface“ deaktiviert ist, da wir bald unsere eigene Benutzeroberfläche erstellen werden. Klicken Sie auf `NEXT` und geben Sie der Klasse den Namen `MovieEditorViewController`.

Wir wollen ein UI mit drei Textfeldern und einem Button haben. Wir fügen daher der Header-Datei die Outlets und Actions hinzu, die wir für eine saubere Kommunikation mit dem UI benötigen. Mit den Textfeldern kann der Anwender den Titel, den Umsatz und die Zusammenfassung bearbeiten, mit dem Button wird die Bearbeitung abgeschlossen. Daher brauchen wir drei Paare aus Instanzvariable und Eigenschaft. Zudem benötigen wir eine Action-Methode, die durch den Button aufgerufen wird. Das letzte Element dieser Klasse ist die Instanz von Movie, die der Anwender bearbeiten wird, also brauchen wir dafür auch noch ein Instanzvariable/Eigenschaft-Paar (vergessen Sie nicht die Vorwärts-Deklaration @class Movie;). Das Interface unserer Klasse sollte damit so aussehen:

ViewControllers/Movie03/Classes/MovieEditorViewController.h

```
@class Movie;

@interface MovieEditorViewController : UIViewController <UITextFieldDelegate>
{
    UITextField *titleField;
    UITextField *boxOfficeGrossField;
    UITextField *summaryField;
    Movie *movie;
}

@property(n nonatomic, retain) IBOutlet UITextField *titleField;
@property(n nonatomic, retain) IBOutlet UITextField *boxOfficeGrossField;
@property(n nonatomic, retain) IBOutlet UITextField *summaryField;
@property(n nonatomic, retain) Movie *movie;

-(IBAction)done;

@end
```

Der Controller muss auf die Eingabe über die Tastatur reagieren, daher fügen wir innerhalb der spitzen Klammern eine Deklaration für das Protokoll UITextFieldDelegate hinzu. Damit wird dem Compiler mitgeteilt, dass diese Klasse alle erforderlichen Methoden dieses Protokolls implementieren wird. Wir haben Protokolle in Kapitel 3, *Grundlagen der iPhone-Entwicklung*, auf Seite 25 kennengelernt, als wir das Protokoll UIApplicationDelegate besprachen. Das Textfeld-Delegate-Protokoll ähnelt dem Anwendungs-Delegate-Protokoll insofern, als dass während der Lebensdauer des Textfelds in bestimmten interessanten Situationen Methoden an den Delegate geschickt werden. Wir werden uns die Details gleich genauer anschauen. Vergessen Sie nicht, die Datei zu speichern, so dass der Interface Builder auch über die neuen Outlets und Actions Bescheid weiß.

Denken Sie ebenfalls daran, `Movie.h` am Anfang der Datei `MovieEditorViewController.m` zu importieren. Und innerhalb des `@implementation`-Blocks müssen wir für jede Eigenschaft, die wir in der Datei `MovieEditorViewController.h` implementiert haben, eine `@synthesize`-Anweisung eintragen.

Jetzt müssen wir die Methode `viewWillAppear:` implementieren, um die Werte des `Movie`-Objekts in die Textfelder zu übernehmen. Dazu werden wir deren Eigenschaft `text` setzen.

ViewControllers/Movie03/Classes/MovieEditorViewController.m

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    self.titleField.text = self.movie.title;
    self.summaryField.text = self.movie.summary;
    NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
    [formatter setNumberStyle:NSNumberFormatterCurrencyStyle];
    self.boxOfficeGrossField.text =
        [formatter stringFromNumber:self.movie.boxOfficeGross];
    [formatter release]; }
```

Beachten Sie, wie die Zahl `boxOfficeGross` mithilfe eines `NSNumberFormatter` in einen String umgewandelt wird. Das ist der gleiche Weg wie in der Implementierung von `viewWillAppear:` im `MovieViewController`.

In der Methode `done` entfernen wir den aktuellen modalen View-Controller mit einem Aufruf von `dismissModalViewControllerAnimated:`

ViewControllers/Movie03/Classes/MovieEditorViewController.m

```
- (IBAction)done {
    [[self parentViewController] dismissModalViewControllerAnimated:YES];
}
```

Wir haben das UI zwar noch nicht gebaut, aber wir werden unseren View-Controller zum Delegate aller Textfelder machen, so dass wir die Eingabe über das Protokoll `UITextFieldDelegate` begleiten können. Wir werden das Protokoll jetzt schon implementieren, aber die Methoden werden erst dann aufgerufen werden, wenn wir im IB den Delegate für die Textfelder setzen.

Im Protokoll `UITextFieldDelegate` müssen wir zwei Methoden implementieren. Die Methode `textFieldShouldReturn:` wird vom Textfeld aufgerufen, wenn der Anwender auf der Tastatur auf die Return-Taste drückt. Das ist für uns genau die richtige Stelle, um dem Feld durch den Aufruf von `resignFirstResponder` den First Responder-Status zu entziehen (womit auch die Tastatur wieder verschwindet). Diese Methode sollte im

Allgemeinen YES zurückgeben, aber wenn Sie die Eingabe überprüfen und das Textfeld davon abhalten wollen, den First Responder-Status zu verlieren, können Sie auch NO zurückgeben. Denken Sie aber daran, dass der Rückgabewert nur ein „Vorschlag“ ist – erhält der Anwender einen Anruf, werden Sie ihn nicht davon abhalten können, das Feld zu verlassen. Ist Ihre Anwendung darauf angewiesen, dass die Daten gültig sind, dürfen Sie sie nicht nur hier validieren.

ViewControllers/Movie03/Classes/MovieEditorViewController.m

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];
    return YES;
}
```

Die Methode `textFieldDidEndEditing:` wird aufgerufen, sobald das Textfeld seinen First Responder-Status verliert.⁴ Das ist der richtige Moment, die Daten auszulesen, die der Anwender eingetippt hat. Wir holen die Werte aus den Feldern und übertragen sie in das `movie`-Objekt. Dies ist der Code:

ViewControllers/Movie03/Classes/MovieEditorViewController.m

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    if(textField == self.titleField) {
        self.movie.title = self.titleField.text;
    } else if(textField == self.boxOfficeGrossField) {
        NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
        [formatter setNumberStyle:NSNumberFormatterCurrencyStyle];
        self.movie.boxOfficeGross =
            [formatter numberFromString:self.boxOfficeGrossField.text];
        [formatter release];
    } else if(textField == self.summaryField) {
        self.movie.summary = self.summaryField.text;
    }
}
```

Da dieser eine View-Controller den Delegate für alle drei Textfelder spielt, müssen wir unterscheiden, in welches Textfeld eingegeben wurde, um zu wissen, welche Eigenschaft des Films aktualisiert werden soll.⁵

⁴ Ein Textfeld verliert seinen First Responder-Status, weil es ihm gesagt wird (wie wir das in der vorherigen Methode gemacht haben) oder weil es dazu vom System gezwungen wird. Letzteres geschieht, wenn das Textfeld vom Bildschirm genommen wird. Wir haben in unserem Beispiel keine Möglichkeit, diese View verschwinden zu lassen, während die Tastatur zu sehen ist, aber später in Kapitel 5, *Tabellen-Views*, auf Seite 83 werden Sie sehen, wie das geschehen kann.

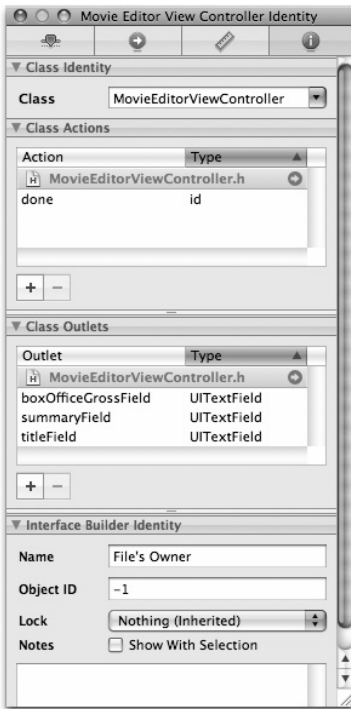
⁵ Haben sich Ihre Fähigkeiten im Umgang mit dem Model-View-Controller-Muster verbessert, werden Sie die Verwendung eines einzelnen Controllers als Delegate für drei Textfelder vielleicht ein wenig unpassend finden. Wir nutzen hier aber nur aus Gründen der Einfachheit eine einzelne Klasse.

Beachten Sie auch, dass wir einen Währungs-Formatter nutzen, um die Daten aus dem Textfeld für das `boxOfficeGross` des Films in eine Zahl umzuwandeln. Daher muss am Anfang ein Währungssymbol stehen, damit der Wert korrekt geparkt werden kann. In einer ausgefeilteren Anwendung würden wir zunächst prüfen, ob es ein Währungssymbol gibt, und dann den passenden Formatter auswählen. Zudem würden wir sicherstellen, dass der vom Formatter zurückgegebene Wert nicht `nil` ist. In dem Fall würden wir den Anwender nämlich darüber informieren, dass der Wert ungültig ist. In dieser Anwendung nutzen wir aber aus Gründen der Übersichtlichkeit einfach nur den Währungs-Formatter.

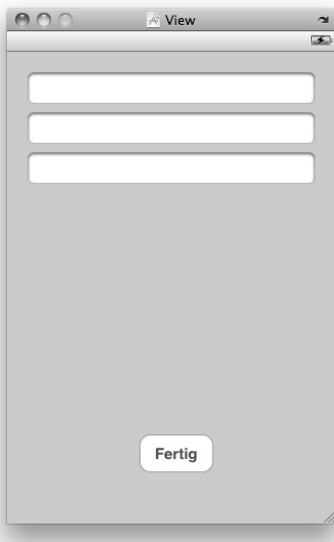
4.7 Erstellen des UI

Nun müssen wir zum Projekt noch eine neue `.nib`-Datei hinzufügen, die die View für unseren neuen View-Controller enthält. Klicken Sie in Xcode bei gedrückter `Ctrl`-Taste auf die Gruppe Resources, wählen Sie `ADD → NEW FILE`, klicken Sie auf `iPhone OS → User Interfaces` und dann auf das Element `View XIB`. Klicken Sie auf `NEXT`, geben Sie der Datei den Namen `MovieEditorViewController.xib` und klicken Sie dann auf `FINISH`. Öffnen Sie `MovieEditorViewController.xib` im IB, indem Sie es doppelt anklicken.

Wir werden das neue Nib so anpassen, dass es die Benutzerschnittstelle für unseren `MovieEditorViewController` wird. Dazu müssen wir als Erstes die Klasse des File's Owner ändern. Wählen Sie dieses Objekt aus und öffnen Sie per `⌘-4` den Identity Inspector. Ändern Sie das Feld `Class` auf `MovieEditorViewController`. Der Inspector sollte nun alle Outlets und Actions anzeigen, die wir dem Header hinzugefügt haben, und wie folgt aussehen:



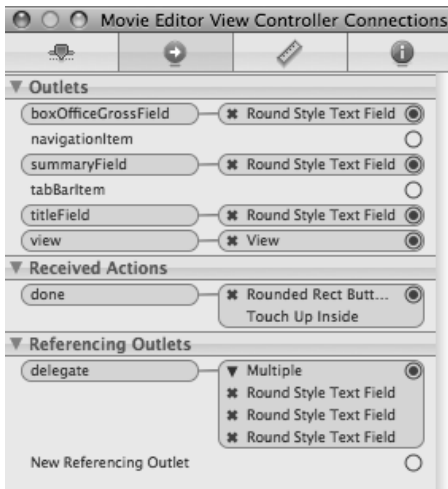
Nachdem der IB nun über unsere Outlets und Actions Bescheid weiß, müssen wir noch das UI und die Verbindungen erstellen. Öffnen Sie die View mit einem Doppelklick. Fügen Sie drei Textfelder und einen Button hinzu. Hier haben wir auch noch die Hintergrundfarbe geändert – ganz ohne Hintergedanken, einfach nur so:



Jetzt haben wir ein UI und den Controller, der es verwaltet. Es bleiben noch die Verbindungen, die wir herstellen müssen. Wählen Sie das Objekt File's Owner und verbinden Sie jedes der Textfeld-Outlets mit dem entsprechenden Textfeld auf dem UI. Verbinden Sie zudem die View des File's Owner mit der View.

Wir mögen die Methode des Ziehens bei gedrückter **Ctrl**-Taste, Sie können aber auch per **⌘-2** den Connections Inspector öffnen und von den Buttons in den linken Bereich jedes der Outlets verbinden. Auch der Button muss noch mit der Action-Methode `done` verbunden werden. Vergessen Sie nicht, jedes der Textfeld-Delegates mit dem File's Owner zu verbinden.

Wenn wir die Verbindungen hergestellt haben, sollte der Connections Inspector in etwa so aussehen:

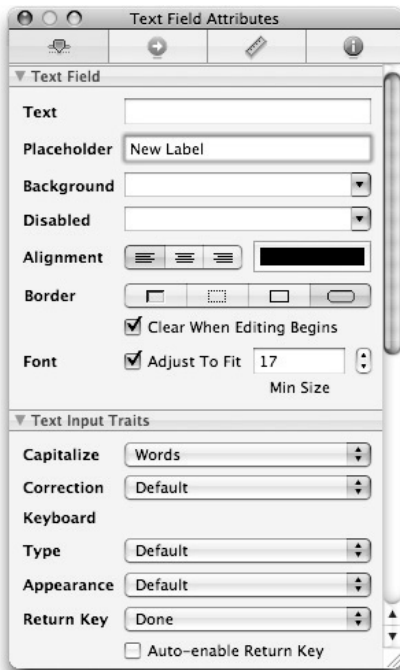


Konfigurieren des Textfelds

Eines der coolsten Features des iPhone ist die Multi-Touch-Tastatur. Ohne echte Tastatur haben Sie mehr Platz für den Bildschirm. Auch wird es deutlich einfacher, unterschiedliche Sprachen zu unterstützen. Immer wenn Sie in einen Textbereich tippen, bringt das iPhone automatisch eine Tastatur auf den Bildschirm. Technisch gesehen, wird dabei das Textfeld der First Responder. Dabei handelt es sich um das Objekt, das als Erstes bestimmte Arten von Events mitgeteilt bekommt (Tastatur-„Eintipper“ sind dabei die wichtigsten). Alle Controls in Cocoa Touch, die mit Text zu tun haben, zeigen eine Tastatur an, wenn sie First Responder werden.

Die Multi-Touch-Tastatur ist für die Anwender eine tolle Sache, aber wir als Entwickler müssen etwas mehr Aufwand treiben, um sie richtig zu nutzen. Die beim Antippen eines Textfelds sichtbar werdende Tastatur ist für eine gute User Experience entscheidend. Wir wollen, dass die richtige Tastatur angezeigt wird, und wir wollen, dass sie korrekt konfiguriert ist, damit Groß- und Kleinschreibung so funktionieren, wie es der Anwender möchte oder erwartet. Wann immer Sie ein Texteingabefeld auf Ihrem UI anzeigen, sollten Sie sich darüber Gedanken machen, was Ihr Anwender erwartet, um ihm so weit wie möglich entgegenzukommen.

Hier ist das Textfeld so konfiguriert, dass Wörter mit Großbuchstaben beginnen und die Korrektur-, Typ- und Erscheinungsoptionen dem Standard entsprechen. Die Return-Taste ist auf „Done“ gesetzt, so dass der Anwender zum Abschluss seiner Eingabe einfach darauf tippen kann.



Wenn Sie sich überlegen, wie sich die Tastatur Ihren Anwendern präsentieren soll, suchen Sie Antworten auf folgende Fragen:

- Wird der Text ungewöhnliche Wörter enthalten? Dann schalten Sie die Korrekturoption ab.
- Handelt es sich bei dem Text um einen Namen? Dann sorgen Sie für große Anfangsbuchstaben bei jedem Wort.

- Geht es um eine Telefonnummer, eine E-Mail-Adresse oder eine URL? Dann nutzen Sie den richtigen Tastaturtyp.
- Was erwartet der Anwender nach Abschluss der Eingabe? Wird eine URL angezeigt, bei Google gesucht, oder werden die Daten gesichert? Wählen Sie den richtigen Text für die Return-Taste.

Nehmen Sie sich die Zeit, die Einstellungen in diesem Bereich sinnvoll vorzunehmen. Dadurch wird die Eingabe „natürlicher“, und die Anwender werden Ihr Programm eher mögen. Übernehmen Sie jedoch einfach nur die Standardvorgaben, können Sie sich damit die ganze User Experience verderben.

Die Verwendung des Platzhaltertexts ist eine gute Möglichkeit, den Anwender darauf hinzuweisen, was im Textfeld eingegeben werden sollte. Egal, ob es sich um das Feld für den Vornamen oder so handelt – durch ein sinnvolles Befüllen des Platzhalters weiß der Anwender besser, was dort einzugeben ist. Wir haben zudem festgelegt, dass die Tastatur die Wörter mit einem großen Anfangsbuchstaben versehen soll, da dieser Text eine Überschrift werden wird, für die der Anwender (zumindest im Englischen) vermutlich solch eine Schreibweise erwartet. Schließlich wird die Return-Taste auf Done eingestellt, damit der Anwender weiß, was er antippen muss, um seine Änderungen abzuschließen.

Es gibt noch eine weitere Einstellung – das Löschen von Feldern, wenn man sie bearbeiten will. Ich mag die Anwendung lieber, wenn die Textfelder den Text zu Anfang des Editierens nicht löschen. Mein Vorschlag ist, beides auszuprobieren, um zu sehen, was Ihnen gefällt. Aber dies ist eine weitere Einstellung, die Sie mit Bedacht treffen sollten. Sie können dieses Feature ein- und ausschalten, indem Sie die Checkbox „Clear When Editing Begins“ im Attributes Inspector dementsprechend setzen.

4.8 Der MovieEditorViewController

Wir haben es fast geschafft. Beide View-Controller und die Benutzerschnittstelle sind so eingerichtet, dass wir Filmdaten anzeigen und bearbeiten können. Aber wir müssen noch den Weg zur zweiten View bauen, in der der Benutzer die Filmdaten bearbeitet. Dies sind die grundlegenden Schritte dafür:

1. Hinzufügen eines Outlets zum MovieViewController für die Instanz von MovieEditorViewController, die wir erzeugen werden.
2. Anpassen der Methode edit des MovieViewController, um den MovieEditorViewController modal anzuzeigen.

3. Erstellen einer Instanz von `MovieEditorViewController` in der `.nib`-Datei von `MovieViewController` und Einrichten der Verbindung zum Outlet.

Das Hinzufügen des Outlet ist für Sie mittlerweile ein alter Hut, daher wechseln wir zu `MovieViewController.h` und ergänzen ein Outlet namens `editingViewController` für den `MovieEditorViewController`. Stellen Sie sicher, dass Sie auch die Vorwärts-Deklaration einbauen:

ViewControllers/Movie03/Classes/MovieViewController.h

```
@class Movie;
▶ @class MovieEditorViewController;

@interface MovieViewController : UIViewController {
    Movie *movie;
    UILabel *titleLabel;
    UILabel *boxOfficeGrossLabel;
    UILabel *summaryLabel;
▶    MovieEditorViewController *editingViewController;
}

@property(n nonatomic, retain) Movie *movie;
@property(n nonatomic, retain) IBOutlet UILabel *titleLabel;
@property(n nonatomic, retain) IBOutlet UILabel *boxOfficeGrossLabel;
@property(n nonatomic, retain) IBOutlet UILabel *summaryLabel;
▶ @property(n nonatomic, retain) IBOutlet
▶    MovieEditorViewController *editingViewController;

-(IBAction)edit;

@end
```

Als Nächstes müssen wir die Methode `edit` so anpassen, dass sie die `Movie`-Instanz für den View-Controller der Edit-View setzt und den neuen Controller anzeigt, statt nur eine Meldung zu protokollieren. Die Anzeige erreichen Sie durch die Methode `presentModalViewController:animated:`⁶. Vergessen Sie auch nicht die Importanweisungen in der Implementierungsdatei von `MovieEditorViewController.h` und die `@synthesize`-Anweisungen für jede Eigenschaft.

ViewControllers/Movie03/Classes/MovieViewController.m

```
-(IBAction)edit {
    self.editingViewController.movie = self.movie;
    [self presentModalViewController:self.editingViewController animated:YES];
}
```

⁶ Es gibt in OS 2.2.1 (und älter) einen Fehler bei modalen Views, der dazu führt, dass `viewWillAppear:` aufgerufen wird, bevor die View geladen ist. Um das zu umgehen, fügen Sie in der `edit`-Methode vor dem Aufruf von `presentModalViewController:animated:` die Zeile `[self.editingViewController view]` hinzu.

4.9 Der Edit View-Controller im Interface Builder

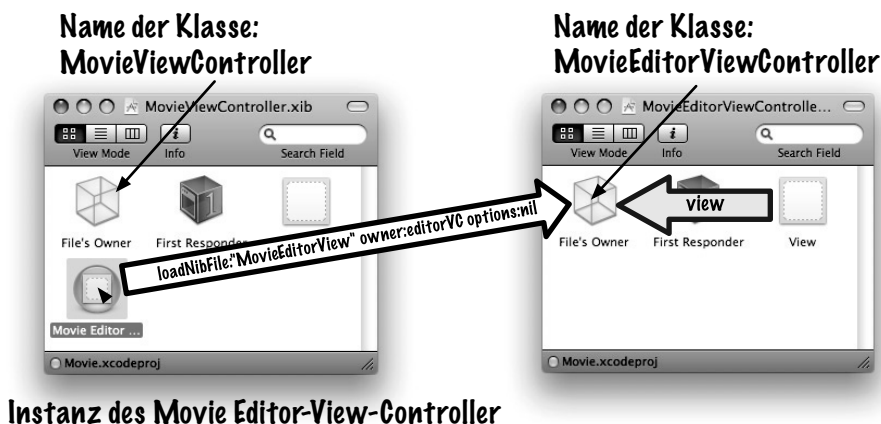
Öffnen Sie die Datei `MovieViewController.xib`. Wir müssen dieser `.nib`-Datei einen View-Controller hinzufügen. Ziehen Sie dazu aus der Library (--) einen View-Controller in das Dokumentenfenster.

Wir müssen die Klasse dieses View-Controller nach `MovieEditorViewController` ändern. Wählen Sie den neuen View-Controller aus, öffnen Sie den Identity Inspector (-) und ändern Sie Class in `MovieEditorViewController`. Als Nächstes müssen wir den `.nib`-Dateinamen für diesen View-Controller setzen. Also öffnen Sie mit - den Attributes Inspector und geben in das Feld „NIB Name“ `MovieEditorViewController` ein. Damit ist die Konfiguration unseres neuen View-Controller abgeschlossen.

Jetzt müssen Sie nur noch die Verbindung vom Outlet `editingView-Controller` im File's Owner zum neuen View-Controller herstellen. Anschließend ist die Anwendung fertig.

Die View des View-Controller

Sie erinnern sich noch: Wenn der View-Controller nach seiner View gefragt wird und sie nicht da ist, prüft er, ob seine `.nib`-Datei angegeben wurde. In diesem Fall lädt er die `.nib`-Datei und übergibt `self` als File's Owner. Und da wir die Eigenschaft `view` in unserer Datei `MovieEditorViewController.xib` gesetzt haben, wird auch die View erzeugt. Dies sind die prinzipiellen Schritte, die vorgenommen werden, wenn die `view` angefordert wird, aber noch nicht vorhanden ist:



Klicken Sie jetzt auf BUILD AND RUN, um sich das Ergebnis im Simulator anzuschauen. Tippen Sie auf den BEARBEITEN-Button, sollte sich die modale View von unten auf den Bildschirm schieben. Wählen Sie eines der Felder aus, tippen Sie einen neuen Wert ein und klicken Sie dann auf FERTIG. Sie sollten zurück auf die Movie-View gelangen, wobei der neue Text im entsprechenden Feld angezeigt wird.

Fantastisch! Sie haben soeben Ihren allerersten View-Controller von Grund auf erstellt. Wenn wir uns mit dem Rest des UIKit befassen werden Sie noch viele View-Controller erstellen. Manche werden etwas komplexer sein, aber die zugrunde liegenden Prinzipien sind immer die gleichen. Der View-Controller betreut eine View und sorgt dafür, dass diese View auf dem Bildschirm im richtigen Zusammenhang angezeigt wird. Zudem dient er als Schnittstelle zu den .nib-Dateien, daher können wir leicht mehrere Elemente auf unserer Oberfläche zusammenbauen. Wenn Sie Navigationen über drei oder vier Ebenen erstellen, werden sich die Details unterscheiden, aber die Grundlagen bleiben gleich.

Kapitel 5

Tabellen-Views

Tabellen-Views (Table Views) sind einer der am häufigsten eingesetzten Bestandteile von UIs im iPhone OS. Eine Menge Standardanwendungen nutzt sehr viele Tabellen-Views, so zum Beispiel Mail, Safari, die Telefonanwendung, iPod, iTunes und viele mehr. Tatsächlich gibt es weniger Standardanwendungen, die keine Tabellen nutzen, als solche, die welche verwenden. Aufgrund der Nützlichkeit von Tabellen und der Tatsache, dass die Anwender sehr gut mit Tabellen umgehen können, ist es sehr wahrscheinlich, dass Ihre Anwendungen einige ihrer Oberflächen mit Tabellen-Views umsetzen werden.

In diesem Kapitel werden wir zeigen, wie man Daten in einem Tabellenformat darstellen kann. Dazu nutzen wir die Klasse *Movie* aus Kapitel 4, *View-Controller*, auf Seite 55. Dort haben wir mit View-Controllern ein einzelnes *Movie*-Objekt angezeigt und bearbeitet. In diesem Kapitel werden wir eine Tabellen-View nutzen, um viele *Movie*-Objekte anzuzeigen und zu einem zweiten View-Controller zu wechseln, in dem wir ein bestehendes Objekt bearbeiten oder ein neues erzeugen können.

5.1 Elemente einer Tabelle

Auf dem iPhone ist eine Tabellen-View eine eindimensionale, vertikal angeordnete Liste von Elementen, die optional in mehrere Abschnitte (Sections) unterteilt ist. Diese Abschnitte machen aus der Liste eine zweidimensionale Datenstruktur. Jeder Abschnitt hat eine beliebige Anzahl von Elementen, so dass ein bestimmtes Element in einer Tabelle durch seinen Abschnitt und die *Zeile* (Row) innerhalb dieses Abschnitts identifiziert wird.

In Abbildung 5.1 können wir die Darstellung von Tabellen-Views im Interface Builder sehen. Dabei sind dort die Namen von US-Bundesstaaten als Abschnittsüberschriften und Städtenamen als Zeilen aufgeführt. Es gibt zwei visuelle Stile für Tabellen – einen „Plain“-Stil, bei dem die Felder horizontal bis zu den Rändern reichen, und einen „Grouped“-Stil, der mit abgerundeten Ecken und Einrückungen arbeitet, um die Zeilen jedes Abschnitts zu gruppieren. Die gruppierte Tabelle in der Abbildung zeigt zwei Abschnitte: einen mit vier Zeilen für den ersten Abschnitt („California“) und einen mit drei Zeilen für den zweiten („New York“).

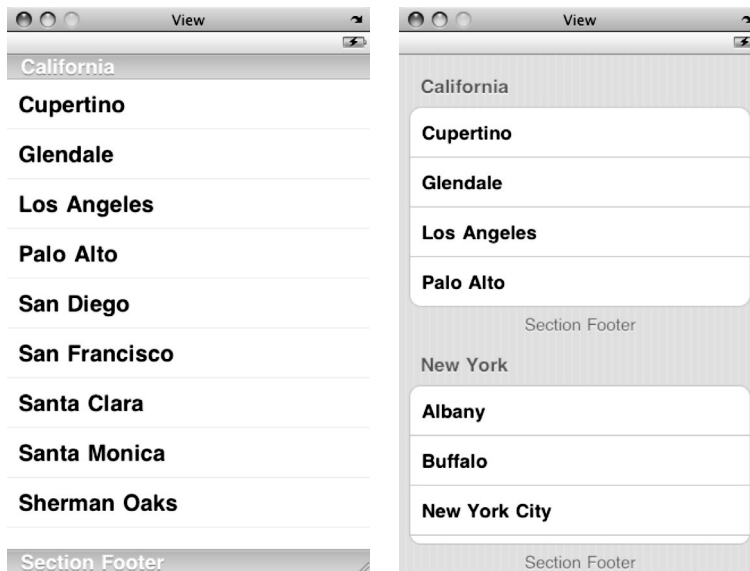


Abbildung 5.1: Zwei UITableView im Interface Builder im Stil „Plain“ (links) und „Grouped“ (rechts)

Eine iPhone-Tabelle besteht aus drei Dingen: einer View, einer Datenquelle (Data Source) und einem Delegate. Sie beginnen mit einer Klasse UITableView, die die Tabelle auf dem Bildschirm anzeigt und auf die Benutzeraktionen reagiert – wie zum Beispiel ein Tippen, um eine Zeile auszuwählen, oder ein Wischen, um ein Element zu löschen. Die UITableView braucht eines oder mehrere andere Objekte, um sinnvoll genutzt werden zu können:

- Eine *Tabellen-View Data Source* ist ein Objekt, das die Beziehung zwischen der sichtbaren UITableView und ihrem Inhalt verwaltet. Die Methoden im Protokoll UITableViewDataSource stellen die Anzahl der Abschnitte und Zeile in der Tabelle bereit, liefern Titel

für Kopf- und Fußzeilen und generieren die Views für jedes Feld. Die Data Source enthält auch Methoden zum Einfügen, Löschen oder Umsortieren von Tabellenzeilen. Die meisten dieser Features sind optional – die einzigen erforderlichen Methoden sind `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath:`.

- Ein *Tabellen-View Delegate* ermöglicht es der Host-Anwendung, die Darstellung und das Verhalten der Tabelle besser zu kontrollieren. Ein Objekt, das das Protokoll `UITableViewDelegate` implementiert, wird über eine ganze Reihe von Benutzeraktionen informiert – zum Beispiel über den Anfang oder das Ende einer Zeilenauswahl oder eines Bearbeitungsvorgangs. Andere Methoden ermöglichen es dem Delegate, angepasste Views für Kopf- und Fußzeilen bereitzustellen und Zellenhöhen festzulegen, die nicht dem Standard entsprechen. Die Idee der Cocoa-Delegate wurde in Abschnitt 3.6, *Verhaltensanpassung durch Delegieren*, auf Seite 44 vorgestellt, und in Abschnitt 4.6, *Den neuen View-Controller erstellen*, auf Seite 70 haben wir ein `UITextFieldDelegate` verwendet, um das Verhalten der Tastatur für ein Textfeld anzupassen.

Um eine Tabelle in Ihrer Anwendung zu nutzen, erzeugen Sie eine `UITableView` – normalerweise im Interface Builder – und verbinden sie mit Objekten, die die Data Source und die Delegate-Protokolle implementieren. Häufig wird der View-Controller, der die Tabellen-View betreut, auch als Data Source *und* als Delegate dienen. Indem Sie den View-Controller zur Data Source für die Tabelle machen, müssen Sie `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath:` aus `UITableViewDataSource` implementieren, ebenso mindestens `tableView:didSelectRowAtIndexPath:` von `UITableViewDelegate`, um auf das Tippen des Anwenders auf eine der Tabellenzeilen reagieren zu können.

5.2 Tabellenbasierte Navigation einrichten

Die `UITableView` wird als Navigationsmethode im ganzen iPhone OS genutzt. In Mail verwenden Sie Tabellen, um einen Account auszuwählen, dann eine Postfach innerhalb dieses Accounts und dann eine Nachricht innerhalb dieses Postfachs. Bei jedem dieser Schritte werden die Daten als Tabelle präsentiert, und durch das Auswählen einer Zeile *navigiert* man zu einer neuen View, wobei man entweder zu einer anderen Tabelle oder zu einer View wechselt, die den Inhalt einer Nachricht anzeigt. Um diese Art von Verhalten in den verschiedenen Anwendungen zu standardisieren, stellt Xcode ein navigationsbasiertes Anwen-

dungs-Projekt-Template bereit, das eine Tabelle als erste View nutzt. Wir werden dieses Template verwenden, um in diesem Kapitel den Umgang mit Tabellen und Navigation zu erlernen.

In Xcode wählen Sie **FILE → NEW PROJECT** und klicken dann dort auf das **NAVIGATION-BASED APPLICATION TEMPLATE**. Stellen Sie sicher, dass die Checkbox „Use Core Data for storage“ deaktiviert ist.

Geben Sie dem Projekt den Namen **MovieTable**. Xcode wird es dann für Sie einrichten und zwei Klassen (**MovieTableAppDelegate** und **RootViewController**) sowie zwei Nibs erstellen (**MainWindow.xib** und **RootViewController.xib**).

Das ist ein etwas aufwendigeres Projekt-Template als die bisher genutzten, daher ist es hilfreich, zu verstehen, wie all die Elemente zusammenwirken. Öffnen Sie die **MainWindow.xib** mit dem Interface Builder und ändern Sie den View-Modus von der Symbolanzeige auf eine Liste (der mittlere Button des View-Mode-Controls). Durch das Aufklappen der Baumstruktur erhalten Sie ein Fenster wie in Abbildung 5.2. Das Nib hat einen *Navigation Controller*, ein Objekt, das wir im Code nutzen werden, um zwischen den verschiedenen Views hin- und herzunavigieren. Der Navigation Controller besitzt zwei Kindobjekte – eine *Navigation Bar* (Navigation Bar), die Sie als Balken am oberen Rand von Navigationsscreens erkennen (wo sie im Allgemeinen Buttons wie **ZURÜCK** oder **BEARBEITEN** aufnimmt) und den **RootViewController**, der ein *Navigation Item*-Objekt besitzt.



Abbildung 5.2: Navigationsobjekte in der Datei **MainWindows.XIB**

Für die Navigation ist also alles vorhanden, aber wo ist die Tabelle? Wenn Sie sich den `RootViewController` anschauen, werden Sie sehen, dass er seine View aus `RootViewController.xib` holt. Öffnen Sie dieses Nib, finden Sie dort standardmäßig ein einzelnes Objekt vom Typ `UITableView`. Was haben wir also bisher erfahren? Eine Navigationsanwendung hat diese Klasse `UINavigationController`, die für die Navigation verantwortlich ist. Gleichzeitig ist sie das übergeordnete Element zu `RootViewController`, bei dem es sich um den View-Controller für die View handelt, die der Anwender als Erstes zu Gesicht bekommt. Diese erste View wiederum ist eine `UITableView`.

5.3 Tabellendaten modellieren

Da dem `RootViewController` die Tabellen-View gehört, wollen wir diese Klasse genauer anschauen. In ihrer Implementierungsdatei `RootViewController.m` finden Sie die Standardimplementierungen für einige der Methoden für die Data Sources und die Delegates, von denen drei nicht auskommentiert wurden: `numberOfSectionsInTableView:`, `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath`. Wenn Sie sich mit dem Interface Builder `RootViewController.xib` anschauen, werden Sie feststellen, dass die Outlets `dataSource` und `delegate` der Tabelle mit File's Owner verbunden sind. Im Endergebnis ist diese Tabelle also schon verknüpft und lauffähig – sie geht davon aus, dass der `RootViewController` als Delegate und Data Source dient und die Klasse die minimale Implementierung dieser Protokolle bereitstellt, damit die Anwendung laufen kann. Die Protokolle sind in `RootViewController.h` nicht explizit deklariert, da der Controller von `UITableViewController` abgeleitet ist, deren Deklaration schon die beiden Protokolle enthält. Wenn Sie einen anderen View-Controller mit einer Tabelle verwenden, dürfen Sie nicht vergessen, dem **@interface** in der Header-Datei den Text `<UITableViewDataSource, UITableViewDelegate>` hinzuzufügen, um anzugeben, dass Sie diese Protokolle implementieren.

Die Standardimplementierung kann für eine Tabelle genutzt werden, die einen Abschnitt mit null Zeilen enthält. Sie enthält auch die Logik für das codebasierte Erstellen von Feld-Views, aber bei null Zeilen wird dieser Code nie genutzt werden. Als Erstes müssen wir daher die Methoden zum Zählen von Abschnitten und Zeilen implementieren – und zwar nicht nur in der trivialsten Version. Das bedeutet, wir müssen ein *Modell* für die Tabellendaten entwickeln, das über den `RootViewController` mit der View verbunden ist. Wir implementieren also das klassische Model-View-Controller-Entwurfsmuster.

Ein Tabellenmodell muss nicht total ausgefuchst sein, es muss nicht einmal wie in anderen Sprachen eine eigene Klasse sein. Für eine Tabelle mit nur einem Abschnitt kann man auch einfach ein NSArray verwenden, das die Objekte enthält, die die Tabelle repräsentieren. Die Länge des Arrays liefert Ihnen die Anzahl an Zeilen, und der Inhalt eines bestimmten Felds kann über die Methode `objectAtIndex:` des Arrays erreicht werden.

Am Anfang dieses Kapitels haben wir gesagt, dass wir die Klasse `Movie` aus dem vorherigen Kapitel für unsere Tabelle wiederverwenden würden. Klicken Sie in `Groups & Files` bei gedrückter Maustaste oder mit der rechten Maustaste auf den Ordner `Classes` und wählen Sie `ADD → EXISTING FILES`. Wechseln Sie zum vorigen Projekt `Movie`,¹ wählen Sie dort den Ordner `Classes` aus, nutzen die Command-Taste (`⌘`), um `Movie.h` und `Movie.m` auszuwählen, und klicken dann auf den Button `ADD`. Im nächsten Dialogfenster stellen Sie sicher, dass die Checkbox „Copy items into destination group's folder (if needed)“ markiert ist, und klicken erneut auf `ADD`, um die Dateien in dieses Projekt zu kopieren.

Fügen Sie in `RootViewController.h` auch noch `#import "Movie.h"` ein, da wir die Klasse `Movie` in unserem View-Controller verwenden werden.

Da wir als Nächstes eine editierbare Liste von `Movies` anbieten wollen, greifen wir auf ein Array zurück, in das wir Elemente einfügen und entfernen können. Daher deklarieren wir im `@interface`-Block von `RootViewController.h` die Instanzvariable `NSMutableArray *moviesArray`; Dieses Array muss initialisiert werden, und wir wollen auch noch ein paar Daten bereitstellen (wir werden dem Anwender später erlauben, seine eigenen Daten hinzuzufügen). Daher aktivieren wir die bereits vorhandene Methode `viewDidLoad` in `RootViewController.m`, indem wir die Kommentarzeichen entfernen, und fügen den hervorgehobenen Code ein, um einen `Movie` zu erzeugen und ihm dem Array hinzuzufügen:

TableViews/MovieTable01/Classes/RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    ▶ moviesArray = [[NSMutableArray alloc] init];
    ▶ Movie *aMovie = [[Movie alloc] init];
    ▶ aMovie.title = @"Spielzeug-Anekdoten" ;
    ▶ aMovie.boxOfficeGross = [NSNumber numberWithInt: 191796233];
    ▶ aMovie.summary =
        @"Haben Sie je geglaubt, Ihre Puppen würden leben?
        Nun, sie tun es." ;
```

¹ In den zu diesem Buch herunterladbaren Beispielen ist das `ViewControllers/Movie02`.

```

► [moviesArray addObject: aMovie];
► [aMovie release];
}

```

Nachdem unser Modell nun ein paar Daten enthält, müssen wir die Methoden von `UITableViewDataSource` so anpassen, dass wir die Daten in die `UITableView` transportiert bekommen. Die Standardimplementierung von `numberOfSectionsInTableView:` liefert 1 zurück, was so weit in Ordnung ist. Aber `tableView:numberOfRowsInSection:` liefert 0, und das ist falsch. Wir wollen, dass stattdessen die Länge des Arrays ermittelt wird:

TableViews/MovieTable01/Classes/RootViewController.m

```

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [moviesArray count];
}

```

Damit wird der Tabellen-View mitgeteilt, dass der eine Abschnitt eine Zeile besitzt. Wenn die Anwendung nun läuft, wird sie die Methode `tableView:cellForRowAtIndexPath:` aufrufen, um eine `UITableViewCell` für diese eine Zeile zu erhalten. Das Template bietet auch hier eine Standardimplementierung an, die per Code ein Feld erzeugt. Wir müssen dieses Feld nur anpassen – direkt nach dem Kommentar `// Configure the cell` und noch vor `return cell;`.

Nun wollen wir herausfinden, welches Element im Array – es gibt im Moment nur eines, aber es wird später noch viele weitere geben – wir für den Feldinhalt nutzen wollen. Dafür können wir die Variable `indexPath` nutzen. Ein `NSIndexPath` ist ein Objekt, das einen Pfad durch eine Baumstruktur als Folge von (bei null beginnenden) Integer-Indexwerten beschreibt. Für das iPhone OS wurde diese Klasse um Eigenschaften erweitert, die speziell für `UITableViews` genutzt werden können: `section` und `row`. Mit anderen Worten – jedes Mal, wenn Sie `tableView:cellForRowAtIndexPath:` verwenden, enthalten `indexPath.section` und `indexPath.row` den Abschnitt und die Zeile, die angefragt wurden.

Direkt vor `return cell;` fügen wir also in der Implementierung folgenden Code ein:

TableViews/MovieTable01/Classes/RootViewController.m

```

Movie *aMovie = [moviesArray objectAtIndex:indexPath.row];
cell.textLabel.text = aMovie.title;

```

Damit wird über den Wert von `indexPath.row` zuerst das Element von `moviesArray` ausgelesen, das zur gewählten Zeile gehört. Dann müssen wir

nur noch den Titel von Movie im Feld anzeigen. Die `UITableViewCell` enthält zwei `UILabel` als Eigenschaften: `textLabel` und `detailTextLabel`. In diesem einfachen Fall setzen wir als Text von `textLabel` den Filmtitel ein.

Für eine einfache Tabelle ist damit alles getan. Klicken Sie auf **BUILD AND RUN**, und Sie werden eine einzeilige Tabelle wie in Abbildung 5.3 sehen.



Abbildung 5.3: Eine einfache UITableView

5.4 Tabellenfelder

Bisher haben wir eine Implementierung einer Data Source umgesetzt, die gerade einmal eine minimale Tabelle auf dem Bildschirm anzeigt. Aber wir können mit einer Tabelle noch viel mehr machen und da wollen wir mit den Tabellenfeldern beginnen. Unsere Klasse `Movie` enthält drei Member-Eigenschaften, aber wir zeigen nur eine davon in der Tabelle an. Lassen Sie uns die Felder besser ausnutzen.

Feldstile

Die generierte Implementierung von `tableView:cellForRowAtIndexPath:` erzeugt ein `UITableViewCell`-Objekt namens `cell`, das wir anpassen, bevor wir es an den Aufrufenden zurückgeben.

Das Standardfeld besitzt drei visuelle Eigenschaften, die wir nutzen können, um unsere Daten anzuzeigen: `textLabel`, `detailTextLabel` und `imageView`. In diesem Beispiel setzen wir den Text von `textLabel`, um die einfache Standarddarstellung zu erhalten. Hätte die Klasse `Movie` ein `UIImage`-Member (wie zum Beispiel einen Screenshot oder ein DVD-Cover), könnten wir die Eigenschaft `image` von `imageView` nutzen, um das Bild auf der linken Seite des Felds anzuzeigen.

Um das `detailTextLabel` nutzen zu können, müssen wir einen anderen Feldstil wählen. Die Idee der Stile ist im iPhone OS 3.0 neu dazugekommen. Es werden vier Stile angeboten:

- `UITableViewCellStyleDefault`: Gibt das `textLabel` linksbündig als schwarzen Text aus. Das `detailTextLabel` wird nicht genutzt. Dies ist der Standard und entspricht den Tabellenfeldern im iPhone 2.x.
- `UITableViewCellStyleSubtitle`: Ausgabe eines großen linksbündigen `textLabel` in Schwarz und auf einer zweiten Zeile darunter eines kleineren grauen `detailTextLabel`, wie in den iPod- oder Musikanwendungen.
- `UITableViewCellStyleValue1`: Anzeige eines großen linksbündigen `textLabel` in schwarz und eines etwas kleineren rechtsbündigen `detailTextLabel` in Blau. Dieses Layout ähnelt den Feldern in den Einstellungsoptionen und ist nur für eine Verwendung in Tabellen im Gruppenstil gedacht.
- `UITableViewCellStyleValue2`: Ausgabe eines kleinen rechtsbündigen `textLabel` in Blau auf der linken und eines kleinen linksbündigen `detailTextLabel` in Schwarz auf der rechten Seite – so wie bei den Kontakten. Auch dieser Button-ähnliche Stil passt eigentlich sinnvoll nur in gruppierten Tabellen.

In Abbildung 5.4 können wir diese vier Stile in einer modifizierten Version der Beispielanwendung sehen. Wir haben den Inhalt der Felder in Abhängigkeit von ihrem Stil geändert, da manche der Layouts für lange Strings nicht sinnvoll sind (insbesondere `UITableViewCellStyleValue2`, dessen linkes Label nach zehn Zeichen abgeschnitten wird). Da die „Value“-Stile eher für die Button-ähnliche Darstellung in gruppierten Tabellen gedacht sind, wurden die Felder im rechten Screenshot jeweils in einen eigenen Abschnitt gesteckt, während die Tabelle im linken Screenshot nur einen Abschnitt mit vier Zeilen enthält.



Abbildung 5.4: UITableView mit den vier vorhandenen UITableViewCellStyle im einfachen und im gruppierten Modus

Nutzen Sie Stile für Tabellenfelder, nicht CGRect

Vor dem iPhone SDK 3.0 war der für `UITableViewCell` genutzte Initialisierer `initWithFrame:reuseIdentifier:`, dem für das Argument `frame` ein `CGRect` übergeben wurde (normalerweise die Konstante `CGRectZero`, da es gar nicht verwendet wurde). Das Template für Navigationsanwendungen hat diesen Initialisierer aufgerufen, ebenso auch jeglicher andere Tabellencode. Aber seit dem iPhone SDK 3.0 ist dieser Initialisierer als veraltet gekennzeichnet, und es wird `initWithStyle:reuseIdentifier:` bevorzugt, der statt des `CGRect` eine der Stilkonstanten erwartet. Es ist wirklich einfach, alten Code zu konvertieren – man muss nur die aufgerufene Methode ändern und den Stil `UITableViewCellStyleDefault` nutzen.

Die angegebenen Stile bieten beim Anzeigen Ihrer Daten eine gewisse Flexibilität, um den Platz auf einem so kleinen Gerät wie dem iPhone sinnvoll zu nutzen. Wenn keiner dieser Stile – mit oder ohne optionale `imageView` – passt, schauen Sie sich Abschnitt 5.7, *Eigene Felder in Tabellen-Views*, auf Seite 104 an, in dem wir zeigen, wie man eigene Feldlayouts erzeugt.

Felder wiederverwenden

Neben einem Stil erwartet der Initializer für eine `UITableViewCell` auch den String `reuseIdentifier`. Es ist sehr wichtig, die Verwendung dieses Objekts zu verstehen, um ausreichend performante Tabellen zu erstellen. Glücklicherweise zeigt uns die Standardimplementierung von `tableView:cellForRowAtIndexPath:` schon, was diese Eigenschaft tut und wie man sie verwendet.

Eine `UITableView` speichert Felder zwischen, um sie später erneut verwenden zu können. Dadurch erhöht sich die Performance, weil bestehende Felder mehrfach eingesetzt werden und nicht immer neue erzeugt werden. Wenn ein Feld nach oben oder unten aus dem Anzeigebereich verschwindet, kann es wiederverwendet werden. Müssen Sie also in `tableView:cellForRowAtIndexPath:` ein Tabellenfeld erstellen, versuchen Sie zunächst, ein bestehendes aus dem Tabellencache zu holen. Wenn das klappt, setzen Sie einfach den Inhalt des Felds zurück. Nur wenn das nicht funktioniert (weil vermutlich keine zwischengespeicherten Felder verfügbar sind), erstellen Sie ein neues.²

Dies ist die Standardimplementierung in `tableView:cellForRowAtIndexPath:`³

TableViews/MovieTable01/Classes/RootViewController.m

```

1  static NSString *CellIdentifier = @"Cell";
2  UITableViewCell *cell =
3      [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
4  if (cell == nil) {
5      cell = [[UITableViewCell alloc]
6              initWithStyle:UITableViewCellStyleDefault
7              reuseIdentifier:CellIdentifier] autorelease];
8  }
```

In Zeile 1 wird ein *Cell Identifier* erzeugt, ein String, der die Art des Felds festlegt, das wir haben wollen. Die Idee dahinter ist, dass Sie bei unterschiedlichen Feldstilen in der gleichen Tabelle (entweder Standardstile oder selbst erstellte Layouts) ein Unterscheidungsmerkmal für den Tabellen-Cache haben, damit Sie ein Feld mit dem Stil zurückerhalten, den Sie suchen. Im Standardfall verwenden Sie nur einen Stil, daher reicht ein beliebig gewählter String wie "Cell" vollkommen aus.

² Der Cache lässt sich damit nur dann nutzen, wenn genug Daten vorhanden sind, um mit der Tabelle den ganzen Bildschirm zu füllen, und wenn der Anwender schon weit genug gescrollt hat, so dass eines oder mehrere Felder vom Bildschirm verschwunden sind.

³ Wir haben den Standardcode nur anders umbrochen, damit er in das Layout dieses Buchs passt.

In den Zeilen 2 bis 3 wird versucht, ein Feld per *Dequeing* zu erhalten, also aus dem Tabellen-Cache auszulesen, wobei die Kennung übergeben wird, um den richtigen Stil zu bekommen. Schlägt das fehl, wird ein neues Feld alloziert und initialisiert.

5.5 Tabellen bearbeiten

Bisher haben wir besprochen, wie man den Inhalt einer Tabelle bereitstellt und wie er dargestellt werden kann. Der nächste Schritt ist, die Tabelle editierbar zu machen. Die Tabelle soll also als Schnittstelle für das Bearbeiten des zugrunde liegenden Modells dienen. Löschen wir eine Zeile aus der Tabelle, soll auch das Objekt aus dem Modell entfernt werden. Wird dem Modell ein Objekt hinzugefügt, soll die Tabelle aktualisiert werden, damit sich diese Änderung dort widerspiegelt.

Lassen Sie uns mit dem Löschen beginnen, denn das ist einfacher. Tatsächlich enthält der auskommentierte Code aus dem Navigations-Anwendungs-Template schon die Grundlagen für das Löschen. Schauen wir uns `tableView:canEditRowAtIndexPath:` an. Die Standardimplementierung (und das Standardverhalten, wenn diese Methode von `UITableViewDataSource` gar nicht implementiert ist) ist, gar keine Zeile bearbeiten zu können. Entfernen Sie die Kommentare vor der Standardimplementierung und ändern Sie sie zu `return YES;`. Haben Sie das erledigt, werden Sie feststellen, dass Sie horizontal über eine Tabellenzeile wischen können, um einen Lösch-Button zu erhalten.

Um das Löschen zu ermöglichen, müssen wir `tableView:commitEditingStyle:forRowAtIndexPath:` implementieren. Die auskommentierte Implementierung enthält einen **if-then**-Block, um die Fälle zu behandeln, in denen der Editierstil `UITableViewCellEditingStyleDelete` und `UITableViewCellEditingStyleInsert` ist. Wir müssen nur den ersten unterstützen. Um ein Löschen umzusetzen, sind zwei Dinge nötig: Wir müssen das entsprechende Objekt aus dem Modell `moviesArray` entfernen und dann die `UITableView` aktualisieren. Für Letzteres stellt `UITableView` die Methode `deleteRowsAtIndexPaths:withRowAnimation:` bereit, die genau das ist, was wir brauchen. Ergänzen Sie die Standardimplementierung um die hervorgehobene Zeile und löschen Sie den **else**-Block für `UITableViewCellEditingStyleInsert`:

TableViews/MovieTable01/Classes/RootViewController.m

```

- (void)tableView:(UITableView *)tableView
    commitEditingStyle: (UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath {
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        // Die Zeile aus der Datenquelle löschen
        [moviesArray removeObjectAtIndex: indexPath.row];
        [tableView deleteRowsAtIndexPaths:
            [NSArray arrayWithObject:indexPath]
            withRowAnimation:UITableViewRowAnimationFade];
    }
}

```

Damit erhält man die Möglichkeit, per Streichen zu löschen, aber manche Anwender wissen gar nicht, dass das geht. Da dies jedoch eine Navigationsanwendung ist, gibt es zum Glück eine Navigationsleiste am oberen Rand des Bildschirms, in der sich ein Bearbeiten-Button wunderbar unterbringen lässt. Wie in anderen Anwendungen ist dessen Standardverhalten, einen ENTPERREN ZUM LÖSCHEN-Button auf der linken Seite jeder Tabellenzeile hinzuzufügen, die editiert werden darf. Wenn dieser angetippt wird, erhält man auf der rechten Seite einen Lösch-Button.

In der von Ihnen „entkommentierten“ Methode `viewDidLoad` ist Ihnen vielleicht der folgenden auskommentierte Code aufgefallen:

TableViews/MovieTable01/Classes/RootViewController.m

```

    // Uncomment the following line to display an Edit button in the
    // navigation bar for this view controller.
    // self.navigationItem.rightBarButtonItem = self.editButtonItem;

```

Sie erinnern sich an das in Abschnitt 5.2, *Tabellenbasierte Navigation einrichten*, auf Seite 85 Geschriebene, das erläutert, dass der RootViewController in `MainView.xib` so eingerichtet wurde, dass UINavigationController ein Kindelement ist. Dies steht für den blauen Balken oberhalb der Tabelle, der normalerweise zum Vor- und Zurückblättern und zum Bearbeiten von Tabellen genutzt wird. Es gibt zwei Eigenschaften, um Buttons in der Bar anzuzeigen: `leftBarButtonItem` und `rightBarButtonItem`. Und schauen Sie sich auf der rechten Seite der Zuweisung die Referenz auf `self.editButtonItem` an. Jeder UIViewController bietet diese Eigenschaft `editButtonItem`, die ein `UIBarButtonItem` zurückgibt, das die Methode `setEditing:animated:` des View-Controller aufruft und ihren Status zwischen BEARBEITEN und FERTIG wechselt.

Die auskommentierte Zeile ist schon fast alles, was wir wollen, nur dass der Button auf die linke Seite wandern soll, um die rechte Seite für einen HINZUFÜGEN-Button frei zu lassen, den wir später noch erstellen werden. Dies ist also die Zeile, die Sie in viewDidLoad brauchen:

```
TableViews/MovieTable01/Classes/RootViewController.m
```

```
self.navigationItem.leftBarButtonItem = self.editButtonItem;
```

Nach BUILD AND RUN sollten Sie nun auf den BEARBEITEN-Button tippen können, durch den der ENTPERREN ZUM LÖSCHEN-Button für alle Zeilen sichtbar wird. In Abbildung 5.5 sehen Sie die Tabelle im Bearbeitungsmodus (mit ein paar mehr Beispieldaten, so dass die Zeilen auch etwas enthalten).

Lokalisierung

Wenn Sie eine Anwendung entwickeln, die nicht englischsprachig ist, werden Sie feststellen, dass solche Standard-Buttons nicht unbedingt übersetzt sind – selbst bei einem deutschsprachig eingestellten iPhone steht auf dem EDIT-Button immer noch EDIT und nicht BEARBEITEN sowie DONE statt FERTIG. In Kapitel 29 wird noch kurz auf die Lokalisierung eingegangen, aber ein kleiner Hack sei schon hier verraten: Nehmen Sie sich eine .nib-Datei Ihres Projekts, rufen Sie im Kontextmenü GET INFO auf und klicken Sie im Tab GENERAL auf MAKE FILE LOCALIZABLE. Nun wechseln Sie wieder auf den GENERAL-Tab der lokalisierten Informationen zurück und klicken auf ADD LOCALIZATION. Tippen Sie im Pop-up „de“ (für deutsche Buttons) ein und klicken Sie auf ADD. Sie müssen gar keine echte Lokalisierung vornehmen, und es müssen auch nicht alle .xib-Dateien lokalisiert werden, aber mit diesem Schritt sollten auch die Standard-Buttons in Ihrer Sprache erscheinen.

5.6 In Tabellen navigieren

Als Nächstes soll der Benutzer eine neue Tabellenzeile einfügen können. Im vorherigen Kapitel haben wir einen MovieEditorViewController entwickelt, der wunderbar dafür geeignet ist, die Daten für ein neues Movie-Objekt einzugeben oder die eines bestehenden zu bearbeiten. Und nachdem das neue Movie-Objekt erstellt wurde, ist es ein Leichtes, es zum Modell hinzuzufügen und die Tabelle zu aktualisieren.

Wie zeigen wir nun das Eingabefenster an? Im vorherigen Kapitel haben wir die Methode `presentModalViewController:animated:` des `UIViewController` genutzt, um zur Edit View zu wechseln. In diesem Fall werden wir etwas Neues erlernen: wie man dafür die Navigationsobjekte nutzen kann. Wir haben das Projekt teilweise auch deshalb als navigationsbasierte Anwendung erstellt, weil es eine gute Ausgangsbasis für unsere Tabelle bietet und weil Navigation ein passender Begriff für den Wechsel zwischen unseren Anzeige- und Bearbeitungszuständen ist.

Die Navigation auf dem iPhone läuft über eine „Drill-Down“-Methode, die Sie vermutlich schon aus Mail, iPod/Musik und den Einstellungen kennen. Im SDK wird dies durch einen `UINavigationController` erledigt, der den Navigationszustand als Stack aus View-Controllern verwaltet. Immer dann, wenn Sie zu einer anderen View wechseln, packen Sie einen neuen `UIViewController` auf den Stack. Wenn Sie zurückgehen, holen Sie den aktuellen View-Controller vom Stack und wechseln zur vorherigen View. Die Navigation geschieht im Code, unabhängig davon, was auf dem Bildschirm zu sehen ist – immer wenn Sie durch das Antippen von Zeilen in einer Tabelle oder von Buttons in der Navigationsleiste navigieren, wird der Stack auf gleiche Weise verwaltet.



Abbildung 5.5: Verwenden des Standard-EditButtonItem, um Zeilen aus einer UITableView zu löschen

Den MovieEditorViewController hinzufügen

Um das auszuprobieren, wollen wir uns den `MovieEditorViewController` aus Sicht der Navigations-API vornehmen. Wir werden den Controller für zwei Dinge nutzen: zum Bearbeiten von Einträgen, die schon in der Tabelle vorhanden sind, und zum Erzeugen neuer Einträge.

Wie bei der Klasse `Movie` müssen Sie die Dateien `MovieEditorViewController.h` und `MovieEditorViewController.m` in den Ordner `Classes` Ihres Projekt kopieren und diese Kopien dann dem Xcode-Projekt bekannt machen. Kopieren Sie auch die Datei `MovieEditorViewController.xib` (wie zuvor mit `ADD → EXISTING FILES`) in die `Resources`-Gruppe des Projekts. In den vorherigen Beispielen wurde diese `Edit View` modal auf dem gesamten Bildschirm angezeigt. In dieser Anwendung ist sie ein Teil der Navigation, daher wird die Navigationsleiste oberhalb der View immer noch Platz einnehmen. Glücklicherweise können wir mit dem Interface Builder eine Navigationsleiste simulieren, um sicherzugehen, dass trotzdem noch alles auf die View passt. Öffnen Sie die `.nib`-Datei im IB, wählen Sie sie aus und lassen Sie den Property Inspector (-) anzeigen. Im Abschnitt „Simulated Interface Elements“ setzen Sie „Top Bar“ auf „Navigation Bar“, um zu sehen, wie die View als Teil der Navigation aussehen wird. In diesem Fall wird der `FER-TIG`-Button nicht vom Bildschirm verschwinden, aber vielleicht wollen Sie seine Position anpassen, so dass er innerhalb der vom IB angezeigten gestrichelten Linien liegt.

Um die Seite zum Bearbeiten anzuzeigen, muss unser `RootViewController` eine Instanz des `MovieEditorViewController` auf den Navigations-Stack schieben. Wir können den View-Controller per Code erzeugen, aber da wir überhaupt nur eine einzige Instanz benötigen, ist es sinnvoll, ihn im Interface Builder anzulegen. Der erste Schritt ist daher, in `RootViewController.h` ein `IBOutlet` zu erstellen. Fügen Sie innerhalb der geschweiften Klammern des `@interface`-Blocks eine Instanzvariable `MovieEditorViewController* movieEditor;` ein und deklarieren Sie dann nach der schließenden geschweiften Klammer die Eigenschaft als `Outlet`:

`TableViews/MovieTable01/Classes/RootViewController.h`

```
@property (nonatomic, retain) IBOutlet MovieEditorViewController
                                     *movieEditor;
```

Wie üblich müssen Sie für diese Eigenschaft in der `.m`-Datei per `@synthesize` Getter und Setter erstellen. Denken Sie auch daran, `#import "MovieEditorViewController.h"` im Header aufzunehmen.

Jetzt können Sie eine Instanz von `MovieEditorViewController` im Interface Builder erstellen. Öffnen Sie `RootViewController.xib` mit dem IB und ziehen Sie einen `UIViewController` aus der Library in das Dokumentenfenster des Nib. Wählen Sie diesen View-Controller aus und verwenden Sie den Identity Inspector (-), um dessen Klasse auf `MovieEditorViewController` zu setzen. Mit dem letzten Schritt wird dieses Objekt mit dem gerade erzeugten Outlet verbunden. Klicken Sie bei gedrückter -Taste oder mit der rechten Maustaste auf FILE'S OWNER (oder zeigen Sie dessen Connections Inspector mit - an) und ziehen Sie eine Verbindung vom `movieEditor` auf das gerade erstellte View-Controller-Objekt. Im IB sind wir nun erst einmal fertig, also sichern Sie Ihre Änderungen.

Ein bestehendes Tabellenelement bearbeiten

Zum Bearbeiten nutzen wir den `MovieEditorViewController`. Wenn der Anwender eine Zeile auswählt, navigieren wir zum Bearbeitungsbildschirm und laden den aktuellen Zustand des ausgewählten Movie-Objekts.

Dazu müssen wir zunächst auf das Selection-Event reagieren. Der `UITableViewDelegate` erhält dieses Event in der Delegate-Methode `tableView:didSelectRowAtIndexPath:`. Das Template für die Navigationsanwendung bietet eine auskommentierte Version davon im `RootViewController`, allerdings erzeugt dieser Beispielcode einen neuen View-Controller per Code. Das brauchen wir nicht, da wir den nächsten View-Controller schon haben – den `movieEditor`, den wir gerade im Interface Builder eingerichtet haben. Also müssen wir ihn nur vorbereiten und dann dorthin navigieren.

Deklarieren Sie in der Header-Datei eine Instanzvariable vom Typ `Movie*` mit dem Namen `editingMovie`. Sie merkt sich, welches Movie-Objekt bearbeitet wird, so dass wir wissen, was in der Tabelle aktualisiert werden muss, wenn wir dorthin navigieren. Haben Sie das erledigt, ist der Rest sehr einfach: sich merken, welcher Film bearbeitet wird, dem `MovieEditorViewController` mitteilen, was er anzeigen soll, und dann mit der Methode `pushViewController:animated:` des `UINavigationController` dorthin navigieren.

TableViews/MovieTable01/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    editingMovie = [moviesArray objectAtIndex:indexPath.row];
```

```

        movieEditor.movie = editingMovie;
        [self.navigationController pushViewController:movieEditor
         animated:YES];
    }

```

Am letzten Schritt ist besonders interessant, wie wir eine Referenz auf den Navigations-Controller erhalten haben, denn schließlich haben wir keine Instanzvariable oder Eigenschaft dafür definiert. Der Navigations-Controller wurde für uns in `MainWindow.xib` erzeugt, und wir haben ihn mit dem IB gar nicht angefasst. Der Trick ist die Eigenschaft `navigationController`, die von der Klasse `UIViewController` definiert und damit vom `RootViewController` geerbt wurde. Diese Eigenschaft (die auch als Instanzmethode aufrufbar ist) schaut in der Objekthierarchie nach, ob sie ein übergeordnetes Objekt findet, das ein `UINavigationController` ist. Dank dieser Methode müssen Sie niemals eine explizite Verbindung zu Ihrem Navigations-Controller herstellen. Ihre Root View-Controller und alle View-Controller, die damit auf den Navigations-Stack geschoben werden, können über diese Eigenschaft den Navigations-Controller erreichen, um damit vorwärts und rückwärts zu navigieren oder um die Navigationsleiste auf dem Bildschirm anzupassen.

Jetzt haben wir für die Bearbeitungs-View alles erledigt. Was bleibt, ist ein Weg zurück auf die Hauptseite. `MovieEditorViewController` enthält eine Methode `done`, die im IB mit dem FERTIG-Button verbunden ist,⁴ allerdings muss die Implementierung angepasst werden. Statt sich selbst als modalen View-Controller zu entsorgen, muss zurück zum vorigen View-Controller navigiert werden:

TableViews/MovieTable01/Classes/MovieEditorViewController.m

```

-(IBAction)done {
    [self.navigationController popViewControllerAnimated:YES];
}

```

Wie Sie sehen, kann der `MovieEditorViewController` auch die geerbte Eigenschaft `navigationController` verwenden, um den `UINavigationController` zu erhalten.

Nun können wir zur Bearbeitungs-View und wieder zurück navigieren, aber dann muss noch die Tabelle aktualisiert werden. Der `RootViewController` erhält bei einer Navigation zurück zu ihm den Callback `viewWillAppear:`, daher können wir dort dafür sorgen, dass die Tabellen-View aktualisiert wird:

4 Hätten wir nicht bereits einen FERTIG-Button in der View, wäre es sinnvoller, einen FERTIG- oder ZURÜCK-Button in der Navigationsleiste unterzubringen. Die Navigation im Beispiel in Kapitel 8, *Datei-I/O*, auf Seite 143 wird genau so vorgenommen.

TableViews/MovieTable01/Classes/RootViewController.m

```

- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    // Tabellen-View aktualisieren, wenn ein Film bearbeitet wurde
    if (editingMovie) {
        NSIndexPath *updatedPath = [NSIndexPath
            indexPathForRow:
                [moviesArray indexOfObject: editingMovie]
            inSection: 0];
        NSArray *updatedPaths =
            [NSArray arrayWithObject:updatedPath];
        [self.tableView reloadRowsAtIndexPaths:updatedPaths
            withRowAnimation:NO];
        editingMovie = nil;
    }
}

```

Zu Beginn unserer Update-Logik prüfen wir, ob ein Film bearbeitet wurde, da diese Methode auch zu anderen Zeitpunkten angerufen wird (zum Beispiel beim Starten der Anwendung). Wenn wir gerade einen Film editiert haben, müssen wir herausfinden, welche Tabellenzeile aktualisiert werden muss. Das erreichen wir, indem wir den Array-Index ermitteln, der `editingMovie` entspricht, dann einen `NSIndexPath` erzeugen, der im Abschnitt 0 der Tabelle diese Zeile anspringt, und den Pfad an die Methode `reloadRowsAtIndexPaths:withAnimation:` der Tabellen-View übergeben.

Ein Element zur Tabelle hinzufügen

Wir wollen auch neue Einträge in die Tabelle einfügen können. Das lässt sich erreichen, indem man dies als Spezialfall des Bearbeitens ansieht. Sobald der Anwender auf einen **+**-Button tippt, fügen wir in das Tabellenmodell still und leise ein leeres Movie ein, ergänzen eine Tabellenzeile und navigieren zur Bearbeitungs-View.

Weiter oben haben wir den `leftBarButtonItem` der Navigationsleiste für das `editButtonItem` genutzt, daher wollen wir den **+**-Button auf der rechten Seite der Navigationsleiste unterbringen. Wir können keinen **+**-Button vom `UIViewController` übernehmen, wie wir das mit dem BEARBEITEN-Button gemacht haben, daher werden wir selbst einen erzeugen.

Wechseln Sie zuerst zu `RootViewController.h` und erstellen Sie eine `IBAction`, um ein Event von dem Button zu behandeln, den wir gleich noch anlegen werden:

TableViews/MovieTable01/Classes/RootViewController.h

```

- (IBAction) handleAddTapped;

```

Da wir mit den Navigationsobjekten arbeiten müssen, die Xcode für uns angelegt hat, werden wir den Interface Builder nutzen, um die Datei `MainWindow.xib` zu öffnen, in der sie definiert sind. Wechseln Sie den Anzeigemodus im Nib-Dokumentenfenster auf die Listen- oder Spaltenansicht und klicken Sie dann doppelt auf das Objekt „Navigation Controller“. Dadurch erhalten Sie ein Fenster mit der Navigationsleiste im oberen und einem View-Platzhalter im unteren Bereich, der angibt, dass er von `RootViewController` geladen wurde. Vielleicht fällt Ihnen auf, dass der BEARBEITEN-Button auf der linken Seite der Navigationsleiste fehlt. Das liegt daran, dass wir ihn zur Laufzeit im Code hinzufügen.

Wechseln Sie zur Library und suchen Sie nach dem Symbol für das „Bar Button Item“. Dieser unterscheidet sich vom normalen „Round Rect Button“, daher achten Sie darauf, dass für die Klasse des genutzten Objekts `UIBarButtonItem` angegeben ist. Ziehen Sie den „Bar Button“ auf die rechte Seite der Navigationsleiste, wo er sich automatisch auf einem hervorgehobenen Zielfeld breitmacht. Damit wird er zum `rightBarButton-Item` der Navigationsleiste. Klicken Sie den „Bar Button“ an, öffnen Sie seinen Identity Inspector (-) und ändern Sie den Identifier in `ADD`. Damit wechselt sein Aussehen zu einem einfachen Pluszeichen (+).

Als Nächstes verbinden wir diesen Button mit der Methode `handleAddTapped`. Dabei gehen wir aber ein bisschen anders vor als bisher. Wenn Sie den Connections Inspector des Buttons öffnen (-) , sehen Sie nicht die übliche Auflistung von Touch-Events, wie zum Beispiel `Touch Up Inside`. Stattdessen gibt es eine einzelne Sent Action namens `selector`. Das liegt daran, dass das `UIBarButtonItem` eine andere Objekthierarchie als die normalen Buttons hat und `UIControl`, `UIView` und `UIResponder` nicht als Superklassen besitzt. Stattdessen hat dieses Objekt die Eigenschaften `target` und `selector`. Tippt man den Bar Button an, wird die von `selector` definierte Methode für das Objekt in `target` aufgerufen. Sie können beide Eigenschaften per Code setzen, aber da wir schon im Interface Builder sind, wollen wir es hier machen.

Um `Selector` und `Target` zu setzen, ziehen wir die Action `selector` aus dem Connections Inspector auf eines der anderen Objekte im Nib. Dieses Mal ziehen wir sie aber *nicht* auf den File's Owner. Da dies die `MainWindow.xib` ist, verweist das Proxyobjekt des File's Owner auf eine generische `UIApplication`. Die Methode `handleAddTapped`, die der Button aufrufen soll, ist in der Klasse `RootViewController` definiert, daher ziehen wir die Verbindung auf das Objekt des Root View-Controller im Nib-Fenster, wie es in Abbildung 5.6 zu sehen ist. Wenn Sie die Maustaste schließlich loslassen, werden die Namen der `IBAction`-Methoden des Ziels angezeigt, und Sie wählen die eine vorhandene aus: `handleAddTapped`.

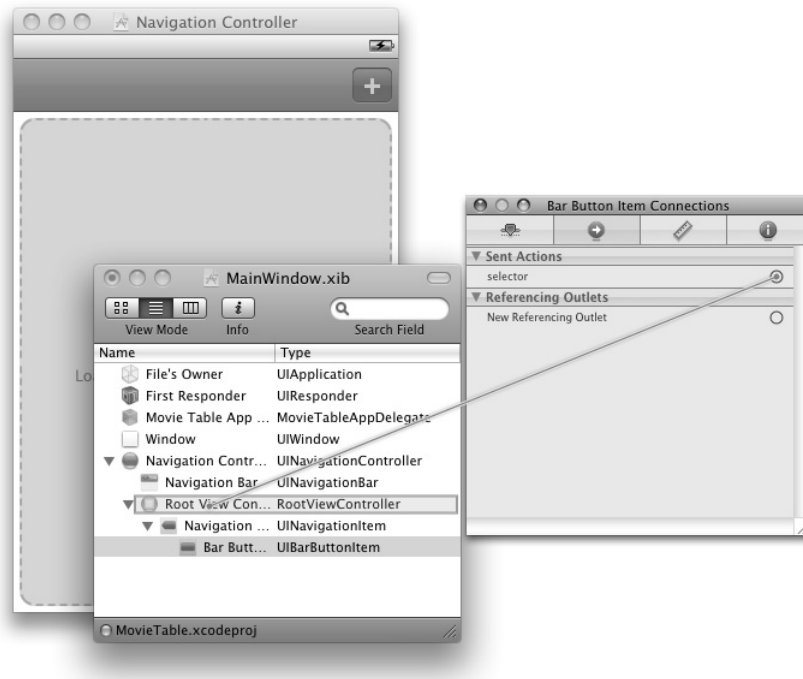


Abbildung 5.6: Den SELECTOR von UIBarButtonItem mit dem UINavigationController verbinden

Speichern Sie nun Ihre Änderungen im IB und kehren Sie zu Xcode zurück. Jetzt können wir die Methode `handleAddTapped` implementieren, die durch ein Antippen des **+**-Buttons aufgerufen wird:

TableViews/MovieTable01/Classes/RootViewController.m

```
-(IBAction) handleAddTapped {
    Movie *newMovie = [[Movie alloc] init];
    editingMovie = newMovie;
    movieEditor.movie = editingMovie;
    [self.navigationController pushViewController:
        movieEditor animated:YES];
    // UITableView (im Hintergrund) mit dem neuen Element
    // aktualisieren
    [moviesArray addObject: newMovie];
    NSIndexPath *newMoviePath =
        [NSIndexPath indexPathForRow:
            [moviesArray count]-1 inSection:0];
    NSArray *newMoviePaths =
        [NSArray arrayWithObject:newMoviePath];
    [self.tableView insertRowsAtIndexPaths:newMoviePaths
        withRowAnimation:NO];
    [newMovie release];
}
```

In dieser Methode wird zunächst ein leeres `Movie`-Objekt erzeugt, das dann als `editingMovie` gesetzt wird. Nun wird zum `MovieEditorView-Controller` navigiert – ähnlich wie beim Code für ein bestehendes `Movie`-Objekt. Anders als dort wird aber hier nach dem Navigieren die Tabellen-View wieder in Ordnung gebracht (während die Tabelle nicht dargestellt wird), indem das neue Objekt dem Modell-Array hinzugefügt und dann `insertRowsAtIndexPaths:withRowAnimation:` aufgerufen wird, um die Tabelle auf den neuesten Stand zu bringen. Das eingefügte `Movie`-Objekt hat leere Felder, aber wenn der Anwender aus der Bearbeitungs-View zurückkehrt, wird es in `viewWillAppear:` aktualisiert – genau wie beim Bearbeiten eines bestehenden Elements.

Fassen wir zusammen. Wir haben das Navigations-Anwendungs-Template genutzt, um eine Anwendung mit einer Tabellen-View einzurichten. Diese wird durch ein Modell unterstützt (ein einfaches `NSMutableArray`), um eine Liste von `Movie`-Objekten bereitzustellen. Nachdem wir uns die verschiedenen Tabellenfeldtypen angeschaut haben, fügten wir die Möglichkeit ein, Elemente aus der Tabelle zu entfernen – entweder über ein horizontales Wischen (durch Implementieren von `tableView:canEditRowAtIndexPath:`) oder über den BEARBEITEN-Button (durch das Ergänzen des Standard-`editButtonItem` und das Implementieren von `tableView:commitEditingStyle:forRowAtIndexPath:`). Dann haben wir uns angeschaut, wie man auf `UINavigationController` zugreifen kann, um zwischen verschiedenen View-Controllern zu navigieren, und den `MovieEditorViewController` genutzt, um ein `Movie`-Objekt zu bearbeiten, das durch die entsprechend ausgewählte Zeile in der Tabelle bestimmt war. Zum Schluss haben wir auch noch die Möglichkeit eingebaut, ein neues `Movie`-Objekt zu bearbeiten, indem man auf einen `+`-Button tippt.

5.7 Eigene Felder in Tabellen-Views

In Abschnitt 5.4, *Tabellenfelder*, auf Seite 90 haben wir die vier Feldstile vorgestellt, die das iPhone OS bietet. Auch wenn damit schon ein weiterer Anwendungsbereich abgedeckt ist, wollen Sie manchmal doch etwas anderes. Wenn Ihr GUI ein eigenes Farbschema nutzt, kann es gut sein, dass der standardmäßige schwarze oder blaue Text auf weißem Hintergrund nicht passt. Und auch wenn Sie mehr als zwei Labels brauchen, sind die verfügbaren Stile nicht so hilfreich.

Es ist mit ein bisschen Arbeit möglich, eigene Tabellenfelder im Interface Builder zu entwerfen und dann dafür zu sorgen, dass Ihre Tabelle diese Felder nutzt. In diesem Abschnitt werden wir mit dieser Technik eine Tabelle erstellen, die alle drei Felder des `Movie`-Objekts anzeigt.⁵

Ein eigenes Tabellenfeld entwerfen

Jede `UITableViewCell` besitzt eine `contentView`, so dass es möglich ist, per Code Subviews zu erzeugen und diese dann der View hinzuzufügen. Dafür gibt es auch schon Beispielcode von Apple. Das Problem ist dabei nur, dass Sie Position, Schrift, Größe und andere Eigenschaften jeder Subview im Code definieren müssen, ohne einen visuellen Editor zur Hand zu haben. Eine andere Variante ist, eine `UITableViewCell` in einer `.nib`-Datei zu erzeugen, die Subviews visuell hinzuzufügen und dann diese `.nib`-Datei zu laden, wenn die Tabelle ein neues Feld benötigt. Genau so werden wir auch vorgehen.

Wählen Sie in Xcode die Gruppe Resources aus und nutzen Sie FILE → NEW FILE, um eine neue Datei zu erstellen. Wählen Sie aus dem Abschnitt „iPhone OS“ den Eintrag „User Interface“ und erzeugen Sie eine leere `.nib`-Datei mit dem Namen `MovieTableCell.xib`. Öffnen Sie diese Datei im Interface Builder. Das Dokument wird nur die beiden üblichen Proxyobjekte enthalten: File's Owner und First Responder. Ziehen Sie aus der Library eine „Table View Cell“ in das Nib-Fenster. Bearbeiten Sie das Objekt durch einen Doppelklick. Es öffnet sich ein kleines Fenster mit der Größe eines normalen Tabellenfelds, das einen grauen Bereich für die *Content View* enthält.

Die Content View ist eigentlich nur eine visuelle Hilfe des IB, ein Platzhalter für die zur Laufzeit erzeugte View, die alle unsere Subviews enthält. Daher platzieren wir unsere UI-Elemente direkt darauf. Die Klasse `Movie` besitzt drei Felder, also nutzen wir drei Labels, um diese Felder in einem einzelnen Tabellenfeld anzeigen zu können. Dabei passen wir Schrift, Farbe, Größe und Layout entsprechend der „Wichtigkeit“ des Felds an. Ziehen Sie drei `UILabel` aus der Library in das Feld, wobei Sie die Positionierungshilfen und den Attributes Inspector (-) nutzen, um Position, Ränder, Farbe und Schrift anzupassen. Bei den Feldern im mitgelieferten Beispiel haben wir folgende Einstellungen verwendet:

- *Filmtitel*: Georgia 17 Punkt, gelber Text, links ausgerichtet nahe dem linken, oberen Rand des Felds
- *Umsätze*: Helvetica 17 Punkt, grüner Text, rechts ausgerichtet nahe dem rechten Rand
- *Zusammenfassung*: Helvetica 10 Punkt, hellblauer Text über die gesamte Breite am unteren Rand

5 Da wir im Projekt sehr viel ändern, um eigene Tabellenfelder zu nutzen, wurden die Codebeispiele zum Herunterladen für diese Übung in zwei Projekte aufgeteilt. Das vorherige Material finden Sie in `MovieTable01`, während das Projekt für eigene Felder in `MovieTable02` liegt.

Unser Felddesign im Interface Builder sehen Sie in Abbildung 5.7. Wir haben hellere Farben genutzt, weil wir vorhaben, für die Tabelle einen schwarzen Hintergrund zu verwenden. Dadurch wird es natürlich etwas schwieriger, den Text vor dem grauen Content-View-Platzhalter zu erkennen. Wir haben noch halbwegs sinnvolle Daten in jedes Feld eingetragen, um herauszufinden, wie viel Platz jeweils benötigt wird und wie das Ganze mit echten Daten aussieht. Speichern Sie `MovieTableViewCell.xib`, öffnen Sie dann `RootViewController.xib` und nutzen Sie den Attributes Inspector der Tabelle, um die Hintergrundfarbe auf schwarz zu setzen. Das müssen wir machen, weil Teile des Tabellenfelds transparent sind und wir eventuell nicht genug Felder haben, um die gesamte Tabelle zu füllen. Hier sollen die leeren Teile der Tabelle den gleichen Hintergrund wie die gefüllten Felder besitzen.



Abbildung 5.7: Entwerfen einer eigenen `UITableViewCell` im Interface Builder

Laden und Anzeigen eines eigenen Tabellenfelds

Wir haben jetzt ein eigenes Tabellenfeld, aber nutzen wir es in der Tabelle? Denn wir haben ja eventuell eine ganze Reihe von Tabellenfeldern. Normalerweise erzeugt die Tabelle per Code immer dann ein neues Feld, wenn wir keines aus dem Tabellen-Cache wiederverwenden konnten. Wollen wir das Feld aus dem Nib nutzen, müssen wir immer dann ein neues, eigenes Feld laden, wenn ansonsten ein neues Feld im Code erzeugt worden wäre.

Dafür gibt es einen interessanten Trick. Wir können das Nib manuell im Code laden und das Feld dort herausholen. Dazu erstellen wir eine `IBOutlet`-Eigenschaft in `RootViewController.h`, die die aus dem Nib geladene `UITableViewCell` speichert.

Jetzt bearbeiten Sie wieder die `MovieTableViewCell.xib` im Interface Builder. Markieren Sie `File's Owner`, zeigen Sie dessen Identity Inspector an ([⌘]-[4]) und ändern Sie die Klasse nach `RootViewController`. Jetzt sollten Sie zum Connections Inspector ([⌘]-[2]) wechseln und das Outlet `nibLoadedCell` mit dem Feldobjekt im Nib-Fenster verbinden kön-

nen. Im IB markieren Sie ebenfalls das Tabellenfeld, zeigen den Attributes Inspector an und setzen den Identifier (das erste Feld) auf `Cell`. Das ist der reuseIdentifier-String, den wir in Abschnitt 5.4, *Tabellenfelder*, auf Seite 90 genutzt haben.

TableViews/MovieTable02/Classes/RootViewController.h

```
@interface RootViewController : UITableViewController {
    // ... andere Instanzvariablen wurden hier weggelassen
    UITableViewCell *nibLoadedCell;
}

@property (nonatomic, retain) IBOutlet MovieEditorViewController *movieEditor;
@property (nonatomic, retain) IBOutlet UITableViewCell *nibLoadedCell;

-(IBAction) handleAddTapped;

@end
```

Das Geheimnis von File's Owner

Die Technik, ein eigenes Tabellenfeld aus einer .nib-Datei zu laden, sollte auch die Natur von File's Owner im Interface Builder klarstellen. Im IB ist File's Owner ein Proxyobjekt, das auf das Objekt verweist, dem die .nib-Datei „gehört“. Sie können die Klasse von File's Owner setzen, um auf die Outlets und Actions eben dieser Klasse zugreifen zu können, aber eigentlich stellen Sie damit nur implizit sicher, dass ein Objekt dieser Klasse der Besitzer der .nib-Datei sein wird, wenn sie geladen ist. Hier sehen Sie die andere Seite der Beziehung `loadNibNamed:owner:options:` lädt das Nib, wobei ein owner-Objekt angegeben wird. Alle Verbindungen zu File's Owner werden beim Laden des Nib zu oder von diesem Objekt hergestellt.

Nun zum überraschenden Teil. In `RootViewController.m` gehen Sie zur Methode `tableView:cellForRowAtIndexPath:` und schreiben den Block `if (cell==nil)` wie folgt um:

TableViews/MovieTable02/Classes/RootViewController.m

```
1 if (cell == nil) {
2     [[NSBundle mainBundle] loadNibNamed:@"MovieTableCell"
3         owner:self options:NULL];
4     cell = nibLoadedCell;
5 }
```

Damit wird das Tabellenfeld nicht mehr per Code erzeugt. Allerdings ist nicht ganz offensichtlich, wie `cell` seinen Wert erhält, denn der wichtigste Schritt geschieht implizit. In Zeile 2 laden wir das Nib `MovieTable-Cell`. Dies liefert ein `NSArray` mit dem Inhalt zurück, über das wir iterieren könnten, um das Objekt mit dem Tabellenfeld zu finden. Aber wir müssen es nicht, da wir ein `Outlet` dieses Felds für die Eigenschaft `nibLoadedCell` deklariert haben. Die `Outlets` werden durch das Laden des Nib verbunden. Wenn also `loadNibNamed:owner:options:` fertig ist, besitzt `nibLoadedCell` schon eine Referenz auf das aus dem Nib geladene eigene Feld, das wir dann in Zeile 4 der lokalen Variablen `cell` zuweisen.

Werte in einem eigenen Tabellenfeld zuweisen

Immer wenn ein neues Feld notwendig ist, wird `loadNibNamed:owner:options:` aufgerufen und ein neues Feldobjekt im Speicher erzeugt. So haben wir am Ende des `if` ein Feld (entweder aus dem Cache oder aus dem Nib geladen), das wir mit den Werten eines `Movie` aus dem Modell füllen müssen. Aber mit einem eigenen Feld können wir nicht mehr länger die Eigenschaften `titleLabel` oder `detailTextLabel` verwenden. Stattdessen brauchen wir einen Weg, auf die Subviews zuzugreifen, die wir im Interface Builder erstellt haben.

Eine Option wäre, eine eigene Unterklasse von `UITableViewCell` zu erzeugen, die `Outlets` in dieser Klasse zu deklarieren und zu verbinden und dann das Feld beim Laden auf diese Klasse zu casten. Der Nachteil ist hier, dass sehr viele Klassen zu schreiben sind – eine für jede Art von Tabellenfeld in Ihrer Anwendung. Daher nutzen wir eine etwas direktere Technik. Öffnen Sie das Feld im Interface Builder und markieren Sie das Titel-Label. Öffnen Sie den Attributes Inspector und scrollen Sie bis zum Feld mit dem Namen „Tag“. Das Tag ist ein einfacher, eindeutiger Integer-Wert, mit dem eine View in einer View-Hierarchie gefunden werden kann. Setzen Sie im Attributes Inspector das Tag des Titel-Labels auf 1, das für das Umsatz-Label auf 2 und das für die Zusammenfassung auf 3.

Jetzt können Sie in `tableView:cellForRowAtIndexPath:` den Text jedes Labels anpassen, indem Sie über die Feldmethode `viewWithTag:` darauf zugreifen.

TableViews/MovieTable02/Classes/RootViewController.m

```
// Anpassen des Felds.
Movie *aMovie = [moviesArray objectAtIndex:indexPath.row];
UILabel *titleLabel = (UILabel*) [cell viewWithTag:1];
titleLabel.text = aMovie.title;
UILabel *boxOfficeLabel = (UILabel*) [cell viewWithTag:2];
boxOfficeLabel.text = [NSString stringWithFormat: @"%d" ,
    [aMovie.boxOfficeGross intValue]];
UILabel *summaryLabel = (UILabel*) [cell viewWithTag:3];
summaryLabel.text = aMovie.summary;
return cell;
```

Jetzt sind wir so weit. Wir haben ein eigenes Felddesign in einem Nib und neuen Tabellencode zum Laden und Befüllen dieses Felds. Klicken Sie auf BUILD AND RUN, um eine Tabelle wie in Abbildung 5.8 zu erzeugen.

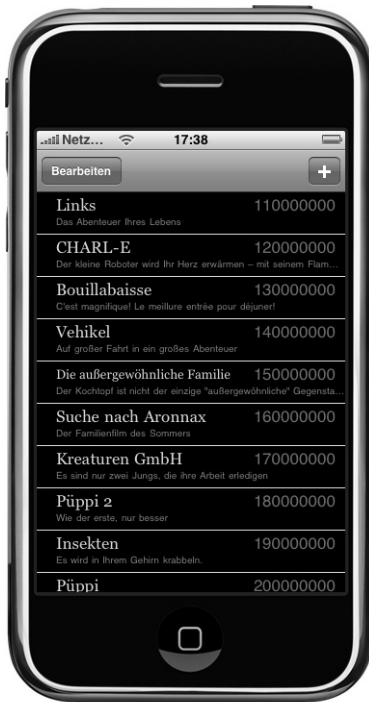


Abbildung 5.8: Eine UITableView mit selbst definierten Feldern

5.8 Tabellendaten sortieren

Häufig sollen Tabellendaten sortiert werden. Glücklicherweise stellen uns Cocoa und Objective-C einige Features bereit, die diese Aufgabe sehr vereinfachen.

Um die Daten sortieren zu können, fügen wir unserer Benutzerschnittstelle zunächst ein Sortier-Control hinzu.⁶ Öffnen Sie `MainWindow.xib` im Interface Builder und klicken Sie doppelt auf das Navigation Controller-Objekt, um es anzuzeigen. Ziehen Sie ein Segmented Control aus der Library in die Mitte der Navigationsleiste. Das `UISegmentedControl` ist ein sehr nützliches Control, das es dem Anwender ermöglicht, einen von mehreren vorgegebenen Werten auszuwählen. Obwohl das Control seine Größe automatisch an den geringen Platz in der Navigationsleiste anpasst, werden nicht viele Optionen hier hinpassen, daher wollen wir nur drei verwenden. Markieren Sie das Segmented Control und öffnen Sie den Attributes Inspector. Setzen Sie die Anzahl der Segmente auf 3 und nutzen Sie das Feld Title, um die Texte für die Segmente auf A-Z, Z-A und \$ zu setzen (oder welches Währungssymbol bei Ihnen am sinnvollsten ist).

Wir müssen auf dieses Segmented Control vom Code aus zugreifen können, also brauchen wir ein Outlet dafür. Deklarieren Sie in `RootViewController.h` die Instanzvariable `UISegmentedControl* sortControl`; und erstellen Sie mit den üblichen `@property`- und `@synthesize`-Anweisungen eine Eigenschaft. Sie müssen auch die folgende Methode deklarieren, die das Event verarbeitet, wenn der Anwender auf das Sortier-Control tippt:

```
TableViews/MovieTable03/Classes/RootViewController.h
```

```
-(IBAction) handleSortChanged;
```

Im IB sollten Sie nun in `MainWindow.xib` in der Lage sein, das Outlet `sortControl` des Root View-Controller mit dem Segmented Control und dessen Event *Value Changed* mit der Methode `handleSortChanged` zu verbinden.

Wir müssen das Array nicht nur als Reaktion auf eine Anforderung des Benutzers über das Segmented Control sortieren. Auch ein Hinzufügen oder Bearbeiten eines Elements macht ein erneutes Sortieren erforderlich – ebenso muss das Array sortiert werden, wenn die Anwendung gestartet wird. Lassen Sie uns also eine Methode `sortMoviesArray`

⁶ Auch für diesen Abschnitt sind die Änderungen am Projekt so groß, dass wir beim herunterladbaren Code ein eigenes Projekt verwenden. Die sortierbare Version finden Sie unter `MovieTable03`.

schreiben (auf die wir gleich eingehen). Jetzt können wir `handleSortChanged` recht einfach implementieren:

TableViews/MovieTable03/Classes/RootViewController.m

```
-(IBAction) handleSortChanged {
    [self sortMoviesArray];
    [self.tableView reloadData];
}
```

Immer wenn sich der Sortiertyp ändert, sortieren wir das Array und weisen die `UITableView` an, alle Daten neu zu laden. Das kann teuer sein, aber ein Sortieren kann eventuell jede Tabellenzeile betreffen, daher lohnt es sich nicht, einzelne Zeilen zu aktualisieren. Zudem müssen wir die beiden Codezeilen auch am Ende von `viewWillAppear:` ergänzen, um die Tabelle beim Starten der Anwendung sowie beim Bearbeiten und Hinzufügen eines Elements (erneut) zu sortieren.

Wie sortieren wir nun? Das ist ganz einfach. Die Methode `sortMoviesArray` muss in der Implementierung zu finden sein, bevor sie von jemandem aufgerufen wird (oder Sie fügen die Signatur der Methode in der Header-Datei ein, wodurch sie allerdings öffentlich sichtbar wird). Um das Sortieren durchzuführen, nutzen wir die Tatsache aus, dass das `NSArray` eine Reihe von Methoden bereitstellt, um eine sortierte Kopie eines Arrays zurückzugeben. `NSMutableArray` sortiert das veränderbare Array dabei selbst. Einige dieser Methoden erwarten Funktionszeiger oder Objective-C-Selektoren, wodurch Sie eine eigene Sortierfunktion schreiben können. Die einfachste Option ist aber, *Sortierdeskriptoren* zu nutzen.

Der `NSSortDescriptor` ist eine Klasse, die ein Sortierkriterium beschreibt. Sie besteht einfach aus einem Schlüssel und einem **BOOL**, das angibt, ob das Sortieren aufsteigend oder absteigend erfolgt. Dabei werden Schlüssel/Wert-Paare genutzt, um auf das zu sortierende Feld zuzugreifen. Der Schlüssel ist ein String, der einen *Schlüsselpfad* definiert – ein durch Punkte getrennter Pfad von per Getter lesbaren Eigenschaften eines Objekts. Für jeden Schritt in diesem Pfad wird das Pfadsegment ausgelesen, indem versucht wird, eine Eigenschaft, Zugriffsmethode oder Instanzvariable mit dem Schlüsselnamen anzusprechen. Der Sortierdeskriptor nutzt dann den Standardselektor `compare:`, der für viele Cocoa-Objekte definiert ist, um den eigentlichen Sortiervorgang durchzuführen.⁷

⁷ Wenn Ihre Eigenschaften eigene Klassen sind, die sich nicht mit `compare:` vergleichen lassen, können Sie den Sortierselektor ändern. Normalerweise sortieren Sie aber nur anhand von Cocoa-Klassen wie `NSString` und `NSNumber`, die auf `compare:` sinnvoll reagieren.

Unsere Movie-Objekte sind sehr einfach, sie bestehen nur aus drei Eigenschaften. Um alphabetisch nach dem Titel zu sortieren, erzeugen wir lediglich einen Sortierdeskriptor, dessen key den Wert title hat – den Namen der Eigenschaft.

Mit dieser Beschreibung im Hinterkopf sehen wir nun, wie einfach es ist, alle drei Sortierarten zu implementieren:

TableViews/MovieTable03/Classes/RootViewController.m

```
- (void) sortMoviesArray {
    NSSortDescriptor *sorter;
    switch (sortControl.selectedSegmentIndex) {
        case 0: // alphabetisch aufsteigend sortieren
            sorter = [[NSSortDescriptor alloc]
                initWithKey:@"title" ascending:YES];
            break;
        case 1: // alphabetisch absteigend sortieren
            sorter = [[NSSortDescriptor alloc]
                initWithKey:@"title" ascending:NO];
            break;
        case 2:
        default: // nach Umsatz sortieren
            sorter = [[NSSortDescriptor alloc]
                initWithKey:@"boxOfficeGross" ascending:YES];
            break;
    }
    NSArray *sortDescriptors = [NSArray arrayWithObject: sorter];
    [moviesArray sortUsingDescriptors:sortDescriptors];
    [sorter release];
}
```

In dieser Implementierung wird ein NSSortDescriptor so eingerichtet, dass er den Sortiertyp widerspiegelt. Dann landet er in einem Array, um von der Methode sortUsingDescriptors: des NSMutableArray genutzt werden zu können. Ein Array wird genutzt, um mehrere Deskriptoren verwenden und damit nach mehreren Feldern sortieren zu können. Wenn zwei Objekte laut dem ersten Deskriptor gleich sind, wird dann nach dem zweiten Deskriptor im Array sortiert (und so weiter).

Mit diesen Änderungen können wir jetzt die Tabelle sortieren. Sie werden bereits direkt nach dem Starten der Anwendung das Ergebnis sehen, da viewWillAppear: beim Starten sortMoviesArray aufruft (siehe Abbildung 5.8).



Abbildung 5.9: Sortieren der Tabelle anhand des Titels

Kapitel 6

Navigation

In diesem Kapitel werden wir uns anschauen, wie man eine Datenhierarchie mit einem Navigations-Controller verwaltet. Es ist wichtig, all das in Kapitel 5, *Tabellen-Views*, auf Seite 83 Gelernte im Hinterkopf zu behalten. Denn zum einen wurde dort eine Anwendung vorgestellt, die auf Basis des Navigation-Controller-Templates entstand, und zum anderen wird ein Großteil der Navigation zwischen unterschiedlichen Tabellen stattfinden. Sie müssen keine Tabellen-View nutzen, um einen Navigations-Controller sinnvoll einzusetzen, es ist nur deutlich sinnvoller, auf dem aufzubauen, was Sie schon kennen. Lassen Sie uns mit einem Beispiel beginnen.

6.1 Navigation in Mail

Die beste Möglichkeit, ein Gefühl dafür zu bekommen, was Sie mit einem Navigations-Controller machen können, ist ein Blick auf die Mailanwendung von Apple auf Ihrem iPhone. Am oberen Rand des Bildschirms, direkt unter der Systemstatusleiste (in der sich der Carrier und die verbundenen Netzwerke sowie andere Dinge befinden), sitzt die Navigationsleiste. In der Mailanwendung ist das der graublaue Bereich direkt unter der weißen Statusleiste. In der Mitte der Navigationsleiste finden Sie das Wort „Accounts“, wenn Sie mehrere E-Mail-Accounts konfiguriert haben, oder ansonsten „Postfächer“. Wir gehen hier davon aus, dass Sie nur einen Account haben. Darunter befindet sich eine Tabellen-View mit Ihren Postfächern.

Lassen Sie uns ein bisschen in ein Postfach hinein navigieren und dabei beobachten, wie die Navigationsleiste aussieht. Wählen Sie Ihren Posteingang. Die Navigationsleiste ändert sich in dreierlei Hinsicht.

Erstens gibt es einen neuen Button auf der linken Seite, der zurück zur View mit den Postfächern verweist, von der wir gerade kamen. Zweitens hat sich der Titel in der Mitte in „Posteingang“ geändert, zusammen mit der Anzahl der ungelesenen Nachrichten. Und drittens gibt es einen neuen Button BEARBEITEN auf der rechten Seite. Wenn Sie den antippen, wird Ihre Tabellen-View mit den Nachrichten in den Bearbeitungsmodus geschaltet, wodurch Sie rote Lösch-Links auf der linken Seite jedes Tabellenfelds erhalten. Der BEARBEITEN-Button hat sich in einen ABBRECHEN-Button verwandelt. Tippen Sie diesen an, verlassen Sie den Bearbeitungsmodus.

Wählen Sie eine Ihrer Nachrichten aus. Die Navigationsleiste ändert sich erneut. Der Zurück-Button, der auf der vorigen View mit „Postfächer“ beschriftet war, nennt sich nun „Posteingang“ und zeigt auch an, wie viele Nachrichten noch nicht gelesen wurden. Mit anderen Worten – der Zurück-Button zeigt den Titel der View an, auf der wir gerade waren. Auch der Titel der aktuellen View hat sich geändert. Dieses mal wird die Nummer der aktuellen Nachricht und die Gesamtanzahl aller Nachrichten angezeigt. Auf der rechten Seite der Navigationsleiste findet sich ein Segmented Control mit einem Pfeil nach oben und einem nach unten, mit dem Sie durch die Liste Ihrer Nachrichten blättern können.

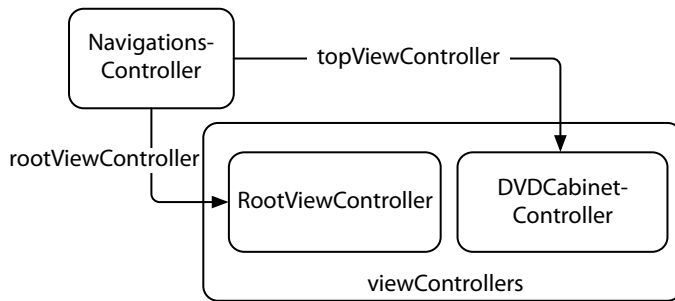
Der Titel und der Zurück-Button helfen dem Anwender gemeinsam, zu verstehen, wo er sich in der Anwendungshierarchie gerade befindet. Durch das Hinzufügen von Buttons für weitere Funktionen, wie zum Beispiel den BEARBEITEN-Button, kann der Anwender die am häufigsten benötigten Aufgaben für diese View schneller erreichen.

6.2 Der Navigations-Controller

Der Navigations-Controller kümmert sich um all die Funktionalität, die Sie brauchen, um eine Datenhierarchie zu verwalten. Das Coole daran ist, dass Sie sich mit dem Navigations-Controller ohne großen Aufwand von den allgemeinen zu den speziellen Informationen und auch wieder zurück bewegen können.

Ein Navigations-Controller verwaltet einen Stack mit View-Controllern. Jeder View-Controller steht für einen Knoten in der Hierarchie. Wenn sich der Benutzer zu den detaillierteren Informationen bewegt, wird jeweils ein weiterer View-Controller mit diesen Informationen auf den Stack befördert. Bewegt sich der Anwender zurück zu den allgemeineren Daten, wird der View-Controller vom Stack geholt, und der View-Controller für die allgemeineren Daten wird wieder der oberste Controller.

Hier ein Diagramm des View-Controller-Stacks für eine DVD-Verwaltungsanwendung. Im restlichen Kapitel werden wir diese Anwendung aufbauen, so dass wir sehen, wie die navigationsbasierten Anwendungen entstehen.

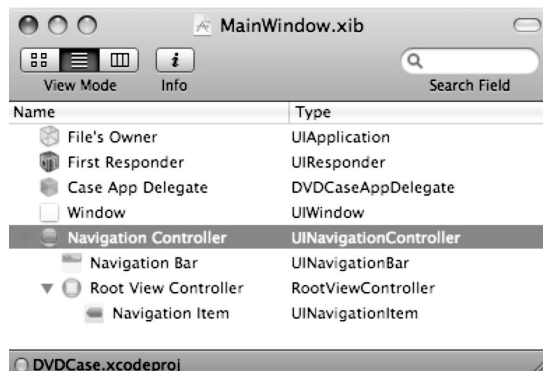


Dieser Navigations-Controller enthält im Stack `viewController` zwei View-Controller. Der auf der linken Seite ist der allgemeinste Controller, der auf der rechten der speziellste. In diesem Fall ist der allgemeinste Controller der `RootViewController` und der speziellste der `DVDCabinetController`.

6.3 Navigationsbasierte Anwendungen

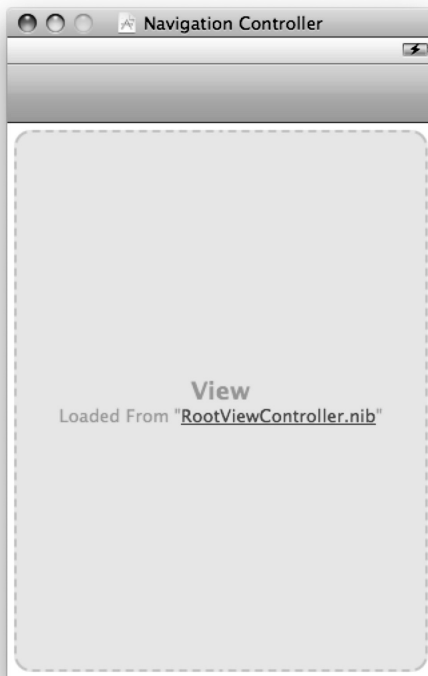
Starten Sie Xcode (wenn es nicht schon läuft) und erzeugen Sie eine neue, navigationsbasierte Anwendung namens `DVDCase`. Unser neues Projekt enthält eine Reihe von Klassen und `.nib`-Dateien. Wir werden zunächst die Datei `MainWindow.xib` unter die Lupe nehmen. Öffnen Sie sie mit einem Doppelklick. Diese `.nib`-Datei enthält die Grundlagen unserer Anwendung, daher wollen wir sie etwas genauer betrachten.

Ändern Sie den Anzeigemodus des Nib-Dokumentenfensters so, dass die Objekte als Liste angezeigt werden. Sie sollten nun in etwa die folgende Darstellung sehen:



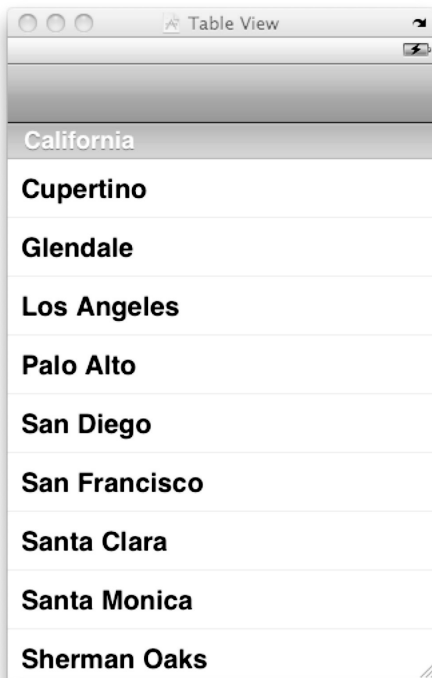
Markieren Sie den Navigations-Controller und klappen Sie ihn auf. Achten Sie auf die beiden Objekte unter dem Navigations-Controller. Das erste ist die Navigationsleiste (Navigation Bar), sie ist verantwortlich für die Darstellung der Kontextinformationen am oberen Rand des Bildschirms. Das zweite ist der Root Controller. Lassen Sie uns zunächst die Navigationsleiste anschauen.

Wenn Sie den Navigations-Controller doppelt anklicken, erhalten Sie ein Fenster wie das folgende:



Der Balken am oberen Rand des Fensters ist die Navigationsleiste. Dort finden sich die Kontextinformationen, die den Anwender darüber unterrichten, wo er sich gerade in der Datenhierarchie befindet. In der Mitte der Navigationsleiste findet sich der Titel des aktuell aktiven View-Controller. Wir können diesen Titel per Code setzen, aber auch im Interface Builder. Auf der linken Seite ist üblicherweise ein Zurück-Button, sofern wir uns nicht im Root Controller befinden. Auf der rechten Seite finden wir häufig einen Aktions-Button – meist einen BEARBEITEN-Button, mit dem wir die Tabellen-View in den Bearbeitungsmodus bringen.

Der Rest des Fensters wird vom Inhalt der `RootViewController.xib` eingenommen, wenn die Anwendung läuft. Bevor wir uns dem Code zuwenden, wollen wir noch einen Blick auf dessen `.nib`-Datei werfen. Öffnen Sie sie, indem Sie auf den blauen Link im Fenster klicken. Schauen Sie sich die View an, indem Sie im Nib-Dokument das Objekt mit dem Namen `Tabellen-View` doppelt anklicken. Sie sollten nun ein Bild sehen, das dem folgenden ähnelt. Beachten Sie, dass sich bei dieser View die Navigationsleiste am oberen Fensterrand befindet, damit wir sehen, wie die View aussieht, wenn die Anwendung läuft.



Beachten Sie ebenfalls, dass die Outlets `dataSource` und `delegate` der Tabellen-View auf `File's Owner` gesetzt sind. Damit kann die Tabellen-View ihre Daten erhalten und auf Events reagieren. Im zum Buch herunterladbaren Code für `RootViewController.m` wurden die Methoden `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath:` implementiert, aber sie sind hier auch noch einmal aufgeführt. Diese Methoden befüllen die initiale Tabellen-View mit Einträgen für „Zuhause“ und „Arbeit“. Mehr Informationen dazu erhalten Sie in Abschnitt 5.3, *Tabellendaten modellieren*, auf Seite 87.

Navigation/DVDCase/Classes/RootViewController.m

```

- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
    return 2;
}
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

static NSString *CellIdentifier = @"Cell" ;

UITableViewCell *cell =
    [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
if (cell == nil) {
    cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
        reuseIdentifier:CellIdentifier]
        autorelease];
}

cell.accessoryType = UITableViewCellAccessoryDisclosureIndicator;

switch (indexPath.row) {
    case 0:
        cell.textLabel.text = @"Zuhause" ;
        break;
    case 1:
        cell.textLabel.text = @"Arbeit" ;
        break;
    default:
        break;
}
return cell;
}

```

In diesem Code setzen wir den `accessoryType` von `cell` auf `UITableViewCellAccessoryDisclosureIndicator`. Damit wird dem Anwender angezeigt, dass durch das Auswählen dieser Zeile ein neuer View-Controller mit mehr Details geöffnet werden wird.

Der `accessoryType` kann eine von drei Optionen als Wert haben. In unserem Beispiel wollen wir den Disclosure Indicator nutzen. Die anderen beiden Optionen sind Detail Disclosure Button und Checkmark. Mit dem Detail Disclosure Button wird der blaue Kreis mit einem Pfeil angezeigt, der deutlich macht, dass der Anwender beim Antippen des Feldes zwei Möglichkeiten hat. Er kann das Feld antippen, um eine Aktivität auszulösen, oder den blauen Pfeil antippen, um zu den Details zu navigieren. Mit der Checkmark-Option kann ein Eintrag ausgewählt werden, ohne dass eine Navigation durchgeführt wird. Die von uns gewählte Variante schließlich sollte verwendet werden, wenn man durch ein Antippen der Zeile die Details des Elements aufrufen will.

Mit diesen beiden .nib-Dateien und dem Code für die Tabellen-View haben wir die Grundlagen einer Navigationsanwendung geschaffen. Nachdem wir nun die Konfiguration der .nib-Dateien betrachtet haben, wollen wir uns den Code für die Navigation anschauen.

6.4 View-Controller auf den Stack schieben

Als Nächstes müssen wir die eigentliche Navigation durchführen. Dazu muss nur die `tableView:didSelectRowAtIndexPath:` implementiert und ein neuer View-Controller auf den Stack des Navigations-Controller geschoben werden. So sieht der Code dazu aus:

Navigation/DVDCase/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if(0 == indexPath.row) {
        self.cabinetController.key = @"home";
        self.cabinetController.title = @"Zuhause";
    } else {
        self.cabinetController.key = @"work";
        self.cabinetController.title = @"Arbeit";
    }
    [self.navigationController pushViewController:self.cabinetController
                                     animated:YES];
}
```

Wir haben den `cabinetController` noch nicht erzeugt, das werden wir aber gleich nachholen. Dieser Controller wird für das Anzeigen des Inhalts der „Schränke“ in einer anderen Tabellen-View zuständig sein.

Das Setzen der Eigenschaft `key` in Abhängigkeit von der Zeile ist notwendig, um zu wissen, welche DVD-Liste angezeigt werden soll. Wir werden uns das Datenlayout noch anschauen, um zu verstehen, was hier passiert. Auch der Titel des View-Controller wird entweder auf „Zuhause“ oder „Arbeit“ gesetzt.

Abgesehen davon, den `cabinetController` anzeigebereit zu machen, damit er den Inhalt eines DVD-Schranks darstellen kann, brauchen wir nur eine einzige Codezeile, um einen neuen View-Controller auf den Stack zu schieben. Das ist ziemlich cool – ein Aufruf von `pushViewController:animated:` erledigt all die anstrengenden Dinge, die notwendig sind, um die View eines View-Controller zu aktivieren und ordentlich einzublenden.

In dieser Methode nutzen wir die Eigenschaft `cabinetController`, die dem Interface von `RootViewController` zusammen mit einer Instanzvariablen und einer `@synthesize`-Anweisung in der Implementierung hinzugefügt werden muss. Wir werden uns das noch detaillierter in Abschnitt 6.5, *Die Navigationsleiste anpassen* auf Seite 122 anschauen.

Die Methode `pushViewController:animated:` ist auch dafür verantwortlich, sich um die Navigationsleiste zu kümmern. Wenn der neue View-Controller auf den Stack geschoben wird, wird auch die Navigationsleiste mit dem `title` des neuen View-Controller und der `title` des Zurück-Button mit dem `title` des vorherigen View-Controller befüllt. Da der Titel des `RootViewController` auf *Schränke* gesetzt ist, wird das der Titel, der auf dem Zurück-Button zu sehen ist. Dies ist der Code, mit dem der Titel gesetzt wird:

Navigation/DVDCase/Classes/RootViewController.m

```
-(void)viewDidLoad {
    [super viewDidLoad];
    self.title = @"Schränke";
}
```

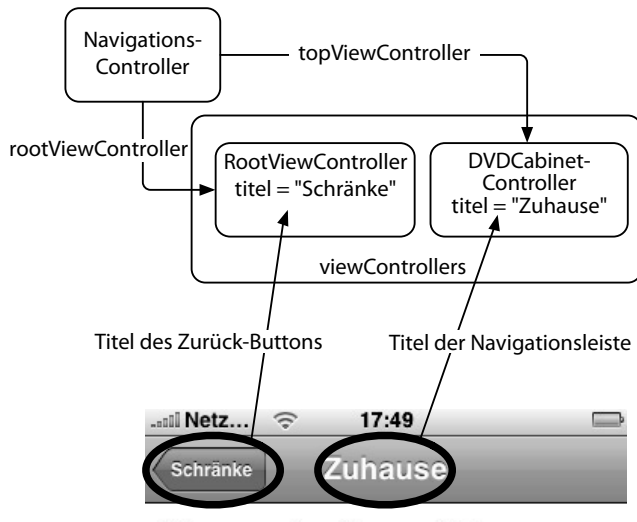
Als Nächstes müssen wir den View-Controller `cabinetController` erstellen, aber zuvor wollen wir uns noch ein bisschen über die Navigationsleiste und ihre Anpassungsmöglichkeiten unterhalten, denn wir können nicht nur am Titel drehen.

6.5 Die Navigationsleiste anpassen

Was in der Navigationsleiste angezeigt wird, können wir über die Eigenschaften unseres View-Controller beeinflussen.

Dieses Bild zeigt, wie der Stack mit View-Controllern den Titel der verschiedenen Elemente der Navigationsleiste setzt. Der `title` des `topViewController` findet sich in der Mitte als Titel der Navigationsleiste wieder. Der Titel des View-Controller direkt „unter“ dem `topViewController` im Stack steht dann auf dem Zurück-Button.

Der Titel in der Mitte der Navigationsleiste wird auf den Titel des aktuellen View-Controller gesetzt. Der rechte Button bleibt meist frei, wird aber auch gern für einen Bearbeiten-Button genutzt.



Der rechte Button kann durch eine eigene Instanz eines `UIBarButtonItem`-Item ersetzt werden. Mit einer eigenen Instanz des „Bar Button Item“ können Sie Ihr eigenes Verhalten für den Button definieren, indem Sie das Target und die Action angeben. Sie können sogar so weit gehen und die View zum Anzeigen des Buttons komplett ersetzen, indem Sie die Eigenschaft `customView` anpassen.

Sie haben beim Anpassen der Darstellung und des Verhaltens der Navigationsleiste in Ihrer Anwendung viele Möglichkeiten. In unserer DVD-Case-Anwendung setzen wir nur den Titel der View-Controller. In einer ausgefeilteren Anwendung würden wir die gesamte Navigationsleiste anpassen. Wir haben uns noch nicht über eigene Views unterhalten, aber das werden wir in Kapitel 19, *Zeichnen in eigenen Views*, auf Seite 415 tun. Im Prinzip können Sie das `UILabel`, das standardmäßig zum Anzeigen des Titels genutzt wird, durch eine eigene View ersetzen, die alles, was Sie wollen, im Titelbereich ausgibt.

Um ein Beispiel dafür zu sehen, öffnen Sie die YouTube-Anwendung und tippen auf Topvideos. Die Titel-View der Navigationsleiste wurde durch ein `Segmented Control` ersetzt. In Ihrer Anwendung können Sie ein beliebiges `UIControl` oder, wie schon gesagt, eine beliebige View nutzen, selbst eine komplett eigene.

Unser neuer View-Controller wird sich um die Liste der DVDs in einem Schrank kümmern. Also müssen wir einen neuen Tabellen-View-Controller und die dazugehörige Benutzerschnittstelle erzeugen.

Dazu sind folgende Schritte notwendig:

1. Erstellen einer neuen Subklasse von UITableViewController.
2. Erstellen einer neuen .nib-Datei, die das UI für unseren neuen Tabellen-View-Controller enthält.
3. Anpassen der .nib-Datei, um eine Tabellen-View darzustellen, und Verbinden der Tabellen-View mit unserem neuen View-Controller.
4. Hinzufügen eines Outlet zum RootViewController, so dass er über den neuen Tabellen-View-Controller Bescheid weiß.
5. Aktualisieren von RootViewController.xib, um dieses Outlet zu setzen.

Klicken Sie in Xcode mit der rechten Maustaste auf die Gruppe Classes und wählen Sie ADD NEW FILE. Erstellen Sie eine Subklasse von UITableViewController (wählen Sie COCOA TOUCH CLASSES OBJECTIVE-C CLASS und dann aus dem Auswahlménü UITableViewController), klicken Sie auf NEXT und geben Sie Ihrem neuen Controller den Namen DVDCabinetController. Jetzt müssen wir das neue Controller-Outlet dem RootViewController hinzufügen. Dies ist der angepasste Code für den Header. Denken Sie daran, die entsprechenden Import- und Synthesize-Anweisungen in der Implementierungsdatei einzufügen.

Navigation/DVDCase/Classes/RootViewController.h

```
@class DVDCabinetController;

@interface RootViewController : UITableViewController {
    DVDCabinetController *cabinetController;
}

@property (nonatomic, retain) IBOutlet DVDCabinetController
    *cabinetController;

@end
```

Nachdem wir nun ein Outlet haben, müssen wir es im IB verbinden. Öffnen Sie die Datei RootViewController.xib und fügen Sie einen neuen View-Controller hinzu. Ändern Sie dessen Klasse im Identity Inspector (-) zu DVDCabinetController. Jetzt stellen Sie die Verbindung von File's Owner zum neuen View-Controller her, indem Sie bei gedrückter -Taste vom File's Owner zum neuen View-Controller ziehen und das Outlet cabinetController auswählen. Setzen Sie den Nib-Name (Attributes Inspector, -) auf DVDCabinetController. Speichern Sie Ihre Arbeit.

Jetzt müssen wir die Datei `DVDCabinetController.xib` erstellen. Klicken Sie in Xcode bei gedrückter `[Ctrl]`-Taste auf die Gruppe Resources und wählen Sie `ADD NEW FILE`. Wählen Sie das Template „View XIB“, klicken Sie auf `NEXT` und geben Sie Ihrer neuen Datei den Namen `DVDCabinetController.xib`. Öffnen Sie diese Datei, um das UI entsprechend unseren Bedürfnissen anzupassen.

Die vorgegebene View in dieser Datei ist eine generische `UIView`, aber wir brauchen eine Tabellen-View. Also löschen wir das View-Objekt und ersetzen es durch eine `UITableView`. Als nächstes müssen wir die Klasse des File's Owner-Objekts auf `DVDCabinetController` setzen. Markieren Sie File's Owner, öffnen Sie den Identity Inspector (`[⌘]-[4]`) und ändern Sie die Klasse. Jetzt können wir die notwendigen Verbindungen herstellen. Ziehen Sie bei gedrückter `[Ctrl]`-Taste vom Outlet view des File's Owner-Objekts zum neuen Tabellen-View-Objekt und wählen Sie dort das Outlet view. Nun verbinden Sie `dataSource` und `delegate` von der Tabellen-View mit File's Owner. Bei jedem UI, das auf einer Tabellen-View basiert, muss man im Allgemeinen folgende Schritte durchführen: Erstellen einer neuen Subklasse von `UITableViewController`, Erstellen einer neuen „View XIB“-Datei, Auswechseln der View und Setzen von Data Source und Delegate.

Nachdem wir unser UI eingerichtet haben, müssen wir nun den Code für den `DVDCabinetController` erstellen. Dabei entspricht dieser Tabellen-View-Controller weitgehend allen anderen bisher gebauten. Die Klasse ist dafür verantwortlich, die Liste der DVDs im ausgewählten Regal anzuzeigen. Um die Daten darzustellen, müssen Sie die Methoden `tableView:numberOfRowsInSection:` und `tableView:cellForRowAtIndexPath:` implementieren, wie Sie das bisher auch schon getan haben.

Damit diese Klasse ihre Aufgaben erfüllen kann, benötigt sie zwei Instanzvariablen: einen Marker, um zu wissen, welches Regal ausgewählt wurde, und einen Container, der die Daten für jedes Regal enthält. Wir verwenden ein `NSString`-Objekt namens `key`, um den Marker zu speichern. In einem `NSDictionary` legen wir die Daten ab. Denken Sie daran, dass wir die Eigenschaft `key` in der Methode `tableView:didSelectRowAtIndexPath:` des `RootViewController` setzen, wenn der Anwender eine der Zeilen „Zuhause“ oder „Arbeit“ auswählt. `data` ist rein intern, daher benötigen wir keine Eigenschaft dafür. Vergessen Sie nicht, für die `key`-Eigenschaft per `@synthesize` Getter und Setter zu erzeugen.

Dies ist der Quellcode für die Header-Datei von DVDCabinetController:

Navigation/DVDCase/Classes/DVDCabinetController.h

```
@interface DVDCabinetController : UITableViewController {
    NSDictionary *data;
    NSString *key;
}

@property(n nonatomic, retain) NSString *key;

@end
```

Hier finden Sie die Deklaration eines Dictionary, in dem sich die Daten befinden. Wir werden gleich auch in der Methode `viewDidLoad` das Initialisieren dieser Daten vornehmen. Die zweite Instanzvariable ist jetzt interessanter. Dieser Schlüssel ist – wenn Sie sich an die Implementierung von `tableView:didSelectRowAtIndexPath:` im `RootViewController` erinnern – unser Weg, um anzugeben, welcher Schrank ausgewählt wurde. Wir werden sehen, wie er verwendet wird, wenn wir uns um die Daten kümmern. Dies ist der Code für das Initialisieren des `data`-Objekts:

Navigation/DVDCase/Classes/DVDCabinetController.m

```
-(void)viewDidLoad {
    [super viewDidLoad];
    NSArray *keys = [NSArray arrayWithObjects:@"home" ,@"work" , nil];
    NSArray *homeDVDs =
        [NSArray arrayWithObjects:@"Thomas der Baumeister" , nil];
    NSArray *workDVDs =
        [NSArray arrayWithObjects:@"Einführung in Blender" , nil];
    NSArray *values =
        [NSArray arrayWithObjects:homeDVDs, workDVDs, nil];
    data = [[NSDictionary alloc] initWithObjects:values forKeys:keys];
}
```

Dieser Code erzeugt ein Dictionary mit den beiden Schlüsselwerten, die wir schon in der Implementierung des `RootViewController` verwendet haben. Auch wenn es besser wäre, hier eine symbolische Konstante für die Schlüssel zu verwenden, war hier das Ziel, den Code möglichst gut lesbar zu halten. In einem umfangreicheren Projekt sollten Sie also besser dafür sorgen, dass Sie Konstanten und keine hartkodierten Werte nutzen.

Jetzt wissen wir, wie das Datenlayout aussieht.

Nun wollen wir uns die Implementierung der Methode `tableView:numberOfRowsInSection:` vornehmen:

Navigation/DVDCase/Classes/DVDCabinetController.m

```
- (NSInteger)tableView:(UITableView *)tableView
  numberOfRowsInSection:(NSInteger)section {
    return [[data valueForKey:self.key] count];
}
```

Wir müssen nur das Array für den entsprechenden DVD-Schrank holen und die Anzahl der Einträge zurückgeben. Wenn der Anwender in der ersten Tabellen-View auf „Zuhause“ tippt, wird der Schlüssel auf `home` gesetzt, und wir erhalten damit das korrekte Array. Das gilt ebenso für den Schlüssel `work`. Hier der Code für `tableView:cellForRowAtIndexPath::`

Navigation/DVDCase/Classes/DVDCabinetController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView
  cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier]
            autorelease];
    }

    cell.textLabel.text = [[data valueForKey:self.key]
        objectAtIndex:indexPath.row];

    return cell;
}
```

Auch hier holen wir wieder das korrekte Array aus dem Daten-Dictionary, um den Text für das Feld auszulesen.

Mit einem letzten Codeabschnitt stellen wir sicher, dass unsere Tabellen-View immer die richtigen Informationen anzeigt. Dies ist der Code für die Methode `viewWillAppear::`

Navigation/DVDCase/Classes/DVDCabinetController.m

```
- (void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}
```

In Abschnitt 4.3, *Den Controller um Outlets und Actions ergänzen*, auf Seite 64 haben wir beschrieben, dass die Methode `viewWillAppear:` immer dann aufgerufen wird, wenn die vom Controller betreute View sichtbar werden soll. Das ist genau die richtige Stelle, um die Daten neu zu laden, damit der Anwender beim Wechsel zwischen „Zuhause“ und „Arbeit“ immer die aktuelle Filmliste vorfindet. Ein Aufruf von `reloadData` sorgt dafür, dass die Tabellen-View den gesamten Inhalt neu lädt. Wie wir schon gesagt haben, kann das ineffizient werden, aber mit diesen wenigen Beispieldaten ist das kein Problem.

Dies ist ein sehr häufig genutztes Muster, um zwei View-Controller (VC) miteinander zu verbinden. Der Eltern-VC kennt den Kind-VC durch eine Verbindung, die im Interface Builder (IB) hergestellt wurde. Der Eltern-VC setzt einen Wert für den Kind-VC, der diesen dann als Schlüssel nutzt, um die anzuzeigenden Daten zu finden. Wenn Sie eine Zweiwegekommunikation benötigen (also auch vom Kind- zum Eltern-VC), können Sie diese Verbindung im IB ebenfalls herstellen oder den File's Owner verwenden.

6.6 View-Controller vom Stack holen

Schließlich wollen wir uns noch anschauen, wie man den Kind-VC vom Stack holt, wenn der Anwender fertig ist. Bei unserer Beispielanwendung werden wir den Kind-VC einfach verwerfen. Hier sehen Sie die Implementierung der Methode `tableView:didSelectRowAtIndexPath:` in unserer Klasse `DVDCabinetController`:

Navigation/DVDCase/Classes/DVDCabinetController.m

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [self.navigationController popViewControllerAnimated:YES];
}
```

Wir rufen die Methode `popViewControllerAnimated:` auf, um unseren View-Controller zu verwerfen. Da die View-Controller in einem Stack geordnet sind, wird der Navigations-Controller die View des `DVDCabinetController` durch die des `RootViewController` ersetzen. Bauen Sie mit BUILD AND RUN die Anwendung und bewundern Sie Ihr Werk.

Wenn Sie eine Anwendung mit einer umfangreichen Hierarchie aufbauen und Ihren Anwendern die Möglichkeit geben wollen, jederzeit direkt an den Anfang der Hierarchie zu gelangen, können Sie das durch einen Aufruf der Methode `popToRootViewControllerAnimated:` erreichen. Außerdem können Sie per `popToViewController:animated:` auch zu einem beliebigen anderen View-Controller in der Hierarchie wechseln.

Die Navigation ist ein elementarer Bestandteil vieler iPhone-Anwendungen, da die realen Daten oft hierarchisch organisiert sind. Bauen Sie eine Anwendung, die mit solchen Daten arbeitet, wird die Arbeit mit dem Navigations-Controller ganz natürlich erscheinen.

Kapitel 7

Zwischen Bereichen wechseln: Tab-Controller

Sie haben View-Controller genutzt, damit die Anwender mehr von Ihrer Anwendung haben. Die Tabellen-View und ihre Freunde helfen Ihnen dabei, Daten in Listen anzuzeigen. Der Navigations-Controller hilft den Anwendern, von den allgemeineren zu den spezifischeren Daten zu wechseln. In diesem Kapitel werden wir den Tab-Controller nutzen, um unsere Anwendung in Funktionsbereiche zu unterteilen.

7.1 Wann man Tabs verwendet

Das Problem ist immer das gleiche – die Benutzerschnittstelle muss so organisiert sein, dass Ihre Anwender nur genau das sehen, was sie auch brauchen. Aber wenn Sie mehrere unterschiedliche Interaktionsbereiche oder verschiedene „Sichtweisen“ auf die gleichen Daten haben, funktioniert das Navigationsparadigma nicht mehr. Schließlich ist die Navigation dazu gedacht, mit jeder neuen View von den allgemeineren zu den spezifischeren Informationen zu gelangen. Wenn Ihre Benutzerschnittstelle nicht in dieses Navigationsparadigma passt, sollten Sie sich den Tab-Controller anschauen.

Der Tab-Controller ist insbesondere dann nützlich, wenn man eine Reihe unterschiedlicher Funktionsbereiche organisieren oder gruppieren will. Jeder Bereich hat sein eigenes TabBar-Element und ist nur dann aktiv, wenn der Benutzer dieses Element antippt. Immer wenn Ihre Anwender auf unterschiedlichen Wegen mit den Informationen arbeiten, kann ein Tab-Controller dabei helfen, Ihre Benutzerschnitt-

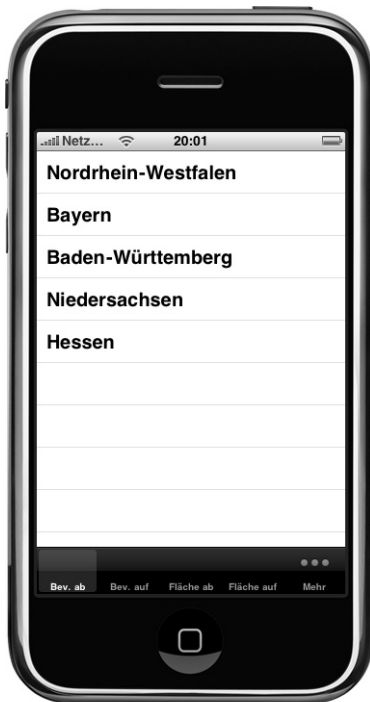
stelle so zu organisieren, dass sich der Anwender auf die Aufgabe konzentrieren kann, die er gerade erledigen will.

Jeder View-Controller erhält seinen eigenen Tab am unteren Rand des Bildschirms. Tippt der Anwender auf den Tab, wird der View-Controller aktiv, der mit diesem Tab verbunden ist. Beim Navigations-Controller sieht das anders aus. Tippt der Anwender auf einen Tabelleneintrag, wird ein neuer View-Controller aktiviert, der detailliertere Informationen über das Element anzeigt.

Ein Tab-Controller zeigt dagegen keine detaillierteren Informationen an, wenn der Anwender auf einen anderen Tab tippt – sondern andere.

Denken Sie an die Uhranwendung, die ein tolles Beispiel für eine Anwendung ist, bei der jeder Tab andere Informationen und andere Funktionen beinhaltet. Wenn Sie zwischen Weltuhr und Wecker wechseln, erwarten Sie keine vertiefenden Informationen, sondern ganz andere Daten.

Die App, die wir in diesem Kapitel erstellen werden, wird sechs verschiedene View-Controller enthalten, die unsere Daten in einer jeweils anderen Sortierreihenfolge enthalten. Hier ein Screenshot mit fünf Bundesländern als Daten:



Wenn der Anwender auf die verschiedenen Elemente in der Tab-Leiste unten tippt, wird jeweils eine andere Sortierung erreicht. Lassen Sie uns also eine Anwendung mit einem Tab-Controller aufbauen.

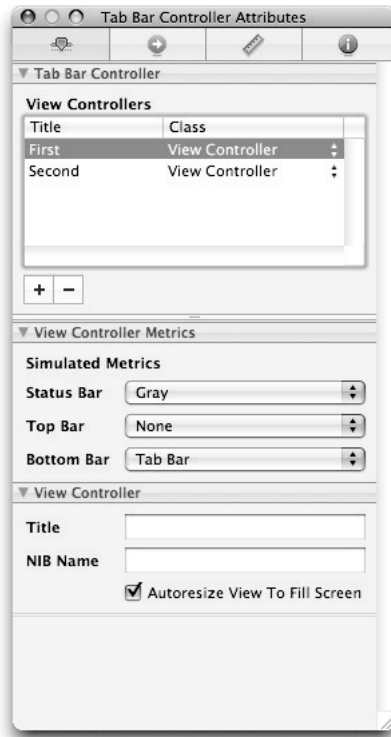
Wenn Sie mit einem Abschnitt Probleme haben, schauen Sie noch einmal in eins der vorherigen Kapitel, um nachzulesen, was Sie genau tun müssen.

7.2 Einen Tab-Controller erstellen

Die erste Beispielanwendung wird zwei verschiedene Möglichkeiten anbieten, die Daten zu sortieren. Da es zwei Wege dafür gibt, nutzen wir auch zwei View-Controller. Der erste View-Controller wird die Bundesländer nach ihrer Bevölkerungsanzahl sortieren, der zweite nach Größe.

Wie für die anderen View-Controller, mit denen Sie schon gearbeitet haben, bietet Xcode auch ein Template für Tab-gesteuerte Anwendungen, durch das schon eine ganze Menge Ressourcen für Sie erzeugt werden. Erstellen Sie in Xcode ein neues Projekt mit dem Template „Tab Bar Application“ und geben Sie ihm den Namen States. Dann klicken Sie auf den Button SAVE.

Dieses neue Projekt lässt sich am besten verstehen, wenn wir uns die Details von zwei Objekten anschauen, die vom Template für uns erzeugt wurden. Öffnen Sie die Datei `MainWindow.xib`, indem Sie sie in Xcode doppelt anklicken. Das Dokumentenfenster sollte fünf Objekte enthalten. Das für uns hier interessante nennt sich `Tab Bar Controller`. Markieren Sie den `Tab Bar Controller` und öffnen Sie seinen `Attributes Inspector` (-). Sie sollten in etwa folgendes Fenster erhalten:



Der Attributes Inspector führt die View-Controller auf, die der Tab-Controller enthält. Sie können diese Liste anpassen, indem Sie View-Controller hinzufügen, entfernen oder umsortieren. Dieser Tab-Controller besitzt zwei Controller mit den Namen First und Second, um die er sich kümmert. Es handelt sich um UINavigationController (also nicht etwas Spezifischeres wie beispielsweise einen Navigations-Controller). Sie werden diese beiden View-Controller bald durch Ihre eigenen View-Controller austauschen, in denen die Bundesländer sortiert sind.

Das nächste Objekt, das wir uns anschauen wollen, ist der States App Delegate. Öffnen Sie dessen Connections Inspector, indem Sie es markieren und - drücken. Dieses Objekt besitzt zwei Outlets, tabBar-Controller und window. Der Anwendungs-Delegate wird informiert, wenn die Anwendung mit dem Laden der Haupt-.nib-Datei fertig ist und nun die Ereignisverarbeitung starten will. Wir nutzen diesen Delegate-Hook, um die Initialisierung abzuschließen und das UI auf dem Bildschirm darzustellen. Der Delegate fügt dem window die vom tabBar-Controller definierte View hinzu. Hier ist der Code:


```
TabBarControllers/States/Classes/StatesAppDelegate.m
```

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    [window addSubview:tabBarController.view];
    [self createData];
}
```

Dieser Code ist recht klar, wir fragen den Tab-Controller nach seiner view und definieren sie dann als Subview von window. Sie haben so etwas in so gut wie jedem Beispiel mit View-Controllern gesehen, interessant ist hier allerdings, woher der Tab-Controller die View erhält. Statt eines Controller-Stacks wie beim Navigations-Controller besitzt der Tab-Controller eine Liste mit Controllern und einen expliziten „ausgewählten Controller“. Die View, die der Tab-Controller zurückgibt, ist die View des `selectedController`. Wir erstellen zudem mit einem Aufruf der Methode `createData` die Daten für unsere Anwendung. Diese Methode werden wir gleich noch betrachten.

Nachdem wir nun die Grundlagen einer einfachen Anwendung mit einem Tab-Controller beleuchtet haben, wollen wir sie mit unseren View-Controllern anpassen, um die Daten unterschiedlich zu sortieren.

7.3 View-Controller in Tab-Controllern

In diesem Beispiel benötigen wir zwei View-Controller – einen für eine Liste der fünf bevölkerungsstärksten Bundesländer und den anderen für die fünf flächenmäßig größten Länder. Die Daten für dieses Beispiel sind im Anwendungs-Delegate fest verdrahtet. In einer echten Anwendung würden Sie eine Reihe von Modellklassen erstellen, die Ihre Daten repräsentieren, und die Daten selbst in einer Datenbank oder Datei speichern (wir bevorzugen eine Datenbank). Für dieses einfache Beispiel verwenden wir aber nur `NSDictionary`-Objekte, um die Daten zu halten. Dies ist der Code für die Daten aus Nordrhein-Westfalen, ausgeschnitten aus der Methode `createData`. Die Daten für die restlichen Bundesländer werden genauso erzeugt und dem Array `data` hinzugefügt.¹

```
TabBarControllers/States/Classes/StatesAppDelegate.m
```

```
NSMutableArray *data = [NSMutableArray array];

[data addObject:[NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithInt:17933064], @"population",
    @"Nordrhein-Westfalen", @"name",
    [NSNumber numberWithInt:34088], @"area", nil]];
```

¹ Nutzen Sie den herunterladbaren Code, da die Reihenfolge wichtig ist.

Nachdem Sie alle Testdaten erzeugt haben, müssen Sie die Eigenschaft `states` auf dieses Array setzen. Der Code sollte so aussehen:

```
TabBarController/States/Classes/StatesAppDelegate.m
```

```
self.states = [NSArray arrayWithArray:data];
```

Vergessen Sie nicht, dem Interface `StatesAppDelegate` die Instanzvariable und Eigenschaft `states` hinzuzufügen und die Eigenschaft in der Implementierung der Klasse `StatesAppDelegate` zu synthetisieren.

Wir erstellen zwei Methoden, um diese Daten auszulesen – einmal nach Bevölkerung sortiert und einmal nach Fläche. Die erste Methode ist:

```
TabBarController/States/Classes/StatesAppDelegate.m
```

```
- (NSArray *)statesByPopulation {
    NSRange range = NSMakeRange(0, 5);
    return [self.states subarrayWithRange:range]; }
```

Da unsere Daten absteigend nach der Bevölkerung erstellt wurden, müssen wir hier nicht viel mehr machen, als sie auf die ersten fünf Bundesländer einzuschränken. Die nächste Methode ist ein bisschen interessanter – sie liefert die Daten nach Fläche sortiert:

```
TabBarController/States/Classes/StatesAppDelegate.m
```

```
- (NSArray *)statesByArea {
    NSSortDescriptor *sorter = [[[NSSortDescriptor alloc]
                                initWithKey:@"area" ascending:NO] autorelease];
    NSArray *sorted = [self.states sortedArrayUsingDescriptors:
                        [NSArray arrayWithObject:sorter]];
    NSRange range = NSMakeRange(0, 5);
    return [sorted subarrayWithRange:range]; }
```

In dieser Methode müssen wir die Daten mithilfe eines `NSSortDescriptor` neu sortieren. Ein Sort Descriptor ermöglicht es uns, eine Eigenschaft und die Sortierrichtung anzugeben, nach der sortiert werden soll. Anschließend rufen wir die Methode `sortedArrayUsingDescriptors:` für unser Bundesland-Array auf und erhalten ein Array zurück, das nach Fläche statt nach Bevölkerungsanzahl sortiert ist. Nun bilden wir einfach eine Untermenge des Arrays, um die ersten fünf Elemente zu erhalten.

Nachdem wir nun unsere Daten haben, müssen wir sie in die entsprechenden Tabellen-Views übertragen. Die beiden View-Controller, die wir aufbauen werden, besitzen im Prinzip den gleichen Code. Sie unterscheiden sich nur in der Methode, die zum Ermitteln der anzuzeigenden Bundesländer aufgerufen wird. Wir müssen dann zwei Klassen erstellen, die wir `ByPopulationViewController` und `ByAreaViewController` nennen. Markieren Sie in Xcode die Gruppe `Classes`, klicken Sie sie bei

gedrückter `[Ctrl]`-Taste an und wählen Sie **ADD NEW FILE OBJECTIVE-C CLASS**. Im Auswahlmenü **SUBCLASS OF** entscheiden Sie sich für „**UITableViewController**“. Dann klicken Sie auf **Next**, geben den Namen **ByPopulationViewController** an und klicken auf **FINISH**. Wir müssen dieser Klasse drei Methoden hinzufügen. Die erste holt die Bundeslanddaten aus dem Anwendungs-Delegate:

TabBarController/States/Classes/ByPopulationViewController.m

```
- (NSArray *)states {
    return [(StatesAppDelegate *)
        [[UIApplication sharedApplication] delegate]
        statesByPopulation];
}
```

Dieser View-Controller wird die Daten sortiert nach Bevölkerungsanzahl zurückgeben, daher ruft er die Methode `statesByPopulation` auf. Denken Sie bitte daran, dass wir das hier nur deshalb so schlicht halten, um uns auf die Tab-Controller konzentrieren zu können. In einem ausgefeilteren Beispiel würden wir eher *Core Data* (siehe Kapitel 11, *Core Data*, auf Seite 225) nutzen, um die Daten zu verwalten, statt sie aus dem Anwendungs-Delegate zu holen.

Als Nächstes müssen wir die Daten der Tabellen-View zur Verfügung stellen (die wir in Kürze erstellen und anpassen werden). In Abschnitt 5.3, *Tabellendaten modellieren*, auf Seite 87 haben Sie erfahren, dass Sie die Zeilenanzahl für einen Abschnitt bereitstellen und dann jedes Feld in diesem Abschnitt füllen müssen. Der Code dazu sieht so aus:

TabBarController/States/Classes/ByPopulationViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section {
    return self.states.count;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if(nil == cell) {
        cell = [[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }
    cell.textLabel.text = [[self.states objectAtIndex:indexPath.row]
        objectForKey:@"name" ];
    return cell;
}
```

Der Code selbst ist recht gradlinig gehalten, und viele seiner Elemente haben wir schon in Kapitel 5, *Tabellen-Views*, auf Seite 83 kennengelernt. Der einzige neue Abschnitt enthält die Verwendung der Methode `objectForKey:`, die einfach über die Angabe des Schlüssels ein Objekt aus dem Dictionary holt.

Jetzt müssen wir diesen View-Controller im Interface Builder anpassen. Wenn die Datei noch nicht geöffnet ist, klicken Sie `MainWindow.xib` in Xcode doppelt an, um den IB zu starten und die Datei zu laden. Wechseln Sie im Dokumentenfenster nun in den Listenmodus.

So können Sie nun zwei View-Controller sehen, die Teil dieses Tab-Controller sind. Dieses Fenster ist für Tab-Controller sehr gut geeignet, da Sie dort die View-Controller auswählen und bearbeiten können, die der Tab-Controller enthält.

Wählen Sie zum Beispiel den ersten View-Controller aus und zeigen Sie dann per - seinen Identity Inspector an. Setzen Sie den Klassennamen auf `ByPopulationViewController`, öffnen Sie per - den Attributes Inspector und ändern Sie den Titel in „Nach Einwohnern“. Jetzt haben wir unseren View-Controller zur Anzeige nach Einwohnern erstellt und angepasst.

Wiederholen Sie diese Schritte für den View-Controller zur Anzeige nach Fläche, wobei Sie den Aufruf von `statesByPopulation` durch den von `statesByArea` ersetzen müssen. Nehmen Sie zudem im IB die gleichen Anpassungen vor wie für den ersten View-Controller. Allerdings wird die View dieses zweiten Controllers aus einer anderen `.nib`-Datei geladen werden (mehr zu Demonstrationszwecken denn aus echter Notwendigkeit).

Wenn Sie fertig sind, sollten Sie in etwa folgendes Bild vor sich haben:



Bevor wir nun die Schnittstelle für den View-Controller zur Anzeige nach Fläche erstellen, wollen wir erst noch die Tabellen-View für den Controller zur Anzeige nach Bevölkerungsanzahl hinzufügen. Löschen Sie im IB die View, die mit der Klasse `ByPopulationViewController` verbunden ist, indem Sie sie markieren und dann auf **Entf** drücken. Lassen Sie als Nächstes per **⌘-⌘-L** die Library anzeigen und wählen Sie aus der oberen Liste Data Views. Markieren Sie dann eine Tabellen-View und setzen Sie sie als Kindelement des `ByPopulationViewController` ein.² Speichern Sie Ihre Arbeit. Nun wollen wir die Schnittstelle für den `ByAreaViewController` erstellen.

Wählen Sie im IB den zweiten View-Controller für den Tab-Controller aus und setzen Sie dann im Attributes Inspector (**⌘-1**) den Dateinamen der .nib-Datei auf `ByAreaView`.³

Jetzt müssen wir die .nib-Datei erzeugen. Klicken Sie in Xcode bei gedrückter **Ctrl**-Taste auf die Gruppe Resources und wählen Sie NEW FILE. Markieren Sie in der Liste auf der linken Seite des Wizards „User Interfaces“, auf der rechten Seite „View XIB“ und klicken Sie dann auf NEXT. Geben Sie den Namen mit `ByAreaView` an und klicken Sie auf FINISH.

Öffnen Sie Ihre neue .nib-Datei und ändern Sie die Klasse des File's Owner-Objekts in `ByAreaViewController`. Anschließend müssen wir das Gleiche wie für den `ByPopulationViewController` durchführen. Löschen Sie zunächst die bestehende View und fügen Sie dann der .nib-Datei eine Tabellen-View hinzu. Danach verbinden Sie das View-Outlet des `ByAreaViewController` mit der Tabellen-View. Schließlich müssen Sie noch die Outlets `dataSource` und `delegate` der Tabellen-View mit File's Owner verbinden.⁴ Sichern Sie Ihre Arbeit.

Ihr Projekt ist jetzt fast fertig. Zurück in `MainWindow.xib`, müssen wir den Titel für beide View-Controller setzen. Dieser Titel wird gebraucht, wenn der Tab-Controller die Titel für die Buttons benötigt. Markieren Sie also den Tab-Controller und drücken Sie dann **⌘-1**. Klicken Sie jeden Titel in der Liste der View-Controller doppelt an und ändern Sie sie in „Nach Bevölkerung“ beziehungsweise „Nach Fläche“.

² Indem Sie die Tabellen-View als Kindelement des `ByPopulationViewController` einsetzen, wird diese View automatisch auch die View des Controllers, daher müssen wir keine Verbindungen herstellen.

³ Das Template-Projekt erzeugt eine .nib-Datei namens `SecondView.xib`. Wir könnten diese Datei umbenennen und wiederverwenden, stattdessen werden wir sie aber ignorieren. Sie können die Datei auch löschen, da das Projekt sie nach dem Erstellen der neuen .nib-Datei nicht mehr benötigt.

⁴ Hier müssen wir die Verbindungen herstellen, da wir die View in einer eigenen .nib-Datei bearbeiten.

Lassen Sie die Anwendung nun laufen. Der erste Button sollte ausgewählt und die Liste der Bundesländer absteigend nach Bevölkerungszahl sortiert sein. Klicken Sie auf die Liste „Nach Fläche“, sollten Sie die fünf flächenmäßig größten Länder angezeigt bekommen.

Diese Anwendung ist mit zwei Tabs ziemlich einfach. Häufig wollen Sie aber mehr Tabs haben. Das Gute ist, dass der `UITabBarController` die Entwicklung solcher Anwendungen stark vereinfacht. Lassen Sie uns dies als Nächstes anschauen.

7.4 Viele Controller

Das Tolle an Tab-Controllern ist, dass sie uns so viel Arbeit abnehmen. Vieles geschieht automatisch. Wenn Sie mehr als fünf Controller angeben, wird ganz von allein ein Button MEHR in den Tab eingefügt. Tippt der Anwender auf diesen Button, zeigt der Tab-Controller einen Navigations-Controller an, der die restlichen Controller zusammen mit einem BEARBEITEN-Button aufführt. Tippt der Anwender wiederum auf diesen BEARBEITEN-Button, kann er die Liste mit Controllern so sortieren, dass die für ihn wichtigsten Tabs in der eigentlichen Tab-Leiste erscheinen und die anderen in den Navigations-Controller wandern.

Tab-Controller machen all das standardmäßig für Sie. Wenn Sie View-Controller zur Liste der Controller hinzufügen (entweder im Interface Builder oder per Code), weiß der Tab-Controller natürlich, ob es fünf oder mehr Controller gibt und dass er dann diese neue Funktionalität aufrufen muss. Der Tab-Controller ermöglicht es uns zudem, zu bestimmen, was er tun soll. Lassen Sie uns zunächst anschauen, wie man kontrollieren kann, welche View-Controller neu angeordnet werden können.

Die Eigenschaft `customizableViewControllers` dient dem Tab-Controller dazu, zu ermitteln, was neu angeordnet werden darf. Befindet sich ein View-Controller nicht in dieser Liste, erlaubt es der Tab-Controller nicht, seine Position in der Liste zu verändern. Standardmäßig finden sich alle View-Controller, die der Tab-Controller verwaltet, auch in dieser Liste. Wenn Sie also nicht wollen, dass bestimmte Controller nicht verschoben werden, müssen Sie sie dort entfernen. Lassen Sie uns ein Beispiel anschauen.

Im vorigen Abschnitt hatten wir nur zwei Controller, die eine Liste mit fünf Bundesländern in unterschiedlicher Sortierreihenfolge ausgaben. In diesem Beispiel erweitern wir diese Liste auf sechs verschiedene Sortierungen: nach Name, nach Bevölkerungsanzahl oder nach Flächengröße. Jede Sortierung kann aufsteigend oder absteigend sein, so dass schließlich sechs Varianten herauskommen. Bei echten Anwendungen würden Sie natürlich kein solches UI erstellen, aber so ist dieses Beispiel nur eine kleine Erweiterung des vorherigen, und wir müssen uns nicht in die Untiefen des UI-Designs begeben.

Sie haben einen Screenshot der Anwendung im Simulator schon am Anfang dieses Kapitels gesehen. Die Elemente am unteren Rand des Bildschirms befinden sich dabei in der im Interface Builder definierten Reihenfolge.

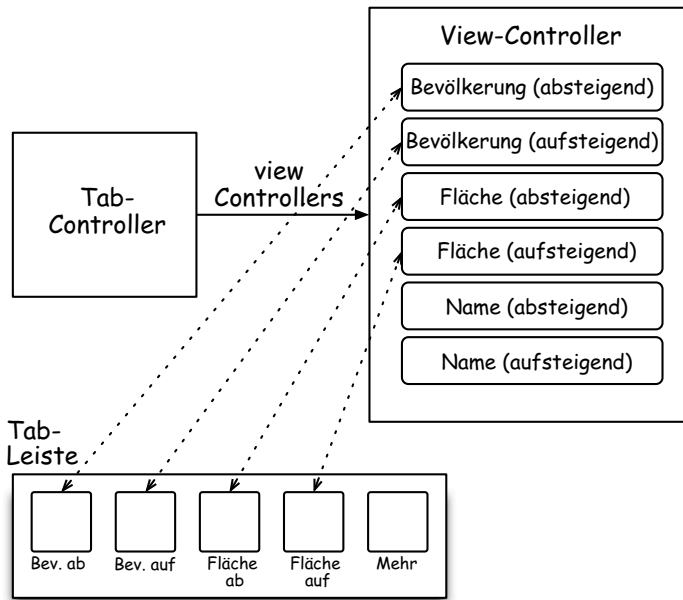
Der Code entspricht weitgehend dem aus dem vorigen Beispiel. Für jeden der zusätzlichen Sortier-Controller wurde eine Methode hinzugefügt, und jeder der Controller ruft die zu ihm passende Methode auf, um die Daten zu erhalten. Hier ein Beispiel für eine der Sortiermethoden:

TabBarController/StatesMore/Classes/StatesAppDelegate.m

```
- (NSArray *)statesAscendingByName {
    NSSortDescriptor *sorter = [[NSSortDescriptor alloc]
                                initWithKey:@"name" ascending:YES autorelease];
    NSArray *sorted = [self.states sortedArrayUsingDescriptors:
                        [NSArray arrayWithObject:sorter]];
    NSRange range = NSMakeRange(0, 5);
    return [sorted subarrayWithRange:range];
}
```

Dieser Code ist den Methoden, die wir weiter oben gesehen haben, sehr ähnlich. Er wird vom Controller `ByNameAscendingViewController` aufgerufen. Der Code für diesen Controller wiederum ähnelt stark dem des Controllers aus dem vorigen Beispiel, daher werden wir hier nicht weiter ins Detail gehen.

Der interessante Teil ist die Art und Weise, wie der Tab-Controller mit den zusätzlichen Controllern umgeht, die nicht auf die Tab-Leiste passen. Wenn Sie mehr als fünf Controller haben, gibt der Tab-Controller die ersten vier aus und ersetzt den fünften durch den MEHR-Button. So sieht die Konfiguration für unser Beispiel aus:



Der Tab-Controller macht das automatisch, indem er das Protokoll `UITabBarControllerDelegate` implementiert und der Delegate der Tab-Leiste wird. Dieses Protokoll enthält eine ganze Reihe von Methoden, aber prinzipiell entscheidet schlicht der Delegate, was geschehen soll, wenn der Mehr-Button angetippt wird. Der Tab-Controller entscheidet, einen Navigations-Controller mit der Liste der zusätzlichen View-Controller auszugeben. Toll daran ist nicht nur, dass das automatisch passiert, sondern auch, dass Sie selbst so etwas ebenfalls erreichen können, indem Sie Ihre eigene Implementierung des `UITabBarControllerDelegate` schreiben. Sie können aber auch etwas komplett anderes machen – zum Beispiel ein Symbol für jeden View-Controller ausgeben, statt den Titel anzuzeigen.

Die gleichen grundlegenden Prinzipien gelten für das Bearbeiten der Reihenfolge der View-Controller. Der Delegate der Tab-Leiste entscheidet, was geschehen soll, und da der Tab-Controller der Standard-Delegate ist, zeigt er eine modale View mit den Symbolen und Titeln der Controller an. Aber auch hier können Sie mit einer eigenen Implementierung etwas ganz anderes anzeigen lassen.

Herzlichen Glückwunsch. Sie haben jetzt alle grundlegenden View-Controller kennengelernt, die Cocoa Touch zu bieten hat. Mit den zur Verfügung stehenden Tools können Sie nun jede Benutzerschnittstelle erzeugen, die Sie auf dem iPhone benötigen. Als Nächstes wollen wir uns mit einigen der Low-Level-APIs befassen, mit denen man auf das Dateisystem des iPhone zugreifen kann.

Kapitel 8

Datei-I/O

Als Entwickler von iPhone-Anwendungen vertrauen Ihnen die Anwender ihre Daten an. Das heißt, Sie müssen diese Daten regelmäßig und häufig genug auf Medien speichern (oder „persistieren“), damit sie nach dem Ausschalten nicht einfach verloren sind. Denn schließlich kann Ihr Anwendungs-Delegate jederzeit die Nachricht `applicationWillTerminate:` erhalten. Dann wollen Sie den Programmmzustand vor dem Beenden der Anwendung zügig speichern und beim nächsten Starten auch wieder laden können. Zudem ist Ihre Anwendung dafür verantwortlich, die Einstellungen des Anwenders zu verwalten und zu berücksichtigen. Und in manchen Fällen ist es aus Gründen der Geschwindigkeit sinnvoll, die Daten einer Anwendung in einer Datenbank zu speichern.

In diesem Kapitel werden wir uns das Dateisystem anschauen. Das ist das wichtigste System, um Daten langfristig zu sichern. Es wird häufig die erste Wahl für die Datenablage sein, aber denken Sie daran, dass es auch noch ein paar andere Systeme gibt, die Sie stattdessen wählen können. Ihre Anwendung kann die vom System angebotenen Möglichkeiten zum Sichern und Laden von Einstellungen nutzen (siehe Kapitel 9, *Einstellungen*, Seite 181), oder Sie entscheiden sich für die Performance einer relationalen Datenbank – entweder über die SQLite3-API (Kapitel 10, *Die Datenbank SQLite*, Seite 205) oder per Core Data (Kapitel 11, *Core Data*, Seite 225). Es gibt noch zwei weitere APIs für eine langfristige Speicherung, aber sie unterscheiden sich von den bisher genannten, weil sie Daten mit anderen Anwendungen gemeinsam nutzen: die Image-APIs (behandelt in Kapitel 20, *Bilder und Fotos zeichnen*, Seite 427) und die Address Book-API (behandelt in Kapitel 23, *Adressbuch*, Seite 471).

Das iPhone OS sitzt auf einem Dateisystem, das mehr oder weniger identisch mit dem auf einem Mac ist. Dateien und Verzeichnisse existieren in der von Ihnen erwarteten Art und Weise – mit Namen, Hierarchien, Attributen im POSIX-Stil und so weiter.¹ Wie Sie in diesem Kapitel sehen werden, haben Sie auf das Dateisystem einen Zugriff, der dem in Cocoa für den Mac entspricht. Sie können durch die Verzeichnishierarchien navigieren, Dateien lesen und schreiben, Verzeichnisse erstellen und löschen, Attribute auslesen und vieles mehr.

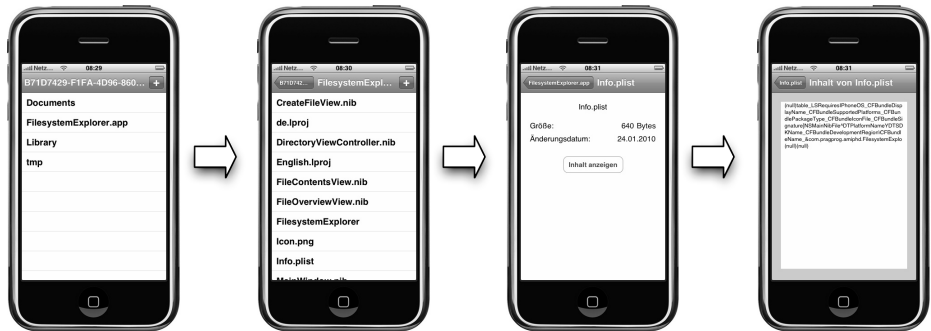


Abbildung 8.1: Navigation mit der Beispielanwendung `FileSystemExplorer`

Eine Einschränkung ist, dass Sie nicht das gesamte Dateisystem sehen können, sondern nur den Inhalt des Home-Verzeichnisses Ihrer Anwendung. Das ist ein Teil des *Sandbox*-Sicherheitsmodells des iPhone OS. Ihre Anwendung kann in dem für sie reservierten Teil des Dateisystems (in ihrem *Sandkasten*) machen, was sie will, hat aber keinen Zugriff auf andere Teile des Dateisystems. Das bedeutet, dass der Code und die Daten anderer Anwendungen sowie die Musik und die Videos des Anwenders, die Systemdaten und alles andere außerhalb Ihres Home-Ordners verbotene Zone ist. Versuche, davon zu lesen oder dort zu schreiben, schlagen fehl.

8.1 Das Dateisystem erforschen

Um die üblichen Datei-I/O-Aktivitäten auszuprobieren, wollen wir eine Anwendung zum Durchstöbern des Dateisystems entwickeln, die in Ihrem Home-Verzeichnis beginnt. Die Anwendung `FileSystemExplorer` wird die folgenden Aufgaben unterstützen:

¹ Ein entscheidender Unterschied: Das Dateisystem auf dem iPhone unterscheidet sich anders als der Mac zwischen Groß- und Kleinschreibung.

- Den Inhalt eines Verzeichnisses in einer UITableView ausgeben
- Im Inhalt dieses Verzeichnisses navigieren durch:
 - eine weitere UITableView für ein ausgewähltes Unterverzeichnis
 - einen Überblick über den Inhalt einer ausgewählten Datei
- Den Inhalt einer Datei lesen und anzeigen
- Ein neues Verzeichnis mit einem vom Anwender vorgegebenen Namen erstellen
- Eine neue Datei mit vom Anwender vorgegebenen Namen und Inhalt erstellen
- Eine ausgewählte Datei löschen

Eine Vorschau auf die Beispielanwendung finden Sie in Abbildung 8.1. Wenn Sie durch die Verzeichnisse navigieren, sehen Sie, dass ihre Inhalte in Tabellen-Views angezeigt werden. Wählen Sie eine Datei aus, erhalten Sie einen Überblick über die Datei, und Sie können ihren Inhalt in einer weiteren View anzeigen lassen. Wir werden eine navigationsbasierte Anwendung schreiben, die die folgenden View-Controller nutzt (mit den entsprechenden im IB entwickelten Views):

- `DirectoryViewController`: Anzeige des Inhalts eines Verzeichnisses als Tabelle. Durch das Auswählen einer Zeile kommen Sie entweder zu einem weiteren `DirectoryViewController` (wenn es sich um ein Verzeichnis handelt) oder zu einem Überblick über eine Datei. Auch gibt es einen `+`-Button, der dazu dient, neue Dateien und Verzeichnisse anzulegen, und Sie können Dateien oder Verzeichnisse per „Wischen“ löschen.
- `FileOverviewViewController`: Ausgabe von Metadaten einer Datei: Größe, Datumswerte und so weiter.
- `FileContentsViewController`: Auslesen des Inhalts einer Datei in eine `UITextView`, einer grafischen Komponente, um (scrollbar) mehrzeilige Texte auszugeben (und eventuell zu bearbeiten).
- `CreateDirectoryViewController`: Ausgabe eines `UITextField`, um einen neuen Verzeichnisnamen abzufragen.
- `CreateFileViewController`: Ausgabe eines Textfelds für einen neuen Dateinamen und einer `UITextView` für den Inhalt der Datei.

Über die Verzeichnisse Ihrer Anwendung

Bevor wir mit dem Navigieren in Dateien und Unterverzeichnissen beginnen, wollen wir einen Blick auf das Home-Verzeichnis werfen. Dieses Verzeichnis gibt es in jeder Anwendung für das iPhone OS, die Sie erzeugen. Es stehen vier Verzeichnisse zur Verfügung, von denen Sie allerdings sehr wahrscheinlich zwei nie genauer beachten werden.

- **Documents:** Dies ist das Verzeichnis, in dem Sie hauptsächlich Dateien für Ihre Anwender speichern. Es ist standardmäßig leer, und Sie können damit anstellen, was Sie wollen.
- **Anwendungsname:** Dieses Verzeichnis ist Ihr Anwendungs-Bundle. Es enthält Nibs, Lokalisierungen, ausführbaren Code und andere Ressourcen. In Abbildung 8.2 sehen Sie den Filesystem-Explorer, wie er sein eigenes Anwendungs-Bundle darstellt. Sehr wahrscheinlich werden Sie in diesem Verzeichnis selbst nur sehr wenig machen. Löschen Sie eine .nib-Datei, während Ihre Anwendung im Simulator läuft, beschwören Sie damit sehr wahrscheinlich einen Crash herauf. Auf dem echten Gerät werden Sie vermutlich gar nicht löschen dürfen. Wenn Sie andererseits Bilder, Töne und weitere Ressourcen aus Ihrem Anwendungs-Bundle auslesen, werden die Pfade in dieses Verzeichnis verweisen.



Abbildung 8.2: Den Inhalt eines Anwendungs-Bundles anzeigen

- **Library:** Dieses Verzeichnis existiert nur als übergeordnetes Verzeichnis zu Preferences. Sie müssen in Preferences selbst keine Dateien schreiben, stattdessen können Sie Preferences-APIs wie `NSUserDefaults`, nutzen. Voreinstellungen werden detaillierter in Kapitel 9, *Einstellungen*, auf Seite 181 behandelt.
- **tmp:** Das „temporäre“ Verzeichnis ist dafür gedacht, nur kurzfristig aufzubewahrende, temporäre Dateien abzulegen, die nicht langfristig gebraucht werden. Sie können es als „Zwischenspeicher“ der Anwendung nutzen, wenn Sie zum Beispiel eine Datei Stück für Stück anlegen wollen, bevor sie über das Netzwerk übertragen wird. Nutzen Sie dieses Verzeichnis, sollten Sie regelmäßig den Inhalt löschen (wie zum Beispiel beim Starten oder Beenden), um nicht unnötig Speicherplatz zu verbrauchen. Beachten Sie, dass die Datensicherung Ihres iPod alles ignoriert, was sich in tmp findet.

8.2 Unser Projekt erstellen

Wie Sie vielleicht schon aufgrund des Ablaufdiagramms erwartet haben, sollten Sie ein neues Xcode-Projekt mit dem Namen `Filesystem-Explorer` erzeugen und dabei das Template für Navigation-based Application nutzen (siehe auch Kapitel 6, *Navigation*, Seite 115). Bei dieser Anwendung werden wir eine ganze Menge Anpassungen vornehmen müssen.

Da dies das dritte Mal ist, dass wir mit einer navigationsbasierten Anwendung arbeiten, hoffen wir, dass Sie mittlerweile damit vertraut sind. Mit diesem Kapitel beginnen wir auch, uns vom UIKit und den Xcode-Tools weg und mehr in Richtung Rest der APIs zu bewegen. Damit Sie sich besser auf das neue Material konzentrieren können, werden wir mehr und mehr Dinge als gegeben ansehen und davon ausgehen, dass Sie über das Folgende Bescheid wissen:

- Um eine Eigenschaft zu erstellen, brauchen Sie eine dazugehörige Instanzvariable, eine **@property**-Deklaration in der `.h`-Datei und (normalerweise) eine **@synthesize**-Anweisung in der `.m`-Datei.
- Wenn Sie eine andere Klasse nutzen, müssen Sie sie per **import** ihre `.h`-Datei einbinden.
- Sie verbinden `IBOutlet`s und `IBAction`s mit dem Interface Builder, um Ihr GUI mit Ihrem Code zu verknüpfen.

- Sie implementieren ein Protokoll, indem sie seinen Namen in spitzen Klammern (<...>) als Teil Ihrer **@interface**-Deklaration aufnehmen und dann seine dazugehörigen Methoden implementieren. Für Delegates machen Sie das sehr häufig.

Idealerweise sollten Sie sich zunächst einmal konzeptionelle Gedanken über die Elemente Ihrer Anwendung machen. Je mehr Anwendungen Sie entwickeln, desto weniger Sorgen werden Sie sich über das Ziehen von diesem auf jenes machen, sondern sich eher überlegen, wie Sie Beziehungen zwischen den Objekten ausdrücken, die Sie in Ihrem Code deklarieren, und denen, die im IB erstellt werden.

Ihren Code refaktorisieren

Lassen Sie uns mit der normalen Tabellen-View beginnen. Die Klasse, die für uns generiert wird, hat den Namen `RootViewController`. In dieser Anwendung ist das kein wirklich beschreibender Name für einen View-Controller. Da wir nicht nur mit einem vorgegebenen Verzeichnis beginnen, sondern auch noch wiederholt in andere Verzeichnisse wechseln, ist das eigentlich keine View, auf die „Root“ passt – sondern eher „Directory“. Das gibt uns die Gelegenheit, wie versprochen die Refaktorisierungsunterstützung von Xcode auszuprobieren.

Um etwas in Ihrem Code zu refaktorisieren – eine Klasse, eine Methode oder einen Member – klicken Sie einfach seinen Namen in einer beliebigen Quelldatei doppelt an und wählen entweder aus dem EDIT-Menü den Eintrag REFACTOR, oder Sie klicken den Namen mit der rechten Maustaste an, um das Kontextmenü aufzurufen. Öffnen Sie also `RootViewController.h`, markieren Sie den Klassennamen `RootViewController` aus der **@interface**-Deklaration und wählen Sie dann EDIT REFACTOR. In das sich öffnende Refaktorisierungsfenster können Sie einen neuen Namen eingeben. Sie erhalten dann eine Übersicht über sämtlichen Code (und mögliche Dateinamen), der sich ändern wird. In Abbildung 8.3 sehen Sie, was passiert, wenn wir `RootViewController` zu `DirectoryViewController` refaktorisieren wollen.²

² Stellen Sie sicher, dass Sie auch die Importe überprüfen – beim Anpassen des Buches an iPhone 3.0 haben wir bemerkt, dass der Refaktorisierer es nicht geschafft hat, `#import "RootViewController.h"` in `FilesystemExplorerAppDelegate.m` nach `#import "DirectoryViewController.h"` zu ändern. Das Projekt ließ sich erst wieder kompilieren, nachdem wir das selbst angepasst haben.

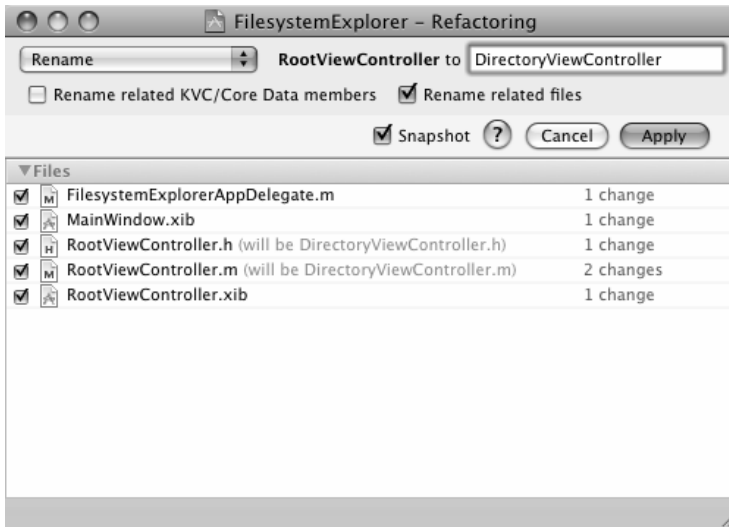


Abbildung 8.3: Die Klasse RootViewController refaktorisieren

Wendet man diese Änderungen an, werden die Quelldateien angepasst (und umbenannt), aber die Datei RootViewController.xib besitzt immer noch ihren alten Namen. Wenn Sie den ebenfalls ändern wollen – Sie müssen nicht, aber Sie sollen wenigstens erfahren, wie es geht –, müssen Sie an ein paar Stellen eingreifen. Sie können zunächst im Resources-Ordner der Liste Groups & Files die Datei RootViewController.xib markieren und im per rechter Maustaste aufrufbaren Kontextmenü den Eintrag RENAME wählen. Damit wird die Datei umbenannt, aber auch Ihre Anwendung ist dann nicht mehr lauffähig.

Um das zu verstehen, klicken Sie doppelt auf MainWindow.xib, um den Interface Builder zu starten. Öffnen Sie das Navigation Controller Objekt, wird dessen View angezeigt – eine graue Navigationsleiste am oberen Rand und ein großer, gestrichelt umrandeter Bereich, in dem steht: „View Loaded from RootViewController“. Und das wäre *schlecht*, denn wir haben doch gerade RootViewController.xib umbenannt, aber MainWindow.xib hat offensichtlich nichts davon mitbekommen. Glücklicherweise lässt sich das recht einfach beheben. Klicken Sie im Vorschaufenster des Navigations-Controller auf die View und öffnen Sie den Attributes Inspector ([⌘]-[1]). Wie Sie in Abbildung 8.4 sehen, müssen Sie nur den Nib-Namen auf den von Ihnen umbenannten View-Controller zurücksetzen.

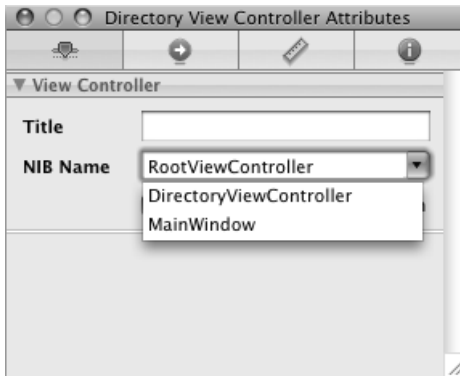


Abbildung 8.4: Die Datei `Mainwindows.xib` refaktorisieren, damit für die erste View eine andere `.nib`-Datei genutzt wird

Verzeichnisse anzeigen

Um den Inhalt eines Verzeichnisses anzuzeigen, benötigt Ihr `DirectoryViewController` eine Instanzvariable vom Typ `NSArray` mit dem Namen `directoryContents`, die einfach den Inhalt eines bestimmten Verzeichnisses als `NSString`-Pfade enthält. Cocoa besitzt keine besonderen Klassen, um Dateien zu repräsentieren, und verwendet häufig `NSString`s, für die es eine ganze Reihe von Instanzmethoden zur Behandlung von Pfaden gibt.³ Tragen Sie diese Instanzvariable in der Header-Datei `DirectoryViewController.h` ein und wechseln Sie dann zur Implementierungsdatei. Mit diesem Array ist es nun sehr einfach, die beiden notwendigen `UITableViewDataSource`-Methoden zum Befüllen der Tabelle für den Verzeichnisinhalt zu schreiben. Als Erstes ist die Anzahl der Zeilen in der Tabelle einfach die Länge des Arrays `directoryContents`, also ersetzen Sie die vorgeschlagene Implementierung von `tableView:numberOfRowsInSection:` durch einen Einzeiler, der die Array-Länge zurückgibt:

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [directoryContents count];
}
```

Analog wird ein Tabellenfeld mit dem Namen der Datei erstellt, bei dem es sich einfach um ein per Index angesprochenes Array-Element handelt.

³ Dateien können auch als `NSURL`s vom Typ `file://` dargestellt werden, und es gibt Methoden, um Dateireferenzen zwischen Strings und URLs zu konvertieren.

Die hervorgehobenen Zeilen wurden der Standardimplementierung von `tableView:cellForRowAtIndexPath:` hinzugefügt:

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)table
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"DirectoryViewCell" ;
    UITableViewCell *cell =
        [table dequeueReusableCellWithIdentifier:
            CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier] autorelease];
    }
    ▶ cell.textLabel.text = (NSString*)
    ▶ [directoryContents objectAtIndex: indexPath.row];
    return cell;
}
```

Aber wie schaffen wir es, diese Daten in die Tabelle zu laden? Irgendwann müssen wir dem View-Controller mitteilen, welches Verzeichnis er darstellt. Eine Möglichkeit ist, den `directoryPath` zu einer Eigenschaft des View-Controller zu machen. Erstellen Sie die Instanzvariable `NSString *directoryPath` und deklarieren Sie sie dann als Eigenschaft:

FileIO/FileSystemExplorer/Classes/DirectoryViewController.h

```
@property (nonatomic, retain) NSString *directoryPath;
```

Wenn die Eigenschaft gesetzt ist, können wir den Verzeichnisisinhalt in das Array laden und die Tabellen-View aktualisieren. Aber dazu müssen wir unsere eigene Implementierung der Eigenschafts-Getter und -Setter schreiben, da eine per `@synthesize` „erzeugte“ Eigenschaft nicht weiß, wie das Array gefüllt werden soll. Stattdessen implementieren wir die Eigenschaft, indem wir die Methoden `directoryPath` und `setDirectoryPath:` selbst implementieren. Dadurch können wir den Inhalt des Verzeichnisses laden, wenn der Pfad gesetzt wird, und dabei auch gleich den Navigationstitel auf den aktuellen Verzeichnisnamen setzen.⁴

⁴ Da wir im Getter nichts Spannendes machen, könnten wir eine normale `@synthesize`-Anweisung nutzen und ihn erzeugen lassen. `setDirectoryPath` würden wir dann per Hand schreiben. Wir haben uns nur gedacht, dass es ganz nützlich wäre, beide Methoden hier zu zeigen, damit Sie sehen, dass hinter den Eigenschaften nichts Mystisches steckt.

```
FileIO/FileSystemExplorer/Classes/DirectoryViewController.m
```

```
- (NSString*) directoryPath {
    return directoryPath;
}

- (void) setDirectoryPath: (NSString*)p {
    [p retain];
    [directoryPath release];
    directoryPath = p;
    [self loadDirectoryContents];
    // Titel des Nav Controller mit dem letzten Pfadelement setzen
    NSString *pathTitle= [directoryPath lastPathComponent];
    self.title = pathTitle;
}
```

Da damit der `directoryPath` per `retain` in Beschlag genommen wird, sollten Sie sicherstellen, dass er in `dealloc` per `release` wieder freigegeben wird – so wie Sie das mit jeder anderen Eigenschaft machen, die das Attribut `retain` nutzt. Die Attribute stehen für Ihren Kontrakt mit jeglichem Code, der Ihre Eigenschaft nutzt, und Ihre Implementierung sollte die deklarierten Attribute in Bezug auf Schreibbarkeit, Atomarität und Speicherverwaltung auch erfüllen.

Nachdem wir jetzt eine Stelle haben, an der eine Methode `loadDirectoryContents` aufgerufen wird, müssen wir mit den File-APIs von Cocoa nun auch den Inhalt eines gegebenen Verzeichnisses einlesen. `NSFileManager` ist die wichtigste Klasse, wenn Sie mit dem Dateisystem arbeiten. Da es keine „Datei“-Klasse gibt, für die Sie Methoden aufrufen können, holen Sie sich stattdessen eine Instanz des Managers – eine gemeinsam genutzte Standardinstanz, die Sie über die Klassenmethode `defaultManager` erhalten – und rufen deren Methoden auf. `NSFileManager` besitzt eine Methode `directoryContentsAtPath:`, die ein `NSArray` mit `NSStrings` zurückgibt, in denen die Dateien oder Verzeichnisse aus diesem Pfad enthalten sind. Es ist also kein Problem für unseren `DirectoryViewController`, den Verzeichnisinhalt zu ermitteln (denken Sie daran, dass Sie diese Methode auch im Header `DirectoryViewController.h` angeben müssen):

```
FileIO/FileSystemExplorer/Classes/DirectoryViewController.m
```

```
-(void) loadDirectoryContents {
    [directoryContents release];
    directoryContents = [[NSFileManager defaultManager]
        directoryContentsAtPath: directoryPath];
    [directoryContents retain];
}
```

Da damit der neue Verzeichnisinhalt per retain in Besitz genommen wird, müssen Sie Ihrer dealloc-Methode auf jeden Fall ein `[directoryContents release];` hinzufügen.

Der erste `DirectoryViewController` wurde schon als Teil der `MainWindow.xib` vom Template erzeugt. Wie können wir ihm dann den ersten `directoryPath` mitteilen, den er nutzen soll? Lassen Sie uns das beim Starten der Anwendung vornehmen. In `FilesystemExplorerAppDelegate.h` deklarieren Sie eine Instanzvariable namens `directoryViewController` für ein `IBOutlet` auf den `DirectoryViewController`. Dann öffnen Sie `MainWindow.xib` mit dem IB, lassen die Verbindungen von `FilesystemExplorerAppDelegate` anzeigen (mit dem Connections Inspector, -) und verbinden `directoryViewController` mit dem Objekt `DirectoryViewController`, das ein Kind des Navigations Controller ist (vermutlich müssen Sie den Listen- oder Spaltenmodus des IB aktivieren, um es zu finden).

Da wir jetzt Zugriff auf den ersten `DirectoryViewController` haben, wollen wir ihm einen `directoryPath` als Ausgangspunkt mitgeben. Das erreichen Sie mit einer Zeile in der Methode `applicationDidFinishLaunching` des `FilesystemExplorerAppDelegate`, nachdem sie das Hauptanwendungsfenster mit `makeKeyAndVisible`: sichtbar gemacht hat.

```
FileIO/FilesystemExplorer/Classes/FilesystemExplorerAppDelegate.m
```

```
// Erste View befüllen
directoryViewController.directoryPath = NSHomeDirectory();
```

Wenn Sie möchten, können Sie jetzt die Anwendung bauen und laufen lassen. Sie erhalten eine Tabelle mit vier Zeilen, die den Inhalt Ihres Home-Verzeichnisses widerspiegeln: Documents, `FilesystemExplorer.app`, Library und tmp. Sie werden auch sehen, dass der Navigations-Titel dem Namen des Home-Verzeichnisses entspricht – dem String mit der langen Anwendungs-ID. Jetzt wollen wir also damit beginnen, uns in diese Verzeichnisse hineinzuarbeiten.

Wie Sie sehen, können wir das Home-Verzeichnis der Anwendung mit einem einfachen Aufruf von `NSHomeDirectory()` erhalten. In Mac Cocoa liefert diese Methode den Pfad des Home-Verzeichnisses des Anwenders zurück, aber auf dem iPhone erhält man das Home-Verzeichnis der *Anwendung* – ein Verzeichnis mit dem Anwendungs-Bundle und den „Hilfs“-Verzeichnissen Documents, Library und tmp.

Dieses Verzeichnis hat einen langen, eindeutigen Kennungs-String als Namen – der gesamte Pfad im Simulator ist dann so etwas wie `/Users/thomas/Library/Application Support/iPhone Simulator/User/Applications/5C73EBC6-BDB3-46E7-B7EC-203A0BA6417B`, wobei das letzte Pfad-element eine künstliche *Anwendungs-ID* ist, die mit jedem neuen Bauen Ihrer Anwendung neu generiert wird. Da Sie im iPhone-Sicherheitsmodell in der Dateisystemhierarchie nicht höher navigieren können, ist der Name dieses Verzeichnisses oder der darüber ziemlich irrelevant.

Wir möchten in jedem Fall darauf hinweisen, dass wir nicht immer das von `NSHomeDirectory()` zurückgegebene Verzeichnis nutzen müssen. Wir wollen ja häufig einen Pfad auf eines seiner Unterverzeichnisse haben, wie zum Beispiel `Documents`. Es gibt eine Technik, Systempfade nach bekannten speziellen Verzeichnissen zu durchsuchen, die anhand des Kopierens einer Datenbankdatei nach `Documents` in Abschnitt 10.3, *Ihre Datenbank auf das iPhone bringen*, auf Seite 212 demonstriert wird.

8.3 Dateiattribute auslesen

Die Beispielanwendung muss wissen, welche Verzeichniselemente Dateien und welche Verzeichnisse sind, um richtig reagieren zu können, wenn der Anwender auf ein Feld tippt – entweder soll der Dateiinhalte angezeigt werden oder die Liste eines neuen Verzeichnisses. Es gibt viele verschiedene Wege, herauszufinden, ob ein gegebener Pfad ein Verzeichnis oder eine Datei repräsentiert. Einer der einfachsten ist die Nutzung der Methode `fileExistsAtPath:isDirectory:` des `NSFileManager`. Der zweite Parameter ist ein Zeiger auf eine `BOOL`-Variable. Kehrt die Methode zurück, ist dieser Wert auf `YES` gesetzt, wenn der Pfad für ein Verzeichnis steht. Eine typische (aber hypothetische) Anwendung sieht daher so aus:

```
BOOL myPathIsDir;
BOOL fileExists = [[NSFileManager defaultManager]
    fileExistsAtPath: selectedPath
    isDirectory: &myPathIsDir];
NSLog (myPathIsDir ? @"Mein Pfad ist ein Verzeichnis " :
    @"Mein Pfad ist eine Datei" );
```

Beachten Sie, wie die *Adresse* der `BOOL`-Variablen an die Methode übergeben und deren Wert nach dem Aufruf geprüft wird. Richtig, das sieht sehr nach C aus. Das muss so sein, weil der Rückgabewert der Methode schon genutzt wird, um anzuzeigen, ob eine Datei an dem angegebenen Pfad überhaupt vorhanden ist. Wenn Sie sich lediglich dafür interessie-

ren, ob ein Pfad auf ein existierendes Objekt verweist, und es Ihnen egal ist, ob es sich dabei um ein Verzeichnis oder eine Datei handelt, können Sie an `isDirectory` einfach `NULL` übergeben oder noch besser die Methode `fileExistsAtPath:` nutzen, die nur ein Argument hat.

Nachdem wir nun ein Verzeichnis von einer Datei unterscheiden können, kann das GUI unseres Dateisystem-Explorers bestimmen, was zu tun ist, wenn der Anwender auf eine Zeile tippt (wodurch die eventverarbeitende Delegate-Methode `tableView:didSelectRowAtIndexPath:` aufgerufen wird). Wenn das ausgewählte Element ein Verzeichnis ist, erstellen Sie einen weiteren `DirectoryViewController` mit `initWithNibName:bundle:`, setzen dessen Eigenschaft `directoryPath` und übergeben den View-Controller an den `UINavigationController`. Um zum bisherigen Verzeichnis zurückkehren zu können, müssen wir gar nichts tun – der Anwender wird einfach den vom Navigations-Controller bereitgestellten Zurück-Button nutzen.

FileIO/FilesystemExplorer/Classes/DirectoryViewController.m

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *selectedFile = (NSString*)
    [directoryContents objectAtIndex: indexPath.row];
    BOOL isDir;
    NSString *selectedPath =
    [directoryPath stringByAppendingPathComponent: selectedFile];
    if ([[NSFileManager defaultManager]
        fileExistsAtPath:selectedPath isDirectory:&isDir] && isDir) {
        DirectoryViewController *directoryViewController =
            [[DirectoryViewController alloc]
             initWithNibName:@"DirectoryViewController"
             bundle:nil];
        [[self navigationController]
         pushViewController:directoryViewController animated:YES];
        directoryViewController.directoryPath = selectedPath;
        [directoryViewController release];
    }
}
```

Mit dieser ersten Hälfte des `if`-Blocks (und den natürlich dafür notwendigen schließenden geschweiften Klammern) können Sie die Anwendung schon starten, um durch die Verzeichnisse vor und zurück zu navigieren. So können Sie zum Beispiel ins Library-Verzeichnis und von dort zu Preferences wechseln, um dann mithilfe des Zurück-Buttons der Navigationsleiste zurückzukehren.

Wenn der Anwender auf eine Zeile tippt, die eine Datei repräsentiert, brauchen wir eine neue View, die von einer Klasse mit dem (von uns vergebenen) Namen `FileOverviewViewController` verwaltet wird. Erzeugen Sie mit `New File` diesen neuen View-Controller und binden

Sie seinen Header per **import** in `DirectoryViewController.m` ein. Wie der `DirectoryViewController` ist dies ein View-Controller mit einer Eigenschaft `filePath`, deren Setter-Methode wir selbst implementieren (ebenfalls wie beim `DirectoryViewController`) und die als Signal zum Aktualisieren des GUI dient.

`FileIO/FileSystemExplorer/Classes/FileOverviewViewController.m`

```
- (NSString*) filePath {
    return filePath;
}

- (void) setFilePath: (NSString*)p {
    [p retain]; [filePath release];
    filePath = p;
    [self updateFileOverview];
    // Titel des Navigations-Controller mit dem
    // letzten Pfadelement setzen
    NSString *pathTitle= [filePath lastPathComponent];
    self.title = pathTitle;
}
```



Joe fragt...

Warum sollen View-Controller per Code erzeugt werden?

In Kapitel 6, Navigation, auf Seite 115 haben wir den zweiten View-Controller für das Beispiel DVDCase im Interface Builder erzeugt, aber für den `FileSystemExplorer` schlagen wir vor, dass Sie die View-Controller per Code anlegen. Der Unterschied ist, dass die Navigation in DVDCase deterministisch ist. Sie geht immer vom `RootViewController` zum `DVDCabinetController`, daher können wir beide Objekte zur Designzeit im IB erzeugen. Durchfortsten wir das Dateisystem, kennen wir die Reihenfolge der View-Controller, die wir benötigen, nicht. Wir können ein Verzeichnis nach dem anderen aufsuchen und damit immer mehr Instanzen des `DirectoryViewController` auf den Navigations-Stack packen, bevor wir zu einer Datei kommen (für die wir einen `FileOverviewViewController` bräuchten). Da wir die `DirectoryViewController` per Code erzeugen müssen, ist es in diesem Fall unserer Meinung nach einfacher, alle Navigations-View-Controller per Code zu erstellen – abgesehen vom ersten, den Xcode automatisch als Teil des `NavigationController` in `MainWindow.xib` anlegt.

Um also in `tableView:didSelectRowAtIndexPath:` des `DirectoryViewController` zur Dateiübersichts-View zu navigieren, erzeugen wir einen `FileOverviewViewController`, übergeben ihn an den Navigations-Controller und setzen `filePath`, damit der neue Controller seine View aktualisiert (was wir als Letztes tun, um sicherzustellen, dass die View aus der `.nib`-Datei geladen wurde).

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
else {
    FileOverviewViewController *fileOverviewViewController =
        [[FileOverviewViewController alloc]
         initWithNibName:@"FileOverviewView"
         bundle:nil];
    [[self navigationController]
     pushViewController:fileOverviewViewController animated:YES];
    fileOverviewViewController.filePath = selectedPath;
    [fileOverviewViewController release];
}
}
```

Um das Ganze einfach zu halten, wird auf der Übersichtsseite nur der Dateiname, die Größe und das letzte Änderungsdatum angezeigt. In Abbildung 8.5 sehen wir, wie diese View aussieht, wenn man sich die Binärdatei `FileSystemExplorer` im Anwendungs-Bundle anschaut. Sie wissen vielleicht mittlerweile schon aus dem Effeft, wie Sie Ihre eigenen View-Controller mit den im IB erzeugten Views verbindet, aber wir wollen die Schritte trotzdem nochmals einzeln durchgehen:

1. Erstellen Sie eine neue Subklasse `UIViewController` und fügen Sie `IBOutlet`s und `IBAction`s in der Header-Datei für die Verbindungen hinzu, die Sie im IB erstellen wollen. In diesem Fall brauchen wir drei `UILabel`-Outlets. Denken Sie daran, diese Outlets durch Instanzvariablen abzudecken und per **@synthesize** die Getter und Setter in `FileOverviewViewController.m` zu erzeugen.

FileIO/FileSystemExplorer/Classes/FileOverviewViewController.h

```
@property (nonatomic, retain) IBOutlet UILabel *fileNameLabel;
@property (nonatomic, retain) IBOutlet UILabel *fileSizeLabel;
@property (nonatomic, retain) IBOutlet UILabel *fileModifiedLabel;
```

Wir brauchen auch noch zwei Methoden – eine zum Aktualisieren der Labels, wenn die Eigenschaft gesetzt wurde, und eine `IBAction`, mit der auf das Antippen eines Buttons reagiert wird, wenn sich der Anwender den Inhalt der Datei anzeigen lassen will.

FileIO/FileSystemExplorer/Classes/FileOverviewViewController.h

- (void) updateFileOverview;
- (IBAction) readFileContents;

2. Erstellen Sie in Xcode ein neues View-Nib mit dem Namen FileOverviewView.xib und öffnen Sie es im IB, um die Benutzerschnittstelle zu erstellen. Verwenden Sie den Identity Inspector (-) , um File's Owner auf Ihre eigene View-Controller-Klasse zu setzen.
3. Verbinden Sie bei gedrückter -Taste die Outlets Ihrer Klasse mit den GUI-Controls und die Events (wie zum Beispiel das Event „Touch Up Inside“ eines Buttons) mit Ihren Actions. Denken Sie auch daran, das Outlet view von File's Owner mit der Main View im IB-Fenster zu verbinden, da Ihre Anwendung ansonsten beim Laden dieser .nib-Datei abstürzen würde (das ist etwas, an das Sie immer denken müssen, wenn Sie Nibs mit dem Template „View XIB“ erstellen).
4. Fügen Sie diesen neuen View-Controller mit pushViewController:animated: Ihrem Navigations-Controller hinzu, so wie wir es vorher schon in tableView:didSelectRowAtIndexPath: des DirectoryViewController getan haben.

Wie bereits erwähnt, können Sie durch einen eigenen Setter für die Eigenschaft filePath zusätzliche Aktionen durchführen, wenn die Eigenschaft gesetzt wird, wie zum Beispiel das Aktualisieren der Labels. Das erreichen wir durch den Aufruf der Methode updateFileOverview. In dieser Methode erstellen wir zunächst Objekte zum Formatieren von Daten, so dass wir beim Befüllen der Labels gut lesbare Zahlen und Datumswerte erhalten.

FileIO/FileSystemExplorer/Classes/FileOverviewViewController.m

```
- (void) updateFileOverview {
    if (self.filePath != NULL) {
        NSString *fileName = [self.filePath lastPathComponent];
        fileNameLabel.text = fileName;

        NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
        [dateFormatter setTimeStyle:NSDateFormatterNoStyle];

        NSNumberFormatter *numberFormatter =
            [[NSNumberFormatter alloc] init];
        [numberFormatter setPositiveFormat: @"#,##0.## Bytes" ];
```




Abbildung 8.5: Anzeige der Dateiattribute

Jetzt nutzen wir wieder die Klasse `NSFileManager`. Sie bietet die Methode `fileAttributesAtPath:traverseLink:`, die `NSDictionary` mit den Attributen der Datei als Schlüssel/Wert-Paare zurückgibt (der Parameter `traverseLink` bestimmt, ob symbolischen Links gefolgt werden soll oder ob man die Attribute für den Link selbst erhalten will – in dieser Anwendung ist das recht unwichtig). Die möglichen Schlüsselwerte sind in der Dokumentation von `NSFileManager` beschrieben. Hier brauchen wir `NSFileSize` und `NSFileModificationDate`, daher geht der Code wie folgt weiter:

`FileIO/FilesystemExplorer/Classes/FileOverviewViewController.m`

```
NSDictionary *fileAttributes =
[[NSFileManager defaultManager]
    fileAttributesAtPath: self.filePath
    traverseLink: YES];
NSDate *modificationDate = (NSDate*)
    [fileAttributes objectForKey: NSFileModificationDate];
NSNumber *fileSize = (NSNumber*)
    [fileAttributes objectForKey: NSFileSize];
fileSizeLabel.text =
    [numberFormatter stringFromNumber: fileSize];
fileModifiedLabel.text =
    [dateFormatter stringFromDate: modificationDate];
[numberFormatter release];
```

```

        [dateFormatter release];
    }
}

```

Wie wir sehen können, sind zwar alle Schlüssel vom Typ `NSString`, aber der Typ der Werte unterscheidet sich je nach Attribut. In diesem Fall ist das Modifikationsdatum ein `NSDate`, während die Dateigröße eine `NSNumber` ist. Die Dokumentation des `NSFileManager` gibt für jedes Attribut den Typ des Werts an. Allerdings gibt es auf dem iPhone wenige Attribute, die Sie verwenden können. Viele der Attribute sind entweder Überbleibsel vom Mac (zum Beispiel `NSFileHFSTypeCode` und `NSFileHFSTypeCreator`), oder es handelt sich um Besitz- und Berechtigungsattribute aus der Unix-Welt, die auf einem iPhone keine Relevanz haben.⁵

8.4 Daten aus Dateien lesen

Der nächste und vermutlich nützlichste Schritt ist das Lesen eines Dateiinhalts. Auch dafür gibt es eine Reihe von Wegen – mehr, als hier sinnvoll zu beschreiben wären.

Vielleicht ist Ihnen in Abbildung 8.5 der Button INHALT ANZEIGEN aufgefallen. Tippen Sie diesen Button an, müssten Sie eine neue View erreichen, die den Inhalt der Datei anzeigt. Dazu müssen Sie die Klasse `FileContentsViewController` erstellen. Verpassen Sie ihr eine Eigenschaft `filePath`, die die Klasse `FileOverviewViewController` setzen kann, um ihr mitzuteilen, welche Datei zu lesen ist.

Jetzt können Sie den Event-Handler `readFileContents` in `FileOverviewViewController.m` implementieren:

FileIO/FileSystemExplorer/Classes/FileOverviewViewController.m

```

- (IBAction) readFileContents {
    FileContentsViewController *fileContentsViewController =
        [[FileContentsViewController alloc]
         initWithNibName:@"FileContentsView"
         bundle:nil];
    fileContentsViewController.filePath = filePath;
    fileContentsViewController.title =
        [NSString stringWithFormat:@"Inhalt von %@",
        [filePath lastPathComponent]];
    [[self navigationController] pushViewController:
        fileContentsViewController animated:YES];
    [fileContentsViewController release];
}

```

⁵ Erstaunlicherweise ist das nützlich klingende `NSFileCreationDate` zwar definiert, aber das Attribut ist im Simulator und auf dem iPhone selbst mit dem iPhone SDK 3.0 immer `NULL`.

Erstellen Sie eine neue .nib-Datei `FileContentView.xib` für die Anzeige des Dateiinhalts, die nur eine `UITextView` für den Dateiinhalt benötigt. Deklarieren Sie dieses Outlet in `FileContentViewController.h` und schauen Sie sich dann Abbildung 8.6 an. So soll das Ergebnis aussehen. Da diese View Teil einer Navigation sein wird, sollten Sie sicherstellen, dass Sie im IB für die View im Attributes Inspector die Navigationsleiste einschalten.

Wenn Sie nun auf BUILD AND RUN klicken, können Sie zu Dateien navigieren, sich ihre allgemeinen Informationen anzeigen lassen und dann auf den Button INHALT ANZEIGEN tippen, um zur Inhalts-View zu navigieren. Allerdings haben wir ihre Member noch nicht befüllt.

Da der Inhalt der Text-View als `NSString` verwaltet wird, beschränkt sich das Laden der Datei in die View auf eine einzelne Codezeile:⁶

```
myString =
    [NSString stringWithContentsOfFile: filePath
                                usedEncoding: NULL
                                error: NULL];
```

Allerdings können wir hier nicht wirklich so einfach vorgehen, da die zu ladende Datei auch etwas anderes als Text enthalten kann (die meisten Dateien, die Sie im Moment anzeigen lassen können, sind Binärdateien). In diesem Fall können Sie einen anderen Einzeiler nutzen, um die Datei in ein `NSData`-Objekt zu laden – einen objektorientierten Wrapper für einen Byte-Puffer.

```
NSData *fileData = [NSData dataWithContentsOfFile: filePath];
```

So erhalten Sie natürlich ein `NSData`, die Text-View benötigt aber immer noch einen `String`. Diese beiden Methoden sind aber auf jeden Fall praktisch, wenn der Inhalt der Datei vollständig in den Speicher passt – was häufig der Fall ist. Behalten Sie sie also im Hinterkopf. Was aber, wenn Sie eine große Datei haben, die Sie nicht unbedingt auf einmal laden wollen? Was, wenn Sie eine Datei parsen oder lediglich ins Netz hochladen wollen und dabei immer nur Abschnitte lesen oder schreiben? In solchen Fällen ist es sinnvoller, den klassischen Weg zu gehen und mit *Streams* zu arbeiten.

⁶ Beachten Sie, dass die in früheren Betas gezeigten Methoden `initWithContentsOfFile:` und `stringWithContentsOfFile:` veraltet sind und stattdessen neue Methoden genutzt werden sollen, die auch die Textkodierung berücksichtigen und Fehler zurückgeben können.

Beim Stream-Konzept sind die Bytes einer Datei ein „Datenstrom“, der in eine Richtung fließt – vom Anfang der Datei zum Ende. Um eine Datei zu lesen, öffnen Sie einen Stream für eine gegebene Datei und lesen die Bytes, bis der Stream endet.

Die wichtigste Klasse für das Lesen aus einer Datei ist `NSInputStream`, eine Subklasse von `NSStream`. Um aus einer Datei lesen und in sie schreiben zu können, müssen Sie Folgendes tun:

1. Öffnen Sie einen Input-Stream für die Datei.
2. Allokieren Sie einen Speicher-Puffer, der die aus dem Stream gelesenen Bytes speichert.
3. Rufen Sie wiederholt `read:maxLength:` auf, um die nächsten paar Bytes aus dem Stream zu lesen. Der erste Parameter ist der Puffer, in den die Bytes kopiert werden sollen, während `maxLength` festlegt, wie viele Bytes Sie mit jedem Lesevorgang verarbeiten (üblicherweise die Größe des Puffers). Diese Methode gibt die Anzahl der gelesenen Bytes zurück. Die kann kleiner als `maxLength` sein. Ist sie 0, ist das Ende des Streams erreicht, und bei -1 ist ein Fehler aufgetreten.
4. Tun Sie etwas mit den in den Puffer eingelesenen Bytes.
5. Haben Sie das Ende des Streams erreicht, schließen Sie ihn.

Im Folgenden wird gezeigt, wie der `FileContentsViewController` einen `NSInputStream` nutzt, um den Inhalt einer Datei in eine `UITextView` zu lesen.⁷ Sie rufen `loadFileContentsIntoTextView` aus `viewDidLoad` auf, so dass der Dateiinhalt in die Text-View geladen wird, bevor wir dorthin navigieren.

FileIO/FileSystemExplorer/Classes/FileOverviewViewController.m

```
- (void) loadFileContentsIntoTextView {
    // Öffnen eines Streams für filePath
    NSInputStream *inputStream = [[NSInputStream alloc]
        initWithFileAtPath: filePath];
    [inputStream open];
    // Lesen und Übertragen in NSTextView
    NSInteger maxLength = 128;
    uint8_t readBuffer [maxLength];
    BOOL endOfStreamReached = NO;
    // Diese Schleife blockt, bis der Stream endet
    while (! EndOfStreamReached) {
```

⁷ Wenn Sie möchten, können Sie diesen Code nutzen, aber wir werden im nächsten Abschnitt eine bessere Alternative präsentieren.

```

NSInteger bytesRead = [inputStream read: readBuffer
                                maxLength:maxLength];
if (bytesRead == 0) {
    endOfStreamReached = YES;
} else if (bytesRead == -1) {
    // TODO - sollte einen Fehlerdialog enthalten
    endOfStreamReached = YES;
} else {
    NSString *readBufferString =
        [[NSString alloc]
         initWithBytesNoCopy: readBuffer
         length: bytesRead
         encoding: NSUTF8StringEncoding
         freeWhenDone: NO];
    [self appendTextToView: readBufferString];
    [readBufferString release];
}
} // while ! endOfStreamReached
[inputStream close];
[inputStream release];
}

```

Aus Gründen der Bequemlichkeit hat dieser Beispielcode eine Methode namens `appendTextToView:`, die einen `NSString` an das Ende des text-Strings der Text-View kopiert. Auf diesem Weg schreiben Sie mit jedem gefüllten Datenpuffer inkrementell in die Text-View, bis der Stream zu Ende ist und Sie den Dateiinhalt anzeigen.

FileIO/FilesystemExplorer/Classes/FileOverviewViewController.m

```

- (void) appendTextToView: (NSString*) textToAppend
{
    fileContentsTextView.text = [NSString stringWithFormat:
        @"%@%" , fileContentsTextView.text, textToAppend];
}

```

Wie Sie am Block zum Einlesen des Streams sehen, wird jeder Byte-Block, der in den Puffer eingelesen wird, in einen `NSString` konvertiert und an die View angehängt. Die hartkodierte Verwendung von `NSUTF8StringEncoding` in diesem Beispiel ist für einfache Textdateien ausreichend, aber nicht für Binärdateien.⁸ In Abbildung 8.6 können Sie sehen, wie `Info.plist`, eine Binärdatei mit Einstellungen, aussieht, wenn sie so eingelesen wird.

⁸ Ein Großteil der Anwendungsinhalte besteht aus Binärdaten, daher werden wir nicht viel Sinnvolles lesen können, bis wir dazu in der Lage sind, selbst neue Dateien anzulegen.



Abbildung 8.6: Der Dateiinhalt

8.5 Asynchrones Lesen von Dateien

Dieser Code hat ein Problem. Die Schleife zum Lesen der Daten hält die Anwendung auf, bis der Stream eingelesen ist. Die Anwendung kann also nichts anderes tun – weder das GUI rendern noch eine Eingabe verarbeiten noch irgendetwas sonst, bis das Ende des Streams erreicht ist. Wenn Dateien größer werden und die Zeit zum Einlesen länger wird, wird das immer weniger akzeptabel.

Das Hauptproblem besteht darin, dass die Methode `read:maxLength:` alles blockiert, bis die Bytes eingelesen werden können. Beim Arbeiten mit Netzwerk-Streams und deren inhärenter Unzuverlässigkeit ist das kritischer, aber es ist immer von Vorteil, einen fehlertoleranten Ansatz zu verfolgen.

Ein `NSStream` kann so eingerichtet werden, dass er *asynchron* arbeitet. Sie können sich für Zwischendurch-Benachrichtigungen registrieren, so dass Sie informiert werden, wenn der Stream wieder Bytes zur Verfügung hat. Gibt es gerade keine, werden Sie auch nicht blockiert. Dazu übergeben Sie dem Stream einen Delegate (zum Beispiel `FileContents-`

ViewController selbst), der die Methode `stream:handleEvent:` implementiert. Dann sorgen Sie dafür, dass der Stream regelmäßig prüft, ob Bytes zur Verfügung stehen. Ist das so, soll er Ihre Delegate-Methode aufrufen. Der entsprechende Aufruf in `viewDidLoad` sieht dementsprechend wie folgt aus (statt des Aufrufs von `loadFileContentsIntoTextView`):

FileIO/FileSystemExplorer/Classes/FileContentViewController.m

```

1 - (void) setUpAsynchronousContentLoad {
2     // Einen Stream für filePath öffnen
3     NSInputStream *inputStream =
4         [[NSInputStream alloc] initWithFileAtPath: filePath];
5     [inputStream setDelegate: self];
6     [inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
7         forMode:NSDefaultRunLoopMode];
8     [inputStream open];
9     [inputStream release];
10 }
```

Der Aufruf von `scheduleInRunLoop:forMode:` in Zeile 6 sieht vielleicht etwas ungewöhnlich aus, aber es ist tatsächlich nur ein Standardaufruf. Er besagt lediglich, dass die Ereignisverarbeitung (die *Run Loop*) des aktuellen Threads diejenige sein soll, die sich um das Prüfen des Streams kümmert. Das ist die ganz normale Einstellung. Zudem wollen Sie, dass die Run Loop im Standardmodus arbeitet, denn es gibt noch eine ganze Reihe esoterischer Ereignisverarbeitungsmodi (von denen die meisten für Mac Cocoa definiert sind und gar nicht auf dem iPhone existieren). Sehr wahrscheinlich werden Sie diese Methode niemals mit anderen Optionen aufrufen.

Der Delegate muss nur die eine Methode `stream:handleEvent:` implementieren. Das erste Argument ist der einzulesende Stream, der das Ereignis ausgelöst hat, das zweite ist ein `NSStreamEvent`, das das aufgetretene Event beschreibt. Diese Events sind Konstanten, die in der Klasse `NSStream` enumeriert und in ihrer Dokumentation beschrieben wurden. Dabei können die folgenden Ereignisse auftreten:

- das Öffnen eines Streams
- die Verfügbarkeit eines Streams zum Lesen oder Schreiben
- ein Fehler bei der Stream-Verarbeitung
- das Ende des Streams

Bei der Implementierung der Methode wird man also üblicherweise ein **switch** für das `NSStreamEvent` nutzen und entscheiden, was zu tun ist. Im `FilesystemExplorer`-Beispiel muss nur das Folgende passieren:

- Wenn Bytes bereit sind, müssen sie aus dem Stream gelesen werden.
- Wenn ein Fehler aufgetreten ist, wird ein Dialogfenster angezeigt, der Stream geschlossen, und die Callbacks werden deregistriert.
- Wenn das Ende des Streams erreicht ist, wird der Stream geschlossen, und die Callbacks werden deregistriert.

Hier eine einfache Implementierung von `stream:handleEvent::`

`FileIO/FilesystemExplorer/Classes/FileContentViewController.m`

```

1  - (void)stream:(NSStream *)theStream
2      handleEvent:(NSStreamEvent)streamEvent{
3      NSInputStream *inputStream = (NSInputStream*) theStream;
4      switch (streamEvent) {
5          case NSStreamEventHasBytesAvailable: {
6              NSInteger maxLength = 128;
7              uint8_t readBuffer [maxLength];
8              NSInteger bytesRead = [inputStream read: readBuffer
9                  maxLength:maxLength];
10             if (bytesRead > 0) {
11                 NSString *bufferString = [[NSString alloc]
12                     initWithBytesNoCopy: readBuffer
13                     length: bytesRead
14                     encoding: NSUTF8StringEncoding
15                     freeWhenDone: NO];
16                 [self appendTextToView: bufferString];
17                 [bufferString release];
18             }
19             break;
20         } // case: Bytes verfügbar
21         case NSStreamEventErrorOccurred: {
22             // Fehlermeldung
23             NSError *error = [theStream streamError];
24             if (error != NULL) {
25                 UIAlertView *errorAlert = [[UIAlertView alloc]
26                     initWithTitle: [error localizedDescription]
27                     message: [error localizedFailureReason]
28                     delegate:nil
29                     cancelButtonTitle:@"OK"
30                     otherButtonTitles:nil];
31                 [errorAlert show];
32                 [errorAlert release];
33             }
34             [inputStream removeFromRunLoop: [NSRunLoop currentRunLoop]
35                 forMode:NSDefaultRunLoopMode];
36             [theStream close];
37             break;

```



```

38         }
39         case NSStreamEventEndEncountered: {
40             [inputStream removeFromRunLoop: [NSRunLoop currentRunLoop]
41              forMode:NSDefaultRunLoopMode];
42             [theStream close];
43         }
44     }
45 }

```

In den Zeilen 4 bis 19 wird das Event „Bytes stehen zum Lesen bereit“ verarbeitet – hier lesen wir aus dem Stream. NSStreams lesen und schreiben Bytes vom Typ `uint8_t`, daher nutzen wir einen Puffer dieses Typs (Zeile 6) und versuchen dann, aus dem Stream in den Puffer zu lesen (Zeile 7). Haben wir erfolgreich mindestens ein Byte gelesen, erstellen wir damit einen neuen String (Zeile 10), wobei wir die `NSUTF8StringEncoding` nutzen, und hängen diesen String an die Text-View an.

Es gibt allerdings einen entscheidenden kritischen Punkt, auf den man bei diesem Vorgehen achten muss. Indem das Laden asynchron erfolgt, entkoppeln wir den Code für die Stream-Behandlung vom Zustand des GUI. Im Allgemeinen ist das nicht schlecht, denn dadurch wird das GUI nicht blockiert, wenn eine große Datei geladen wird. Aber es bedeutet auch, dass der Anwender von der View für den Dateiinhalte wegnavigieren kann, bevor der Stream komplett geladen ist. Weil es in dem Fall keinen Grund gibt, den Stream weiter einzulesen, kann das sogar zu einem Absturz führen.⁹

Wenn also der Anwender von dieser View wegnavigiert, wollen wir das Laden stoppen. Wir können dazu die Nachricht `viewWillDisappear:` nutzen. In diesem Fall trennen wir den Stream von der Run Loop und schließen ihn – genau so wie beim Erreichen des Stream-Endes oder bei einem Fehler. Dazu muss sich der Stream aber in einer Instanzvariablen befinden. Deklarieren Sie also in der Header-Datei `NSInputStream` `*asyncInputStream` und schreiben Sie die Methode `setUpAsynchronousContentLoad` wie folgt um:

FileIO/FileSystemExplorer/Classes/FileContentsViewController.m

```

- (void) setUpAsynchronousContentLoad {
    [asyncInputStream release];
    // einen Stream für filePath öffnen
    asyncInputStream =
        [[NSInputStream alloc] initWithFileAtPath: filePath];
    [asyncInputStream setDelegate: self];
    [asyncInputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]

```

⁹ In früheren Versionen dieses Kapitels ist genau dies aufgrund eines schon frei gegebenen Objekts geschehen. Ein Leser hat diesen Bug gefunden und ihn auf der Errata-Seite gemeldet.

```

        forMode:NSDefaultRunLoopMode];
    [asyncInputStream open];
}

```

Beachten Sie, dass der Code für die Stream-Behandlung gleich ist. Der einzige Unterschied liegt im Umgang mit der Instanzvariablen. Jetzt können wir das Einlesen des Streams stoppen, wenn der Anwender auf den Zurück-Button tippt und die View verworfen wird.

FileIO/FileSystemExplorer/Classes/FileContentViewController.m

```

- (void) viewWillDisappear: (BOOL) animated {
    [asyncInputStream removeFromRunLoop:[NSRunLoop mainRunLoop]
        forMode:NSDefaultRunLoopMode];
    [asyncInputStream close];
    [super viewWillDisappear: animated];
}

```

8.6 Dateien und Verzeichnisse erstellen und löschen

Nachdem wir uns nun durch die Verzeichnisse bis zu den Dateien und ihren Inhalten vorarbeiten können, wollen wir uns ein paar weitere datei-bezogene Aufgaben anschauen. Wir werden dem Anwender die Möglichkeit geben, ein neues Verzeichnis oder eine neue Datei zu erzeugen oder das ausgewählte Element aus einer Verzeichnis-View zu löschen.

Um ein Element hinzuzufügen, werden wir dem `DirectoryViewController` einen **+**-Button hinzufügen. Das können wir in der Methode `viewDidLoad` mit `@selector` und dem Namen einer Methode erreichen, die aufgerufen wird, wenn man den Button antippt:

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```

-(void)viewDidLoad {
    [super viewDidLoad];
    UIBarButtonItem *addButton = [[[UIBarItem alloc]
        initWithBarItemSystemItem:UIBarItemSystemItemAdd
        target:self
        action:@selector(showAddOptions)] autorelease];
    self.navigationItem.rightBarButtonItem = addButton; }

```

Sobald der Button angetippt wird, fügen wir ein `UIActionSheet` ein, das ein modales Sheet mit Buttons nach vorne bringt (siehe Abbildung 8.7). Dabei werden ein Titel für das Sheet und Namen für die Buttons mitgegeben. Dazu gehört auch ein `ABBRECHEN`-Button, der das Sheet einfach wieder ausblendet. Es gibt einen weiteren optionalen roten Button für „destruktive“ Aktivitäten, aber den benötigen wir hier nicht. Der Methode kann eine variable Anzahl an Argumenten für die „normalen“ Buttons übergeben werden. Dabei müssen Sie ein abschließendes `NULL`-Argument mitgeben, um die Liste der Button-Texte zu beenden.



Abbildung 8.7: Ein UIActionSheet für Dateiaktivitäten anzeigen

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (void) showAddOptions {
    NSString *sheetTitle = [[NSString alloc]
        initWithFormat:@"%\" \"%\" bearbeiten" ,
        [directoryPath lastPathComponent]];
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle: sheetTitle
        delegate: self
        cancelButtonTitle: @"Abbrechen"
        destructiveButtonTitle: NULL
        otherButtonTitles: @"Neue Datei" ,@"Neues Verzeichnis" , NULL];
    [actionSheet showInView: self.view];
    [sheetTitle release];
    [actionSheet release];
}
```

Damit das funktioniert, muss unsere Klasse das `UIActionSheetDelegate` implementieren und eine Callback-Methode bereitstellen, wenn einer der Buttons angetippt wird.

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (void)actionSheet:(UIActionSheet *)actionSheet
    clickedButtonAtIndex:(NSInteger)buttonIndex {
    if (buttonIndex == 0)
        [self createNewFile];
    else if (buttonIndex == 1)
        [self createNewDirectory];
}
```

Wir müssen also ganz offensichtlich die zwei Methoden `createNewFile` und `createNewDirectory` schreiben, damit sich dieser Code kompilieren lässt.

Lassen Sie uns mit dem Anlegen neuer Verzeichnisse beginnen. Wieder einmal ist der `NSFileManager` die Klasse für die dateibezogene Funktionalität. Zunächst lohnt es sich allerdings, zu prüfen, ob Sie überhaupt in dem Verzeichnis schreiben dürfen, dem Sie etwas hinzufügen wollen. Dazu reicht ein einfacher Aufruf von `isWritableFileAtPath:`, der uns bei einem positiven Ergebnis zu einer neuen Klasse `CreateDirectoryViewController` weiterleitet, die wir als Nächstes erstellen werden.

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (void)createNewDirectory {
    BOOL canWrite = [[NSFileManager defaultManager]
        isWritableFileAtPath: self.directoryPath];
    if (! canWrite) {
        NSString *alertMessage =
            @"Kann nicht in dieses Verzeichnis schreiben";
        UIAlertView *cantWriteAlert =
            [[UIAlertView alloc] initWithTitle:@"Nicht erlaubt:"
            message:alertMessage
            delegate:nil
            cancelButtonTitle:@"OK"
            otherButtonTitles:nil];
        [cantWriteAlert show];
        [cantWriteAlert release];
        return;
    }
    CreateDirectoryViewController *createDirectoryViewController =
        [[CreateDirectoryViewController alloc]
            initWithNibName: @"CreateDirectoryView"
            bundle:nil];
    createDirectoryViewController.parentDirectoryPath =
        directoryPath;
    createDirectoryViewController.directoryViewController = self;
    createDirectoryViewController.title = @"Neues Verzeichnis" ;
    [[self navigationController]
        pushViewController:createDirectoryViewController animated:YES];
    [createDirectoryViewController release];
}
```

Wir werden eine weitere View benötigen, auf der der Anwender den Namen des neuen Verzeichnisses eingeben kann (siehe Abbildung 8.8). Der `CreateDirectoryViewController` braucht eine Eigenschaft für den `parentDirectoryPath` und eine weitere für den `directoryViewController`, der ihn erzeugt hat (aus Gründen, die wir später noch erläutern werden). Definieren Sie für das GUI ein `IBOutlet` für das Feld mit dem Verzeichnisnamen. Sie werden auch `UITextFieldDelegate` implementieren wollen, um die Tastatur wieder verschwinden lassen zu können (mit `resignFirstResponder`), wenn der Anwender auf Return tippt (auch wenn das für solch eine einfache Form nicht unbedingt notwendig ist). Dann nutzen Sie `viewDidLoad`, um einen Speichern-Button in der Navigationsleiste anzuzeigen, der die Methode `createNewDirectory` aufruft:

FileIO/FilesystemExplorer/Classes/CreateDirectoryViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIBarButtonItem *saveButton =
        [[UIBarButtonItem alloc]
         initWithBarButtonSystemItem: UIBarButtonSystemItemSave
         target: self
         action: @selector(createNewDirectory)];
    self.navigationItem.rightBarButtonItem = saveButton;
    [saveButton release];
}
```

In `createNewDirectory` erzeugen Sie das Verzeichnis, indem Sie einen String für den Pfad aufbauen und dann den `NSFileManager` nutzen:

FileIO/FilesystemExplorer/Classes/CreateDirectoryViewController.m

```
- (void) createNewDirectory {
    [directoryNameField resignFirstResponder];
    NSString *newDirectoryPath =
        [parentDirectoryPath stringByAppendingPathComponent:
         directoryNameField.text];
    [[NSFileManager defaultManager]
     createDirectoryAtPath:newDirectoryPath
     attributes: nil];
    [directoryViewController loadDirectoryContents];
    [directoryViewController.tableView reloadData];
    [self.navigationController popViewControllerAnimated:YES];
}
```

Der zweite Parameter in `createDirectoryAtPath:attributes:` ist ein `NSDictionary`, dessen Schlüssel die schon weiter oben beschriebenen Attributkonstanten sind. Viele davon sind auf dem iPhone irrelevant, insbesondere für Verzeichnisse, daher wird hier häufig auf ein `NULL`-Dictionary zurückgegriffen. Als Letztes müssen wir noch den vorigen `DirectoryViewController` (eine Eigenschaft in dieser Klasse) benach-



Abbildung 8.8: Erstellen eines neuen Verzeichnisses

richtigen, damit er den Inhalt des Verzeichnisses neu lädt und seine Tabellen-View aktualisiert, bevor wir zurücknavigieren.

Haben Sie einmal ein paar Verzeichnisse erstellt, werden Sie sie auch wieder löschen wollen. Das können Sie im `DirectoryViewController` machen, indem über `tableView:commitEditingStyle:forRowAtIndexPath:` die „Wisch“-Geste ausgewertet wird. Wenn wir den Löschwischer erkennen, erhalten wir den zu löschenden Pfad. Für diesen prüfen wir, ob er nicht schreibgeschützt ist.

FileIO/FileSystemExplorer/Classes/DirectoryViewController.m

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {
    // Löschwischer
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        NSString *selectedFile = (NSString*)
            [directoryContents objectAtIndex: indexPath.row];
        NSString *selectedPath =
            [directoryPath stringByAppendingPathComponent:
             selectedFile];
```

```

BOOL canWrite =
    [[NSFileManager defaultManager]
     isWritableFileAtPath: selectedPath];
if (! canWrite) {
    // UIAlert, der angibt,
    // dass der Pfad schreibgeschützt ist

```

Wenn wir löschen dürfen, nutzen wir den `NSFileManager`, der eine einfache Methode zum Löschen bereitstellt: `removeItemAtPath:error:..`

FileIO/FilesystemExplorer/Classes/DirectoryViewController.m

```

NSError *err = nil;
if (! [[NSFileManager defaultManager]
      removeItemAtPath: selectedPath error:&err]) {
    // UIAlert, der meldet, dass Löschen fehlgeschlagen ist

```

Beachten Sie, dass diese Methode zwei Dinge tut, um die Ergebnisse des versuchten Löschens bekanntzugeben. Sie gibt ein `BOOL` zurück, das zeigt, ob die Action erfolgreich war. Übergeben Sie auch die Adresse eines Objekts vom Typ `NSError`, erhält dieses Objekt eine etwas ausführlichere Beschreibung dazu, warum das Löschen fehlschlug. Sie können dann dieses Fehlerobjekt nutzen, um dem Anwender Feedback zu geben.

War das Löschen erfolgreich, aktualisieren wir die Liste der Dateien und die entsprechende Tabellen-View:

FileIO/FilesystemExplorer/Classes/DirectoryViewController.m

```

NSArray *deletedPaths = [NSArray arrayWithObject: indexPath];
[self loadDirectoryContents];
[self.tableView deleteRowsAtIndexPaths: deletedPaths
      withRowAnimation: YES];
}

```

Wenden wir uns jetzt der anderen Option im `DirectoryViewController` zu, mit der wir etwas erzeugen können. Dafür benötigen wir noch einen View-Controller `CreateFileViewController` und die in Abbildung 8.9 präsentierte View. Diese Klasse benötigt zwei `IBOutlets` für den Dateinamen (`UITextField`) und den Dateinhalt (`UITextView`). Zudem brauchen wir Eigenschaften für den `DirectoryViewController`, der die View erzeugt hat, und den `parentDirectoryPath`, in dem die neue Datei erzeugt werden soll.

FileIO/FilesystemExplorer/Classes/CreateFileViewController.h

```

@property (nonatomic,retain) IBOutlet UITextField *fileNameField;
@property (nonatomic,retain) IBOutlet UITextView *fileContentsView;
@property (nonatomic,retain) NSString *parentDirectoryPath;
@property (nonatomic,retain) DirectoryViewController *directoryViewController;

```



Abbildung 8.9: Erzeugen einer neuen Datei

Sobald Sie den `CreateFileViewController` und die entsprechende `CreateFileView.xib` erzeugt haben, können Sie zu ihm in der Methode `createNewFile` navigieren, die von `actionSheet:clickedButtonAtIndex:` des `DirectoryViewController` aufgerufen wird.

Um die Daten aus dem Text in eine neue Datei zu kopieren, sind zwei Schritte notwendig – das Erstellen der leeren Datei und das Schreiben der Daten in diese Datei.

Dafür gibt es zum Glück Methoden, die einen ganzen `NSString` oder `NSData` unter einem gegebenen Pfad ablegen. Schauen Sie sich dazu die Methoden `writeToFile:atomically:encoding:error:` von `NSString` und `writeToFile:atomically:` sowie `writeToFile:options:error:` von `NSData` an.

Sie können auch ein `NSData` als Inhalt an die Methode `createFileAtPath:contents:attributes:` des `NSFileManager` übergeben.

Um allerdings sicherzustellen, dass wir die Benutzerschnittstelle während des Schreibens der Daten nicht blockieren, können wir auch hier die asynchrone Natur von `NSStreams` nutzen.

8.7 Daten in Dateien schreiben

Die Stream-Methode für das Schreiben in eine Datei entspricht weitgehend der für das Lesen aus einer Datei:

1. Öffnen eines Streams für eine Datei.
2. Wiederholtes Schreiben von Datenblöcken aus einem Speicherpuffer in den Stream.
3. Schließen des Streams, wenn alle Daten geschrieben wurden.

Die Subklasse von `NSStream` zum Schreiben von Streams ist `NSOutputStream`, die wichtigste Methode `write:maxLength:`. Als Subklasse von `NSStream` erben wir die weiter oben schon verwendete Methode `scheduleInRunLoop:forMode` für eine asynchrone, nicht blockierende Ein- und Ausgabe. Lassen Sie uns anschauen, wie man damit arbeitet.

Zuerst zeigen wir den `CreateFileViewController` an, dem wir einen Button `SPEICHERN` in der Navigationsleiste spendieren, um unseren asynchronen Stream anzulegen.

`FileIO/FileSystemExplorer/Classes/CreateFileViewController.m`

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UIBarButtonItem *saveButton =
        [[UIBarButtonItem alloc]
         initWithBarButtonSystemItem: UIBarButtonSystemItemSave
         target: self action:
             @selector(setUpAsynchronousContentSave)];
    self.navigationItem.rightBarButtonItem = saveButton;
}
```

Als Nächstes müssen wir die Methode `setUpAsynchronousContentSave` schreiben, die dieser Button aufrufen wird. Um zwischen dieser Methode und dem Callback zu vermitteln, brauchen wir drei Instanzvariablen in der Header-Datei: den Ausgabe-Stream, einen Schreibpuffer und einen Zeiger, der auf den Teil des Puffers weist, den wir schreiben wollen.

`FileIO/FileSystemExplorer/Classes/CreateFileViewController.h`

```
NSOutputStream *asyncOutputStream;
NSData *outputData;
NSRange outputRange;
```

Jetzt können wir uns um `setUpAsynchronousContentSave` kümmern. Diese Methode wird `createFileAtPath:` der Klasse `NSFileManager` aufrufen, um die neue Datei zu erzeugen, dann den Stream einrichten und ihn mit der Run Loop verbinden.

FileIO/FileSystemExplorer/Classes/CreateFileViewController.m

```

1  - (void) setUpAsynchronousContentSave {
2      [asyncOutputStream release];
3      [outputData release];
4      outputData = [[fileContentsView.text
5          dataUsingEncoding: NSUTF8StringEncoding] retain];
6      outputRange.location = 0;
7      NSString *newFilePath = [parentDirectoryPath
8          stringByAppendingPathComponent: fileNameField.text];
9      [[NSFileManager defaultManager] createFileAtPath:newFilePath
10         contents:nil attributes:nil];
11     asyncOutputStream = [[NSOutputStream alloc
12         initWithFileAtPath: newFilePath append: NO];
13     [asyncOutputStream setDelegate: self];
14     [asyncOutputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
15         forMode:NSDefaultRunLoopMode];
16     [asyncOutputStream open];
17 }

```

In Zeile 4 wird die NSString-Repräsentation der Text-View in einen NSData-Block aus Bytes konvertiert, den wir für die Stream-APIs nutzen können. Wir initialisieren auch einen NSRange, um uns das nächste Byte zu merken, das in den Stream geschrieben werden soll. Nach dem Erzeugen der Datei legen wir den Stream an (Zeile 11), setzen den Delegate auf **self**, so dass diese Klasse die Callbacks erhält (Zeile 13), verbinden den Stream mit der Run Loop (Zeile 14) und öffnen den Stream (Zeile 16).

Da die asynchrone API von NSStream erbt, ist der Callback der gleiche wie beim Lesen von Daten: `stream:handleEvent:`. Natürlich erhalten wir nun beim Schreiben in den Stream andere Events, wie zum Beispiel `NSStreamEventHasSpaceAvailable`, das anzeigt, dass der Stream bereit ist, einen Schreibbefehl zu akzeptieren.

FileIO/FileSystemExplorer/Classes/CreateFileViewController.m

```

1  - (void) stream: (NSStream *) theStream handleEvent:
2      (NSStreamEvent) streamEvent {
3      NSOutputStream *outputStream = (NSOutputStream *) theStream;
4      BOOL shouldClose = NO;
5      switch (streamEvent) {
6          case NSStreamEventHasSpaceAvailable: {
7              uint8_t outputBuf [1];
8              outputRange.length = 1;
9              [outputData getBytes:&outputBuf range:outputRange];
10             [outputStream write: outputBuf maxLength: 1];
11             if (++outputRange.location == [outputData length]) {
12                 shouldClose = YES;
13             }
14             break;
15         }
16     }
17 }

```

```

15         case NSStreamEventErrorOccurred: {
16             // Fehler ausgeben
17             NSError *error = [theStream streamError];
18             if (error != NULL) {
19                 UIAlertView *errorAlert = [[UIAlertView alloc]
20                     initWithTitle: [error localizedDescription]
21                     message: [error localizedFailureReason]
22                     delegate:nil
23                     cancelButtonTitle:@"OK"
24                     otherButtonTitles:nil];
25                 [errorAlert show];
26                 [errorAlert release];
27             }
28             shouldClose = YES;
29             break;
30         }
31         case NSStreamEventEndEncountered:
32             shouldClose = YES;
33     }
34     if (shouldClose) {
35         [outputStream removeFromRunLoop:[NSRunLoop currentRunLoop]
36             forMode:NSDefaultRunLoopMode];
37         [theStream close];
38
39         // Vorige Seite aktualisieren und View schließen
40         [directoryViewController loadDirectoryContents];
41         [directoryViewController.tableView reloadData];
42         [self.navigationController popViewControllerAnimated:YES];
43     }
44 }

```

Da der Stream nun Bytes akzeptiert, wird in Zeile 6 ein Puffer des Typs `uint8_t` angelegt, der für `write:maxLength:` der Klasse `NSOutputStream` gebraucht wird. Wir haben hier die Pufferlänge 1 genutzt, um das Beispiel einfacher zu halten (es sind weniger Prüfungen notwendig). Dann kopieren wir ein Byte aus dem `NSData` in Zeile 8 in diesen Puffer und schreiben das Byte in Zeile 9 in den Stream. Wenn wir das Ende der zu schreibenden Daten erreicht haben, setzen wir in Zeile 10 ein Flag, das auch bei Fehlern und dem Ende des Streams genutzt wird. Tritt einer dieser Fälle ein, deregistrieren wir den Stream bei der Run Loop (Zeile 35) und schließen ihn (Zeile 37). Schließlich bringen wir das GUI auf Vordermann, indem wir dafür sorgen, dass der `DirectoryViewController`, der diesen `CreateFileViewController` erzeugt hat, den Inhalt des Verzeichnisses neu lädt und sein GUI aktualisiert. Dann navigieren wir zurück zum `DirectoryViewController` (Zeilen 40–42).

Somit haben wir alle Funktionen für den `DirectoryViewController` implementiert. Sie können Verzeichnisse und Dateien erstellen, sie in den Verzeichnis-Views anzeigen lassen und die so angelegten Dateien antippen, um sich deren Inhalt anzeigen zu lassen. Vielleicht ist Ihnen auch das unterschiedliche Verhalten zwischen dem Simulator und dem eigentlichen Gerät aufgefallen. Im Simulator können Sie Dateien und Verzeichnisse innerhalb des Anwendungs-Bundles erzeugen, auf dem iPhone selbst jedoch nicht. Tatsächlich sollten Sie gar nicht erst versuchen, Ihre Daten außerhalb der Ordner Documents und tmp schreiben zu wollen. Aber in diesen Ordnern dürfen Sie beliebige Verzeichnisstrukturen und Dateiinhalte erzeugen.

8.8 Eigenschaftslisten und NSCoder

Nachdem wir beschrieben haben, wie einfach es ist, Objekte vom Typ `NSString` und `NSData` wegzuschreiben oder das Ganze manuell mit einem `NSOutputStream` zu erledigen, werden Sie sich vielleicht fragen, ob damit alle Wege aufgeführt sind, wie Sie Ihre Daten persistieren können. Nun, nicht ganz. Es gibt zwei weitere Optionen beim Schreiben strukturierter Daten, die wir kurz anreißen werden.

Sie haben `.plist`-Dateien schon in Abschnitt 3.4, *Arbeiten mit Xcode und dem Interface Builder*, auf Seite 30 kennengelernt, als wir die Datei `Info.plist` vorstellten, die die Grundeinstellungen Ihrer Anwendung enthält. Diese *Eigenschaftslisten* (Property Lists) sind serialisierte Darstellungen strukturierter Anwendungsdaten.¹⁰ Cocoa unterstützt Eigenschaftslisten aus folgenden Klassen:

- `NSNumber`
- `NSString`
- `NSData`
- `NSDate`
- `NSArray`
- `NSDictionary`

¹⁰ Im iPhone OS 2.x bestand die Datei `Info.plist` der Anwendung aus für Menschen lesbarem XML, aber in 3.0 hat sich das Format geändert.

Die Strukturen in Eigenschaftslisten können beliebig tief verschachtelt sein, so dass Sie zum Beispiel ein Array mit Dictionaries aus String-Datum-Mappings nutzen und dies mit wenigen Zeilen Code persistieren könnten:

```
xmlData = [NSPropertyListSerialization
            dataFromPropertyList: somePlistObject
            format:NSPropertyListXMLFormat_v1_0
            errorDescription:&error];
[xmlData writeToFile:somePath atomically:YES]
```

Genauso einfach ließe sich diese Struktur mit der Methode `propertyListFromData:mutabilityOption:format:errorDescription:` zurück in den Speicher lesen.

Und wenn Sie Objekte haben, die nicht aus den von den Eigenschaftslisten unterstützten Typen bestehen, können Sie das Protokoll `NSCoding` implementieren, das es den diversen Subklassen von `NSCoder` erlaubt, den Zustand des Objekts abzulegen. Diese Art von Vorgehen ist eher für Unternehmensanwendungen interessant, aber die Klassen sind vorhanden, falls Sie sie auf dem iPhone nutzen wollen. Weitere Informationen zu diesem Ansatz finden Sie in Apples *Archives and Serialization Programming Guide for Cocoa* [App08a]. Wir nutzen diese Technik hier in Abschnitt 13.7, *Spiellogik im Netz*, auf Seite 308 mit der `NSCoder`-Subklasse `NSKeyedArchiver`, um Name/Wert-Paare in ein `NSData` zu stecken und sie per Netz zu verschicken.

Kapitel 9

Einstellungen

Ihre iPhone-Anwendung wird immer wieder gestartet und auch wieder beendet. Jedes Mal, wenn der Anwender die Home-Taste drückt oder einen eingehenden Anruf erhält, wird Ihre Anwendung sofort beendet. Ihre Anwender werden es Ihnen danken, wenn die Dinge beim nächsten Aufruf der App genauso sind wie bei der letzten Ausführung – dazu gehört auch das Sichern und Wiederherstellen der Einstellungen.

Es gibt letztlich zwei Vorgehensweisen, um auf dem iPhone Einstellungen zu verwalten. Diese beiden Wege schließen sich technisch nicht gegenseitig aus, aber sie sind, auch in den Ergebnissen, so unterschiedlich, dass man das durchaus so sehen kann. Es ist sehr wichtig, einen Ansatz zu wählen und dann auch dabei zu bleiben, denn wenn Sie das nicht tun, befinden sich Ihre Einstellungen an zwei völlig verschiedenen Stellen, und das schafft für Sie und Ihre Anwender doppelt so viel Arbeit.

Wir werden mit einem Ansatz beginnen, der für Sie mehr Arbeit bedeutet, es dem Anwender aber einfacher macht: Ihre Anwendung wird ihre eigenen Einstellungen verwalten und eine Benutzerschnittstelle dafür anbieten.

9.1 Anzeige einer umklappbaren Einstellungs-View in Ihrer Anwendung

Im Prinzip sollte die Idee, die Einstellungen einer Anwendung selbst zu verwalten, nach dem Lesen von Kapitel 8, *Datei I/O*, auf Seite 143 recht leicht umsetzbar sein, da dort die verschiedenen Formen des Zugriffs auf das Dateisystem des iPhone behandelt wurden. Sie entscheiden sich beim Lesen und Schreiben Ihrer Daten aus einer und in eine Datei

für eine Strategie, und fertig ist die Einstellungs-Persistierungs-Strategie. Wählen Sie diesen Weg, ist es recht sinnvoll, Ihre Einstellungen als `NSDictionary` mit Name/Wert-Paaren abzulegen, das Sie mit `writeToFile:atomically:` speichern und mit `initWithContentOfFile:` wieder einlesen können. Ein großer Vorteil dieses Vorgehens ist, dass die gespeicherten Dateien Eigenschaftslisten sind (siehe Abschnitt 8.8, *Eigenschaftslisten und NSCoder*, Seite 178), so dass die Werte Ihrer Einstellungen `NSArray`s oder eingebettete `NSDictionary`-Objekte sein können (natürlich lassen sich auch die einfacheren Member-Typen von Eigenschaftslisten nutzen – `NSString`, `NSDate`, `NSNumber` oder `NSData`).



Abbildung 9.1: Die Main View umklappen, um eine Einstellungs-View anzuzeigen

Tatsächlich ist das Verwalten der Einstellungen in Ihrer eigenen Anwendung weniger eine Frage des Abspeicherns, als mehr eine der Präsentation. Wenn Sie die Benutzereinstellungen verwalten wollen, brauchen Sie ein GUI, damit der Anwender Zugriff darauf hat. Aber wo soll das bei dem begrenzten Platz eines iPhone-Bildschirms angezeigt werden?

Wenn Sie all die Anwendungen gebraucht haben, die das iPhone dabei hat, kennen Sie die Antwort – auf der „Rückseite“ Ihrer View. Die ein-

facheren Anwendungen, die dem iPhone und dem iPod touch beiliegen – Wetter und Aktien –, haben einen kleinen Button in der unteren rechten Ecke, der mit einem animierten „Umklappen“ ein Einstellungs-GUI so präsentiert, als würde es sich auf der Rückseite der Main View befinden.

Die Beispielanwendung mit dem Namen `FlippingPreferableClock` zeigt, wie man diese Technik einsetzt (siehe Abbildung 9.1). Diese Anwendung gibt einfach die aktuelle Uhrzeit und den Namen einer Zeitzone aus. Die von der Uhr verwendete Zeitzone und das Format der Uhrzeit sind die Einstellungen, die von der Beispielanwendung geändert werden können. Die Main View besteht nur aus diesen beiden UILabels und einem UIButton, mit dem zur Einstellungs-View gewechselt werden kann.

Xcode stellt ein Template für solche Anwendungen bereit, das wir auch für dieses Beispiel nutzen werden. Wenn Sie eine Utility-Anwendung erstellen, erzeugt Xcode für Sie eine große Zahl an Klassen und Nibs:

- Eine klassische Anwendungs-Delegate-Klasse mit dem `MainViewController` als Eigenschaft.
- Eine „Main View“ für die eigentliche Anwendungs-View, die beim Starten erscheint und die eigentliche Funktionalität enthält. Diese wird durch eine eigene Klasse `MainView` implementiert. Dazu gibt es einen `MainViewController` und eine .nib-Datei `MainView.xib`. Die View enthält einen INFO-Button in der unteren rechten Ecke, der mit einer Action `showInfo` verbunden ist. Diese Methode lädt den `FlipsideViewController` aus einem Nib, setzt die Eigenschaft `delegate` des „umgeklappten“ View-Controller auf den `MainViewController` und die in 3.0 neue Eigenschaft `modalTransitionStyle` so, dass beim Animieren des View-Wechsels „umgeblättert“ wird. Dann wird der `Flipside View-Controller` modal angezeigt.
- Eine „Flipside View“ für das Konfigurations-GUI, auch wieder bestehend aus einer eigenen `FlipsideView`-Klasse, einem `FlipsideViewController` und einer `FlipsideView.xib`. Die View besitzt einen `UINavigationController` `FERTIG`, der die Methode `done` aufruft. Dieser Methode wird der `delegate` übergeben (also der `MainViewController`), und sie ruft dessen `flipsideViewControllerDidFinish` auf, damit der `Main View-Controller` weiß, dass er auf die eventuell geänderten Einstellungen reagieren und zudem den `Flipside View-Controller` entfernen muss.
- Die anderen üblichen Dateien, wie zum Beispiel `main.m`, die zum Starten der Anwendung benötigt werden.

Ohne selbst eine Zeile Code schreiben zu müssen, können wir ein Anwendungsprojekt `FlippableClock` erstellen, **BUILD AND RUN** anklicken und die View über die Buttons **INFO** und **FERTIG** hin- und herklappen. Nachdem wir damit ein bisschen gespielt haben, wollen wir die eigentliche Funktionalität einbauen.

Achtung: Änderungen an der Utility Application im iPhone 3.0

Die frühesten Betas des iPhone SDK besaßen keinen Code für umklappbare Anwendungen und erwarteten, dass die Entwickler ihre eigene Animation einbauten. Glücklicherweise enthielt das finale 2.0-SDK ein Utility Application Template mit einem `RootViewController`, der zwischen den `Main` und `Flipside View-Controllern` wechselte. Beide Vorgehen wurden in früheren Betaversionen dieses Buchs behandelt.

Im iPhone SDK 3.0 erhielt der `UINavigationController` eine neue Eigenschaft `modalTransitionStyle`, mit der das Utility Application Template direkt zwischen zwei View-Controllern wechseln kann (über `presentModalViewController:animated:` und `dismissModalViewControllerAnimated:`), ohne die Klasse `RootViewController` zu benötigen. Wir nutzen diesen neuen, einfacheren Ansatz in diesem Kapitel. Sie werden die neuen Wechselstile immer dann nützlich finden, wenn Sie einen modalen View-Controller anzeigen müssen.

Allerdings lassen sich Anwendungen, die den `modalTransitionStyle` nutzen – also auch alle Apps, die mit dem neuen Utility Application Template erstellt wurden – nicht so kompilieren, dass sie auf dem iPhone OS 2.x laufen. Wir haben die alten Beispiele im herunterladbaren Code als `FlippingPreferableClockFor2.0` hinterlegt, falls Sie sie benötigen.

9.2 Einstellungen verwalten

Wir haben gesagt, dass der I/O-Teil beim Verwalten der Einstellungen der einfachere Teil ist, zumindest einfacher als das Verwalten der umklappbaren Views. Daher wollen wir uns diesem Teil auch als Erstes vornehmen. In `MainViewController.h` werden wir per **#define** Schlüssel für ein Einstellungs-Dictionary und ein paar Standardwerte definieren sowie im **@interface**-Abschnitt ein `NSMutableDictionary` deklarieren, in dem die Einstellungen zur Laufzeit und ein Pfad auf die Einstellungsdatei abgelegt sind:

```
FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.h
```

```
#define TWENTY_FOUR_HOUR_PREF_KEY @"24HourDisplay"
#define TIME_ZONE_PREF_KEY @"TimeZone"
#define DEFAULT_TWENTY_FOUR_HOUR_PREF @"YES"
#define DEFAULT_TIME_ZONE_PREF @"Europe/Berlin"

@interface MainViewController : UIViewController
    <FlipsideViewControllerDelegate>{

    NSMutableDictionary *clockPrefs;
    NSString *prefsFilePath;
```

Damit können wir einen Pfad auf eine Einstellungsdatei im Anwendungs-Bundle erstellen und versuchen, den Inhalt der Datei als NSMutableDictionary zu laden. Gibt es die Datei noch nicht, erstellen wir ein Dictionary aus den Standardwerten:

```
FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m
```

```
- (void) initPrefsFilePath {
    NSString *documentsDirectory =
    [NSHomeDirectory() stringByAppendingPathComponent:@"Documents" ];
    prefsFilePath = [documentsDirectory stringByAppendingPathComponent:
        @"flippingprefs.plist" ];
    [prefsFilePath retain];
}

- (void) loadPrefs {
    if (prefsFilePath == nil)
        [self initPrefsFilePath];
    if ([[NSFileManager defaultManager] fileExistsAtPath: prefsFilePath]) {
        clockPrefs = [[NSMutableDictionary alloc]
            initWithContentsOfFile: prefsFilePath];
    }
    else {
        clockPrefs = [[NSMutableDictionary alloc] initWithCapacity: 2];
        [clockPrefs setObject: DEFAULT_TIME_ZONE_PREF
            forKey: TIME_ZONE_PREF_KEY];
        [clockPrefs setObject: DEFAULT_TWENTY_FOUR_HOUR_PREF
            forKey: TWENTY_FOUR_HOUR_PREF_KEY];
    }
    NSString *prefTimeZone = [clockPrefs objectForKey: TIME_ZONE_PREF_KEY];
    BOOL uses24Hour = [(NSString*)
        [clockPrefs objectForKey: TWENTY_FOUR_HOUR_PREF_KEY] boolValue];
    [self setClockToTimeZoneName: prefTimeZone uses24Hour: uses24Hour];
}
```

Ihnen fällt vielleicht auf, dass dieser Code mit einem Aufruf der sehr praktischen Methode `setClockToTimeZoneName:uses24Hour:` endet. Das nutzen wir, um das GUI sowohl beim Starten (durch ein Hinzufügen von `[self loadPrefs];` zu Ihrer Methode `viewDidLoad`) als auch beim Ändern der Einstellungen auf der Flipside View zu aktualisieren.

Natürlich haben wir das GUI noch gar nicht erstellt, daher wollen wir uns ihm als Nächstes zuwenden.

Das Template stellt .h- und .m-Dateien für `MainView` und `FlipsideView` bereit, wodurch Sie beide ganz nach eigenem Wunsch rendern können, aber für dieses Beispiel müssen wir nur den View-Controller und die .xib-Dateien anpassen. Erstellen Sie in `MainViewController.h` `IBOutlet`s für `UILabel`s mit den Namen `timeLabel` und `timeZoneLabel`. Klicken Sie doppelt auf `MainView.xib` (nicht auf `MainWindow.xib`, die nur eine Vorlage ist), um die Datei im IB zu öffnen. Ziehen Sie zwei Labels auf die View, passen Sie deren Stil und Größe nach Wunsch an und verbinden Sie sie mit den eben definierten Outlets.

Nachdem wir nun `IBOutlet`s zu `timeLabel` und `timeZoneLabel` haben, können wir sie durch die weiter oben erwähnte Methode `setClockToTimeZoneName:uses24Hour:` aktualisieren. Dazu fügen Sie der Header-Datei die Instanzvariablen `NSString* timeZoneName` und `NSDateFormatter *clockFormatter` hinzu und fügen die Methode in der Implementierungsdatei noch vor `loadPrefs` ein, durch die sie aufgerufen wird:

FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m

```
-(void) setClockToTimeZoneName: (NSString*) tz uses24Hour: (BOOL) u24h
{
    [timeZoneName release];
    [tz retain];
    timeZoneName = tz;
    // Time Formatter mit 12h-/24h- und
    // Zeitzoneneinstellung versehen
    if (clockFormatter == nil) {
        clockFormatter = [[NSDateFormatter alloc] init];
    }
    // Formate finden Sie unter
    // http://unicode.org/reports/tr35/tr35-4.html#Date_Format_Patterns
    [clockFormatter setTimeZone: [NSTimeZone timeZoneWithName: tz]];
    if (u24h)
        [clockFormatter setDateFormat: @"HH:mm:ss" ];
    else
        [clockFormatter setDateFormat: @"h:mm:ss a" ];
}
```

Der `NSDateFormatter` erledigt hier den größten Teil der Arbeit: Er erhält den Namen einer Zeitzone und einen Formatierungs-String, um den richtigen Text für das `timeLabel` zu erzeugen.¹

¹ Beachten Sie, dass sich der `NSDateFormatter` in bestimmten Verhaltensweisen an den lokalen und Systemeinstellungen orientiert. Vor allem wenn Sie in den Einstellungen eine 24-Stunden-Anzeige gewählt haben, ignoriert der `NSDateFormatter` jeglichen Format-String, der durch Ihre Anwendung gesetzt wird, und gibt *immer* die Stunden im 24h-System aus.

Als Letztes müssen wir zum Anzeigen der Uhrzeit noch den Formatter benutzen, um den Wert des `timeLabel` zu setzen. Fügen Sie die folgende Methode `updateClockView` hinzu:

FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m

```
- (void) updateClockView {
    if (clockFormatter == NULL) {
        timeLabel.text = @"";
        timeZoneLabel.text = @"";
        return;
    }
    NSDate *dateNow = [NSDate date];
    timeLabel.text = [clockFormatter stringFromDate: dateNow];
    timeZoneLabel.text = timeZoneName;
}
```

Rufen Sie diese Methode am Ende von `viewDidLoad` durch `[self updateClockView];` auf. Wenn Sie Ihren Code jetzt bauen und ausführen, werden Sie die aktuelle Uhrzeit in der Standardzeitzone auf Ihrem Bildschirm angezeigt bekommen. Bisher haben wir auch noch nichts für die Einstellungen gemacht. Lassen Sie uns also die View umklappen, so dass der Anwender die Einstellungen bearbeiten kann.

9.3 Einstellungen ändern und aktualisieren

Für die „Rückseite“ braucht Ihr `FlipsideViewController.h` `IBOutlet`-Eigenschaften für einen `UISwitch` namens `twentyFourHourSwitch` und eine `UIPickerView` namens `timeZonePicker` (vergessen Sie nicht, sie auch zu synthetisieren). Zudem benötigen Sie eine Instanzvariable vom Typ `NSArray` mit dem Namen `timeZoneNames`, um den Picker zu befüllen. Klicken Sie doppelt auf die Datei `FlipsideView.xib`, um sie im IB zu bearbeiten, fügen Sie den Schalter sowie den Picker hinzu, verbinden Sie sie mit den Outlets und ergänzen Sie ein paar passende Labels.

Ein Picker entspricht in vielen Dingen einer Tabelle, daher müssen Sie die Outlets `delegate` und `datasource` des Pickers mit File's Owner verbinden. Wir werden diese Verbindungen nutzen, damit die Flipside ihren Picker mit den bekannten Zeitzonen befüllen kann. Fügen Sie in `FlipsideViewController.h` am Ende der `@interface`-Anweisung die Protokolldeklaration `<UIPickerViewDataSource, UIPickerViewDelegate>` hinzu. In der Implementierungsdatei müssen Sie jetzt an eine Liste mit den bekannten Zeitzonen kommen. Dafür gibt es in der Klasse `NSTimeZone` die Methode `knownTimeZoneNames` (die allerdings die Namen der Zeitzonen leider nur auf Englisch liefert). Das Ganze verpacken wir in einer Methode `loadTimeZoneNames`, die Sie aus `viewDidLoad` aufrufen können.

FileIO/FlippingPreferableClockFor3.0/Classes/FlipsideViewController.m

```
- (void) loadTimeZoneNames {
    if (timeZoneNames)
        return;
    NSMutableArray *unsortedTimeZoneNames =
        [NSTimeZone knownTimeZoneNames];
    timeZoneNames =
        [unsortedTimeZoneNames sortedArrayUsingSelector:
         @selector(caseInsensitiveCompare:)];
    [timeZoneNames retain];
}
```

Ist das Array gefüllt, können Sie die benötigten Data Source- und Delegate-Methoden für den Zeitzonen-Picker implementieren:

FileIO/FlippingPreferableClockFor3.0/Classes/FlipsideViewController.m

```
- (NSInteger) pickerView: (UIPickerView*) pickerView
    numberOfRowsInComponent: (NSInteger) component {
    return [timeZoneNames count];
}

- (NSInteger) numberOfComponentsInPickerView: (UIPickerView *) pickerView
{
    return 1;
}

- (NSString *) pickerView: (UIPickerView *) pickerView
    titleForRow: (NSInteger) row
    forComponent: (NSInteger) component {
    return (NSString*) [timeZoneNames objectAtIndex: row];
}
```

Jetzt haben wir einen Picker, der alle bekannten Zeitzonen anzeigen kann. Wenn wir zur Flipside View navigieren, sollten wir den Picker natürlich auf die aktuelle Zeitzone der Main View setzen und auch den 24h-Schalter entsprechend voreinstellen. Dazu machen Sie `clockPrefs` von `MainViewController` zu einer Eigenschaft und lassen diese von der `Flipside` auslesen, damit sie ihren Status korrekt setzen kann. Sie erledigen das am besten in der Methode `viewDidLoad` des `FlipsideViewController` nach `[self loadTimeZoneNames];`. Sie müssen auch noch ein `#import "MainViewController.h"` einfügen, da Sie `delegate` auf einen `MainViewController` casten und dessen Eigenschaft `clockPrefs` ansprechen.

FileIO/FlippingPreferableClockFor3.0/Classes/FlipsideViewController.m

```
[self loadTimeZoneNames];
// Werte aus der Main View übernehmen
MainViewController *mainVC = (MainViewController*) delegate;
NSString *timeZone =
    [mainVC.clockPrefs objectForKey:TIME_ZONE_PREF_KEY];
```

```

NSString *twentyFourHourPref =
    [mainVC.clockPrefs objectForKey:TWENTY_FOUR_HOUR_PREF_KEY];
[timeZonePicker
    selectRow: [timeZoneNames indexOfObject:timeZone]
    inComponent: 0 animated:NO];
twentyFourHourSwitch.on = [twentyFourHourPref boolValue];

```

Die Anwendung wird also gestartet, und die Zeitzone sowie die 24h-Einstellung, bei denen es sich um Instanzvariablen im MainViewController handelt, werden vom Dateisystem geladen oder mit Standardwerten belegt. Wenn man zur Flipside View navigiert, werden die aktuellen Werte genutzt, um den initialen Status des Zeitzones-Pickers sowie des 24h-Schalters zu setzen. Lassen Sie uns nun annehmen, dass der Anwender einen oder beide Werte ändert. Wenn er dann auf FERTIG tippt, müssen die neuen Werte an den MainViewController übermittelt werden. Wo sollten wir das am besten machen?

Das Template erzeugt den FlipsideViewController mit einer Methode `done`, die im MainViewController die Methode `flipsideViewControllerDidFinish` aufruft. Wir könnten unseren Update-Code also in beide Klassen einbauen. Das führt zu einer interessanten Designentscheidung: Sollte der MainViewController seine Einstellungen auch von der Flipside View verändern lassen können (beziehungsweise von beliebigem Code), oder sollte er die Werte aus der Flipside View holen? Um möglichst wenig Informationen herauszugeben, werden wir den zweiten Ansatz nutzen und in `flipsideViewControllerDidFinish` des MainViewController die Werte aus der Flipside auslesen.

Anstatt aber nun den Schalter und den Picker nach außen sichtbar zu machen, wollen wir in `FlipsideViewController.h` zwei Getter-Methoden mit den Namen `selectedTimeZone` und `uses24Hour` (mit den folgenden Signaturen) deklarieren und in `FlipsideViewController.m` implementieren.

FileIO/FlippingPreferableClockFor3.0/Classes/FlipsideViewController.m

```

- (NSString*) selectedTimeZone {
    return [timeZoneNames objectAtIndex:
        [timeZonePicker selectedRowInComponent: 0]];
}
- (BOOL) uses24Hour {
    return twentyFourHourSwitch.on;
}

```

Jetzt kann der MainViewController seine Anzeige aktualisieren, indem er die gewählten Werte von der Flipside holt, Uhrzeit und Zeitzone in der Main View anpasst und die geänderten Einstellungen auf der Platte speichert. Standardmäßig enthält `flipsideViewControllerDidFinish`:

eine Zeile, um den Flipside View-Controller zu verwerfen. Wir werden unseren Code direkt davor einfügen.

FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m

```
- (void)flipsideViewControllerDidFinish:
    (FlipsideViewController *)controller {
    timeZoneName = [controller selectedTimeZone];
    BOOL uses24Hour = [controller uses24Hour];
    NSString *selected24HourDisplayS = uses24Hour ? @"YES" : @"NO" ;
    [clockPrefs setObject: timeZoneName forKey: TIME_ZONE_PREF_KEY];
    [clockPrefs setObject: selected24HourDisplayS
        forKey: TWENTY_FOUR_HOUR_PREF_KEY];
    // Einstellungen im Documents-Ordner speichern
    [self savePrefs];

    // Anzeige anpassen
    [self setClockToTimeZoneName: timeZoneName uses24Hour: uses24Hour];
    [self updateClockView];

    // Aus dem Template
    [self dismissModalViewControllerAnimated:YES];
}
```

Diese Methode nutzt die Methoden `selectedTimeZone` und `uses24Hour`, die wir für den `FlipsideViewController` geschrieben haben, und trägt die Werte in den `clockPrefs` ein. Bevor dann die Zeitzone der Uhr aktualisiert und das Umklappen vollzogen wird, ruft sie die Methode `savePrefs` auf, um die Eigenschaften auch im Dateisystem abzulegen. Da die Eigenschaften in einem `NSDictionary` gespeichert sind, reicht ein Einzeiler, um sie als `plist` zu sichern.

FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m

```
- (void) savePrefs {
    [clockPrefs writeToFile: prefsFilePath atomically: YES];
}
```

Und damit gibt es für unsere lokal verwalteten Einstellungen einen vollständigen Kreislauf – aus der Datei in den `MainViewController`, in dem sie zum Formatieren von `timeLabel` und `timeZoneLabel` genutzt werden – von dort zum `FlipsideViewController`, wo sie den initialen Zustand des Pickers und Schalters in der Flipside View bestimmen – dann zurück zum `MainViewController`, in dem sie zum Aktualisieren der Anzeige genutzt und wieder im Dateisystem abgelegt werden.

Durch das leicht zu speichernde `NSDictionary` hatten wir natürlich auch nicht viel Arbeit beim Speichern oder Laden der Einstellungen und konnten unsere Kreativität beim liebevollen Anordnen der Felder auf der Anwendung freien Lauf lassen. Wenn Sie ein anderes Schema

zum Persistieren Ihrer Einstellungen nutzen wollen, wie zum Beispiel Ihr eigenes Dateiformat, die `NSUserDefaults` (siehe Abschnitt 9.6, *Einstellungen laden, die in der System-App konfiguriert wurden*, Seite 200) oder sogar eine Datenbank (siehe Kapitel 10, *Die Datenbank SQLite*, Seite 205), müssen Sie nur Änderungen in `savePrefs` und `loadPrefs` vornehmen.

9.4 Abstecher: Sekündliches Aktualisieren der Uhr

Es hat eigentlich nichts mit den Einstellungen zu tun, aber die Technik, mit der das Uhrzeit-Label konstant aktualisiert wird, sollte man kennen. Der Trick ist, einen `NSTimer` zu erzeugen – ein Objekt, das eine Methode regelmäßig aufruft –, um den Inhalt des Uhrzeit-Labels immer wieder zu erstellen und zu aktualisieren.

Zunächst brauchen Sie eine Methode, die regelmäßig aufgerufen werden soll. Wir haben bereits eine, da wir einfach immer wieder `updateClockView` aufrufen können.

Als Nächstes erstellen Sie einen `NSTimer`, um diese Methode mit einer gewissen Wiederholrate (in Sekunden) aufzurufen. Deklarieren Sie die Instanzvariable `clockViewUpdateTimer` in `MainViewController.h` und fügen Sie dann in der Implementierungsdatei eine Methode `startClock` ein. Diese Methode muss vor `viewDidLoad` stehen, da sie von dort aus aufgerufen wird.

FileIO/FlippingPreferableClockFor3.0/Classes/MainViewController.m

```
-(void) startClock {
    // Da der erste Callback des Timers erst nach dem Intervall
    // kommt, ein einzelnes Refresh durchführen
    [self updateClockView];
    // Jetzt den Timer so einrichten, dass er updateClockView
    // regelmäßig aufruft
    clockViewUpdateTimer = [NSTimer
        scheduledTimerWithTimeInterval:0.2
        target:self
        selector:@selector(updateClockView)
        userInfo:NULL
        repeats:YES];
}
```

Damit werden die Labels zunächst einmal direkt aktualisiert, und danach wird ein `NSTimer` so eingerichtet, dass er immer wieder das Gleiche tut. Die Klassenmethode `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` ist ein ganz schöner Brocken, aber lassen Sie uns trotzdem alles aufführen, was wir für regelmäßige Callbacks

brauchen: wie oft ein Callback kommen soll, welche Methode aufzurufen ist (eine Kombination aus dem Objekt `target` und der Methodensignatur `selector`), ein Objekt, das dem Callback übergeben wird (und das wir hier nicht benötigen), und die Entscheidung, ob `NSTimer` den Callback immer wieder aufrufen soll oder nur ein einziges Mal. Beachten Sie, wie Sie sich auf eine Methode über das Schlüsselwort `@selector` beziehen können. Sie erhalten so den Selector-Typ, den Sie für den Parameter `selector` benötigen.

Haben Sie diese Methode hinzugefügt, ersetzen Sie den Aufruf `[self updateClockView];` in `viewDidLoad` durch einen Aufruf zum Starten der Uhr: `[self startClock];`. Damit wird die View sofort aktualisiert und dann regelmäßig wieder aufgefrischt. Bauen Sie die Anwendung erneut und lassen Sie sie wieder laufen – die Uhr sollte nun sekundlich die aktuelle Uhrzeit anzeigen.

9.5 Die Systemanwendung für Einstellungen nutzen

Die zweite Option für Einstellungen ist, sie gar nicht in der eigenen Anwendung zu verwalten. Stattdessen soll sich die Einstellungsanwendung des iPhone OS darum kümmern.

Bei diesem Vorgehen stellen Sie einfach eine Bundle-Datei bereit, in der die vom Anwender konfigurierbaren Einstellungen beschrieben sind – ihre Typen, die möglichen Werte, die vom Benutzer lesbaren Strings und so weiter. Die Einstellungsanwendung stellt ein GUI dafür bereit – Sie müssen die Werte in Ihrer Anwendung einfach nur einlesen. In Abbildung 9.2 sehen Sie, wie die Einstellungsanwendung die beiden Einstellungen für die Uhrzeitanwendung bereitstellt, die durch ein Einstellungs-Bundle definiert sind.

Wir werden dieses Vorgehen in einer umgestalteten Uhrzeitanwendung umsetzen. Erstellen Sie in Xcode ein View-basiertes Anwendungsprojekt mit dem Namen `BundlePreferableClock`. Auch diese Anwendung wird eine regelmäßig aktualisierte Uhr anzeigen, bei der der Benutzer einstellen kann, ob eine 12h- oder eine 24h-Uhr genutzt werden soll und welche Zeitzone zu verwenden ist. Da unsere Anwendung nicht für ein Einstellungs-GUI verantwortlich ist, brauchen wir kein Umklappen, sondern nur eine einzelne Main View.



Abbildung 9.2: Einstellungen, wie sie durch `Settings.Bundle` definiert wurden

Bei dieser Anwendung werden wir mit den Einstellungsdaten beginnen und die Benutzerschnittstelle erst am Ende bauen.

Lassen Sie uns damit beginnen, ein Symbol der Größe 29 x 29 im PNG-Format zu erzeugen, das in der Einstellungsanwendung angezeigt werden kann. Diese Datei *muss* den Namen `Icon-Settings.png` haben – wenn sie nicht gefunden wird, skaliert das iPhone OS das Symbol Ihrer Anwendung (`Icon.png`) und nutzt dieses. Klicken Sie mit gedrückter `Ctrl`-Taste (oder mit der rechten Maustaste) auf die Gruppe „Resources“, wählen Sie `ADD EXISTING FILES` und dann `Icon-Settings.png`. Achten Sie darauf, dass Sie im zweiten Dialogfenster „Copy item into destination group’s folder (if needed)“ markiert haben. In den zum Buch herunterladbaren Beispielen haben wir unser Symbol mit einem blauen Hintergrund versehen, um es vom eigentlichen Anwendungssymbol zu unterscheiden.



Abbildung 9.3: Ein neues Einstellungs-Bundle erstellen

Die Datei Settings.bundle erstellen

Als Nächstes müssen wir angeben, welche Einstellungen die Systemanwendung verwalten soll. Wählen Sie das oberste Projektsymbol in GROUPS & FILES aus und anschließend den Menüeintrag NEW FILE. Im sich öffnenden Fenster entscheiden Sie sich für das SETTINGS BUNDLE-Template, wie in Abbildung 9.3 zu sehen.

Damit erhalten Sie in Ihren GROUPS & FILES den Eintrag Settings.bundle. Das ist ein Ordner, der standardmäßig zwei Elemente enthält: eine Datei Root.plist und einen Lokalisierungsordner en.lproj. Das ist der Minimalinhalt eines Einstellungs-Bundles.

Schauen Sie sich per Doppelklick Root.plist an. Sie erhalten einen Editor wie den in Abbildung 9.4. Die mitgelieferten Einträge enthalten eine StringsTable-Referenz, um Lokalisierungen zu finden, und ein Array namens PreferenceSpecifiers mit vier Einträgen. Das sind Beispiele, an denen Sie das Format des Einstellungs-Bundles ablesen können. Sie sollten sie allerdings löschen, wenn Sie Ihre eigenen Einstellungen eintragen wollen.

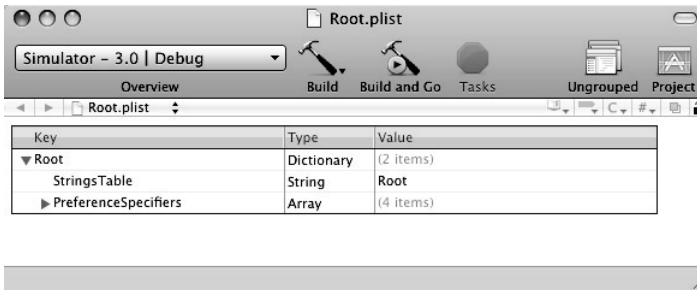


Abbildung 9.4: Initialer Inhalt der Datei Root.plist des Einstellungs-Bundles

Jedes Element des Arrays PreferenceSpecifiers ist ein Dictionary mit Name/Wert-Paaren, die eine Einstellung definieren. Der Inhalt des Verzeichnisses ist ein bisschen von der Art der Einstellung abhängig, aber meist sind folgende Schlüssel dabei:

- **Type:** Eine von mehreren möglichen Konstanten, die den Typ der Einstellung beschreiben und damit auch festlegen, wie der Anwender sie konfiguriert: ein Schalter für Boolesche Werte, ein Textfeld für Strings und so weiter.
- **Key:** Der String, der in der Anwendung genutzt wird, um zur Laufzeit auf die Einstellung zuzugreifen.
- **DefaultValue:** Der Wert, der als Standard angezeigt werden soll.

Neben diesen Schlüsseln gibt es abhängig vom Typ der Einstellung noch andere Schlüssel/Wert-Paare. Die verfügbaren Typen sind:

- **PSTextFieldSpecifier** zeigt ein editierbares Textfeld mit einem optionalen Titel an.
- **PSTitleValueSpecifier** zeigt einen nicht editierbaren Titel für Werte an, auf die nur lesend zugegriffen werden kann.
- **PSToggleSwitchSpecifier** gibt einen An-/Ausschalter für Boolesche Werte aus.
- **PSSliderSpecifier** nutzt ein Slider Control für eine Einstellung, deren Wert eine Zahl zwischen einem minimalen und einem maximalen Wert ist.
- **PSMultiValueSpecifier** zeigt eine Tabelle möglicher Werte an, aus denen der Anwender einen Eintrag auswählen kann.

- `PSGroupSpecifier` gruppiert andere Einstellungen unter einem gemeinsamen Titel.
- `PSChildPaneSpecifier` bietet einen Link auf eine Unterseite, die in einer eigenen `.plist`-Datei definiert ist.



Joe fragt...

Wann nutze ich eine Flippable View und wann ein Bundle?

Ganz offensichtlich sollten die beiden Einstellungsparadigmen, die in diesem Kapitel vorgestellt werden, nicht gleichzeitig verwendet werden. Denn wenn Sie einen Teil der Einstellungen in einer Flippable View und einen anderen Teil durch die Einstellungsanwendung verwalten lassen, bedeutet das doppelte Arbeit – für Sie und für Ihre Anwender. Welcher Ansatz ist also der richtige?

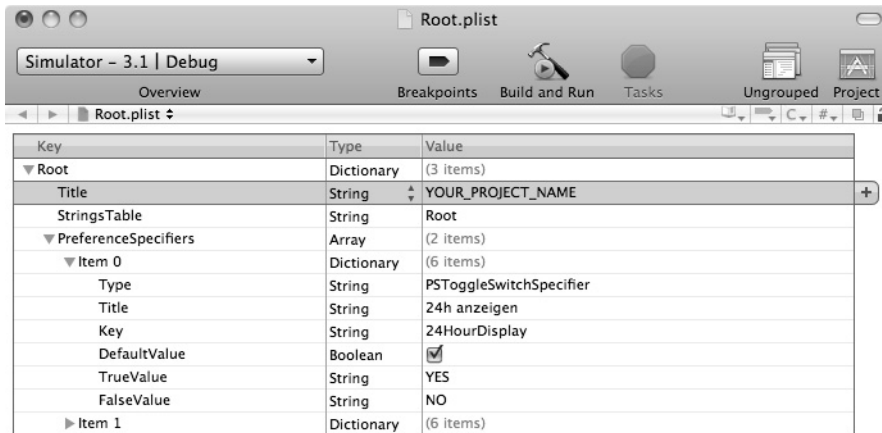
Die von Apple mitgelieferten Standard-Apps könnten das Gefühl aufkommen lassen, dass nur die einfachsten Anwendungen ihre Einstellungen selbst verwalten. Dafür gibt es einen guten Grund. Hat man so wenig Platz auf dem Bildschirm – fünf Buttons am unteren und vielleicht zwei am oberen Rand des Bildschirms –, können Anwendungen wie Mail und Safari nicht auch noch eine Ecke für einen Einstellungs-Button abzwiegen.

Zudem scheint es so zu sein, dass bei den einfachen Anwendungen die Einstellungen häufiger geändert werden. Wenn Sie auf Reisen sind, ändern Sie regelmäßig die Städte, für die Sie sich das Wetter anzeigen lassen. Aber kümmern Sie sich je um die Einstellungen von Safari?

Es mag auf den ersten Blick etwas seltsam erscheinen, die Einstellungen an einem ganz anderen Ort als in der Anwendung abzulegen, aber die Benutzer werden beim Ausführen Ihrer App daran gewöhnt sein. Ist Ihre Anwendung also nicht wirklich simpel oder vielleicht auch ein Spiel mit einem eigenen, sehr unterschiedlichen GUI, werden Sie Ihre Einstellungen eher in einem Bundle verwalten.

Um also ein Einstellungs-Bundle für die Uhr einzurichten, brauchen wir nur zwei Einträge im Array `PreferenceSpecifiers`: einen vom Typ `PSToggleSwitchSpecifier`, um zwischen 12h- und 24h-Anzeigen zu wechseln, und einen vom Typ `PSMultiValueSpecifier`, um eine Zeitzone auszuwählen. Wir können mit dem Schalter beginnen. Löschen Sie die vier Beispieleinträge aus dem Array, indem Sie sie mit der rechten Maustaste anklicken und den Menüeintrag `CUT` auswählen. Dann kli-

cken Sie auf das Array `PreferenceSpecifiers`. Sie werden feststellen, dass in der Zeile ganz rechts ein Button erscheint. Dies ist der ADD-Button, mit dem Sie eine neue Tabellenzeile anlegen. Klicken Sie ihn an, um eine neue Zeile zu erstellen, und setzen Sie dann den Typ der Zeile auf „Dictionary“. Sie können diese Zeile jetzt anklicken, um dem Dictionary Einträge hinzuzufügen. Um die 12h-/24h-Einstellung zu definieren, müssen Sie dem Dictionary die folgenden Schlüssel/Wert-Paare hinzufügen (siehe auch Abbildung 9.5).



Key	Type	Value
▼ Root	Dictionary	(3 items)
Title	String	YOUR_PROJECT_NAME
StringsTable	String	Root
▼ PreferenceSpecifiers	Array	(2 items)
▼ Item 0	Dictionary	(6 items)
Type	String	PSToggleSwitchSpecifier
Title	String	24h anzeigen
Key	String	24HourDisplay
DefaultValue	Boolean	☒
TrueValue	String	YES
FalseValue	String	NO
▶ Item 1	Dictionary	(6 items)

Abbildung 9.5: Eine Bool-Einstellung anlegen

- **Type:** Der String `PSToggleSwitchSpecifier`.
- **Title:** Ein kurzer String für den Anwender, zum Beispiel „24h anzeigen“.
- **Key:** Ein String wie „24HourDisplay“, den Ihre Anwendung als Dictionary-Schlüssel nutzen kann, um diesen Einstellungswert zu finden.
- **DefaultValue:** Ein Boolescher Standardwert.
- **TrueValue:** Der String, den Ihre Anwendung erhalten soll, wenn der Benutzer diese Einstellung angeschaltet hat.
- **FalseValue:** Der String, den Ihre Anwendung erhalten soll, wenn der Benutzer diese Einstellung abgeschaltet hat.

Das Bereitstellen der Zeitzonen ist aufwendiger. Die Einträge, die der Picker in der FlippingPreferableClock (siehe Abschnitt 9.1, *Anzeige einer umklappbaren Einstellungs-View in Ihrer Anwendung*, Seite 181) angezeigt hat, wurden über die Methode `knownTimeZoneNames` der Klasse `NSTimeZone` ausgelesen. Dadurch wurden mehrere Hundert Zeitzone-einträge erstellt. Wie Sie sich vorstellen können, ist die Verwendung des Point-and-Click-Interface von Xcode nicht gerade hilfreich, wenn man all diese Tabelleneinträge einfügen möchte.

Glücklicherweise ist die Eigenschaftsliste des GUI nicht der einzige Weg, `.plist`-Dateien zu bearbeiten. Wenn Sie bei gedrückter `Ctrl`-Taste oder mit der rechten Maustaste auf `Root.plist` klicken, werden Sie sehen, dass das Untermenü `OPEN AS` vier Einträge bereithält: „XML Property List“, „Text Property List“, „Source Code File“ und „Plain Text File“. Mit einem der beiden letzten Einträge öffnen Sie die Eigenschaftsliste als Text, wie in Abbildung 9.6 zu sehen. Das ist deutlich praktischer, wenn man große Textblöcke einfügen und anpassen will – wie zum Beispiel die Namen von Zeitzonen.



Abbildung 9.6: `Root.plist` als Text bearbeiten

Am Anfang der Datei finden wir die XML-Repräsentation der Einstellungskennung für die 12h-/24h-Einstellung. Jetzt müssen wir eine weitere für die Zeitzonennamen hinzufügen. Dabei wird es eine Ken-

nung des Typs `PSMultiValueSpecifier` sein, die zwei Arrays erwartet – eines mit dem Namen `Titles` und das andere mit `Values`. Damit wird festgelegt, wie die Einstellungen dem Anwender präsentiert werden und mit welchen Werten die Anwendung arbeitet.²

Geben Sie also folgenden XML-Code als neues Kindelement dict des Arrays `PreferenceSpecifiers` ein:³

```
<dict>
    <key>Type</key>
    <string>PSMultiValueSpecifier</string>
    <key>Title</key>
    <string>Zeitzone</string>
    <key>Key</key>
    <string>TimeZone</string>
    <key>DefaultValue</key>
    <string>Europe/Berlin</string>
    <key>Titles</key>
    <array>
        <string>Berlin</string>
        <string>Paris</string>
        <string>London</string>
    </array>
    <key>Values</key>
    <array>
        <string>Europe/Berlin</string>
        <string>Europe/Paris</string>
        <string>Europe/London</string>
    </array>
</dict>
```

Durch dieses Bundle können Sie nun Ihre Einstellungen in der Systemanwendung anzeigen und bearbeiten, sobald Sie die Anwendung im Simulator oder auf Ihrem iPhone mit einem `BUILD AND RUN` installiert haben (ja, auch wenn Ihre Anwendung noch gar nichts macht).

Lassen Sie einmal versuchsweise das Beispielprogramm laufen und beenden Sie es mit dem Home-Button. Starten Sie die Einstellungsanwendung, sollten Sie nun Ihre App mit dem im Bundle angegebenen Namen und dem Icon-`Settings.png` wie in Abbildung 9.7 vorfinden.

Selbst ohne Code in der eigentlichen Anwendung ist das Einstellungs-GUI damit fertig – ein deutlicher Vorteil im Vergleich zu einem selbst entwickelten GUI mit dem dazugehörigen Speicherschema, wie wir es in der ersten Hälfte des Kapitels gemacht haben. Jetzt muss die Anwendung die Einstellungen nur noch einlesen und auch umsetzen.

² Ja, es ist eine Katastrophe, beide Arrays synchron zu halten. Wir wissen nicht, warum nicht einfach ein Array mit Titel/Wert-Dictionaries gewählt wurde.

³ Der herunterladbare Beispielcode enthält viel mehr Städte. Wir haben hier nur ein paar gezeigt, um den Code im Buch lesbarer zu gestalten.



Abbildung 9.7: Systemanwendung mit der Anzeige der konfigurierbaren Apps

9.6 Einstellungen laden, die in der System-App konfiguriert wurden

Nachdem das Bundle nun fertig ist, ist das Einlesen der Einstellungen in Ihrer Anwendung ziemlich einfach. Wir beginnen damit, die Schlüssel für Ihre Einstellungen (die natürlich denen entsprechen müssen, die Sie in den Einstellungskennungen in `Root.plist` definiert haben) und die Standardwerte zu definieren:

```
FileIO/BundlePrefereableClock/Classes/BundlePrefereableClockViewController.m
```

```
NSString *TwentyFourHourPrefKey = @"24HourDisplay";
NSString *TimeZonePrefKey = @"TimeZone";
NSString *DefaultTimeZonePref = @"Europe/Berlin";
```

Um nun die Einstellungen einlesen zu können, holen Sie sich über die Klassenmethode `standardUserDefaults` ein Objekt vom Typ `NSUserDefaults`. Damit können Sie Einstellungen als Strings, Dictionaries, `BOOLs` und so weiter mit den Methoden `stringForKey:`, `dictionaryForKey:`, `boolForKey:` etc. auslesen. Definieren Sie in `BundlePreferable-`

ClockViewController.h die Instanzvariablen `BOOL show24Hour` und `NSString *timeZoneName` und fügen Sie dann in der .m-Datei die folgende Methode ein, um die Einstellungen in diese Instanzvariablen zu lesen:

FileIO/BundlePrefereableClock/Classes/BundlePrefereableClockViewController.m

```
- (void) loadPrefs {
    // App-Standard setzen
    timeZoneName = DefaultTimeZonePref;
    // Benutzereinstellungen lesen
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSString *userTimeZone = [defaults stringForKey:TimeZonePrefKey];
    if (userTimeZone != NULL)
        timeZoneName = userTimeZone;
    [userTimeZone release];
    show24Hour = [defaults boolForKey:TwentyFourHourPrefKey];
}
```

Ist das `NSUserDefaults`-Objekt erst einmal eingelesen, ist es recht einfach, die Einstellungswerte Schritt für Schritt auszulesen, wobei Sie auch alle Einstellungen mit einem einzelnen Aufruf von `dictionaryRepresentation` erhalten können. Wir wollen aber auch darauf hinweisen, dass `stringForKey:` den Wert `NULL` zurückgeben kann, wenn der Anwender keinen Wert eingetragen hat. Dann sollten Sie in Ihrem Code auf die Standardwerte zurückgreifen.

Sie werden `loadPrefs` in `viewDidLoad` aufrufen, so dass die Einstellungen beim Laden der View ausgelesen werden.

Da wir gerade bei der View sind – wir haben immer noch nicht das Uhr-GUI erstellt, das diese Eigenschaften nutzt. Wie bei der ersten Uhr-App werden Sie im IB Labels für die Zeit und den Zeitzonennamen erzeugen müssen, die dann mit `IBOutlet`s `timeLabel` und `timeZoneLabel` verbunden werden. Deklarieren Sie auch eine Instanzvariable `NSTimer *clockViewUpdateTimer`, um die Uhrzeit zu aktualisieren, und einen `NSDateFormatter *clockFormatter` für die Formatierung.

FileIO/BundlePrefereableClock/Classes/BundlePrefereableClockViewController.m

```
- (void) setClockFormatter {
    if (clockFormatter == nil) {
        clockFormatter = [[NSDateFormatter alloc] init];
    }
    if (show24Hour)
        [clockFormatter setDateFormat: @"H:mm:ss" ];
    else
        [clockFormatter setDateFormat: @"h:mm:ss a" ];
    // Auch Zeitzone aktualisieren
    [clockFormatter setTimeZone: [NSTimeZone timeZoneWithName:
        timeZoneName]];
}
```

```

- (void) updateClockView {
    NSDate *dateNow = [NSDate date];
    timeLabel.text = [clockFormatter stringFromDate: dateNow];
    timeZoneLabel.text = timeZoneName;
}

- (void) startClock {
    [self updateClockView];
    if (!clockViewUpdateTimer) {
        clockViewUpdateTimer =
            [NSTimer scheduledTimerWithTimeInterval:0.2
             target:self
             selector:@selector(updateClockView)
             userInfo:NULL
             repeats:YES];
    }
}

- (void) viewDidLoad {
    [super viewDidLoad];
    [self loadPrefs];
    [self setClockFormatter];
    [self startClock];
}

```

Wenn Sie auf den Titel oder das Symbol der Anwendung klicken, gelangen Sie zur ersten Seite mit den Einstellungen (siehe auch Abbildung 9.2).

Versuchen Sie, im Simulator oder auf dem iPhone die Einstellungen hier zu ändern, und wechseln Sie dann wieder zurück zu `BundlePreferableClock`. Die Änderungen sollten sich in der Anzeige widerspiegeln. Sie müssen bedenken, dass ein `BUILD AND RUN` in Xcode dafür sorgt, dass das neue Einstellungs-Bundle über die bisher vorgenommenen Einstellungen kopiert wird. Wenn Sie also die Einstellungen testen wollen, sollten Sie sicherstellen, dass Sie im Simulator bleiben und so lange nicht zu Xcode zurückkehren, bis Sie alles ausgetestet haben.

Ein kombinierter Ansatz

Im ersten Abschnitt dieses Kapitels haben wir uns nicht groß damit beschäftigt, wie die Eigenschaften gespeichert werden, sondern nur darauf hingewiesen, dass es kein Problem ist, einfach `.plist`-Dateien im Documents-Verzeichnis Ihrer App in einem beliebigen Format abzulegen. Aber selbst wenn Sie Ihre eigene Einstellungsoberfläche anbieten wollen – über eine Flippable View oder auf andere Art und Weise –, müssen Sie nicht ausschließlich mit den Datei-APIs arbeiten.

Schauen Sie sich die Dokumentation zu `NSUserDefaults` an, das wir zum Laden der Einstellungen genutzt haben, die in der Systemanwendung gesetzt wurden. Neben den Einstellungs-„Gettern“ wie `boolForKey:`, `integerForKey:` und `stringForKey:` gibt es auch entsprechende „Setter“-Methoden (`setBoolForKey:`, `setIntegerForKey:`) und ein `setObjectForKey:`, das mit jedem der Eigenschaftslistentypen funktioniert.

Wenn Sie sich also dazu entscheiden, Ihr eigenes Einstellungs-UI zu nutzen, haben Sie die Wahl, entweder die Dateien selbst zu verwalten (wenn das für Ihre Daten sinnvoller ist) oder die Einstellungsdatenbank durch `NSUserDefaults` zum Lesen und Schreiben der Eigenschaften zu verwenden.

Kapitel 10

Die Datenbank SQLite

Haben Sie schon einmal Code für eine Datenbank geschrieben? Ja? Lassen Sie uns raten – die Datenbank lief auf dem Server, der Code war Teil einer Unternehmensapplikation, und die Lizenz für die Datenbank war teurer als Ihr Haus. Stimmt's?

Und nun denken Sie sich folgendes Szenario: eine mächtige Datenbank ... in Ihrer Tasche ... für alle Anwendungen, die sinnvoll damit nutzbar sind ... *kostenlos*. Das bekommen Sie alles mit der in die iPhones eingebauten SQLite-Datenbank.

Leser, die vorher noch nie eine Datenbank per Code angesprochen haben, haben Glück, denn die Mächtigkeit einer relationalen Datenbank sorgt häufig dafür, dass Ihre Anwendung schneller und zuverlässiger wird, als Sie durch einfache Dateien je erreichen könnten. Schauen Sie sich zum Beispiel die iPod-Anwendung an. Würden Sie sich nur auf die Metadaten in den MP3- und AAC-Dateien stützen, würde das Finden und Abspielen eines bestimmten Lieds über seinen Titel oder aller Lieder eines Künstlers eine sehr teure Operation sein, da Sie Tausende von Dateien öffnen und einlesen müssten. Mit einer Datenbank ist es nur eine Frage des sinnvollen Aufbaus einer Abfrage, wie zum Beispiel `select * from music where artist like "%Zappa%"`¹. In Bruchteilen einer Sekunde kann man dann die Ergebnisse erhalten.

Das iPhone OS enthält die Datenbank SQLite, eine einfache, leichtgewichtige Datenbank, die sich selbst als „die weltweit meistverbreitete SQL-Datenbank-Engine“ bezeichnet. Sie erfahren mehr über SQLite auf der Homepage unter <http://sqlite.org>.

1 Wie immer mag Ihr Geschmack auch anders sein.

In diesem Abschnitt werden wir die Techniken erlernen, mit denen Sie SQLite-Datenbanken in Ihrer Anwendung erstellen und nutzen können.



Joe fragt...

Sollte ich stattdessen nicht eher Core Data nutzen?

Vielleicht. Schauen Sie sich dieses Kapitel und Kapitel 11, *Core Data*, auf Seite 225 an. Dann merken Sie sofort, dass es einen riesigen Unterschied zwischen beiden gibt. Die direkte Anwendung der SQLite-API geschieht auf unterster Ebene, und die Datenbanktabellen werden direkt angesprochen. Das ist für eine direkte Datenablage manchmal gar nicht so schlecht, aber wenn Sie komplexe Relationen modellieren müssen – zum Beispiel 1:n- oder n:m-Relationen –, wird es schwierig. Mit Core Data können Sie auf der Objektebene arbeiten, wodurch das Leben einfacher wird, wenn jedes Objekt mit vielen anderen Objekten verbunden ist und Sie diese Beziehungen auch verwalten müssen. Core Data ist zudem dazu gedacht, die so verwalteten Objekte in einer UITableView zu präsentieren, während Anwendungen, die die Datenbank direkt nutzen, ihre eigenen Tabellendatenmodelle bereitstellen müssen (wie wir es in diesem Kapitel tun).

Wenn Sie bestehenden SQL-Code portieren oder im Team schon SQL-Erfahrung vorhanden ist, bevorzugen Sie vielleicht die Arbeit mit SQLite. Jedoch hat Apple dadurch, dass es Core Data auf das iPhone gebracht hat, klargestellt, dass dies das zu bevorzugende Framework für das Persistieren von Daten auf dem iPhone und auf dem Desktop ist.

10.1 Ihre Datenbank erstellen

Praktischerweise läuft auf Ihrem Mac ebenfalls SQLite Version 3, was sich durch ein einfaches `which` an der Befehlszeile ermitteln lässt:

```

⇐ Yuna:~ cadamson$ which sqlite3
⇒ /usr/bin/sqlite3

```

Wir werden `sqlite3` nutzen, um eine Datenbank zu erstellen, die Sie dann mit Ihrer Anwendung mitliefern können.

Als Beispiel werden wir in diesem Abschnitt einen durch eine Datenbank unterstützten „Einkaufszettel“ erstellen, dem der Anwender neue Elemente hinzufügen und die vorhandenen auflisten kann.

Bevor wir die iPhone-Anwendung selbst erstellen, wollen wir die Datenbank aufbauen und mit ein paar Daten bestücken. Wie die meisten Datenbanken arbeitet auch SQLite mit der Structured Query Language (SQL). Dabei handelt es sich um einen weit verbreiteten und unterstützten Standard für Datenbankbefehle, die durch Menschen lesbar sind. Zudem lassen sich diese Anweisungen leicht zur Laufzeit per Code erstellen. Wie die meisten Datenbanken hat SQLite einen interaktiven Modus, der es Ihnen ermöglicht, SQL-Befehle einzugeben und direkt ausführen zu lassen. Öffnen Sie also Terminal (oder xterm oder etwas Ähnliches) und wechseln Sie mit `cd` zum Verzeichnis `Database-ShoppingList`. Es existiert schon eine Datenbankdatei `shopping.db`, die Sie am besten löschen oder umbenennen, damit Sie selbst eine neue aufbauen können.

Jetzt starten Sie durch den Aufruf des Befehls `sqlite3` eine SQLite-Session zusammen mit einer Datenbank:

```

⇒ Yuna:DatabaseShoppingList cadamson$ sqlite3 shopping.db
⇒ SQLite version 3.4.0
Enter ".help" for instructions
sqlite>

```

Damit wird die Datei `shopping.db` geöffnet (wenn sie noch nicht existiert, wird sie in dem Moment erstellt, in dem Sie Daten hineinschreiben). Jetzt können Sie direkt damit beginnen, SQL-Befehle einzugeben. Da es noch keine Daten in der leeren Datenbank gibt, werden wir zuerst eine Tabelle für den Einkaufszettel erstellen, bei der jede Zeile den Namen der Ware, den Preis, einen Gruppencode und das Datum, an dem es eingetragen wurde, enthält. Dafür braucht man nicht mehr als den Befehl `create table`, gefolgt von einem `go`.²

```

⇒ sqlite> create table shoppinglist (key integer primary key,
⇒     ...> item text,
⇒     ...> price double,
⇒     ...> groupid integer,
⇒     ...> dateadded date);
⇒ sqlite> go

```

Damit wird die Struktur der Tabelle `shoppinglist` erstellt, aber kein Inhalt eingefügt. Das erreichen Sie durch den SQL-Befehl `insert`:

```

⇒ sqlite> insert into "shoppinglist" (item, price, groupid, dateadded)
⇒     ...> values ("iPhone 3G", "199", 1, DATETIME('NOW'));
⇒ sqlite> go

```

² Beachten Sie, dass die `...>` in der Ausgabe im Buch Einrückungen von `sqlite3` sind, bei denen wir beim Eingeben die Return-Taste gedrückt haben. Da das Semikolon – und nicht die Return-Taste – einen Befehl beendet, können Sie die Zeilen ganz nach Wunsch umbrechen.

Starten Sie eine SQL-Abfrage, um sich den gerade hinzugefügten Eintrag anzeigen zu lassen:

```

⇐ sqlite> select * from shoppinglist
⇐ ...> go
⇒ 1|iPhone 3G|199.0|1|2008-06-28 12:35:24

```

Sie können noch mehr Daten für die Anwendung eintragen, indem Sie eine Skriptdatei nutzen, die wir in den herunterladbaren Beispielcode mit aufgenommen haben. Beenden Sie `sqlite3` mit dem Befehl `.quit`, um zur Befehlszeile zurückzugelangen. Löschen Sie die Datei `shopping.db` durch `rm shopping.db`. Der Download enthält ein Skript namens `make-table-script`, mit dem noch mehr Einträge eingefügt werden. Sie können dieses Skript ausführen, indem Sie es `sqlite3` übergeben:

```

⇐ Yuna:DatabaseShoppingList cadamson$ sqlite3 shopping.db < make-table-script

```

Jetzt wechseln wir wieder in den interaktiven Modus von SQLite, um uns die soeben erstellten Daten anzuschauen:

```

⇐ Yuna:DatabaseShoppingList cadamson$ sqlite3 shopping.db
⇒ SQLite version 3.4.0
  Enter ".help" for instructions
⇐ sqlite> select * from shoppinglist
⇐ ...> go
⇒ 1|Marzipan|3.99|0|2008-06-28 12:42:23
  2|Mac Pro|2499.99|1|2008-06-28 12:42:23
  3|iPhone 3G|199.0|1|2008-06-28 12:42:23
  4|Chips|2.49|0|2008-06-28 12:42:23
  5|TK-Pizza|3.75|0|2008-06-28 12:42:23
  6|Alu-Tastatur|169.95|1|2008-06-28 12:42:23
  7|Atemfrei|0.98|0|2008-06-28 12:42:23
  8|ADC-Mitgliedschaft|500.0|1|2008-06-28 12:42:23
  9|iPhone Einzelmitgliedschaft|99.0|1|2008-06-28 12:42:23
  sqlite>

```

Für dieses Beispiel haben wir versucht, Dinge zu finden, die Geeks typischerweise einkaufen. Daher gibt es auch nur zwei Gruppencodes: 0 steht für Lebensmittel und 1 für Technik. Denn mal ganz ehrlich – was brauchen Sie mehr, solange Sie etwas zum Essen und zum Spielen haben?³

Wenn Sie bisher noch nicht viel mit Datenbanken gearbeitet haben, wundern Sie sich vielleicht über dieses `key`-Feld, das wir angelegt und als Primärschlüssel gekennzeichnet haben, ohne es je zu füllen. Es ist aber in der Datenbankprogrammierung übliche Praxis, jeder Zeile eine eindeutige ID zu verpassen, die oft von der Datenbank selbst erzeugt

³ Tatsächlich haben wir überlegt, noch eine dritte Gruppe für Actionfiguren mit aufzunehmen, aber festgestellt, dass es viel zu viel Aufwand ist, sich authentisch klingende Superhelden- und RPG-Charakternamen auszudenken, die dann nicht gleich von der Urheberrechtspolizei einkassiert würden.

wird, um später zweifelsfrei auf eine Zeile verweisen zu können. Wenn Sie zum Beispiel mehrere Einträge für „TK-Pizza“ haben, die alle gleich sind, und einen davon aktualisieren oder löschen wollen, brauchen Sie eine eindeutige ID, um der Datenbank sagen zu können, was sie löschen soll (zum Beispiel `delete from shoppinglist where key=5`). Es ist außerdem unbedingt notwendig, wenn Sie später Zeilen aus mehreren Tabellen kombinieren wollen – zum Beispiel wenn Sie eine zweite Tabelle haben, in der verschiedene Läden aufgeführt sind, in denen Sie die Elemente der Einkaufsliste kaufen können. In solch einem Fall würde sich jede Zeile der Tabelle `itemsatstores` auf die `id` einer Zeile in der Tabelle `shoppinglist` beziehen.

10.2 Die Beispielanwendung erstellen

Um ein bisschen mit Datenbankcode herumzuspielen, wird unsere iPhone-Anwendung zwei Funktionen anbieten: das Hinzufügen von Einträgen zum Einkaufszettel und das Betrachten der Einträge. Wir werden dazu eine Tab-Leiste nutzen, bei der jeweils ein Tab (und damit ein View-Controller) für das Hinzufügen und für das Anzeigen gedacht ist. Erstellen Sie also in Xcode eine neue Tab Bar Application, die wir `DatabaseShoppingList` nennen werden. Standardmäßig besitzt die Anwendung zwei Tabs, was für unsere Zwecke genau richtig ist. Allerdings haben wir uns dazu entschlossen, mit Refactor den `FirstViewController` in `AddItemViewController` umzubenennen.

Standardmäßig besitzt die `MainWindow.xib` zwei View-Controller für die beiden Tabs. Genau das wollen wir ja auch haben, allerdings tragen sie den Namen `First` und `Second`. Suchen Sie den „Selected First View-Controller (First)“ im Interface Builder (Sie werden dafür zunächst den Anzeigemodus des IB auf Liste oder Spalten umschalten müssen), erweitern Sie ihn, um die „Tab Bar Items“ zu erhalten, öffnen Sie den Identity Inspector und ändern Sie den Titel zu „Eintrag hinzufügen“. Genau ändern Sie den Titel des zweiten Tab Bar View-Controller in „Nach Preis anzeigen“. Wir haben im herunterladbaren Beispiel hier auch noch kleine PNG-Symbole ergänzt.

`MainWindow.xib` enthält standardmäßig eine View für den ersten Tab. Statt eine eigene `.nib`-Datei zu erstellen, ist es auch in Ordnung, diese erste View anzupassen. Löschen Sie irgendwelche vorhandenen Inhalte und fügen Sie Textfelder für den Namen und den Preis eines Eintrags hinzu sowie einen Picker für den Typ, einen Button HINZUFÜGEN und ein Label am unteren Rand, mit dem wir jeden neuen Eintrag bestätigen werden.



Sie müssen in `AddItemViewController.h` Outlets für die Textfelder, den Picker und das Label ergänzen und eine Action für das Antippen des Buttons definieren. Zudem müssen Sie den Picker-Delegate und die Data Source-Methoden sowie `UITextFieldDelegate` deklarieren, um die Tastatur auszublenden, wenn der Anwender auf Return tippt. Dann verbinden Sie diese Outlets und Header aus den GUI-Komponenten mit dem `AddItemViewController`. Verbinden Sie zudem den delegate der Textfelder mit diesem View-Controller, ebenso delegate und data-source des Pickers.

SQLite/DatabaseShoppingListComplete/Classes/AddItemViewController.h

```
@interface AddItemViewController : UIViewController <UIPickerViewDelegate,
    UIPickerViewDataSource, UITextFieldDelegate> {
    UITextField *itemNameField;
    UITextField *priceField;
    UIPickerView *groupPicker;
    UILabel *statusLabel;
}
- (IBAction) addShoppingListItem: (id) sender;

@property (nonatomic, retain) IBOutlet UITextField *itemNameField;
@property (nonatomic, retain) IBOutlet UITextField *priceField;
@property (nonatomic, retain) IBOutlet UIPickerView *groupPicker;
@property (nonatomic, retain) IBOutlet UILabel *statusLabel;

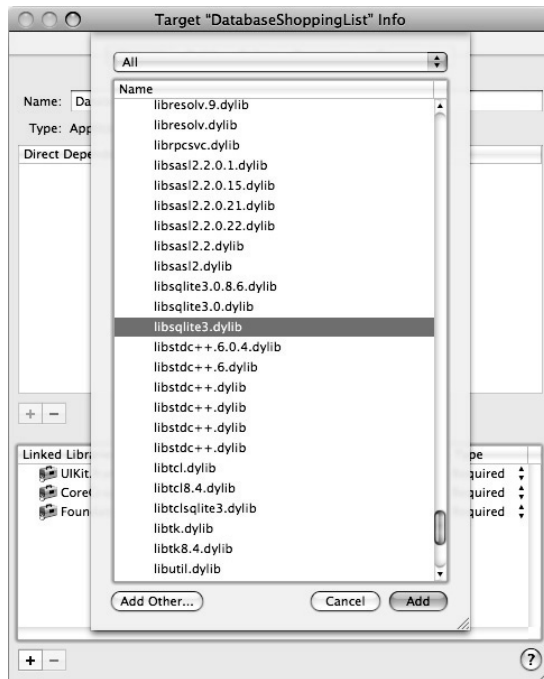
@end
```

Der Delegate des Pickers und die Data-Source-Methoden haben nichts mit der Datenbank zu tun und können jetzt schon geschrieben werden.

10.3 Ihre Datenbank auf das iPhone bringen

Sie haben eine Datenbankdatei und die Grundlagen einer Anwendung. Jetzt können Sie in Ihrem Code auf den Inhalt der Datenbank zugreifen. Dazu werden Sie die C-API von SQLite nutzen, die nicht standardmäßig in den Xcode-Projekten enthalten ist. Sie müssen also bei jeder iPhone-Anwendung, die SQLite verwendet, zwei Dinge durchführen:

Ergänzen Sie das Xcode-Projekt um die Datei `libsqlite3.dylib`. Klappen Sie in „Groups & Files“ das Targets-Element auf, markieren Sie `DatabaseShoppingList` und lassen Sie mit - oder dem INFO-Button in der Toolbar den Inspector anzeigen. Klicken Sie auf GENERAL und schauen Sie sich die untere Hälfte des Fensters an: Sie sehen dort die Frameworks, die aktuell im Projekt verlinkt sind. Mit dem **+**-Button tragen Sie ein neues Framework ein. Wählen Sie aus der Liste `libsqlite3.dylib`. Im Beispielcode haben wir das schon erledigt.



In jeder Klasse, die die SQLite-API nutzt, müssen Sie die Header-Datei mit aufnehmen:

```
#include <sqlite3.h>
```

Sie müssen auch Ihre Datenbankdateien in das Anwendungs-Bundle kopieren. Das können Sie durch Ziehen der Dateien aus dem Finder in

den Resources-Ordner in Ihrem Xcode-Projekt erreichen oder indem Sie das Resources-Verzeichnis mit gedrückter `Ctrl`-Taste anklicken und `ADD EXISTING FILE` wählen.

Wenn Ihre Datenbank im Anwendungs-Bundle liegt, schreiben Sie dort keine neuen Datensätze hinein. Der Grund dafür ist einfach – Sie haben keinen Schreibzugriff auf das Bundle Ihrer eigenen Anwendung. Stattdessen sollten Sie die Datenbankdatei in einen anderen Ordner kopieren – mit ziemlicher Sicherheit werden Sie Documents nutzen – und dann mit dieser Datei arbeiten.

Kopieren in das Documents-Verzeichnis

Bevor Sie nun also irgendeine Funktion von SQLite nutzen, kopieren Sie die Datenbankdatei an eine andere Stelle (auf die sich die Klasse `DatabaseShoppingListAppDelegate` über eine `NSString`-Eigenschaft `dbFilePath` beziehen wird). Im Beispielcode ist dies eine Datei im Verzeichnis Documents, deren Name durch den Wert der Konstanten `DATABASE_FILE_NAME` bestimmt wird. Um diesen Pfad zu setzen, wenn die Anwendung gestartet wird, suchen wir zunächst das Verzeichnis Documents.



Joe fragt...

Muss ich meine Datenbank zunächst auf dem Desktop-Rechner anlegen?

Technisch gesehen – nein. Wie Sie in den nächsten paar Abschnitten sehen werden, unterscheiden sich die SQL-Anweisungen, die Sie im Code erstellen, nicht von denen, die Sie in der Befehlszeilenanwendung `sqlite3` eingeben, daher können Sie alles mit der API zur Laufzeit ausführen, auch den Befehl `CREATE TABLE`. Wenn Sie das tun, erzeugt der erste Aufruf von `sqlite3_open()` die Datenbankdatei, wenn sie nicht schon existiert – so wie bei `sqlite3`.

Es kann hochdynamische Fälle geben, in denen dieses Vorgehen sinnvoll ist, aber häufig ist es praktischer und performanter, Ihre Datenbank auf dem Desktop anzulegen, sie als Teil Ihres Anwendungs-Bundles mit auszuliefern und eine Kopie für die Benutzerdaten zu erstellen. Damit können Sie auch leicht die ursprüngliche Datenbank wiederherstellen. Kopieren Sie einfach die ursprüngliche Datenbankdatei aus dem Bundle.

```

NSArray *searchPaths =
    NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentFolderPath = [searchPaths objectAtIndex: 0];
dbFilePath = [documentFolderPath stringByAppendingPathComponent:
    DATABASE_FILE_NAME];

```

Vielleicht ist Ihnen aufgefallen, dass wir hier ein anders vorgehen als bei der weiter oben gezeigten Funktion `NSHomeDirectory()` nutzen (Abschnitt 8.2, *Unser Projekt erstellen*, Seite 147), um das Home-Verzeichnis der Anwendung zu finden. Der hier gezeigte Code – Apples bevorzugte Technik, um den Ordner Documents zu finden – sucht in bekannten Pfaden nach eventuell interessanten Elementen. In diesem Fall zeigen Sie Ihr Interesse am Ordner Documents durch das Übergeben der Konstanten `NSDocumentDirectory` und dem anschließenden Auslesen des ersten Pfads aus dem zurückgegebenen Array.

Jetzt folgt der einfache Schritt, die Datenbankdatei im Anwendungs-Bundle zu finden und sie über den `NSFileManager` in das Verzeichnis Documents kopieren zu lassen. In diesem Beispiel sind die Konstanten `DATABASE_RESOURCE_NAME` und `DATABASE_RESOURCE_TYPE` einfach `NSString`s für den Dateinamen und die Erweiterung der Datenbankdatei: `data-base` und `db`.

SQLite/DatabaseShoppingListComplete/Classes/DatabaseShoppingListAppDelegate.m

```

if (![NSFileManager defaultManager] fileExistsAtPath: dbFilePath)) {
    // Keine DB gefunden, also kopieren
    NSString *backupDbPath = [[NSBundle mainBundle]
        pathForResource:DATABASE_RESOURCE_NAME
        ofType:DATABASE_RESOURCE_TYPE];
    if (backupDbPath == nil) {
        // Keine Backup-Kopie gefunden, abbrechen
        return NO;
    } else {
        BOOL copiedBackupDb = [[NSFileManager defaultManager]
            copyItemAtPath:backupDbPath
            toPath:dbFilePath
            error:nil];
        if (!copiedBackupDb) {
            // Kopieren fehlgeschlagen, abbrechen
            return NO;
        }
    }
}
return YES;

```


10.4 Ihre Datenbank auf dem iPhone verwenden

Nachdem die Datenbankdatei nun in das Verzeichnis Documents Ihrer Anwendung kopiert ist, können Sie sie jetzt über die SQLite-C-API ansprechen. Die ganze API ist recht klein, und Sie werden im Allgemeinen immer dem gleichen Muster von Funktionsaufrufen folgen:

1. `sqlite3_open()` – Vorbereiten der Datenbank.
2. `sqlite3_prepare()` – Initialisieren einer vorbereiteten SQL-Anweisung.⁴
3. `sqlite3_step()` – Ausführen der Anweisung, wodurch entweder in die Datenbank geschrieben wird (beim Einfügen, Löschen und Aktualisieren) oder eine Zeile einer Ergebnismenge zum Lesen aufbereitet wird.
4. `sqlite3_column_NNN()` – Auslesen typisierter Daten aus der aktuellen Ergebnismenge, wobei *NNN* einen Datentyp angibt, wie zum Beispiel `bytes()`, `int()`, `text()` und so weiter.
5. `sqlite3_finalize()` – Freigeben von Ressourcen, die für die Datenbank genutzt wurden.⁵
6. `sqlite3_close()` – Schließen der Datenbank.

Werte in die Datenbank einfügen

Lassen Sie uns nun überlegen, was wir für die Beispielanwendung brauchen. Sie muss die Einträge anzeigen, die sich momentan in der Datenbank befinden – das kann sie durch Abfragen der Tabelle `shoppinglist` und dem Aufbau eines GUI für die Ergebnisse erreichen. Zudem muss der Anwender neue Einträge einfügen können. Wir wollen mit dem Hinzufügen beginnen, da das ein bisschen einfacher ist.

Wenn der Anwender auf den Button `HINZUFÜGEN` tippt, wird die Methode `addShoppingListItem:` aufgerufen. Sie prüft zunächst den Inhalt des Text- und des Preisfelds. Stehen dort sinnvolle Werte, holen wir die Eigenschaft `dbFilePath` aus dem Anwendungs-Delegate und nutzen sie, um die Datenbank mit `sqlite3_open()` zu öffnen.

4 Technisch gesehen, sollte man `sqlite3_prepare_v2()` bei jeder neueren SQLite-Entwicklung vorziehen.

5 Einer unserer Fachkorrektoren hat uns daran erinnert, dass nicht finalisierte Anweisungen zu Datenverlust oder zerstörten Daten führen können. Halten Sie also Datenbankanweisungen im Speicher, müssen Sie sie eventuell abschließend in `applicationWillTerminate` finalisieren.

SQLite/DatabaseShoppingListComplete/Classes/AddItemViewController.m

```
if ([itemNameField.text length] == 0) ||
    ([priceField.text length] == 0) ||
    ([priceField.text doubleValue] == 0.0))
    return;

sqlite3 *db;
int dbrc; // Rückgabecode der DB
DatabaseShoppingListAppDelegate *appDelegate
    = (DatabaseShoppingListAppDelegate*)
        [UIApplication sharedApplication].delegate;
const char* dbFilePathUTF8 = [appDelegate.dbFilePath UTF8String];
dbrc = sqlite3_open (dbFilePathUTF8, &db);
if (dbrc) {
    NSLog (@"Konnte DB nicht öffnen:" );
    return;
}
```

Da Sie es mit einer prozeduralen C-API zu tun haben, müssen Sie bei der Arbeit mit SQLite etwas anders vorgehen, als Sie es vielleicht bei Cocoa und Objective-C gewohnt sind. Vor allem arbeiten Sie nicht mit Objekten, stattdessen müssen Sie Strukturen erzeugen, die Sie den verschiedenen Funktionsaufrufen mitgeben. Sie sehen das hier an `db`, einem Zeiger auf eine Struktur vom Typ `sqlite3`. Statt sie per `alloc` anzulegen, rufen Sie die Funktion `sqlite3_open()` auf und übergeben eine Adresse, an der Sie einen Zeiger auf eine neu erzeugte `sqlite3-struct` haben möchten.

Ihnen fällt eventuell auch auf, dass Sie aufgrund der C-API den Datenbankpfad als Zeiger auf einen C-String übergeben müssen – ein `const char*`, beendet durch ein `\0`-Zeichen. Zum Glück gibt die Methode `UTF8String` von `NSString` genau so etwas zurück – freundlicherweise auch noch per Autorelease. Beachten Sie auch, dass alle SQLite-Funktionen einen „Rückgabecode“ zurückgeben. Dabei handelt es sich um eine der Konstanten, die in `sqlite3.h` definiert sind. In den meisten Fällen ist 0, der Wert für die Konstante `SQLITE_OK`, das Zeichen für Erfolg, und ein Wert ungleich null weist auf einen Fehler oder andere Ausnahmebedingungen hin.

Ist die Datenbank nun geöffnet, können Sie eine `INSERT`-Anweisung erstellen, um Ihren Datensatz einzufügen. So funktioniert das bei der Einkaufszettel-App:⁶

6 Der Backslash am Ende jeder Zeile im String `insertStatementNS` ist nicht notwendigerweise Teil der Syntax – es handelt sich um ein Zeilenumbruchzeichen, um den String für den Beispielcode schöner zu formatieren. Sie müssen die Strings nicht so umbrechen, aber lesbarer werden sie damit auf jeden Fall.

SQLite/DatabaseShoppingListComplete/Classes/AddItemViewController.m

```

sqlite3_stmt *dbps; // Database Prepared Statement
NSString *insertStatementNS = [NSString stringWithFormat:
    @"insert into \" shoppinglist\"\\
    (item, price, groupid, dateadded)\\
    values (\\\"%@\\\" , %@, %d, DATETIME('NOW'))",
    itemNameField.text,
    priceField.text,
    [groupPicker selectedRowInComponent: 0]];
const char *insertStatement = [insertStatementNS UTF8String];
dbrc = sqlite3_prepare_v2(db, insertStatement, -1, &dbps, NULL);
dbrc = sqlite3_step(dbps);

```

In diesem Block reservieren wir zunächst einen Zeiger auf eine **struct** für eine vorbereitete Anweisung (`sqlite3_stmt`), dann bauen wir eine SQL-INSERT-Anweisung zusammen, indem wir die Werte für einen Format-String mit dem Inhalt der UITextFieldFields und dem ausgewählten Eintrag des Gruppen-Pickers besetzen. Der sich so ergebende String ist dann eine INSERT-Anweisung – genau wie die, die Sie vorher im interaktiven Modus an der Befehlszeile eingegeben haben. Anschließend nutzen Sie die Funktion `sqlite3_prepare_v2()`, um mit der C-kompatiblen, null-terminierten Version dieses Strings eine vorbereitete Anweisung zu erzeugen. Dabei wird -1 als Stringlänge angegeben, damit SQLite die Länge des Strings selbst anhand des Null-Bytes herausfindet (so müssen wir uns nicht um Mehrbyte-Zeichen im UTF-8-String kümmern).

Um die Anweisung dann auch auszuführen, rufen Sie `sqlite3_step()` auf. Gibt diese Methode `SQLITE_DONE` zurück, wurde Ihre Zeile der Datenbank hinzugefügt.

Hat die Anweisung ihren Zweck erfüllt, sollten Sie nun aufräumen und alle Ressourcen freigeben, die sie benötigt hat. Sie können die Struct `sqlite3` wiederverwenden, um mehr vorbereitete Anweisungen zu erstellen, aber im Beispielcode haben wir uns dazu entschieden, nach jedem Insert stets aufzuräumen und alles zu schließen.

SQLite/DatabaseShoppingListComplete/Classes/AddItemViewController.m

```

// Fertig mit der DB. Anweisung finalisieren und schließen
sqlite3_finalize(dbps);
sqlite3_close(db);

```

Eine letzte Aufgabe in `AddItemViewController` ist noch, die virtuelle Tastatur auszublenden, wenn der Anwender auf Return tippt. Das muss auch deshalb sein, weil wir sicherstellen wollen, dass der Benutzer auf den Button HINZUFÜGEN tippen kann.



Wir haben schon die delegates der beiden Textfelder mit dem View-Controller verbunden, jetzt brauchen wir nur noch eine einfache Methode, um den First Responder abzumelden:

SQLite/DatabaseShoppingListComplete/Classes/AddItemViewController.m

```
-(BOOL)textFieldShouldReturn:(UITextField *)textField {
    [textField resignFirstResponder];
    return YES;
}
```

Werte aus der Datenbank lesen

Nachdem wir eine Zeile in die Datenbank eingefügt haben, zeigt sich, dass das Auslesen von Zeilen auf nahezu dem gleichen Weg funktioniert. Der größte Unterschied beim Ausführen einer Abfrage – abgesehen vom Inhalt der vorbereiteten Anweisung – ist, dass Sie nach dem Aufruf der Funktion `sqlite3_step()` über die Ergebnisse iterieren, um die von Ihrer Abfrage zurückgegebenen Werte auszulesen.

Aufzucht und Hege von SQL

Die in diesem Beispiel genutzten SQL-Anweisungen sind für eine realistische Anwendung viel zu einfach, aber wir können uns so auf die API von SQLite konzentrieren. Im Speziellen kümmern wir uns nicht darum, die Eingabedaten zu validieren – zum Beispiel Anführungszeichen zu maskieren oder zu verhindern, dass der Benutzer versucht, über die Daten eigenen, böartigen SQL-Code einzuschleusen (eine sogenannte SQL Injection).

Nicht, dass wir denken, solche Techniken wären unwichtig – sie sind sehr wichtig –, aber wir können hier keine umfassende Einführung in SQL liefern. Es gibt Tausende von Büchern über SQL (Echt! Wir haben nachgezählt!). Wenn Sie ernsthaft mit dem Gedanken spielen, die iPhone-Datenbank zu nutzen, haben Sie entweder schon eines davon gelesen, oder Sie werden es bald tun.

Im Beispiel `DatabaseShoppingList` kann der gesamte Inhalt der Tabelle `shoppinglist` jedes Mal in den Speicher gelesen werden, wenn der View-Controller für den Einkaufszettel die Message `viewWillAppear:` erhält. Eine Methode `loadDataFromDb` fragt die Datenbank ab und puffert die Ergebnisse für die Nutzung als Data Source einer `UITableView` – der Tabellen-View, die wir als einzige View im zweiten Nib eingerichtet haben. `loadDataFromDb` erzeugt dafür ein `NSArray` mit `NSDictionary`s, in dem jedes Dictionary einer Zeile der Datenbanktabelle entspricht und in dem die Felder (Name, Art und Preis) als Name/Wert-Paare enthalten sind. So können wir `tableView:numberOfRowsInSection:` ganz einfach implementieren, indem wir die Länge des Arrays zurückliefern.

Das Öffnen der Datenbank und das Vorbereiten einer Anweisung mit `sqlite3_open()` und `sqlite3_prepare_v2()` funktioniert genau so, wie weiter oben schon gesehen, mit der offensichtlichen Ausnahme, dass der SQL-String ein anderer ist:

SQLite/DatabaseShoppingListComplete/Classes/ListByPriceViewController.m

```
NSString *queryStatementNS =
@"select key, item, price, groupid, dateadded\
from shoppinglist order by price";
```

Beachten Sie, wie diese Abfrage die Felder angibt, die wir von der Abfrage erhalten wollen. Die Ergebnismenge enthält *ausschließlich* diese Felder in dieser Reihenfolge. Die Reihenfolge wird dann wichtig, wenn Sie angeben, welche Spalten Sie auslesen wollen.

Als Nächstes wird wiederholt `sqlite3_step()` aufgerufen und dabei geprüft, ob der Rückgabecode den Wert `SQLITE_ROW` hat. Wenn dem so ist, steht eine Zeile der Ergebnisdaten für Sie bereit, die Sie über die „Column“-Funktionen von SQLite auslesen können. Diese Funktionen sind in `sqlite3.h` definiert:

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
sqlite_int64 sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
sqlite3_value *sqlite3_column_value(sqlite3_stmt*, int iCol);
```

Wie Sie sehen können, stehen Funktionen für die Rückgabe von BLOBs (Binary Large Objects, die als untypisierte Zeiger zurückgegeben werden), **bytes**, **doubles**, **ints** und Strings (als Zeiger auf null-terminierte UTF-8-Strings⁷) bereit, zudem Metadatenfunktionen, um die Namen und Typen von Spalten zu ermitteln. Jede Funktion erwartet als Parameter die Anweisung, die Sie mit `sqlite3_step()` ausgeführt haben, aber auch einen Spaltenindex, der sich auf die in Ihrer SQL-SELECT-Anweisung definierte Reihenfolge bezieht (und nicht notwendigerweise auf die Reihenfolge der Spalten in der Datenbanktabelle selbst).

Mit dem folgenden Code können wir über die Abfrageergebnisse iterieren, wobei wir die C-basierten Repräsentationen von SQLite in Cocoa-Objekte umwandeln, die wir dann für eine spätere Verwendung durch die Tabelle in ein `NSDictionary` stecken.

SQLite/DatabaseShoppingListComplete/Classes/ListByPriceViewController.m

```
while ((dbrc = sqlite3_step (dbps)) == SQLITE_ROW) {
    int primaryKeyValueI = sqlite3_column_int(dbps, 0);
    NSNumber *primaryKeyValue = [[NSNumber alloc]
        initWithInt: primaryKeyValueI];
    NSString *itemValue = [[NSString alloc]
        initWithUTF8String:(char*) sqlite3_column_text (dbps, 1)];
    double priceValueD = sqlite3_column_double (dbps, 2);
    NSNumber *priceValue = [[NSNumber alloc]
        initWithDouble: priceValueD];
    int groupValueI = sqlite3_column_int(dbps, 3);
    NSNumber *groupValue = [[NSNumber alloc]
        initWithInt: groupValueI];
    NSString *dateValueS = [[NSString alloc]
        initWithUTF8String: (char*)sqlite3_column_text (dbps, 4)];
    NSDate *dateValue = [dateFormatter dateFromString: dateValueS];
```

⁷ Bei SQLite 3 kann die Datenbank auch mit UTF-16-Texten umgehen.

```

NSMutableDictionary *rowDict =
    [[NSMutableDictionary alloc] initWithCapacity: 5];
[rowDict setObject: primaryKeyValue forKey: PRIMARY_ID_KEY];
[rowDict setObject: itemValue forKey: ITEM_KEY];
[rowDict setObject: priceValue forKey: PRICE_KEY];
[rowDict setObject: groupValue forKey: GROUP_ID_KEY];
[rowDict setObject: dateValue forKey: DATE_ADDED_KEY];
[shoppingListItems addObject: rowDict];
// Jegliches Interesse an den Werte-Objekten aufgeben
[dateValues release];
[primaryKeyValue release];
[itemValue release];
[priceValue release];
[groupValue release];
[rowDict release];
}

```

Werfen Sie mal einen Blick auf den Umgang mit Datumswerten. Vielleicht ist Ihnen im vorigen Codeabschnitt aufgefallen, dass es keine C-Funktion zum Auslesen eines Datumswerts gibt. Lesen Sie den Wert als numerischen Wert aus (in der Erwartung, die Anzahl an Millisekunden seit einem bestimmten Startdatum zu erhalten), bekommen Sie nur das Jahr zurück. Stattdessen müssen Sie eine String-Repräsentation des Datums auslesen und diese dann mit dem NSDateFormatter parsen. Sie müssen nur berücksichtigen, dass Datumswerte in SQLite keine Zeitzone enthalten und dass Zeiten, die durch die entsprechenden Funktionen der Datenbank erzeugt werden – wie zum Beispiel `DATETIME('NOW')` –, immer in UTC erzeugt werden. Das ist eigentlich sogar ein recht gutes System für eine konsistente Behandlung von Uhrzeiten: Konvertieren Sie Zeiten vor dem Einfügen immer nach UTC, da Sie so davon ausgehen können, dass alle Zeiten in der Datenbank UTC-Zeiten sind. Nach dem Auslesen müssen Sie sie dann nur noch in die entsprechende lokale Zeitzone konvertieren.

Im obigen Code haben wir also einen NSDateFormatter genutzt, um die String-Repräsentation der Datenbank in ein NSDate umzuwandeln. Dieser Formatter wird in `initWithNibName:bundle:` initialisiert.

SQLite/DatabaseShoppingListComplete/Classes/ListByPriceViewController.m

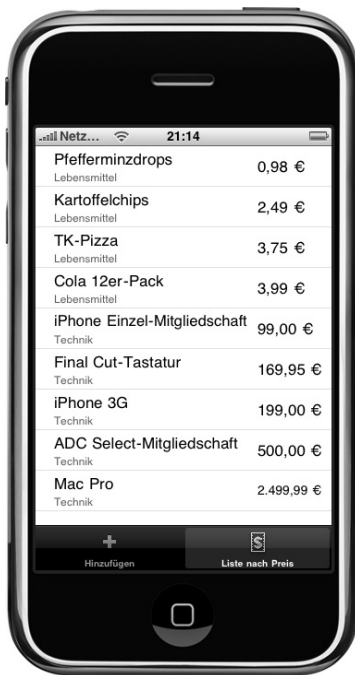
```

dateFormatter = [[NSDateFormatter alloc] init]; [
dateFormatter setTimeZone: [NSTimeZone timeZoneWithAbbreviation:@"UTC" ]];
[dateFormatter setDateFormat: @"yyyy-MM-dd HH:mm:ss" ];

```

Dann erstellen wir einen Formatter, setzen seine Zeitzone auf UTC und stellen den von der Datenbank zurückgegebenen String im Locale Data Markup Language-(LDML-)Format (aka *Unicode Technical Standard 35* [Uni04]) zur Verfügung.

Dadurch dass `loadDataFromDb` (aufgerufen von `viewWillAppear:`) die Instanzvariable `shoppingListItems` als Array erzeugt hat, kann die Methode `tableView:cellForRowAtIndexPath:` einen Eintrag aus dieser Tabelle über den Index ermitteln, ihn in ein `NSDictionary` casten und dann die Name/Wert-Paare nutzen, um ein Tabellenfeld zu füllen. Gibt es in den Feldern mehrere Elemente, ist dies der richtige Zeitpunkt, um eigene im IB entworfene Tabellenfelder zu nutzen, wie sie in Abschnitt 5.7, *Eigene Felder in Tabellen-Views*, auf Seite 104 vorgestellt wurden. Das herunterladbare Beispiel enthält Felder für den Namen, den Preis und die Gruppe – Sie können Ihre Felder natürlich auch ganz anders zusammenstellen.



Nachdem ein Feld entweder aus der Tabelle geholt oder aus dem Nib geladen wurde, ist das Setzen des Werts schnell geschehen, indem man ihn aus dem Array mit Dictionaries lädt, das weiter oben in `loadDataFromDb` erstellt wurde.

SQLite/DatabaseShoppingListComplete/Classes/ListByPriceViewController.m

```
UILabel *itemLabel = (UILabel*) [myCell viewWithTag:1];
UILabel *groupLabel = (UILabel*) [myCell viewWithTag:2];
UILabel *priceLabel = (UILabel*) [myCell viewWithTag:3];
NSDictionary *rowVals =
    (NSDictionary*) [shoppingListItems objectAtIndex: indexPath.row];
NSString *itemName = (NSString*) [rowVals objectForKey: ITEM_KEY];
```



```

itemLabel.text = itemName;
int groupid = [(NSNumber*) [rowVals objectForKey: GROUP_ID_KEY] intValue];
groupLabel.text = GROUP_NAMES [groupid];
NSNumber *price = (NSNumber*) [rowVals objectForKey: PRICE_KEY];
priceLabel.text = [priceFormatter stringFromNumber: price];

```

Dies ist ein ziemlich einfaches Beispiel für die Verwendung der SQLite 3-Datenbank auf dem iPhone. SQLite 3 ist eine sehr mächtige Datenbank mit Triggern, JOINS (einschließlich einer Auswahl von OUTER JOINS) und vielen Elementen des SQL-92-Standards. Die Website von SQLite sagt sogar, dass es einfacher ist, die SQL-92-Features aufzuführen, die nicht implementiert sind (werfen Sie dazu einen Blick auf *SQL Features That SQLite Does Not Implement* [Hip09]), als die, die vorhanden sind. Wenn Ihre Daten also in einer Datenbank abgelegt werden sollten, ist auf dem iPhone vermutlich schon alles vorhanden, was Sie benötigen.

Kapitel 11

Core Data

Wir hoffen, dass Sie in Kapitel 10, *Die Datenbank SQLite*, auf Seite 205 davon überzeugt wurden, eine Datenbank zu verwenden. Wenn Sie aber zur Mehrheit der Entwickler gehören, die SQL jetzt nicht so wirklich spannend finden, haben Sie Glück. Core Data ist ein erstaunlich leicht zu nutzender Wrapper für SQL, mit dessen Hilfe wir Objective-C-Entwickler unsere Zeit damit verbringen können, über Objekte statt über Abfragen nachzudenken.

Verstehen Sie uns nicht falsch – SQL ist eine tolle Sprache und ausgesprochen mächtig. Aber sie unterscheidet sich konzeptionell total von Objekten, und es kann ausgesprochen anstrengend sein, immer wieder zwischen diesen beiden Denkweisen wechseln zu müssen. Hier spielt Core Data seine Stärken aus – es ist so mächtig wie die Datenbank, aber man kann in seiner Objektwelt bleiben. Sie können Core Data nutzen, ohne sich jemals darum kümmern zu müssen, dass die Daten in einer relationalen Datenbank landen. Das ist das Schöne an diesem Framework.¹

In diesem Kapitel werden wir uns anschauen, wie man Core Data nutzt, um die persistenten Daten in einer Anwendung zu erzeugen und zu verwenden. Holen Sie sich bitte das Projekt CoreData/Conference aus dem herunterladbaren Code-Bundle. In diesem Kapitel werden einige der Details weggelassen, die wir umfassend in den vorigen Kapiteln behandelt haben, und es wird hilfreich sein, den vollständigen Code verfügbar zu haben, mit dem Sie Ihren eigenen Code vergleichen können. Ein Großteil des Codes, den Sie für eine funktionierende Anwendung benö-

¹ Natürlich hilft ein Verständnis von SQL und relationalen Datenbanken im Allgemeinen ungemein beim Debuggen, aber für die reine Anwendung und das Verstehen von Core Data ist es nicht notwendig.

tigen, ist bei der Beschreibung absichtlich weggelassen worden, da wir diese Konzepte schon vorher behandelt haben. Themen wie das Erstellen neuer View-Controller und der dazugehörigen .nib-Dateien werden gar nicht angesprochen. Lassen Sie uns die Anwendung anschauen, die wir bauen wollen.

11.1 Die Konferenzanwendung

Um die Core Data-APIs kennenzulernen, werden wir eine Anwendung namens „Conference“ bauen. Diese App soll den Benutzern dabei helfen, die Vorträge zu verwalten, die sie auf einer Konferenz besuchen wollen. Für jeden Track in der Konferenz werden wir eine Liste mit Vorträgen anzeigen. Die Anwender wählen dann die Vorträge aus, an denen sie teilnehmen wollen.

Die Konferenzanwendung besitzt vier Views. Lassen Sie uns nacheinander jede anschauen. Die Start-View enthält eine Liste mit Tracks. Der Benutzer kann Tracks hinzufügen und löschen, und wenn er den BEARBEITEN-Button angetippt und ein Track ausgewählt wird, erreicht er die Bearbeitungs-View für die Track-Informationen. So sieht die App im Bearbeitungsmodus mit drei hinzugefügten Tracks aus:



Die Liste der angezeigten Tracks kommt aus Core Data. Alle Track-Objekte werden vom persistenten Speicher geholt, und diese Liste wird auch das zugrunde liegende Datenmodell für die Tabelle sein. Wir werden in Abschnitt 11.5, *Track-Tabellen-View* auf Seite 243 viel über die Klassen sprechen, die Core Data bereitstellt, um diese Aufgabe erledigen zu können – dort behandeln wir den *RootViewController*, der hinter dieser Tabellen-View steckt.

Tracks können der Liste durch Antippen des +-Buttons in der oberen rechten Ecke hinzugefügt werden. Durch dieses Antippen bitten wir Core Data, das neue Objekt zu erstellen und es im *Managed Object Context* unterzubringen, so dass es persistiert werden kann.² Tippt der Anwender auf einen der roten Buttons in den Tabellenzeilen, kann er das Objekt löschen. Nach den Änderungen speichern wir diese dann, indem wir Core Data anweisen, sie zu persistieren.

Die nächste View ermöglicht es dem Anwender, den Namen und die Zusammenfassung des Tracks zu bearbeiten. Dieser Screenshot zeigt den gewählten IT-Track:



2 Wir werden im nächsten Abschnitt den Managed Object Context genauer betrachten.

Wurde der Name oder die Zusammenfassung eines Tracks bearbeitet, werden die Eigenschaften des Tracks aktualisiert. Diese Änderungen werden ebenfalls registriert, aber wir müssen dafür keinen Code schreiben. Core Data beobachtet die Objekte und merkt sich jede Änderung. Wenn der Anwender fertig ist, werden sie ebenfalls persistiert.

Befindet sich die Track-Liste nicht im Bearbeitungsmodus, bringt die Auswahl eines Tracks den Anwender zur Liste der Vorträge für diesen Track. Hier sehen Sie einen Screenshot dieser Liste für den iPhone-Track:



Tippt man auf den +-Button, wird – wie Sie sicher erwartet haben – ein neues Session-Objekt erzeugt. Wir nutzen Core Data, um das neue Objekt anzulegen und es in der Datenbank zu persistieren.

Und schließlich bringt Sie das Auswählen eines bestimmten Vortrags zum Bearbeitungsbildschirm für diese Session. Einen Screenshot dieser View finden Sie auf der nächsten Seite.

Wenn der Anwender mit den Änderungen an den Vortragsinformationen fertig ist, wird der Text wieder in das Session-Objekt geschrieben, was Core Data erneut registriert. Tatsächlich beobachtet Core Data alle Objekte, die von ihm verwaltet werden. Alle Änderungen, die wir an von Core Data betreuten Objekten vornehmen, werden protokolliert, um sie persistieren zu können.



Wie Sie anhand der Beschreibung sehen können, werden wir Core Data intensiv dazu nutzen, die Funktionalität unserer Konferenzanwendung umzusetzen. Wir haben noch nicht genauer darüber gesprochen, wie Core Data all das ermöglicht oder wie wir es einrichten müssen, daher wird der nächste Abschnitt erläutern, wie Core Data initialisiert wird, und dazu werden wir ein wenig demonstrieren, wie es arbeitet.

11.2 Der Core Data-Stack

Lassen Sie uns mit dem neuen Projekt beginnen. Unser Datenmodell ist listenorientiert und hierarchisch, daher wollen wir mit dem navigationsbasierten Anwendungs-Template beginnen. Geben Sie ihm den Namen Conference und vergessen Sie nicht, das Kästchen „Use Core Data for storage“ zu markieren.

Vieles von dem, was durch das Template angelegt wird, ist mittlerweile bekannt. Die Klassen für den Anwendungs-Delegate und den Root View-Controller sowie die dazugehörigen .nib-Dateien kennen Sie schon. Aber was sich nun durch das Aktivieren der Checkbox „Use

Core Data für storage“ geändert hat, wollen wir uns doch anschauen. Bevor wir mit der Implementierung unserer Anwendung beginnen, schauen wir uns den Anwendungs-Delegate an, um herauszufinden, was dazugekommen ist.

Zunächst einmal enthält die App-Delegate-Klasse (ConferenceApp-Delegate) deutlich mehr. Hier fanden wir bisher ein window und einen viewController vor, aber jetzt gibt es eine Reihe neuer Eigenschaften, die mit Core Data verbunden sind. Dies ist die Header-Datei:

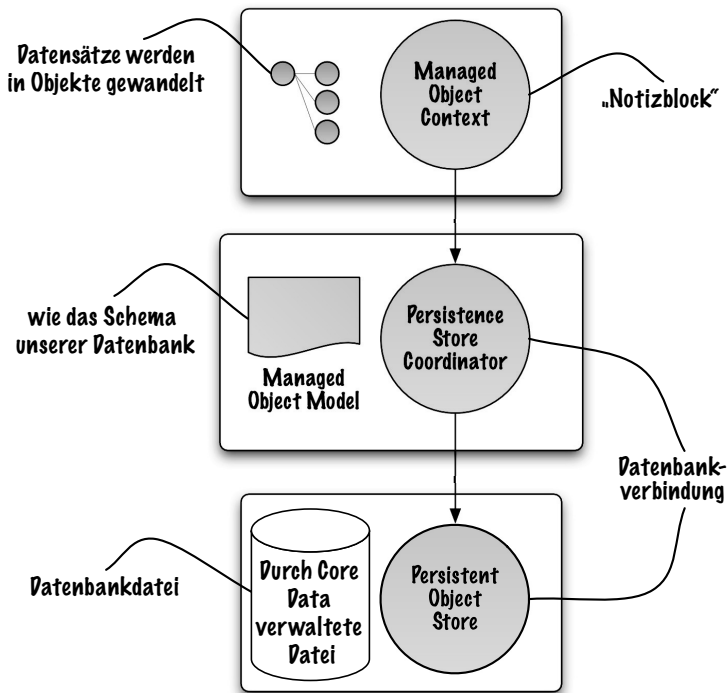
CoreData/Conference01/Classes/ConferenceAppDelegate.h

```

1  @interface ConferenceAppDelegate : NSObject <UIApplicationDelegate> {
2      NSManagedObjectModel *managedObjectModel;
3      NSManagedObjectContext *managedObjectContext;
4      NSPersistentStoreCoordinator *persistentStoreCoordinator;
5
6      UIWindow *window;
7      UINavigationController *navigationController;
8  }
9
10 @property (nonatomic, retain, readonly)
11     NSManagedObjectModel *managedObjectModel;
12 @property (nonatomic, retain, readonly)
13     NSManagedObjectContext *managedObjectContext;
14 @property (nonatomic, retain, readonly)
15     NSPersistentStoreCoordinator *persistentStoreCoordinator;
16 @property (nonatomic, readonly)
17     NSString *applicationDocumentsDirectory;
18 @property (nonatomic, retain) IBOutlet UIWindow *window;
19 @property (nonatomic, retain) IBOutlet
20     UINavigationController *navigationController;
21
22 -(IBAction)saveAction:sender;
23
24 @end

```

Diese fünf neuen Eigenschaften arbeiten mit dem Core Data-Stack zusammen. Dieser Objekt-Stack ist die Grundlage für die Funktionsweise von Core Data. Es handelt sich dabei um Objekte, die Core Data seine wirklich coolen Features ermöglichen. Die vier Objekte, die diesen Stack bilden, sind der *Persistent Object Store*, der *Persistent Store Coordinator*, der *Managed Object Context* und das *Managed Object Model*. Im folgenden Diagramm sind diese vier Objekte und ihre Beziehungen zueinander dargestellt. Wir werden jedes dieser Objekte noch genauer betrachten.



Der *Persistent Object Store* (oder auch POS) führt alle „Übersetzungen“ auf unterster Ebene aus der Objektsprache in Daten aus, aber er kümmert sich auch um das Öffnen und Schließen der zugrunde liegenden Datei. Er wird durch den Persistent Store Controller erzeugt, verwaltet und intensiv genutzt. Schauen Sie sich als Beispiel den SQLite-POS an. Dort werden Objekte in den POS gesteckt, und heraus kommt SQL-Code, der dann an die SQLite-API übergeben wird, was eine Änderung an der Datenbankdatei bewirkt. Wenn Objekte benötigt werden, fordert der Persistent Store Coordinator diese an. Sie werden dann vom POS in SQL-Abfragen umgewandelt. Der SQL-Code erreicht schließlich die SQLite-API, und die zurückgelieferten Datensätze werden vom Persistent Store Coordinator genutzt, um Objekte zu erzeugen.

Core Data wird mit drei POS-Implementierungen ausgeliefert: SQLite, Binary und Memory. Ich empfehle, immer die SQLite-Implementierung zu nutzen, da sie sich am leichtesten debuggen lässt und man auch am einfachsten sieht, was tatsächlich persistiert wird. Die Binary-Implementierung ist gut zu verwenden, wenn Sie wirklich kleine Datensätze haben, den Performance-Overhead von SQL vermeiden wollen und Ihre Daten sehr einfach sind. Die Memory-Implementierung ist für temporäre Daten einsetzbar, die nicht über den POS hinaus persistiert werden müssen.

Der *Persistent Store Coordinator* (PSC) dient als generalisierende Hülle für den Persistent Object Store. Im allgemeinsten Fall können Sie mehrere Object Stores haben, auf dem iPhone wird das aber eher selten passieren. Dort, wo der Persistent Object Store speziell auf einen bestimmten Speicher ausgerichtet ist, ist der Persistent Store Coordinator allgemeiner. Er nutzt das Managed Object Model, um Form und Anordnung der Objekte zu verstehen, die persistiert werden sollen.

Das *Managed Object Model* (MOM) enthält die Beschreibung – oder die Metadaten – Ihres Modells. Hier beschreiben Sie die *Entitäten*³ (Entities) und deren Eigenschaften, die das Modell Ihrer Anwendung bilden. In Conference werden wir die beiden Entitäten Session und Track haben.

Das letzte Objekt im Stack – mit dem wir am meisten interagieren werden – ist der *Managed Object Context* (MOC). Der MOC wird als „Notizblock“ genutzt. Objekte werden über den Stack in den MOC gezogen und dann dort gehalten, während wir sie ändern. Alles, was im MOC neu angelegt, gelöscht und aktualisiert wird, verbleibt dort, bis wir den MOC anweisen, die Änderungen zu speichern. Dann werden die Änderungen durch den Stack gereicht und mit jedem Schritt näher an die Sprache des POS gebracht, wo schließlich native Befehle daraus werden (zum Beispiel SQL-Anweisungen beim SQLite-POS) die an den Persistierungsspeicher gehen.

Dieser Objekt-Stack wird für uns durch den Anwendungs-Delegate angelegt. Der gesamte Code zum Aufbau des Stacks wird generiert, wenn wir dieses kleine Kästchen „Use Core Data for storage“ markieren. Lassen Sie uns anschauen, wie diese Objekte erzeugt werden.

11.3 Aufbau des Core Data-Stack

Ihnen ist vermutlich schon aufgefallen, dass alle mit Core Data in Verbindung stehenden Eigenschaften in der Header-Datei von `ConferenceAppDelegate` schreibgeschützt sind. Das liegt daran, dass alle *abgeleitet* (oder bei Bedarf erstellt) werden. Statt sich auf irgendwelchen äußeren Objekte zu verlassen, die diese Objekte erstellen und einrichten, werden sie vom Anwendungs-Delegate erzeugt. Lassen Sie uns die vom Template erzeugte Implementierung anschauen und untersuchen, wie der Stack erstellt wird.

3 Eine Entität (Entity) ist etwas, das modelliert ist, wie in unserem Beispiel die Session. Eine Entität kümmert sich um die Features des modellierten Gegenstands, wie zum Beispiel seine Attribute und Beziehungen zu anderen Entitäten.

Fangen wir mit dem unteren Ende des Stacks beim Persistent Store Coordinator und dem Persistent Object Store an. Dies ist der Code:

CoreData/Conference01/Classes/ConferenceAppDelegate.m

```

1 - (NSPersistentStoreCoordinator *)persistentStoreCoordinator {
2   if (persistentStoreCoordinator != nil) {
3     return persistentStoreCoordinator;
4   }
5
6   NSURL *storeUrl = [NSURL fileURLWithPath:
7                       [[self applicationDocumentsDirectory]
8                        stringByAppendingPathComponent:@"Conference.sqlite" ]];
9
10      NSError *error = nil;
11      persistentStoreCoordinator =
12      [[NSPersistentStoreCoordinator alloc]
13       initWithManagedObjectModel:[self managedObjectModel]];
14      if (![persistentStoreCoordinator
15          addPersistentStoreWithType:NSSQLiteStoreType
16          configuration:nil
17          URL:storeUrl
18          options:nil
19          error:&error]) {
20        // Fehlerbehandlung
21      }
22
23      return persistentStoreCoordinator;
24  }

```

In dieser Methode geschehen drei Dinge. Zunächst holen wir uns das Documents-Verzeichnis und fügen am Ende ein `Conference.sqlite` hinzu. Mit diesem Pfad erzeugen wir eine URL für unsere Datenbankdatei. Als Nächstes erzeugen wir in Zeile 11 eine Instanz von `NSPersistentStoreCoordinator` mit dem Managed Object Model. Und schließlich konfigurieren wir ab Zeile 14 einen neuen Persistent Object Store. Der POS nutzt SQLite und legt die Datenbank in der durch `storeURL` spezifizierten Datei ab. Das letzte Argument ist ein Zeiger auf einen Error. Geht etwas schief, wird `error` auf eine Instanz von `NSError` gesetzt, in der detailliertere Informationen über den Fehler enthalten sind.

Die Eigenschaft `applicationDocumentsDirectory` wird genutzt, um das Documents-Verzeichnis der Anwendung zu finden. Sie wird bei jedem Aufruf der Methode neu ermittelt. Weitere Informationen finden Sie in Abschnitt 10.3, *Ihre Datenbank auf das iPhone bringen*, auf Seite 212.

Mit diesen wenigen Zeilen Code haben wir den Core Data-Stack mit einer SQLite-Datenbankdatei verbunden. Lassen Sie uns nun anschauen, wie wir nun das Managed Object Model laden. Dies ist der Code:

CoreData/Conference01/Classes/ConferenceAppDelegate.m

```

- (NSManagedObjectModel *)managedObjectModel {
    if (managedObjectModel != nil) {
        return managedObjectModel;
    }
    managedObjectModel = [[NSManagedObjectModel mergedModelFromBundles:nil]
                           retain];
    return managedObjectModel;
}

```

In diesem Code nutzen wir `mergedModelFromBundles:`, um alle Modelldateien zu holen, die sich in unserem Anwendungs-Bundle befinden, und sie in einem MOM zusammenzuführen. Diese Klassenmethode durchsucht das gesamte Anwendungs-Bundle nach Modelldateien, die dann geladen und im Gesamt-MOM zusammengefasst werden. In der Praxis haben die meisten iPhone-Anwendungen allerdings nur ein Modell.

Das letzte Element des Stacks ist der Managed Object Context. Dies ist der Code zum Erzeugen des MOC:

CoreData/Conference01/Classes/ConferenceAppDelegate.m

```

- (NSManagedObjectContext *) managedObjectContext {
    if (managedObjectContext != nil) {
        return managedObjectContext;
    }

    NSPersistentStoreCoordinator *coordinator =
        [self persistentStoreCoordinator];
    if (coordinator != nil) {
        managedObjectContext = [[NSManagedObjectContext alloc] init];
        [managedObjectContext setPersistentStoreCoordinator: coordinator];
    }
    return managedObjectContext;
}

```

Dieser Code erledigt zwei Dinge: Er holt sich den Persistent Store Coordinator und nutzt ihn, um den Managed Object Context zu konfigurieren.

Nachdem wir nun gesehen haben, wie der Core Data-Stack erzeugt wird, und wir uns darüber unterhalten haben, wie man ihn nutzt, werden Sie jetzt erfahren, wie Sie Core Data in Conference verwenden. Wir beginnen mit dem Modellieren der Klassen, die unser Modell darstellen.

11.4 Modellieren

Da Sie nun wissen, wie Core Data mit dem Template eingerichtet wird, wollen wir unserer Anwendung Leben einhauchen, indem wir unser Managed Object Model bauen.

Das Managed Object Model ist ein zentraler Teil der Interaktion von Core Data mit Ihrem Modell. Core Data nutzt die Entität, wenn es zwischen der Objective-C-Welt und der SQLite-Welt übersetzt. Das Modell entspricht zwar dem Schema Ihrer Datenbank, enthält aber noch weitere Informationen zur „Übersetzung“ von Objective-C nach SQL und zurück. Vereinfacht gesagt, entspricht jede Entität einer Tabelle in der Datenbank und jedes Attribut einer Spalte darin. Auf der Seite von Objective-C entspricht jede Entität einer Klasse und jedes Attribut einer Eigenschaft dieser Klasse. Es gibt natürlich viele, viele Details beim Abbilden von Objekten in Datensätzen und umgekehrt, die das MOM berücksichtigen muss, aber dank der in Xcode vorhandenen großartigen Tools müssen Sie das gar nicht vollständig durchschauen. Lassen Sie uns das UI kurz anschauen. Klicken Sie in Xcode doppelt auf `Conference.xcdatamodel`. Sie sollten nun etwas sehen, das mehr oder weniger wie der folgende Screenshot aussieht:



Wir werden diese Template-Daten bearbeiten, so dass sich unsere Entitäten Track und Session dort wiederfinden. Im Prinzip müssen wir folgende Schritte durchführen:

- Die vom Template vorbereitete Entität Event nach Track ändern.
- Die Attribute name und trackAbstract sowie die Relation sessions zur Entität Track hinzufügen.
- Die Entität Session erstellen.
- Die Attribute name, sessionAbstract und sessionID und die Relation track zur Entität Session hinzufügen.
- Die Klassen Track und Session erzeugen und diese neuen Klassen zum Projekt hinzufügen.

Mit diesen beiden Entitäten und den Relationen zwischen beiden erhält Core Data die Informationen, die es benötigt, um sie persistieren zu können.

Benennen Sie also als Erstes die Entität Event in Track um. Markieren Sie die Entität und ändern Sie dann im Attributes Inspector den Namen von Event in Track. Ändern Sie auch den Klassennamen von NSManagedObject in Track. Track hat zwei Attribute, name und trackAbstract. Beide sind Strings ohne Constraints (Länge, Regex und so weiter). Klicken Sie auf den +-Button unter der Eigenschaftsliste (rechts von der Entitätsliste) und wählen Sie ADD ATTRIBUTE. Wenn Sie die Attribute hinzugefügt haben, sollte das Ganze in etwa so aussehen, wie in der folgenden Abbildung.

Achten Sie darauf, bei beiden Attributen „Optional“ zu deaktivieren. Das ist zwar nicht unbedingt notwendig, hilft aber dabei, zu verhindern, dass falsche Daten in die Datenbank gelangen. Wie Ihnen vermutlich schon aufgefallen ist, gibt es noch eine ganze Menge weiterer Möglichkeiten, die Constraints anzupassen, die Core Data auf Ihre Attribute anwendet. Wenn Sie zum Beispiel einen regulären Ausdruck (auch bekannt als *Regex*) im Attributes Inspector angegeben haben, stellt Core Data sicher, dass jeder Wert, der in dieses Attribut geschrieben wird, zur Regex passt, bevor er gespeichert wird. Schlägt die Übereinstimmung mit einem der Constraints fehl, wird ein Validierungsfehler geworfen. Mit diesem Fehler können wir eine für den Anwender verständliche Mitteilung erzeugen, damit dieser weiß, was er korrigieren muss. Wir werden uns damit jetzt nicht genauer befassen, aber die *Core Data documentation* [App09b] enthält mehr Informationen über die Ausgabe von Fehlern an den Anwender.



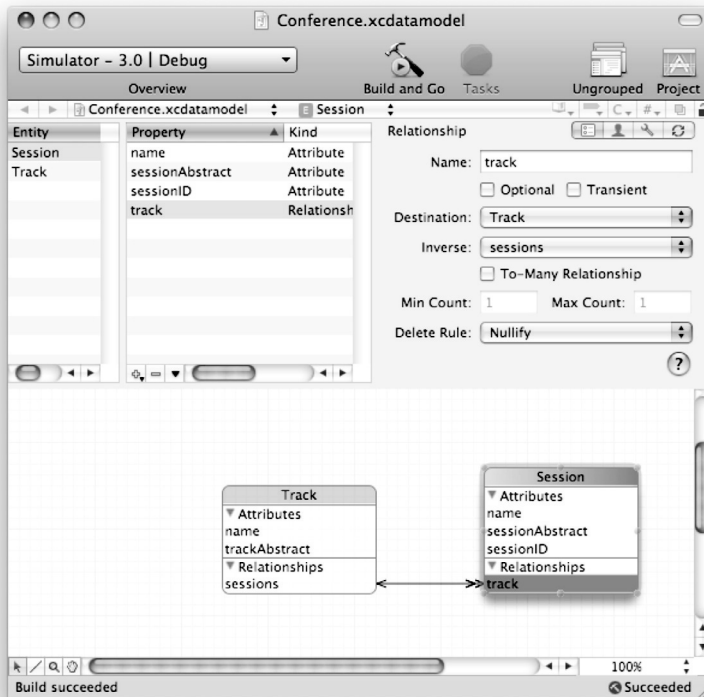
Achten Sie darauf, bei beiden Attributen „Optional“ zu deaktivieren. Das ist zwar nicht unbedingt notwendig, hilft aber dabei, zu verhindern, dass falsche Daten in die Datenbank gelangen. Wie Ihnen vermutlich schon aufgefallen ist, gibt es noch eine ganze Menge weiterer Möglichkeiten, die Constraints anzupassen, die Core Data auf Ihre Attribute anwendet. Wenn Sie zum Beispiel einen regulären Ausdruck (auch bekannt als *Regex*) im Attributes Inspector angegeben haben, stellt Core Data sicher, dass jeder Wert, der in dieses Attribut geschrieben wird, zur Regex passt, bevor er gespeichert wird. Schlägt die Übereinstimmung mit einem der Constraints fehl, wird ein Validierungsfehler ausgelöst. Mit diesem Fehler können wir eine für den Anwender verständliche Mitteilung erzeugen, damit dieser weiß, was er korrigieren muss. Wir werden uns damit jetzt nicht genauer befassen, aber die *Core Data documentation* [App09b] enthält mehr Informationen über die Ausgabe von Fehlern an den Anwender.

Nachdem wir nun die Entität Track haben, müssen wir noch die Entität Session erstellen. Unter der Liste mit Entitäten (in der sich momentan nur Track befindet) klicken Sie auf den Button +. Geben Sie der neuen

Entität den Namen *Session*, ändern Sie den Klassennamen ebenfalls in *Session* und fügen Sie die drei String-Attribute *sessionID*, *name* und *sessionAbstract* hinzu. Auch hier sollten die Attribute von *Session* keine Constraints besitzen, und „Optional“ sollte abgeschaltet sein. Wenn Sie fertig sind, müsste der Bildschirm in etwa so aussehen:

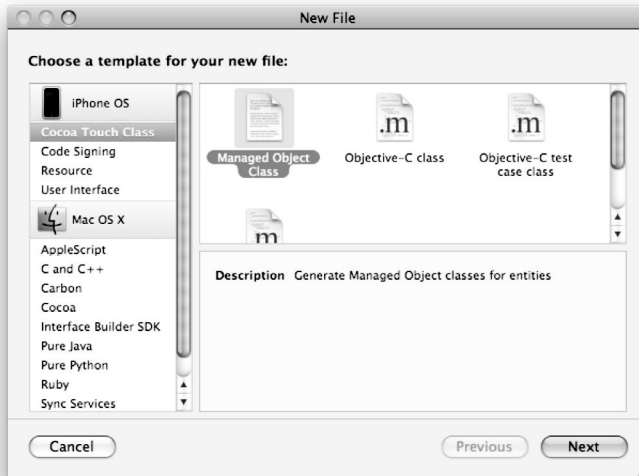


Jetzt haben wir beide Entitäten eingerichtet und müssen zwischen ihnen noch die Relation herstellen. Markieren Sie die Entität *Track* und klicken Sie auf den **+**-Button unter der Eigenschaftsliste (oben links). Wählen Sie **ADD RELATIONSHIP** und geben Sie der neuen Relation den Namen *sessions*. Sorgen Sie dafür, dass das Ziel der Relation die Entität *Session* ist, markieren Sie die Checkbox „To-Many Relationship“ und setzen Sie die „Delete Rule“ auf „Cascade“. Als Nächstes müssen wir der Entität *Session* die Relation in Gegenrichtung hinzufügen. Markieren Sie diese Entität, fügen Sie eine Relation mit dem Namen *track* hinzu, setzen das Ziel auf *Track* und definieren Sie die „Inverse“ als „sessions“. Setzen Sie die „Delete Rule“ für die neue Relation auf „Nullify“ und schalten Sie „Optional“ aus. Wenn Sie fertig sind, sollte das Modell so aussehen:

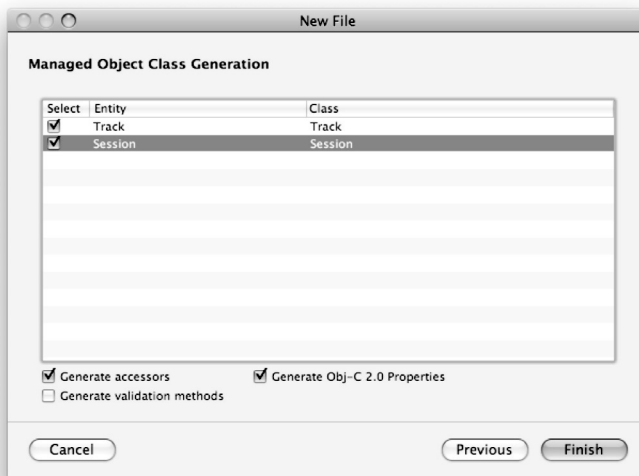


Der nächste Schritt ist das Generieren der Klassen, die zur Repräsentation dieses Modells in unserer Anwendung genutzt werden. Fügen Sie in Xcode eine neue Gruppe namens „Model Classes“ ein und markieren Sie sie. Wechseln Sie zurück zum Modell, markieren Sie alle Entitäten mit - und wählen Sie dann aus dem Menü FILE NEW FILE. Markieren Sie in der nun erscheinenden Dialogbox die „Managed Object Class“ (siehe nächsten Screenshot).⁴

⁴ Wenn sich der Modell-Editor beim Auswählen des Menüeintrags FILE NEW FILE nicht im Vordergrund befindet, erhalten Sie im New File Wizard keine Option „Managed Object Class“.



Auf der nächsten Seite des Wizard klicken Sie auf NEXT. Sie müssen hier nichts ändern, aber darauf achten, dass das Ziel „Conference“ markiert ist. Auf der letzten Seite des Wizards kontrollieren Sie, ob die Checkbox neben Session und Track markiert ist, so dass der Generator beide Entitäten nutzt, wenn er die Klassen erstellt. Die Seite sollte in etwa so aussehen:



Nach dem Klick auf den Button FINISH sollten Sie für die beiden Klassen Track und Session je eine Header- und eine Implementierungsdatei erhalten.

Lassen Sie uns den Code anschauen, den der Generator erzeugt hat. Dies ist die Header-Datei für die Klasse Session:

CoreData/Conference02/ModelClasses/Session.h

```
@class Track;

@interface Session : NSObject {
}

@property (nonatomic, retain) NSString sessionID;
* @property (nonatomic, retain) NSString name;
* @property (nonatomic, retain) NSString sessionAbstract;
* @property (nonatomic, retain) Track * track;

@end
```

Das ist jetzt nicht vollkommen unerwartet. Der Generator hat für jedes der Attribute und für jede Relation im Modell eine Eigenschaft generiert, wobei alle ihre Objektwerte „in Besitz“ nehmen. Als Nächstes werfen wir einen Blick auf die Implementierung. Dies ist der Code der .m-Datei:

CoreData/Conference02/ModelClasses/Session.m

```
@implementation Session

@dynamic sessionID;
@dynamic name;
@dynamic sessionAbstract;
@dynamic track;

@end
```

Nun, hier steht nicht wirklich viel drin. Das neue und interessante Element ist die Nutzung von `@dynamic` für die Eigenschaften. Diese Deklaration teilt dem Compiler mit, dass die Eigenschaften zur Laufzeit Get-/Set-Methoden haben werden, so dass er sich nicht darum kümmern muss, sie bereitzustellen. Core Data liefert uns diese Methoden, daher brauchen wir uns keine Gedanken darum zu machen.

Schauen wir als Nächstes die Klasse Track an. Dies ist die Header-Datei:

CoreData/Conference02/ModelClasses/Track.h

```

1  @class Session;
2
3  @interface Track : NSObject {
4  }
5
6  @property (nonatomic, retain) NSString * trackAbstract;
7  @property (nonatomic, retain) NSString * name;
8  @property (nonatomic, retain) NSSet* sessions;
9
10 @end
11
12 @interface Track (CoreDataGeneratedAccessors)
13 -(void)addSessionsObject:(Session *)value;
14 -(void)removeSessionsObject:(Session *)value;
15 -(void)addSessions:(NSSet *)value;
16 -(void)removeSessions:(NSSet *)value;
17 @end

```

Die obere Hälfte ist vermutlich erneut das, was Sie erwartet haben – eine Eigenschaft für jedes Attribut und jede Relation der Entität. Die Kategorie `CoreDataGeneratedAccessors` ist neu, daher wollen wir sie genauer betrachten. Eine Kategorie in Objective-C ist eine Möglichkeit, einem Objekt Methoden hinzuzufügen. In Zeile 12 wird die Kategorie `CoreDataGeneratedAccessors` für die Klasse `Track` deklariert. Alle in dieser Kategorie deklarierten Methoden (bis zum `@end`) werden Teil des Interface. Schauen wir uns jetzt die Implementierung an:

CoreData/Conference02/ModelClasses/Track.m

```

@implementation Track

@dynamic trackAbstract;
@dynamic name;
@dynamic sessions;
@end

```

Das ist nicht sehr viel ... Die in der Kategorie `CoreDataGeneratedAccessors` deklarierten Methoden werden von Core Data zur Laufzeit generiert (daher der Name der Kategorie). Wir brauchen nur die Kategorie in der Header-Datei, damit wir die Methoden ohne Compiler-Warnung aufrufen können. Diese Objective-C-Magie wird Ihnen präsentiert von: Key-Value Coding (KVC). Die Dokumentation zum *Key-Value Coding* [App08e] von Apple enthält dazu alle Details. Hier müssen Sie sich aber nicht darum kümmern. Es reicht, zu wissen, dass Sie `addSessionsObject:` für jede Instanz von `Track` aufrufen können.

Nachdem unser Modell nun fertig ist, kann Core Data es persistieren. Wir wollen uns daher den Code anschauen, der notwendig ist, damit die Tabellen-View `Track` funktioniert.

11.5 Track-Tabellen-View

Es ist gut, dass Sie bereits wissen, wie Tabellen-Views funktionieren. Sie fragen ihre Data Source nach der Anzahl an Abschnitten, der Anzahl an Zeilen in jedem Abschnitt und dann nach dem Feld für eine bestimmte Position. Dieser Teil unserer Aufgabe ist schon wohlbekannt. Neu ist der Adapter, der zwischen Core Data und der API für die Tabellen-View-Data Source sitzt. Die Klasse für diesen Adapter ist `NSFetchedResultsController` (oder kurz `FRC`). Wir wollen schauen, wie er im `RootViewController` genutzt wird, und danach einen Blick auf den Code werfen, den wir zum Anlegen verwenden werden.

Lassen Sie uns mit den grundlegenden Data Source-Methoden beginnen. Die erste ist `numberOfSectionsInTableView:` – wie Sie wissen, soll diese Methode der Tabellen-View die Anzahl an Abschnitten für die anzuzeigenden Daten liefern. Dazu fragen wir einfach den `Fetched Results Controller`:

CoreData/Conference02/Classes/RootViewController.m

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return [[fetchedResultsController sections] count];
}
```

Der `FRC` weiß, wie viele Abschnitte es in den Daten gibt, die er verwaltet, daher kann er diese Zahl einfach zurückgeben. Die nächste wichtige Methode der Data Source ist `tableView:numberOfRowsInSection:`. Diese Methode gibt die Anzahl an Zeilen für einen bestimmten Abschnitt zurück. Auch hier weiß der `Fetched Results Controller`, wie viele Zeilen es sind. Der Code sieht so aus:

CoreData/Conference02/Classes/RootViewController.m

```
-(NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    return [[[fetchedResultsController sections] objectAtIndex:section]
        numberOfObjects];
}
```

Die Methode `tableView:cellForRowAtIndexPath:` wird aufgerufen, um das Feld zu füllen und anzuzeigen. Und wieder hilft uns der `Fetched Results Controller` mit der Methode `objectAtIndex:`. Dies ist der Code:

CoreData/Conference02/Classes/RootViewController.m

```
1 - (void)configureCell:(UITableViewCell *)cell withTrack:(Track *)track {
2     cell.textLabel.text = track.name;
3     cell.detailTextLabel.text = track.trackAbstract;
4 }
5
```

```

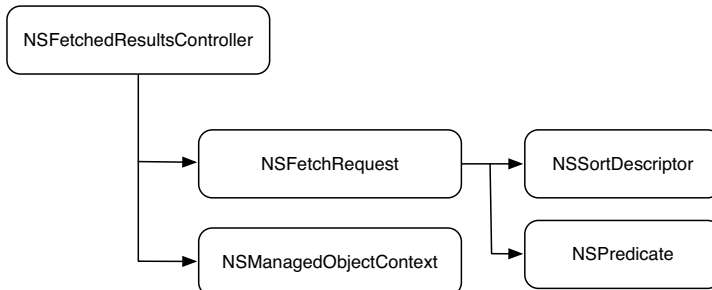
6 - (UITableViewCell *)tableView:(UITableView *)tableView
7   cellForRowAtIndexPath:(NSIndexPath *)indexPath {
8     static NSString *CellIdentifier = @"Cell" ;
9
10    UITableViewCell *cell =
11      [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
12    if (cell == nil) {
13      cell = [[UITableViewCell alloc]
14              initWithStyle:UITableViewCellStyleSubtitle
15              reuseIdentifier:CellIdentifier] autorelease];
16    }
17
18    Track *track = [fetchResultsController objectAtIndex:indexPath:indexPath];
19    [self configureCell:cell withTrack:track];
20
21    return cell;
22 }

```

Die erste Methode in diesem Codeblock richtet das Feld ein, indem es den Text von `textLabel` auf den Wert von `name` des Tracks setzt und den Text von `detailTextLabel` auf `trackAbstract`. Das Füllen des Felds wurde in eine eigene Methode ausgelagert, da wir zwei Stellen haben, an denen es aufgerufen wird. Die erste ist `tableView:cellForRowAtIndexPath:`, die zweite werden wir uns gleich ansehen.

Da der FRC das Datenmodell kennt, kann er uns ohne Probleme das benötigte Objekt mitteilen. Der Code für `tableView:cellForRowAtIndexPath:` enthält bis Zeile 18 keine besonderen Dinge. Hier kommt der FRC wieder ins Spiel. Wir fragen nach dem Objekt für einen Indexpfad, das uns auch prompt geliefert wird. Wie Sie sehen können, ist das Füllen einer Tabelle durch den FRC eine Kleinigkeit. Lassen Sie uns jetzt anschauen, wie wir den FRC erzeugen und welche zusätzliche Funktionalität wir dort nutzen können.

Der `Fetch Results Controller` ist – wie schon gesagt – ein Adapter, um das Auslesen und Anzeigen von Daten aus Core Data in einer `UITableView` zu erleichtern. Dazu nutzt der FRC eine ganze Reihe anderer Objekte. Das folgende Diagramm zeigt alle Elemente:



Der Fetched Results Controller nutzt den Fetch Request und den Managed Object Context, um an die Daten zu gelangen. Der Fetch Request kann optional einen Sortierdeskriptor oder ein Prädikat nutzen. Die Sortierreihenfolge sorgt dafür, dass das Ergebnis des Fetch sortiert wird – wie eigentlich nicht anders zu erwarten. Das Prädikat beschränkt die Anzahl an Objekten, die der Fetch liefert. Wir können hier nicht ins Detail gehen, werfen Sie dazu besser einen Blick in *Core Data: Apple's API for Persisting Data Under Mac OS X* [Zar09].

Neben der Vereinfachung beim Füllen der Tabellen-View kümmert sich der FRC sehr aggressiv um den Speicher. Er stellt sicher, dass nur die Objekte, die tatsächlich benötigt werden, im Speicher gehalten werden, und schmeißt die anderen raus, wenn eine Speicherwarnung ausgelöst wird.

Dies ist der Code zum Erstellen des FRC für die Track-Tabellen-View:

CoreData/Conference02/Classes/RootViewController.m

```

1  - (NSFetchedResultsController *)fetchedResultsController {
2      if (fetchedResultsController != nil) {
3          return fetchedResultsController;
4      }
5
6      NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
7      NSEntityDescription *entity =
8      [NSEntityDescription entityForName:@"Track"
9       inManagedObjectContext:managedObjectContext];
10     [fetchRequest setEntity:entity];
11
12     NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
13     initWithKey:@"name" ascending:YES];
14     [fetchRequest setSortDescriptors:
15     [NSArray arrayWithObject:sortDescriptor]];
16
17     NSFetchedResultsController *aFetchedResultsController =
18     [[NSFetchedResultsController alloc]
19      initWithFetchRequest:fetchRequest
20      managedObjectContext:managedObjectContext
21      sectionNameKeyPath:nil
22      cacheName:@"Root" ];
23     aFetchedResultsController.delegate = self;
24     self.fetchedResultsController = aFetchedResultsController;
25
26     [aFetchedResultsController release];
27     [fetchRequest release];
28     [sortDescriptor release];
29
30     return fetchedResultsController;
31 }
```

In diesem Code werden drei Dinge getan: der Fetch Request wird erzeugt, ein Sortierdeskriptor hinzugefügt und dann das FRC-Objekt angelegt. Der Fetch Request besitzt viele Features, die es uns ermöglichen, das Ermitteln der Daten zu optimieren. In diesem einfachen Beispiel spielt das keine Rolle, aber für große Datenmengen ist es keine schlechte Idee, sich in die Details einzuarbeiten (*Core Data* [Zar09] und *Core Data Documentation* [App09b] sind beides gute Referenzen).

Als Nächstes wird die Sortierreihenfolge festgelegt. Hier wollen wir, dass die Track-Objekte absteigend nach dem Inhalt der Eigenschaft `name` sortiert werden. Für unsere Anwender ist das wichtig. Geben wir beim FRC-Fetch-Request keine Sortierreihenfolge an, kann die Reihenfolge bei jedem Datenbankabruf unterschiedlich ausfallen.⁵

Wir erzeugen den FRC mit dem Fetch-Request-Objekt, das wir gerade aufgesetzt haben, und dem Managed Object Context, den wir vom Anwendungs-Delegate erhalten. (Wie haben diesen Code nicht gezeigt, aber er findet sich in der Methode `applicationDidFinishLaunching:` des `ConferenceAppDelegate` im herunterladbaren Code.) Für den Abschnittsschlüssel nutzen wir `nil`, weil wir keine Abschnitte haben wollen. Bei Anwendungen, die einen Abschnitt brauchen, würden Sie hier den Namen einer Eigenschaft angeben, die der FRC nutzen würde, um zu bestimmen, zu welchem Abschnitt das Objekt gehört.

Der FRC kümmert sich um die Abschnitte für unsere Tabellen-View, wir müssen nur einen Schlüsselpfad angeben. Das ist ein weiterer „magischer“ Effekt des Key-Value Coding. Und auch hier ist die Apple-Dokumentation *Key-Value Coding* [App08e] für weitere Details hilfreich. Der Fetched Results Controller nutzt den Schlüsselpfad, um die Werte zu erhalten, die die Abschnittsnamen darstellen. Bei jedem ermittelten Objekt ruft der FRC die Methode `valueForKeyPath:` auf. Mehrdeutigkeiten werden dann eliminiert und die Namen daraus gebildet. Wenn Sie dieses Feature nutzen, ist das Debuggen einfacher, wenn Sie die Tabellen-View im Grouped Style darstellen oder die Methode `tableView:titleForHeaderInSection:` implementieren,⁶ so dass die Abschnitte offensichtlich sind.

Wir setzen auch den Cache-Namen für unseren FRC auf `Root`. Der Fetched Results Controller puffert eine ganze Reihe von Informationen zu den Abschnitten unter diesem Namen. Beim Starten prüft der FRC,

⁵ Datenbanken speichern ihre Daten in ungeordneten Mengen.

⁶ Sie erhalten die Abschnittsinformation aus dem FRC über einen Aufruf von `[frc.sections objectAtIndex:section]`. Den Titel bekommen Sie dann über die Eigenschaft `name` der Abschnittsinformation.

ob ein Cache mit diesem Namen vorhanden ist und ob er ihn nutzen kann. Wenn das der Fall ist, verwendet er diesen, statt die Abschnittsinformationen neu zu berechnen.

Nun setzen wir in Zeile 23 unseren RootViewController als delegate des Fetched Results Controller ein. Dies wird im nächsten Abschnitt wichtig, in dem wir neue Tracks hinzufügen wollen.

Abgesehen vom Formatieren und Anpassen des Entitätsnamens entspricht dieser Code dem, was das Template schon für uns vorbereitet hat.

11.6 Die Tracks einlesen

Der Fetched Results Controller hat es uns wirklich leicht gemacht, die Daten in unserer Tabellen-View anzuzeigen. Jetzt wollen wir schauen, wie der FRC selbst die Daten erhält.

Da wir möchten, dass die Tracks schon beim Starten angezeigt werden, setzen wir den Code zum Einlesen in die Methode viewDidLoad:

CoreData/Conference02/Classes/RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];

    // Edit- und Add-Buttons hinzufügen.
    self.navigationItem.leftBarButtonItem = self.editButtonItem;

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
                                   initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                                   target:self
                                   action:@selector(insertTrack)];
    self.navigationItem.rightBarButtonItem = addButton;
    [addButton release];

    NSError *error = nil;
    if (![self.fetchedResultsController performFetch:&error]) {
        // Fehler-Handling ...
    }

    self.title = @"Tracks" ;
}
```

Nach dem Hinzufügen der Buttons schicken wir ein performFetch: an den Fetched Results Controller. Der FRC liest dann die durch den Fetch Request angeforderten Objekte aus der Datenbank. Im Hintergrund werden alle Datenstrukturen eingerichtet, die notwendig sind, um die Anzahl der Abschnitte und andere Informationen zurückgeben zu können.

11.7 Tracks ändern

Sie wissen jetzt, in welcher Weise der Fetched Results Controller uns dabei hilft, die Daten in die Tabellen-View zu bekommen. Schauen wir uns also nun an, was passiert, wenn wir diese Daten ändern. Der FRC unterstützt uns per Delegation dabei, die Tabellen-View mit den Änderungen synchron zu halten. Bevor wir in die Details gehen, schauen wir uns die Methode `insertTrack` an, um zu erfahren, wie wir neue Objekte einfügen. Dies ist der Code:

CoreData/Conference02/Classes/RootViewController.m

```
- (void)insertTrack {
    self.firstInsert = [self.fetchedResultsController.sections count] == 0;

    NSManagedObjectContext *context =
        [self.fetchedResultsController managedObjectContext];
    NSEntityDescription *entity =
        [self.fetchedResultsController.fetchRequest entity];
    Track *track =
        [NSEntityDescription insertNewObjectForEntityForName:[entity name]
         inManagedObjectContext:context];

    [track setValue:@"Track" forKey:@"name" ];
    [track setValue:@"Dies ist ein toller Track" forKey:@"trackAbstract"
];

    NSError *error = nil;
    if (![context save:&error]) {
        // Fehler-Handling ...
    }
}
```

Zunächst prüfen wir in dieser Methode, ob es das erste Mal ist, dass etwas eingefügt wird. Das Ergebnis nutzen wir später in den FRC-Delegate-Methoden. Anschließend erzeugen wir ein neues Track-Objekt. Da wir unsere Objekte mit Core Data verwalten, müssen wir sie auch über die Core Data-Methoden erzeugen. Die Implementierung von `insertNewObjectForEntityForName:inManagedObjectContext:` stellt dabei sicher, dass die richtigen Initialisierungsvorgänge durchgeführt werden. Dann nutzen wir die KVC-Methode `setValue:forKey:`, um die Eigenschaften `name` und `trackAbstract` zu setzen. Auch wenn wir die Eigenschafts-Set-Methoden nutzen könnten (also `track.name = @"Track";`), wird sehr häufig der KVC-Ansatz genutzt.

Als Letztes bitten wir den Kontext durch den Aufruf der Methode `save:`, die Änderungen zu speichern. Wir ignorieren eventuelle Fehler, was natürlich in einer produktiven Anwendung kein gutes Vorgehen ist. `NSError` ist eine allgemeine Klasse, die viele Daten enthalten kann. Es ist

schwierig, eine generelle Vorgehensweise, wie mit ihnen umzugehen ist, zu beschreiben. Allerdings kann man auf manche Fehler gut reagieren, auf andere nicht. Wenn es zum Beispiel beim Öffnen der Datenbankdatei Probleme gibt, können Sie oder Ihr Anwender wenig tun, um das zu umgehen. Ist es jedoch ein einfacher Fehler, zum Beispiel, dass das Objekt in einem anderen Managed Object Context geändert wurde, können Sie dem Anwender eine `UIAlertView` präsentieren, in der er gefragt wird, ob er widerrufen oder überschreiben möchte. Während der frühen Entwicklungsphasen protokollieren wir üblicherweise die Fehler mit `NSLog` in der Konsole. Als Teil des „Abrundens“ der Anwendung schauen wir uns dann genauer an, was für Fehler auftreten, um zu entscheiden, welche davon der Anwender zu sehen bekommen soll. Fehlerbehandlung ist immer sehr individuell zu betrachten – Sie sollten sich auf jeden Fall Gedanken darüber machen, wie Sie in Ihrer App mit Fehlern umgehen.

Nachdem wir nun gesehen haben, wie wir Tracks hinzufügen, wollen wir sie auch löschen können. Da Sie ja nun Tabellen-Views aus dem Effeff beherrschen, wissen Sie, dass wir die Methode `tableView:commitEditingStyle:forRowAtIndexPath:` implementieren, um die Änderungen des Anwenders zu bestätigen (zum Beispiel über den roten Lösch-Button). Dies ist der Code für die Methode:

CoreData/Conference02/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {
        NSManagedObjectContext *context =
            [fetchResultsController managedObjectContext];
        Track *track = [fetchResultsController objectAtIndex:indexPath.index];
        [context deleteObject:track];

        // Speicherns des Context.
        NSError *error;
        if (![context save:&error]) {
            // Fehler-Handling ...
        }
    }
}
```

In dieser Methode holen wir den ausgewählten Track und weisen den Kontext an, das Objekt zu löschen, indem wir die Methode `deleteObject:` aufrufen. Dann bitten wir den Kontext, die Änderungen mithilfe der Methode `save:` zu speichern.

Ihnen ist beim Einfügen wie auch beim Löschen vielleicht aufgefallen, dass wir die Tabellen-View nicht aktualisieren. Das liegt daran, dass der FRC uns über eine seiner Delegate-Methoden mitteilt, wann eine Änderung verarbeitet wurde.

Als Delegate des Fetched Results Controller erhalten wir auf jeden Fall die Hinweise zu den Änderungen am Persistierungszustand unserer Daten. Es gibt im Protokoll `NSFetchedResultsControllerDelegate` vier Delegate-Methoden. Die Methode `controllerWillChangeContent:` lässt `delegate` wissen, dass der FRC die Daten ändern wird. An dieser Stelle kann man die Tabellen-View durch `beginUpdates` wissen lassen, dass Sie nun an den Zeilen und Abschnitten arbeiten werden. Sind die Änderungen abgeschlossen, schickt der FRC an seinen Delegate `controllerDidChangeContent:`. Hier können Sie der Tabellen-View durch den Aufruf von `endUpdates` mitteilen, dass man mit dem Ändern fertig ist.

Während des Persistierens der Daten schickt der FRC für jedes geänderte Objekt ein `controller:didChangeObject:atIndexPath:forChangeType:newIndexPath:`. Dies ist die Implementierung:

CoreData/Conference02/Classes/RootViewController.m

```
- (void)controller:(NSFetchedResultsController *)controller
    didChangeObject:(id)anObject
      atIndexPath:(NSIndexPath *)indexPath
    forChangeType:(NSFetchedResultsControllerChangeType)type
      newIndexPath:(NSIndexPath *)newIndexPath {

    if(NSFetchedResultsControllerChangeUpdate == type) {
        [self configureCell:
            [self.tableView cellForRowAtIndexPath:indexPath]
            withTrack:anObject];
    } else if(NSFetchedResultsControllerChangeMove == type) {
        [self.tableView reloadSections:[NSIndexPath indexSetWithIndex:0]
            withRowAnimation:UITableViewRowAnimationFade];
    } else if(NSFetchedResultsControllerChangeInsert == type) {
        if(!self.firstInsert) {
            [self.tableView insertRowsAtIndexPaths:
                [NSArray arrayWithObject: newIndexPath]
                withRowAnimation:UITableViewRowAnimationRight];
        } else {
            [self.tableView insertSections:[NSIndexPath alloc] initWithIndex:0]
                withRowAnimation:UITableViewRowAnimationRight];
        }
    } else if(NSFetchedResultsControllerChangeDelete == type) {
        NSInteger sectionCount = [[fetchedResultsController sections] count];
        if(0 == sectionCount) {
            NSIndexPath *indexes = [NSIndexPath indexSetWithIndex:indexPath.section];
            [self.tableView deleteSections:indexes
                withRowAnimation:UITableViewRowAnimationFade];
        }
    }
}
```

```

    } else {

        [self.tableView deleteRowsAtIndexPaths:
         [NSArray arrayWithObject:indexPath]
         withRowAnimation:UITableViewRowAnimationFade];
    }
}
}

```

In dieser Methode aktualisieren wir die Tabellen-View in Abhängigkeit von der Art der Änderung. Bei Updates rufen wir die vorher schon behandelte Methode `configureCell:withTrack:` auf. Wenn etwas verschoben wird, aktualisieren wir den gesamten Abschnitt, da die Sortierreihenfolge der Elemente in der Track-Liste auf einem editierbaren Feld (der Eigenschaft `name`) basiert. Wenn sich dieser Wert geändert hat, ist es effizienter, den ganzen Abschnitt zu aktualisieren, als nur die einzelnen Felder. Nach einem Einfügen müssen wir unterscheiden, ob es sich um das erste Element oder um ein folgendes handelt. Da die Anwendung ganz ohne Daten startet, wird der Tabellen-View beim ersten Aufruf von `numberOfSectionsInTableView:` mitgeteilt, dass es keine Abschnitte gibt. Fügen wir die erste Zeile ein, fügen wir auch einen Abschnitt ein. Weist man die Tabellen-View an, eine Zeile in einen nicht existierenden Abschnitt einzufügen, gerät die View durcheinander. Beim Löschen schließlich teilen wir der Tabellen-View mit, welche Zeilen gelöscht werden, bis wir an das letzte Objekt geraten. Löschen wir dieses, müssen wir die Tabellen-View auch bitten, den Abschnitt zu löschen, da sie ansonsten ähnliche Probleme wie beim Einfügen bekommen kann.

Wenn die Anwender Änderungen in einer „Bearbeitungs-View“ vornehmen (wie das häufig der Fall ist), ist dies genau die richtige Stelle, die Tabellen-View zu aktualisieren. Aber denken Sie daran, dass Sie die Tabellen-View mit sehr großer Wahrscheinlichkeit nicht in dieser Methode aktualisieren wollen, wenn Sie in einem Hintergrund-Thread eine große Menge an Objekten importieren. In solchen Fällen laden Sie stattdessen einfach die Tabellen-View in `controllerDidChangeContent:` neu.

Jetzt sind wir mit dem Überblick über Core Data fast durch. Wir haben Folgendes erfahren:

- Die Objekte des Core Data-Stack und deren Zusammenarbeit, um uns Daten zu liefern und Änderungen im Persistent Store vorzunehmen.

- Die Verwendung von Core Data-Objekten, um einen NSFetchedResultsController einzurichten.
- Das Befüllen und Synchronhalten einer Tabellen-View mit einem NSFetchedResultsController.

Bei der Arbeit mit Tabellen-Views fehlt jetzt noch ein Aspekt: wie Core Data uns bei der Navigation in Objektdetails hilft.

11.8 Navigation

Die Navigation auf einer Tabellen-View ist dazu gedacht, uns von einem allgemeineren Datenset in ein spezifischeres zu leiten. In unserem Fall gibt es zwei Arten spezifischerer Daten. Die eine ist die Liste der Vorträge, aus denen unser Track besteht, die andere betrifft die Daten über den Track selbst. Denken Sie daran, dass wir in die Vortragsliste navigieren wollen, wenn ein Track ausgewählt wird – befindet sich die Tabellen-View aber im Bearbeitungsmodus, wollen wir zu einer View navigieren, auf der der Anwender die Eigenschaften name und track-Abstract des Tracks bearbeiten kann.

Dies ist der dazu notwendige Code:

CoreData/Conference02/Classes/RootViewController.m

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    Track *track = [[self.fetchedResultsController objectAtIndexPath:indexPath];
    if(YES == self.editing) {
        self.trackEditingVC.selectedTrack = track;
        [self.navigationController pushViewController:
            self.trackEditingVC animated:YES];
    } else {
        self.sessionsVC.selectedTrack = track;
        self.sessionsVC.title = track.name;
        [self.navigationController pushViewController:
            self.sessionsVC animated:YES];
    }
}
```

Wie vermutlich von Ihnen erwartet, haben wir einen View-Controller für jedes mögliche Ziel. Wenn wir uns im Bearbeitungsmodus befinden, schieben wir den trackEditingVC auf den Navigations-Stack. Solange dieser aktiv ist, braucht er Zugriff auf den aktuell ausgewählten Track, daher setzen wir vorher noch seine Eigenschaft selectedTrack.

Befinden wir uns nicht im Bearbeitungsmodus, nutzen wir den `sessionsVC`. Dieser muss den ausgewählten Track ebenfalls kennen, daher setzen wir auch dort dessen Eigenschaft `selectedTrack`. Zudem setzen wir noch `title` auf den Namen des Tracks.

Da wir die Navigationsmechanismen schon in früheren Kapiteln genauer behandelt haben, werden wir hier nicht mehr so ins Detail gehen. Stattdessen wollen wir ein paar interessante Methoden dieser anderen View-Controller hervorheben, um zu zeigen, wie wir die verwalteten Objekte (Instanzen von `Track` und `Session`) zusammen mit `Core Data` nutzen, um die Funktionalität zu implementieren.

Zuerst schauen wir uns den `TrackEditingViewController` an. Wenn der Anwender die Text-View von `trackAbstract` bearbeitet hat, wird die Delegate-Methode `textViewDidEndEditing:` aufgerufen (weil der View-Controller der Delegate der Text-View ist). Dies ist der Code:

CoreData/Conference02/Classes/TrackEditingViewController.m

```
- (void)textViewDidEndEditing:(UITextView *)textView {
    self.selectedTrack.trackAbstract = textView.text;
}
```

Bewundern Sie die Einfachheit dieser Methode! Wir ändern nur den Wert der Eigenschaft – `Core Data` kümmert sich um den Rest. Mit dieser einen Codezeile weiß `Core Data`, dass sich das `Track`-Objekt geändert hat. Zudem weiß `Core Data` auch, dass gespeichert werden muss, und es sorgt durch einen einfachen Methodenaufruf dafür, dass dies geschieht. Das funktioniert nicht nur für so einfache Dinge wie `String`-Werte, sondern auch für Beziehungen, wie wir gleich beim `Sessions-ViewController` sehen werden.

Wenn der Anwender mit seinen Änderungen fertig ist, tippt er auf den `FERTIG`-Button oben links. Dieser Button ruft die Methode `done` auf, die folgenden Code enthält:

CoreData/Conference02/Classes/TrackEditingViewController.m

```
- (void)done {
    [self.abstractText resignFirstResponder];
    [self.nameField resignFirstResponder];
    [self.navigationController popViewControllerAnimated:YES];
}
```

Indem für die beiden Texteingabeobjekte `resignFirstResponder` aufgerufen wird, ist sichergestellt, dass die eingegebenen Werte in das Objekt `selectedTrack` übertragen werden. Dann weisen wir den `navigationController` an, den obersten View-Controller vom Stack zu holen (das

Objekt `TrackEditingViewController`). Wenn der View-Controller die Kontrolle abgibt und seine View vom Stack holt, erhält er auch einen Aufruf von `viewWillDisappear:`. Wir haben uns dazu entschlossen, das Speichern in dieser Methode unterzubringen. Dies ist der Code:

CoreData/Conference02/Classes/TrackEditingViewController.m

```
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    NSError *error = nil;
    if (![self.selectedTrack.managedObjectContext save:&error]) {
        // Fehler-Handling ...
    }
}
```

Das sieht mittlerweile vertraut aus, oder? Durch den Aufruf der Methode `save:` werden alle Änderungen in die Datenbank übertragen. Beachten Sie, dass wir den `managedObjectContext` vom `selectedTrack` erhalten. Alle verwalteten Objekte wissen, zu welchem Managed Object Context sie gehören, dadurch ist das Speichern so supereinfach.

Nachdem wir uns um den View-Controller zum Bearbeiten gekümmert haben, wollen wir im `SessionsViewController` nach weiterem interessantem Code schauen. Dieser soll das Verständnis darüber erleichtern, wie die Objekte in Ihrem Modell durch Core Data bearbeitet werden. Sie erinnern sich: In der Implementierung von `tableView:didSelectRowAtIndexPath:` des `RootViewController` wird beim Schieben des `sessionsVC` auf den Navigations-Stack dessen Eigenschaft `selectedTrack` gesetzt. Dies ist der Code für die Set-Methode:

CoreData/Conference02/Classes/SessionsViewController.m

```
- (void)setSelectedTrack:(Track *)track {
    if(track != _selectedTrack) {
        [_selectedTrack release];
        _selectedTrack = [track retain];
        self.fetchedResultsController = nil;
        [self.tableView reloadData];
    }
}
```

Das ist mehr oder weniger eine Standard-Set-Methode, bis wir zu der Stelle kommen, an der die Eigenschaft `fetchedResultsController` auf `nil` gesetzt wird. Um zu verstehen, warum das notwendig ist, wollen wir einen Blick auf die Implementierung der Get-Methode für die Eigenschaft `fetchedResultsController` werfen:

CoreData/Conference02/Classes/SessionsViewController.m

```

1 - (NSFetchedResultsController *)fetchedResultsController {
2     if (nil == _fetchedResultsController) {
3         NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
4         NSManagedObjectContext *context = self.selectedTrack.managedObjectContext;
5         NSEntityDescription *entity =
6             [NSEntityDescription entityForName:@"Session"
7              inManagedObjectContext:context];
8         [fetchRequest setEntity:entity];
9
10        NSPredicate *pred = [NSPredicate predicateWithFormat:@"track = %@",
11                            self.selectedTrack];
12        [fetchRequest setPredicate:pred];
13
14        NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
15                                            initWithKey:@"name" ascending:YES];
16        [fetchRequest setSortDescriptors:
17            [NSArray arrayWithObject:sortDescriptor]];
18
19        NSFetchedResultsController *aFetchedResultsController =
20            [[NSFetchedResultsController alloc] initWithFetchRequest: fetchRequest
21                                                  managedObjectContext:context
22                                                  sectionNameKeyPath:nil
23                                                  cacheName:@"Sessions" ];
24        aFetchedResultsController.delegate = self;
25        self.fetchedResultsController = aFetchedResultsController;
26        NSError *error = nil;
27        if (![self.fetchedResultsController performFetch:&error]) {
28            // Fehler-Handling ...
29        }
30        [aFetchedResultsController release];
31        [fetchRequest release];
32        [sortDescriptor release];
33    }
34    return _fetchedResultsController;
}

```

Ein Großteil dieses Codes sieht genau so aus wie die Methode, die wir beim `RootViewController` begutachtet haben – abgesehen von Entitätsnamen und dem Prädikat. In Zeile 11 tun wir etwas, das etwas unerwartet sein könnte. Wir erzeugen ein `NSPredicate`, das die Liste der `Session`-Objekte auf die einschränkt, die mit dem `selectedTrack` verbunden sind. So können wir alle Vorteile des `Fetched Results Controller` nutzen, haben den Inhalt aber trotzdem auf die richtigen Vorträge eingeschränkt. Wir können auch die Tabellen-View genauso aktualisieren wie beim `RootViewController`, nämlich über die Methoden des `NSFetchedResultsControllerDelegate`.

So wie das Erstellen des Fetched Results Controller abgesehen von kleinen Unterschieden immer nahezu identisch ist, entspricht auch das Hinzufügen und Entfernen von Vorträgen dem von Tracks. Lassen Sie uns zunächst den Code zum Löschen anschauen. Hier ist die Implementierung der Methode `tableView:commitEditingStyle:forRowAtIndexPath:` aus dem `SessionsViewController`:

CoreData/Conference02/Classes/SessionsViewController.m

```

1 - (void)tableView:(UITableView *)tableView
2   commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
3   forRowAtIndexPath:(NSIndexPath *)indexPath {
4
5     if (editingStyle == UITableViewCellEditingStyleDelete) {
6       Session *session = [self.fetchedResultsController
7                           objectAtIndex:indexPath:indexPath];
8       [self.selectedTrack removeSessionsObject:session];
9       NSManagedObjectContext *context =
10        self.selectedTrack.managedObjectContext;
11       [context deleteObject:session];
12       // Kontext sichern
13       NSError *error;
14       if (![context save:&error]) {
15         // Fehler-Handling ...
16       }
17     }
18 }

```

Beachten Sie, dass dieser Code – abgesehen vom anderen Typ (Track statt Session) – der gleiche wie beim `RootViewController` ist. Der einzige echte Unterschied findet sich in Zeile 8. Neben dem Entfernen des Vortrags aus dem Managed Object Context entfernen wir ihn auch aus der Vortragsliste des Tracks.

Lassen Sie uns nun das Einfügen neuer Vorträge anhand dieses Codes betrachten:

CoreData/Conference02/Classes/SessionsViewController.m

```

1 - (void)insertSession {
2   self.firstInsert =
3     [self.fetchedResultsController.sections count] == 0;
4   NSString *nextId = [self nextSessionIdentifier];
5   // Eine neue Instanz der Entität erstellen, die durch den
6   // Fetched Results Controller verwaltet wird.
7   NSManagedObjectContext *context =
8     self.selectedTrack.managedObjectContext;
9   Session *session =
10    [NSEntityDescription insertNewObjectForEntityForName:@"Session"
11     inManagedObjectContext:context];
12
13   // Konfigurieren des neuen verwalteten Objekts
14   [session setValue:@"Session" forKey:@"name" ];

```

```

13     [session setValue:@"Tolle Session" forKey:@"sessionAbstract" ];
14     [session setValue:nextId forKey:@"sessionID" ];
15
16     [self.selectedTrack addSessionsObject:session];
17
18     // Kontext sichern
19     NSError *error = nil;
20     if (![context save:&error]) {
21         // Fehler-Handling ...
22     }
23 }

```

Es gibt hier zwei Codeabschnitte, die sich von dem zum Einfügen eines Tracks unterscheiden. So fügen wir in Zeile 16 den neu erzeugten Vortrag zum `selectedTrack` hinzu. Achten Sie auf die Verwendung der Methode `addSessionsObject:`. Denken Sie daran, dass diese Methode unserer Klasse `Track` automatisch durch Core Data hinzugefügt wird. Auch wenn wir keine Implementierung bereitstellen, können wir sie so ohne Fehler oder Warnung aufrufen.

Die nächste interessante Zeile in dieser Methode ist Zeile 3. Die Session hat eine eindeutige Session-Kennung, und wir nutzen diese Methode, um die eindeutige ID zu berechnen, indem wir den größten Wert in der Datenbank um eins erhöhen. So sieht der Code aus:

CoreData/Conference02/Classes/SessionsViewController.m

```

- (NSString *)nextSessionIdentifier {
    NSString *nextId = @"Session01" ;

    NSManagedObjectContext *ctx =
        self.selectedTrack.managedObjectContext;
    NSEntityDescription *entity = [NSEntityDescription
                                    entityForName:@"Session"
                                    inManagedObjectContext:ctx];
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:entity];
    NSString *predFormat = @"sessionID = max(sessionID)" ;
    NSPredicate *pred = [NSPredicate predicateWithFormat:predFormat];
    [fetchRequest setPredicate:pred];

    NSError *error = nil;
    NSArray *values = [ctx executeFetchRequest: fetchRequest error:&error];
    if(0 != [values count]) {
        Session *session = [values objectAtIndex:0];
        NSString *maxId = [session valueForKey:@"sessionID" ];
        NSString *number = [maxId stringByTrimmingCharactersInSet:
                            [NSCharacterSet letterCharacterSet]];
        NSString *name = [maxId stringByTrimmingCharactersInSet:
                            [NSCharacterSet decimalDigitCharacterSet]];
    }
}

```

```

    NSNumberFormatter *formatter =
        [[[NSNumberFormatter alloc] init] autorelease];
    NSNumber *value = [formatter numberFromString:number];
    nextId = [NSString stringWithFormat:@"%02d", name,
        [value intValue]+ 1];
}
return nextId;
}

```

In dieser Methode erledigen wir drei Dinge: Zunächst erstellen wir einen Fetch Request, dann nutzen wir diesen, um die Session mit der größten sessionID zu ermitteln, und schließlich parsen wir die sessionID, um den Integer-Wert zu erhalten, so dass wir ihn erhöhen können. Der interessante Teil dieser Methode ist die Verwendung von Aggregatfunktionen in unserem Prädikat. Prädikate sind sehr mächtige Hilfsmittel, die wir einschränken müssen, um von der Datenbank genau das zu erhalten, was wir benötigen. Die Dokumentation *Predicate Programming Guide* [App08g] von Apple enthält viele Details zu Prädikaten.

Wie Ihnen jetzt vermutlich klar geworden ist, ist Core Data ein mächtiges Framework, das es uns ermöglicht, Daten in einer Datenbank sehr einfach anzusprechen, indem wir ein bisschen Modellierung vornehmen und dieses Modell dann verwenden. Wir haben aber nur an der Oberfläche dessen gekratzt, was möglich ist.

Kapitel 12

Verbindung zum Internet

Einer der revolutionärsten Aspekte des iPhone ist sein Einfluss auf den mobilen Internetzugriff. Vor dem iPhone war das Browsen im Web oder das Durchführen anderer netzwerkbasierter Aufgaben so unattraktiv, dass sich nur wenige Anwender damit beschäftigt haben und noch weniger davon begeistert waren. So war es im Februar 2008 ein Schock, als Googles Leiter des Bereichs Mobile Operations auf einer Konferenz erzählte, dass Google 50 Mal häufiger von iPhone-OS-Geräten angesprochen wird als von anderen mobilen Browsern.¹ Im März gab die Analysefirma M:Metrics bekannt, dass Studien zeigten, 84,4% der iPhone-Anwender würden News und andere Informationen über ihre Telefone abrufen – im Vergleich zu 13,1% im gesamten Mobilfunkmarkt und 58,2% bei Smartphone-Nutzern.²

Sie müssen uns also nicht erst erzählen, dass der Netzwerkzugriff für Ihren Anwender ausgesprochen wichtig ist. Bei vielen iPhone-Anwendungen ist gerade das Netzwerk der entscheidende Faktor. In diesem Kapitel werden Sie lernen, wie Sie die wichtigsten Netzwerk-Features des iPhone nutzen: das Laden von Webseiten, das Lesen und Schreiben von Netzwerk-Streams und das Parsen von Daten, die von Webservices bereitgestellt werden.

12.1 Ein Browser in zehn Minuten mit der UIWebView

Für viele Netzerkanwendungen ist es vielleicht *die* typische Aufgabe, eine Webseite zu laden. Sie wollen sich sicher nicht unbedingt mit dem ausgezeichneten Safari-Browser messen, der dem iPhone schon beiligt

1 Google Homes in on Revenues from Phones [PT08]

2 iPhone Users Love That Mobile Web [Sto08]

– tatsächlich sollten Sie das auch gar nicht –, aber vielleicht haben Sie eine Netzwerkkomponente, die sich am einfachsten in einer Webseite darstellen lässt.

Wenn Sie genau das brauchen, haben Sie Glück, denn die Funktionalität der WebKit-Engine von Safari steht Ihnen in Form der `UIWebView` zur Verfügung.

Diese Komponente lässt sich so einfach nutzen und ist dabei gleichzeitig so mächtig, dass wir direkt am Anfang dieses Kapitels in zehn Minuten einen einfachen Browser schreiben werden, der mit echten Webseiten umgehen kann. Tatsächlich haben wir sogar dem Beispielprojekt den Namen `TenMinuteBrowser` gegeben. Sie finden es im Verzeichnis `NetworkIO`.

Starten Sie also Ihre Stoppuhr und legen Sie los.

Das Projekt aufsetzen

Starten Sie ein neues Xcode-Projekt mit dem *View-based Application*-Template. Geben Sie ihm den Namen `TenMinuteBrowser`. Damit werden die Header- und Implementierungsdateien für `TenMinuteBrowserAppDelegate` und `TenMinuteBrowserViewController` erzeugt. Sie müssen den App-Delegate gar nicht anfassen.

Unser Browser wird nur ein URL-Feld und eine große View mit dem Seiteninhalt haben. Gehen Sie also in den `TenMinuteBrowserViewController` und fügen Sie `IBOutlet`s für `urlField` (ein `UITextField`) und `webView` (eine `UIWebView`) hinzu. Da wir das Textfeld ordentlich behandeln und die virtuelle Tastatur verschwinden lassen wollen, wenn die Eingabe beendet ist, fügen Sie außerdem die Protokolldeklaration `UITextFieldDelegate` hinzu. Schließlich ergänzen Sie noch eine Instanzmethode, die beim Antippen des Go-Buttons ausgeführt werden soll. Sinnvollerweise geben Sie ihr den Namen `handleGoTapped`. Ihr `@interface` sollte nun in etwa so aussehen:

```
NetworkIO/TenMinuteBrowser/Classes/TenMinuteBrowserViewController.h
```

```
@interface TenMinuteBrowserViewController : UIViewController
    <UITextFieldDelegate> {
        IBOutlet UITextField *urlField;
        IBOutlet UIWebView *webView;
    }

    -(IBAction) handleGoTapped;

@end
```

Das Browser-GUI im IB erstellen

Okay, eine Minute ist rum, aber wir können auch schon in den Interface Builder wechseln. Klicken Sie doppelt auf `TenMinuteBrowserView-Controller.xib`, um den IB zu starten. Beim Nib ist File's Owner schon auf Ihre View-Controller-Klasse gesetzt mit einer einzelnen View, die darauf wartet, konfiguriert zu werden. Klicken Sie sie doppelt an, um sie zu öffnen, und fügen Sie drei Objekte hinzu:

- Ein `UITextField` am oberen Rand für die URL.
- Ein `UIButton` mit dem Text „Go“.
- Und eine `UIWebView`, die den größten Teil der View unter dem Textfeld und dem Go-Button einnimmt.

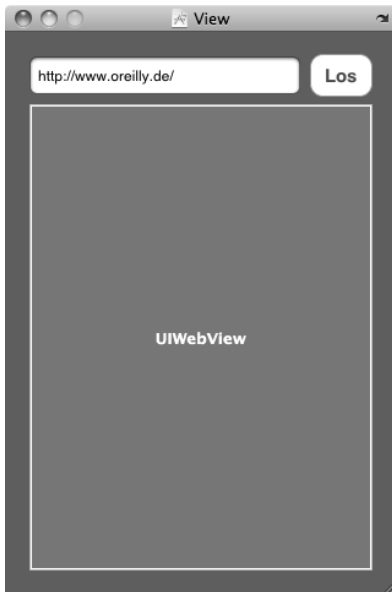


Abbildung 12.1: View für einen einfachen Browser im Interface Builder

Da Sie schon dabei sind, können Sie das Textfeld natürlich mit der URL Ihrer Lieblings-Website vorbefüllen oder die Hintergrundfarbe der View setzen. Auf jeden Fall sollte das Ergebnis aber mehr oder weniger wie das in Abbildung 12.1 aussehen.

Wie geht's Ihnen? Vier Minuten rum, so in etwa? Keine Panik, wir schaffen das schon. Als Nächstes werden wir die GUI-Komponenten mit den Outlets und Actions verbinden. Klicken Sie bei gedrückter `Ctrl`-Taste auf FILE'S OWNER, um dessen Liste mit Outlets und Actions zu öffnen.

- Verbinden Sie das Outlet `urlField` mit dem Textfeld.
- Verbinden Sie das Outlet `webView` mit der `UIWebView`.
- Verbinden Sie die Action `handleGoTapped` mit dem Go-Button, wobei Sie das Event „Touch Up Inside“ auswählen.
- Schließlich klicken Sie noch bei gedrückter `Ctrl`-Taste auf das Textfeld, um dessen Outlets anzuzeigen, und verbinden dessen Delegate mit File's Owner.

Speichern Sie alles und beenden Sie den Interface Builder.

Den Browsers implementieren

Werfen Sie einen Blick auf die Uhr. Schaffen wir es in weniger als zehn Minuten? Keine Angst, es ist nicht mehr weit. Zurück in Xcode, öffnen Sie `TenMinuteBrowserViewController.m`. Wir müssen eine Methode schreiben, die die URL aus dem Textfeld ausliest und sie der Web-View zum Laden übergibt. Diese Methode wird aufgerufen, wenn der Anwender auf den Go-Button oder auf der Tastatur auf RETURN tippt.

Wenn Sie sich die Dokumentation zu `UIWebView` anschauen – später, im Moment läuft ja die Uhr –, werden Sie sehen, dass es eine einfache Methode `loadRequest:` besitzt, die ein Argument erwartet – einen `NSURLRequest`. Also müssen Sie zum Laden der Webseite nur ein Objekt dieser Klasse bekommen ... das Sie aus einer `NSURL` erhalten können ... das sich wiederum aus einem `NSString` bauen lässt ... der wiederum der Typ der Eigenschaft `text` des Textfelds ist. Zum Laden der Webseite brauchen wir also nur ein paar Zeilen Code:

NetworkIO/TenMinuteBrowser/Classes/TenMinuteBrowserViewController.m

```
-(void) loadURL {
    NSURL *url = [[NSURL alloc] initWithString: urlField.text];
    NSURLRequest *request = [[NSURLRequest alloc] initWithURL: url];
    [webView loadRequest: request];
    [request release];
    [url release];
}
```

Jetzt müssen Sie diese Methode nur noch aufrufen, wenn der Go-Button angetippt wird, wobei auch die Tastatur noch ausgeblendet werden sollte:

NetworkIO/TenMinuteBrowser/Classes/TenMinuteBrowserViewController.m

```
-(IBAction) handleGoTapped {
    [urlField resignFirstResponder];
    [self loadURL]; }

```

Sie können auch noch die Methode `textFieldShouldReturn:` des Textfelds nutzen, um über die Return-Taste ebenfalls die Seite laden zu lassen:

NetworkIO/TenMinuteBrowser/Classes/TenMinuteBrowserViewController.m

```
-(BOOL)textFieldShouldReturn:(UITextField *)textField {
    if (textField == urlField) {
        [self handleGoTapped];
    }
    return YES;
}

```

Immer noch unter zehn Minuten? Gut, denn Sie sind fertig. Klicken Sie auf **BUILD AND RUN**, um Ihre Anwendung im Simulator auszuführen. Geben Sie eine vollständige URL ein (also einschließlich `http://`), tippen Sie auf **GO**, und das Ergebnis sollte so wie in Abbildung 12.2 aussehen.



Abbildung 12.2: Rendern einer Webseite mit der UIWebView

Sie haben noch fünf weitere Minuten?

Wie Sie sehen, wird Ihnen der Großteil der Arbeit von der `UIWebView` abgenommen. Haben Sie die Seite einmal geladen, kümmert sich diese View – im Hintergrund unterstützt durch die `WebKit`-Engine, die `HTML` rendert, `JavaScript` interpretiert und sich um die Netzwerkkommunikation kümmert – um all Ihre Webinteraktionen, einschließlich des Abschickens von Formularen, des Navigierens zu neuen Seiten, des Ausführens von Browseranwendungen auf Clientseite und so weiter.

Wenn Sie noch ein paar Minuten mehr erübrigen können, ist es ganz einfach, diese Anwendung zu einem vollständigen Browser auszubauen. Was dazu am offensichtlichsten fehlt, sind die üblichen Navigations-Buttons, um zur vorherigen beziehungsweise nächsten Seite zu gelangen. Diese lassen sich wirklich simpel über die Methoden `goForward` und `goBack` der `UIWebView` implementieren. Sie könnten auch einen „Busy“-Indikator anzeigen lassen, der erscheint, wenn die View gerade eine Seite lädt. Dazu stellen Sie einen Delegate bereit, der das Protokoll `UIWebViewDelegate` implementiert. Dieses besitzt die Callbacks `webViewDidStartLoad:`, `webViewDidFinishLoad:` und `webView:didFailLoadWithError:`.

Sie könnten auch ein Drehen des iPhone unterstützen, so dass sich der Browser auch quer nutzen ließe. Die `UIWebView` funktioniert diesbezüglich wie jede andere `UIView`: Sie überschreiben `shouldAutorotateToInterfaceOrientation:`, damit auch eine horizontale Ausrichtung unterstützt wird. Dann öffnen Sie die View im Interface Builder, markieren die Web-View und aktivieren die horizontalen und vertikalen Springs im Size Inspector.

Selbst wenn Sie keinen Browser entwickeln wollen, besitzt die `UIWebView` andere interessante Anwendungsmöglichkeiten. Das `UIKit` besitzt zwar keine Textkomponente, die optisch an iPhone-Apps ausgerichtet ist, aber Sie können `HTML` mit `CSS` aufbrezeln und diesen Code in einer `UIWebView` anzeigen lassen. Das ist eine ausgezeichnete Möglichkeit, eine „About“-View für Ihre Anwendung anzuzeigen, denn dort können Sie Links auf die Homepage Ihrer Anwendung unterbringen, E-Mail-Adressen für Fragen, ja sogar direkt anwählbare Telefonnummern – alles durch ein bisschen `HTML`.

Und statt eine Seite aus dem Web zu laden, können Sie Ihre `HTML`-, `CSS`- und Bilddateien in das Anwendungs-Bundle aufnehmen und sie dann dort aufrufen. Im Beispielprojekt zeigen wir die Seite `about-book.html` aus dem Bundle an, wenn die URL mit dem String `about:` beginnt. Um eine URL aus einem Pfad im Bundle zu erstellen, müssen Sie den Pfad-String nur in eine `NSURL` umwandeln.

```
NetworkIO/TenMinuteBrowser/Classes/TenMinuteBrowserViewController.m
```

```
NSString *aboutPath =
    [[NSBundle mainBundle] pathForResource:@"aboutbook"
                                     ofType:@"html" ];
url = [[NSURL alloc] initWithFileURLWithPath: aboutPath];
```

Erstellen Sie daraus wie vorher einen `NSURLRequest` und laden Sie diesen in der `UIWebView`. Dann wird Ihre lokale Datei in der Web-View angezeigt, wie in Abbildung 12.3 zu sehen.



Abbildung 12.3: Anzeige einer lokalen „About“-Seite mit der `UIWebView`

12.2 Daten über das Netzwerk lesen

Natürlich ist nicht alles im Internet eine Webseite. Ja, selbst über Port 80 ist nicht alles eine Webseite. Sie wollen vielleicht Daten über einen Webservice austauschen oder sich mit einem Webradio verbinden. Viele Anwendungen haben ihre eigenen Gründe, sich mit dem Netz zu verbinden und die Daten zu ganz individuellen Zwecken zu nutzen. Wenn Sie eine URL öffnen und deren Daten auslesen wollen, macht das URL Loading System von Cocoa es zum Glück außerordentlich leicht.

Das URL Loading System ist eine Sammlung von Klassen, durch die Sie mit den vier grundlegenden URL-Protokollen arbeiten können: `http://`, `https://`, `ftp://` und `file://`. Wir wollen anhand eines Beispiels zeigen, was Sie tun müssen, um den Quelltext einer Webseite einlesen (oder – technisch gesehen – den Inhalt einer beliebigen unterstützten URL) und anzeigen zu können. Sie finden das Beispielprojekt Simple-CocoaURLReader unter NetworkIO. Der Anwender erhält ein ähnliches UI wie bei dem einfachen Browser aus *Abschnitt 12.1, Ein Browser in zehn Minuten mit der UIWebView*, auf Seite 259, nur dass der Quelltext statt in einer UIWebView in einer einfachen UITextView angezeigt wird. In Abbildung 12.4 sehen Sie ein Beispiel dazu mit dem Quelltext der Seite `http://www.oreilly.de/`.



Abbildung 12.4: Anzeige des Quelltexts einer Webseite, der über das URL Loading System geladen wurde

Wie erwartet, sind die UI-Elemente dieser App identisch mit denen des WebKit-Demos. Tippt man auf den Button LADEN oder auf die Return-Taste, wird die Methode `loadURL` aufgerufen, die dafür verantwortlich ist, den Inhalt der durch die im Textfeld eingegebene URL definierten Seite zu laden. Im Browserbeispiel war mit dem Aufruf von `loadRequest`: schon alles erledigt. Hier ist das erst der Anfang.

Im URL Loading System nehmen Sie den `NSURLRequest` und erstellen daraus eine `NSURLConnection`. Die Anfrage legt fest, was Sie haben wollen und wie Sie es haben wollen – ihre Methoden geben die URL zurück, zeigen, ob die HTTP-Methode GET oder POST ist, welche Daten Sie bei einem POST übermitteln wollen und so weiter.³ Eine dazugehörige `NSURLResponse`, die die Antwort des Servers darstellt, wird später automatisch generiert. Mit diesen beiden Objekten repräsentiert die `NSURLConnection` den Vorgang des Verbindens mit dem Server, das Abschicken der Anfrage und das Erhalten der Antwort. Und alles ist so entworfen, dass Sie mit dieser Verbindung stark asynchron interagieren, wie Sie gleich sehen werden.

NetworkIO/SimpleCocoaURLReader/Classes/URLLoaderViewController.m

```
NSURLRequest *request = [[NSURLRequest alloc] initWithURL: url];
NSURLConnection *connection = [[NSURLConnection alloc]
                               initWithRequest:request
                               delegate:self];

[connection release];
[request release];
```

Doch, doch, das ist schon richtig – Sie erzeugen eine `NSURLConnection`, um sie direkt danach per `release` wieder freizugeben, weil Sie sie nie wieder benötigen. Aber was zum Teufel soll das?

Die Verbindung wird mit einem `delegate` eingerichtet – in diesem Fall `self` –, daher erhält Ihr Objekt einen Weckruf, wenn es etwas gibt, um das Sie sich kümmern müssen. Im URL Loading System werden Sie über alles mögliche Interessante informiert – das Öffnen oder Schließen der Verbindung, das Empfangen von mehr Daten, über Fehler und Authentifizierungsanfragen – und zwar durch Callbacks an die diversen Methoden eines Delegate-Objekts. Diese Methoden sind in der Dokumentation zu `NSURLConnection` beschrieben, die wichtigsten davon zeigen wir Ihnen im Folgenden:

- `connection:didReceiveResponse:` zeigt an, dass genug Daten empfangen wurden, um ein Objekt vom Typ `NSURLResponse` zu erstellen.
- `connection:didReceiveData:` stellt einen `NSData`-Wrapper um die zuletzt über die Verbindung empfangenen Daten bereit.
- `connectionDidFinishLoading:` zeigt an, dass der Download normal beendet wurde.

³ Diese Eigenschaften können nur gelesen werden. Mit `NSMutableURLRequest` können Sie auch vom Standard abweichende Werte setzen.

- `connection:didFailWithError:` zeigt an, dass der Download fehlgeschlug. Dabei wird ein `NSError` mitgegeben, der den Fehler beschreibt.

Um den Quelltext hinter einer URL anzuzeigen, ist die Implementierung dieser Callbacks ziemlich einfach. Wenn Sie die Verbindung erstellen, leeren Sie die `UITextView` und zeigen über die Methode `startAnimating` einer `UIActivityIndicatorViewView` an, dass das iPhone gerade beschäftigt ist. Erhalten Sie eine Antwort, tun Sie ... nichts.



Joe fragt...

Was ist, wenn ich etwas anderes als HTTP oder FTP brauche?

Wir haben dem URL Loading System so viel Platz eingeräumt, weil es sich dabei um eine dieser 80-20- (oder 90-10-) Situationen handelt. Die überwältigende Mehrheit des Netzwerkverkehrs läuft als HTTP über Port 80 ab. Da dieser Port so gut wie nie durch Proxies oder Firewalls blockiert wird, haben die Entwickler in den letzten 10 oder 20 Jahren jede denkbare Funktionalität darüber realisiert – von Remote Procedure Calls bis hin zum Streamen von Audiodaten. Man kann sich durchaus fragen, ob es nicht sinnvoller gewesen wäre, die Netzwerk-Admins hätten besser mehr Ports offen gelassen, so dass die verschiedenen Dienste auch auf verschiedenen Ports arbeiten könnten, statt die Bedeutung des Port 80 so zu überfrachten. Aber wir schweifen ab ...

Wenn Sie ein Protokoll implementieren müssen, das eben nicht HTTP oder FTP ist, kommen Sie im Prinzip wieder zur klassischen Socket-Programmierung zurück: Öffnen von Input- und/oder Output-Streams für einen Host und Port sowie Lesen und Schreiben von Daten anhand eines (eventuell nur informellen) Protokolls. Cocoa auf dem iPhone hilft hier nicht. Auf dem Mac gibt es eine Klasse `NSHost`, die Sie an die Methode `getStreamsToHost:port:inputStream:outputStream:` der Klasse `NSStream` übergeben können, aber diese gehört nicht zur öffentlichen iPhone-API und sollte daher auch nicht genutzt werden. Zur Information: Die Funktionalität dieser fehlenden Cocoa-Methode wird durch die CFNetwork-Funktion `CFStreamCreatePairWithSocketToHost()` bereitgestellt.

Sie dürfen natürlich gern einen Blick in das `NSURLResponse`-Objekt werfen, um zu schauen, ob sich dort für Sie interessante Daten befinden (wenn Sie zum Beispiel keinen Spinning-Indikator, sondern eine `UIProgressView` nutzen, setzen Sie deren Maximalwert auf den Wert von `expectedContentLength`, zählen beim Empfangen der Daten die Bytes und aktualisieren den Fortschrittsbalken).

Sie interessieren sich vermutlich am meisten für das Empfangen von Daten. Die Delegate-Methode `connection:didReceiveData:` wird wiederholt mit neuen Daten aufgerufen, bis das Ende des Streams erreicht ist. Egal was Sie mit den Daten beim Download machen wollen – Sie machen es hier.

NetworkIO/SimpleCocoaURLReader/Classes/URLLoaderViewController.m

```
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data {
    NSLog(@"connectionDidReceiveData" );
    NSString *newText = [[NSString alloc]
        initWithData:data
        encoding:NSUTF8StringEncoding];
    if (newText != NULL) {
        [self appendTextToView:newText];
        [newText release];
    }
}
```

Die Implementierung des Beispielcodes ist ziemlich trivial – aus dem `NSData` wird ein `NSString` erstellt und an die `UITextView` angefügt. Ihre Erfordernisse sind sehr wahrscheinlich andere. Wenn Sie zum Beispiel den gesamten Inhalt der URL im Speicher halten müssen, könnten Sie ein Objekt vom Typ `NSMutableData` erzeugen, an das Sie jedes empfangene `NSData`-Objekt anhängen. Sie könnten auch das Herunterladen einer Datei implementieren, indem Sie einen Output-Stream für eine Datei öffnen (siehe Abschnitt 8.7, *Daten in Dateien schreiben*, Seite 175) und jeden empfangenen Datenblock dorthin schreiben.⁴

Die anderen beiden Methoden, die Sie in so gut wie jeder App berücksichtigen müssen, sind die für die Abschluss- und Fehler-Events. Sie müssen mit dem URL Loading System nicht explizit irgendwelchen verbindungsbezogenen Ressourcen schließen oder freigeben, daher geht es hier meist nur um UI-Aufgaben, wie zum Beispiel das Abschalten eines Aktivitätsindikators oder das Ausgeben einer Fehlermeldung, wenn die Verbindung abgebrochen wurde.

⁴ Cocoa für Mac OS X besitzt zu diesem Zweck eine Klasse `NSURLDownload`, auf dem iPhone OS ist sie aber nicht vorhanden.

```
NetworkIO/SimpleCocoaURLReader/Classes/URLLoaderViewController.m
```

```
- (void) connectionDidFinishLoading: (NSURLConnection*) connection {
    [activityIndicator stopAnimating];
}
- (void) connection: (NSURLConnection *)connection
    didFinishWithError: (NSError *)error {
    UIAlertView *errorAlert = [[UIAlertView alloc]
        initWithTitle: [error localizedDescription]
        message: [error localizedFailureReason]
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    [errorAlert show];
    [errorAlert release];
    [activityIndicator stopAnimating];
}
```

Und das ist es schon. Kein Babysitting einer Verbindung, keine Ressourcen, die nach jedem Abschluss und jedem Abbruch sorgfältig wieder freigegeben werden müssen, keine nervtötenden Details, mit denen Sie sich wie in anderen Netzwerk-APIs herumschlagen müssen. Zudem ist das URL Loading System schon asynchron entworfen worden. Bei anderen APIs ist es notwendig, dass das UI geblockt ist, bis alle Bytes angekommen sind, aber hier können Sie Event-gesteuert vorgehen.⁵

12.3 HTTP-Authentifizierung

Eine Sache kann unser asynchroner Reader für uns nicht machen – es ist ihm nicht möglich, eine passwortgeschützte URL zu lesen. Anfragen in einer Umgebung, in der eine Authentifizierung erforderlich ist, werden meist mit dem HTTP-Antwortstatus 401 (Unauthorized) und einer für den Anwender lesbaren Beschreibung des Ablehnungsgrunds beantwortet. Glücklicherweise ist auch der Umgang mit einer HTTP-Authentifizierung im URL Loading System sehr einfach.

Eine passwortgeschützte Website einrichten

Als Erstes werden Sie nun natürlich einen eigenen Authentication Realm einrichten wollen, mit dem Sie testen können. Es ist nicht möglich, herauszufinden, ob Ihr Authentifizierungscode korrekt arbeitet,

⁵ Sie können `NSURLConnection` dazu zwingen, in einem synchronen, blockierenden Modus zu arbeiten, indem Sie `sendSynchronousRequest:returningResponse:error:` aufrufen, wenn Sie der Meinung sind, dass Sie das wirklich brauchen. Aufgrund der hohen Latenz und dem Fehlerpotenzial von drahtlosen Verbindungen raten wir Ihnen aber, es eher nicht zu tun.

wenn Sie keine echte passwortgeschützte Website nutzen. Glücklicherweise findet sich auf Ihrem Mac schon der Apache Webserver, den Sie für Ihre Tests verwenden können.

Gehen Sie zu den Systemeinstellungen Ihres Macs, wählen Sie FREI-GABEN und schalten Sie die Webfreigabe ein, wenn sie nicht schon aktiv ist (siehe Abbildung 12.5).



Abbildung 12.5: Webfreigabe in den Systemeinstellungen aktivieren

Damit wird Ihr Apache Webserver gestartet. Merken Sie sich die Adresse Ihres Computers, da Sie diese später auf dem iPhone oder im Simulator nutzen werden.

Als Nächstes müssen Sie ein paar der Konfigurationsdateien des Apache bearbeiten. Auf dem Mac OS X befinden sich diese Dateien an verschiedenen Stellen im Dateisystem. Wir werden zunächst ein Verzeichnis im Web Root erstellen – dem Ordner, der die oberste Ebene Ihrer Website ist – und dieses dann durch das Einrichten einer Konfigurationsdatei per Passwort schützen.

Standardmäßig ist Ihr Web-Wurzelverzeichnis (Apache-Web-Root) das Verzeichnis `/Library/WebServer/Documents`. Navigieren Sie also im Finder dorthin und erstellen Sie einen Ordner mit einem einfachen Namen wie `iphone`.⁶ Jetzt erzeugen Sie in diesem Verzeichnis mit dem Texteditor Ihrer Wahl eine einfache Webseite. Unsere hat nur ein bisschen Text und einen Screenshot aus einem früheren Kapitel:

```
<html><head><title>OK!</title></head>
<body>
<h1>Sie haben es geschafft!</h1>

</body>
</html>
```

Der nächste Schritt ist, dieses Verzeichnis durch die Apache-Authentifizierung zu sichern. Dazu müssen Sie Dateien in `/etc/apache2/` bearbeiten. Dieses Verzeichnis ist aber im Finder nicht sichtbar. Wechseln Sie also im Terminal mit `cd /etc/apache2` dorthin. Sie finden hier die Apache-Konfiguration, insbesondere `httpd.conf` und alle Dateien, die dort importiert werden. Zum Bearbeiten benötigen Sie Root-Zugriff und einen Texteditor, den Sie von der Befehlszeile aus starten können, wie zum Beispiel `vi`, `emacs` oder die Befehlszeilen-Launcher für andere beliebige Mac-Texteditoren – zum Beispiel `mate` für TextMate oder `bbedit` für BBEdit. Zunächst bearbeiten Sie die Datei `httpd.conf`:⁷

```
⇒ Yuna:apache2 cadamson$ sudo emacs httpd.conf
Password:
```

Sie könnten das Authentifizierungsverzeichnis direkt hier in der Hauptdatei `httpd.conf` angeben, aber das ist schlechte Praxis, da man dann leicht durcheinanderbringen kann, welche Einträge von Mac OS X und welche von Ihnen kommen. Stattdessen springen Sie an das Ende der Datei und fügen eine Direktive ein, durch die alle Konfigurationsdateien aus dem Verzeichnis `users` inkludiert werden.

```
# Hinzufügen eines vhost, um das iphone-Verzeichnis zum Testen zu
# authentifizieren
Include /etc/apache2/users/*.conf
```

Jetzt können Sie Ihr passwortgeschütztes Verzeichnis einrichten. Wechseln Sie in das Verzeichnis `users` und erstellen Sie mit dem Editor eine neue Datei beliebigen Namens, solange sie auf `.conf` endet, damit das `<Include>` aus der Hauptkonfigurationsdatei sie auch findet. `iphone-test-dir.conf` wäre zum Beispiel eine gute Wahl. In dieser Datei werden Sie die Apache-Direktive `<Directory>` nutzen, um festzulegen, dass Ihr Verzeichnis `iphone` eine Sonderbehandlung benötigt, nämlich den Passwortschutz. Die Direktive wird detailliert in der Apache-Dokumentation *Authentication, Authorization and Access Control* [Apa09] beschrieben, aber ein einfaches Beispiel sollte auch so verständlich sein:

6 Um Dateien in diesem Verzeichnis erstellen und bearbeiten zu können, müssen Sie eventuell Administrator auf Ihrem Mac sein.

7 Bei `vi` und `emacs` werden Sie den Editor mit `sudo` starten müssen, damit Sie auch Schreibrechte haben. Beim BBEdit und TextMate ist das nicht notwendig – Sie werden beim ersten Speichern nach dem Administratorpasswort gefragt.

```
#
# Einfacher Passwortschutz für das Verzeichnis iphone
#
<Directory "/Library/WebServer/Documents/iphone" >
    AuthType Basic
    AuthName "Authentifiziere oder stirb"
    AuthUserFile /etc/apache2/iphonedirpasswd
    Require user chris
</Directory>
```

In dieser Direktive steht Folgendes:

- Das fragliche Verzeichnis ist `/Library/WebServer/Documents/iphone`.
- Es soll das Apache-Authentifizierungsmodul „Basic“ mit Benutzernamen und Passwort genutzt werden.
- Es wird ein lesbarer Name für den Authentication Realm angegeben: „Authentifiziere oder stirb“.
- Die Passwortdatei für Benutzernamen und Passwörter für diesen Realm wird festgelegt.
- Die zulässigen Benutzernamen werden aufgeführt – in diesem Fall `chris`. Sie können mehrere Benutzer angeben oder Gruppen erstellen, aber für Ihre Tests sollte ein Benutzer ausreichend sein.

Die Authentifizierungsdirektive weist Apache an, in der Datei `/etc/apache2/iphonedirpasswd` nach Benutzernamen und Passwörtern zu schauen, also müssen Sie diese wohl anlegen. Apache-Anwender erzeugen Sie mit dem Befehl `htpasswd`, den Sie als Root ausführen müssen.

```
⇒ Yuna:apache2 cadamson$ sudo htpasswd -c /etc/apache2/iphonedirpasswd chris
Password:
New password:
Re-type new password:
Adding password for user chris
Yuna:apache2 cadamson$
```

Dieser Befehl weist `htpasswd` an, die definierte Datei (über die Option `-c`) zu erstellen und den Benutzernamen einzutragen, den wir erzeugen wollen. Ein wenig verwirrend sind die nächsten drei Zeilen. Das erste `Password:` kommt von der Shell, die Sie für den `sudo`-Befehl nach dem Administratorpasswort fragt. Das `New password`, nach dem zwei Mal gefragt wird, ist das Passwort für den Apache-Benutzer, den Sie anlegen wollen. Da Sie es auf einer iPhone-Tastatur eingeben wollen, sollten Sie es vielleicht einfach halten – `test` ist möglicherweise besser als `ph33rth3l337hax0rd00d`.

Mit der so erzeugten Passwortdatei und der richtigen Konfiguration für den Apache können Sie alles nun erst mal mit Ihrem Desktop-Browser testen. Starten Sie den Apache mit `sudo apachectl restart` neu und navigieren Sie dann in das passwortgeschützte Verzeichnis, indem Sie dessen Namen an Ihre Webadresse anhängen. In unserem Fall war das `http://192.168.0.108/iphone/`, Ihre IP-Adresse sieht vermutlich anders aus. Ist alles korrekt konfiguriert, sollte Ihr Webbrowser ein Pop-up anzeigen, in dem der Benutzername und das Passwort abgefragt werden. Wenn das nicht funktioniert, werfen Sie zunächst einen Blick in die Apache-Logs in `/var/log/apache2/`, um zu sehen, ob beim Neustart des Apache irgendwelche Fehler gemeldet wurden.

Umgang mit der HTTP-Authentifizierung

Nachdem Sie nun eine passwortgeschützte Seite haben, die Sie zum Testen nutzen können, wollen wir Ihrer Anwendung eine Unterstützung der HTTP-Authentifizierung ermöglichen.

Im URL Loading System wird die HTTP-Authentifizierung durch die Delegate-Callback-Methode `connection:didReceiveAuthenticationChallenge:` signalisiert, die ein `NSURLAuthenticationChallenge`-Objekt bereitstellt. Dieses Objekt repräsentiert den Zustand des Challenge/Response-Dialogs zwischen Ihrem Client und dem Server. Sie können zum Beispiel die Methode `protectionSpace` nutzen, um den „Realm“ auf dem Server zu erhalten, den die Authentifizierung benötigt, oder Sie verwenden die Methode `previousFailureCount`, um herauszufinden, ob die Antwort aus einem fehlgeschlagenen Versuch stammt, einen Benutzernamen und ein Passwort anzugeben.

Am wichtigsten ist vielleicht die Methode `sender`, die ein Objekt zurückgibt, das das Protokoll `NSURLAuthenticationChallengeSender` implementiert. Da dieses Objekt den Absender der Challenge darstellt, interagieren Sie damit, um die Authentifizierungs-Challenge zu beantworten. Dabei können Sie entweder antworten, indem Sie die Credentials bereitstellen (`useCredential:forAuthenticationChallenge:`), oder versuchen, ohne Credentials weiterzukommen (`continueWithoutCredentialForAuthenticationChallenge:`), oder aufgeben und die Anfrage abbrechen (`cancelAuthenticationChallenge:`).

Lassen Sie uns versuchen, dies umzusetzen. Im Beispielcode implementieren wir `connection:didReceiveAuthenticationChallenge:`, indem wir uns den Fehlerzähler anschauen:

NetworkIO/SimpleCocoaURLReader/Classes/URLLoaderViewController.m

```

1  - (void)connection:(NSURLConnection *)connection
2      didReceiveAuthenticationChallenge:
3      (NSURLAuthenticationChallenge *)challenge {
4      if ([challenge previousFailureCount] != 0) {
5          // Wenn Fehlerzähler > 0, ist User/Passwort
6          // zurückgewiesen worden
7          NSString *alertMessage =
8              @"Benutzername oder Passwort falsch";
9          UIAlertView *authenticationAlert =
10             [UIAlertView alloc]
11                 initWithTitle:@"Authentifizierung fehlgeschlagen"
12                 message:alertMessage
13                 delegate:nil
14                 cancelButtonTitle:@"OK"
15                 otherButtonTitles:nil];
16             [authenticationAlert show];
17             [authenticationAlert release];
18             [alertMessage release];
19             [activityIndicator stopAnimating];
20     } else {
21         // Nach Authentifizierung fragen
22         AuthenticationChallengeViewController
23             *challengeController =
24             [[AuthenticationChallengeViewController alloc]
25                 initWithNibName:@"AuthenticationChallengeView"
26                 bundle:[NSBundle mainBundle]
27                 loader:self
28                 challenge: challenge];
29         [self presentViewController:challengeController
30             animated:YES];
31         [challengeController release];
32     }
33 }

```

Wenn der Fehlerzähler größer als null ist (Zeile 4), sind wir hier aufgrund einer falschen Antwort aus Benutzername und Passwort gelandet und zeigen daher eine Fehlermeldung an (Zeilen 7–18). Ist es allerdings eine neue Challenge, geben wir eine modale View zur Eingabe von Benutzername und Passwort aus (Zeilen 22–30).

Dieser `AuthenticationChallengeViewController` stellt eine einfache View bereit (siehe Abbildung 12.6), die Felder für den Benutzernamen und das Passwort enthält. Die beiden Actions übergeben die Kontrolle zurück an den aufrufenden View-Controller – entweder durch den Aufruf einer „Cancel“-Methode oder durch das Bereitstellen von Benutzername und Passwort an eine „OK“-Methode.

Beide Methoden geben den modalen View-Controller frei und holen sich den `NSURLAuthenticationChallengeSender` aus der Challenge. Der Unterschied liegt nur darin, dass die Cancel-Methode die Nachricht `cancelAuthenticationChallenge:` schickt, während die „OK“-Methode ein `NSURLCredential`-Objekt mit Benutzernamen und Passwort erzeugt und dies als Antwort auf die Challenge liefert:

NetworkIO/SimpleCocoaURLReader/Classes/URLLoaderViewController.m

```
- (void) handleAuthenticationOKForChallenge:
    (NSURLAuthenticationChallenge *) aChallenge
    withUser: (NSString*) username
    password: (NSString*) password {
    // Versuch, auf Challenge zu reagieren
    NSURLCredential *credential = [[NSURLCredential alloc]
        initWithUser:username
        password:password
        persistence:NSURLCredentialPersistenceForSession];
    [[aChallenge sender] useCredential:credential
        forAuthenticationChallenge:aChallenge];
    [credential release];
    [self dismissModalViewControllerAnimated:YES];
}
```



Abbildung 12.6: GUI für Benutzername und Passwort

Beachten Sie, dass der Initializer von `NSURLCredential` ein Argument `persistence` erhält. Dabei kann es sich um eine der drei `NSURLCredentialPersistence`-Konstanten handeln: `NSURLCredentialPersistenceNone` für Credentials, die direkt wieder vergessen werden, `NSURLCredentialPersistenceForSession` für Credentials, die für die Dauer der aktuellen Sitzung gespeichert werden, und `NSURLCredentialPersistencePermanent` für Credentials, die dauerhaft im Schlüsselbund des Anwenders abgelegt werden. Das tatsächliche Verhalten scheint sich zwischen Simulator und iPhone zu unterscheiden – auf dem iPhone überleben „permanente“ Credentials den Neustart der Anwendung (und sogar das Entfernen und Neuinstallieren der Anwendung), die anderen aber nicht.

So weit sind Sie fertig – egal ob Benutzername und Passwort akzeptiert werden oder nicht. Wenn es klappt, erhalten Sie Aufrufe von `connection:didReceiveData:` mit dem Inhalt der URL, auf die Sie jetzt Zugriff haben. Wenn nicht, erhalten Sie einen weiteren Aufruf von `connection:didReceiveAuthenticationChallenge:`, und dieses Mal wird der Wert von `previousFailureCount` der `NSURLAuthenticationChallenge` nicht null sein, die Challenge wird also eine ungültige Kombination aus Benutzername und Passwort erhalten.

12.4 XML von Webservices parsen

Jetzt wollen wir einen Moment innehalten und darüber nachdenken, was Sie mit den Daten machen wollen, die Sie über das Netzwerk erhalten. Unser Beispiel mit der Anzeige von heruntergeladenem HTML in einer Text-View lässt sich zwar einfach programmieren, ist aber nicht besonders sinnvoll. Schließlich werden Sie die heruntergeladenen Daten auch *parsen* wollen.

In vielen Fällen kann man auf einen Service über eine wie auch immer geartete Netzwerk-API zugreifen, die festlegt, was ein Client sendet (und wie er es tut) und was der Service darauf antwortet. Allgemein können Web-APIs, die HTTP nutzen, als *Webservices* bezeichnet werden.⁸ Und sehr häufig sind die vom Service gelieferten Daten in XML codiert.

Lassen Sie uns ein konkretes Beispiel nehmen. Twitter besitzt eine sehr offene, einfach zu verwendene API, die im Developer-Wiki unter <http://apiwiki.twitter.com/> detailliert beschrieben ist. Diese REST-API kann

⁸ Auch wenn manche Definitionen Webservices darauf beschränken, dass sie gewissen Standards erfüllen müssen, wie zum Beispiel SOAP, wird der Begriff immer mehr für jegliche HTTP-basierte Kommunikation zwischen zwei Rechnern verwendet.

sogar über einen Browser angesprochen werden. So ist zum Beispiel die „Public Timeline“, die die aktuellsten Änderungen aller Benutzer zeigt, unter `http://twitter.com/statuses/public_timeline.format` verfügbar, wobei `format` einen der Werte `xml`, `json`, `rss` und `atom` annehmen kann. Um also eine XML-View der Timeline zu erhalten, müssen Sie nur die URL `http://twitter.com/statuses/public_timeline.xml` laden. Sie können diese URL sogar im Beispiel `SimpleCocoaURLLoader` nutzen und sich die XML-Antwort in der Text-View anzeigen lassen. Hier ein Fragment des zurückgegebenen XML-Codes, es wurde ein wenig angepasst, um die Struktur und einige der wichtigsten Felder hervorzuheben:

```
<?xml version="1.0" encoding="UTF-8"?>
<statuses type="array">
<status>
  <created_at>Sun Apr 19 17:02:43 +0000 2009</created_at>
  <id>1559022041</id>
  <text>Wife and kids shopping. Must write and finish webservices
    section this afternoon.</text>
  <source>web</source>
  [...]
  <user>
    <id>12604932</id>
    <name>Chris Adamson</name>
    <screen_name>invalidname</screen_name>
    <location>Grand Rapids, MI</location>
    <description>I write, edit, and code stuff. I also raise
      children and sometimes clean things</description>
    [...]
  </user>
</status>
[...]
```

Lassen Sie uns nun einen Schritt weitergehen und die nützlichen Informationen auslesen. Wir werden nur die Namen der Anwender und den Statustext ermitteln – ihre *Tweets*.

Erstellen Sie ein neues, View-basiertes Projekt `SimpleTwitterClient` und verwenden Sie den IB, um `SimpleTwitterClientViewController.xib` so zu bearbeiten, dass es eine große Text-View, einen Button `TWEETS AKTUALISIEREN` und einen Aktivitätsindikator anzeigt (siehe Abbildung 12.7). Deklarieren Sie dann in der Header-Datei `IBOutlet`-Eigenschaften für `activityIndicator` und `tweetsView` und eine IB-Action-Methode `updateTweets` für den Button. Verbinden Sie die beiden Outlets und das Button-Event „Touch Up Inside“ entsprechend.

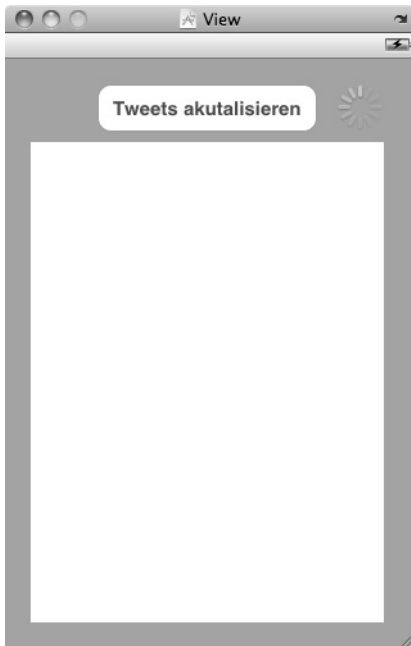


Abbildung 12.7: Die View `SimpleTwitterClientViewController.XIB` im IB bearbeiten

Die Twitter-Daten laden

Beim Implementieren der Methode `updateTweets` müssen wir zwei Dinge tun: Auslesen der XML-Daten über den Twitter-Webservice und Parsen der Teile, die uns interessieren. Sie wissen schon, wie Sie den ersten Schritt umsetzen können. Sie erstellen einen `NSURLRequest` und daraus eine `NSURLConnection`. Dann schreiben wir die Delegate-Methoden, die sich um das Empfangen der Daten und die Fehlerbehandlung kümmern. Viele der Twitter-APIs erfordern eine HTTP-Authentifizierung; wenn Sie also diese Features unterstützen wollen, müssen Sie auch Callbacks wie `connection:didReceiveAuthenticationChallenge:` implementieren.

Aber was wollen wir mit den Daten machen? Lassen Sie uns schauen, was wir zum Parsen benötigen. Die Klasse `NSXMLParser` besitzt zwei Initializer-Methoden: `initWithContentsOfURL:` und `initWithData:`. Es gibt zwei gute Gründe, nicht die erste Methode zu nutzen – sie kann das GUI beim Herunterladen der Daten blockieren, und sie kommt nicht mit HTTP-Authentifizierung klar. Daher werden wir mit `initWithData:` arbeiten, wozu wir den heruntergeladenen XML-Code in einer Variablen vom Typ `NSMutableData` sammeln und diese dann dem Parser übergeben.

Definieren Sie daher in der Header-Datei `NSMutableData *tweetsData`. Jetzt wollen wir die Methode `updateTweets` implementieren, um `tweetsData` zu allozieren und mit dem Laden von der URL zu beginnen.

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
-(IBAction) updateTweets {
    tweetsView.text = @"";
    [tweetsData release];
    tweetsData = [[NSMutableData alloc] init];
    NSURL *url = [NSURL URLWithString:
        @"http://twitter.com/statuses/public_timeline.xml"];
    NSURLRequest *request = [[NSURLRequest alloc] initWithURL: url];
    NSURLConnection *connection = [[NSURLConnection alloc]
        initWithRequest:request
        delegate:self];
    [connection release];
    [request release];
    [activityIndicator startAnimating];
}
```

Nachdem die Anfrage initialisiert wurde, beginnt das URL Loading System mit dem Aufruf unserer Delegate-Callback-Methoden. Dabei wird sehr wahrscheinlich zunächst `connection:didReceiveResponse:` aufgerufen, gefolgt von einer Reihe von Delegate-Callbacks von `connection:didReceiveData:`, die die XML-Daten liefern. Hier müssen wir nur jeden neuen Datenblock an `tweetsData` anhängen.

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
-(void)connection:(NSURLConnection *)connection didReceiveData: (NSData *)data
{
    [tweetsData appendData: data];
}
```

Wenn das URL Loading System schließlich das Stream-Ende erreicht, ruft es noch einmal `connection:didFinishLoading:` auf. Wir können den `activityIndicator` wieder ausblenden und eine noch zu schreibende Methode `parseTweets` aufrufen.

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
-(void) connectionDidFinishLoading: (NSURLConnection*) connection {
    [activityIndicator stopAnimating];
    [self startParsingTweets]; }
```

Die Twitter-Daten parsen

Jetzt haben wir alle XML-Daten in einem `NSData`-Objekt gesammelt. Um sie zu parsen, nutzen wir einen `NSXMLParser`. Es gibt beim Parsen von XML-Daten zwei unterschiedliche Ansätze. Ein *DOM-Parser* erzeugt eine

Baumstruktur im Speicher, in der jeder Knoten einem Element des XML-Markup entspricht. Wenn Sie die tiefe Struktur des Baums brauchen, kann das sehr hilfreich sein, aber oft ist das gar nicht nötig. Zudem müssen Sie bei DOM-Parsern häufig warten, bis der gesamte Baum geparkt wurde, bevor Sie etwas mit den Daten anstellen können. Das alternative Vorgehen läuft über den *Event-gesteuerten Parser*, bei dem der Parser interessierte Beteiligte beim Parsen jedes XML-Tags informiert und den „Zuhörer“ entscheiden lässt, ob etwas zu tun ist. Das ist potenziell leichtgewichtiger und erlaubt Ihrem Code, während des Parsens auch noch nützlich tätig zu werden – zum Beispiel die zwar noch unvollständigen, aber trotzdem schon hilfreichen Daten anzuzeigen.

Der NSXMLParser folgt diesem Event-getriebenen Vorgehen. Schauen Sie sich seine Dokumentation an, finden Sie ein sehr vertrautes Muster. Nach dem Erzeugen des Parsers setzen Sie einen Delegate⁹ und erhalten danach Aufrufe verschiedener Callback-Methoden, wenn der XML-Code geparkt wird: `parserDidStartDocument:`, `parser:foundCharacters:`, `parser: didStartElement:namespaceURI:qualifiedName:attributes:` und so weiter. Das ist das gleiche asynchrone Muster wie in Abschnitt 12.2, *Daten über das Netzwerk lesen*, auf Seite 265 und noch weiter oben in Abschnitt 8.5, *Asynchrones Lesen von Dateien*, auf Seite 164.

Nachdem Sie nun wissen, wie Sie aufgerufen werden, müssen Sie sich noch Gedanken über die Daten machen, die in diesen Callbacks geliefert werden. Lassen Sie uns die XML-Daten, die wir von Twitter erhalten, nochmals aus einem anderen Blickwinkel betrachten. Ein Überblick über die Daten ist in Abbildung 12.8 als Baum dargestellt.

Die Twitter-Timeline-APIs schicken XML-Daten mit einem einzelnen `<statuses>`-Element zurück, das eine Reihe von Status-Kindelementen enthält. Jedes davon ist ein Tweet, dessen Elemente einen Zeitstempel, eine eindeutige ID und weitere Metadaten sowie ein `<text>`-Element mit dem Tweet-Text selbst bereitstellen. Der `<status>` enthält zudem ein `<user>`-Element, das den Autor des Tweet enthält. Dieses Element besitzt wiederum drei Kindelemente, die den Anwender durch `<name>`, `<screen_name>` und `<location>` beschreiben.

9 NSXMLParser nutzt ein *informelles* Delegate-Protokoll – es gibt nur eine Liste mit Callback-Methoden in der Dokumentation, kein definiertes Protokoll, das Ihre Klasse deklarieren muss.

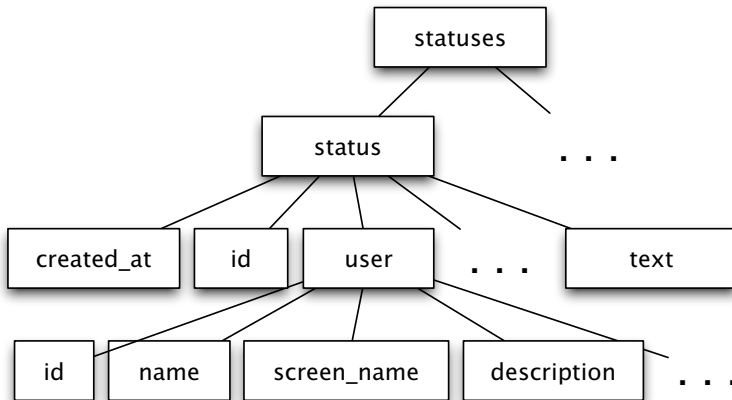


Abbildung 12.8: Diagramm der XML-Antwort der Twitter-Webservice-API

Es handelt sich hier also um eine verschachtelte Baumstruktur, aber wir müssen uns gar nicht unbedingt selbst darum kümmern. Innerhalb jedes `<status>` kann sich unser einfaches Beispiel auf die Verarbeitung der Tags `<text>` und `<name>` beschränken und den Rest ignorieren. Lassen Sie uns also diese Einschränkung berücksichtigen.

Wir erzeugen zunächst einen `NSXMLParser` mit dem heruntergeladenen `tweetData` und lassen ihn direkt mit dem Parsen beginnen.

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```

- (void) startParsingTweets {
    NSXMLParser *tweetParser = [[NSXMLParser alloc]
                                initWithData:tweetData];
    tweetParser.delegate = self;
    [tweetParser parse];
    [tweetParser release];
}

```

Damit werden direkt die Callback-Methoden aufgerufen, wenn es denn welche gibt. Wir müssen uns also überlegen, was jede dieser Methoden tun soll. Letztendlich wollen wir einen großen String mit Name/Nachricht-Paaren aufbauen, um ihn in die Text-View zu übertragen. Also brauchen wir einen `NSMutableString`, an den wir wiederholt etwas anfügen können, während jeder Tweet geparkt wird. Um einen Tweet zu erstellen, müssen wir auf die `status`-Elemente achten. Sobald eines begonnen hat, warten wir auf `name`- und `text`-Elemente, die als Name/Wert-Paare in einem `NSMutableDictionary` gespeichert werden können, bis das `status`-Element endet und wir den Tweet an den `tweetsString` anhängen. Innerhalb jedes dieser Elemente benötigen wir nur den Text, aber es gibt da ein Problem: `parser:foundCharacters:` garantiert nicht,

dass der gesamte Text eines Tags in einem Callback zurückkommt. Das passiert sogar ausgesprochen selten. Wir brauchen also einen NSMutableString, um den Text eines Elements zu sammeln, bis es beendet ist.

Mit all diesem Wissen können Sie nun die folgenden Instanzvariablen in der Header-Datei definieren:

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.h

```
NSMutableString *tweetsString;
NSMutableDictionary *currentTweetDict;
NSString *currentElementName;
NSMutableString *currentText;
```

Jetzt sind wir so weit. Als Erstes kümmern wir uns um die Nachricht `parser:didStartDocument:`. Wenn wir diesen Callback erhalten, können wir den `tweetsString` allozieren:

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
- (void)parserDidStartDocument:(NSXMLParser *)parser {
    [tweetsString release];
    tweetsString = [[NSMutableString alloc]
        initWithCapacity: (20 * (140 + 20)) ];
    currentElementName = nil;
    currentText = nil; }
```

Da Tweets nur 140 Zeichen lang sein können und die Webservice-API von Twitter höchstens 20 Tweets pro Aufruf liefert, können wir ungefähr abschätzen, wie groß ein String sein muss, den wir benötigen. Fügen Sie weitere 20 Zeichen für einen Benutzernamen und Leerraum hinzu, allozieren wir unseren NSMutableString mit genug Platz für 20 160-Zeichen-Elemente, bevor er seine Größe anpassen muss.

Sobald das Parsen des Dokuments begonnen wurde, müssen wir in den Callbacks die einzelnen Elemente abhandeln. Wir erhalten für jedes einzelne Element Callbacks von `parser:didStartElement:` in der klassischen Reihenfolge: erst `statuses`, dann deren erstes `status`-Kindelement, dann das erste Kindelement von `status` (vielleicht `created_at`) und so weiter, bis wir zu einem Element ohne Kindelemente kommen. Wir erhalten für dessen Text einen oder mehrere Callbacks von `parser:foundCharacters:` und dann ein `parser:didEndElement:`. Der Parser kehrt nun zum Elternelement zurück und findet dort entweder ein weiteres Kindelement oder beginnt mit dem Text des Elternelements und so weiter.

Wir wollen uns gar nicht um jedes Element kümmern. Eigentlich interessieren uns nur drei: `status`, `name` und `text`. Das erste ist ein besonderes, da wir durch dieses erfahren, dass wir auf seine Kindelemente

achten müssen. Die anderen sind die, aus denen wir das `currentTweetDict` erstellen. Um auf sie zu lauschen, fügen wir der Header-Datei ein `NSSet *interestingTags` hinzu. Zum Einrichten legen wir zunächst per **#define** fest, welche Elemente uns interessieren:

```
NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m
```

```
#define INTERESTING_TAG_NAMES @"text" ,@"name" , nil
```

Dann initialisieren wir sobald wie möglich (zum Beispiel in `viewDidLoad`) das `NSSet` mit diesen Werten. Beachten Sie, dass `initWithObjects`: eine Objektliste variabler Länge übernehmen kann, die durch `nil` beendet wird. Wir könnten diese Liste hier direkt codieren, aber die Verwendung von **#define** am Anfang der Datei erhöht die Übersicht und Lesbarkeit:

```
NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m
```

```
- (void) viewDidLoad {
    [super viewDidLoad];
    interestingTags = [[NSSet alloc] initWithObjects:
                                                                INTERESTING_TAG_NAMES];
}
```

Jetzt kümmern wir uns endlich um die Elemente, an denen wir interessiert sind, wenn der Parser sie erreicht. Wir beginnen ein Element mit dem Callback `parser:didStartElement:namespaceURI:qualifiedName:attributes::`:

```
NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m
```

```
1 - (void)parser:(NSXMLParser *)parser didStartElement:
2                                     (NSString *)elementName
3                                     namespaceURI:(NSString *)namespaceURI
4                                     qualifiedName:(NSString *)qualifiedName
5                                     attributes:(NSDictionary *)attributeDict {
6     if ([elementName isEqualToString:@"status" ]) {
7         [currentTweetDict release];
8         currentTweetDict = [[NSMutableDictionary alloc]
9                             initWithCapacity: [interestingTags count]];
10    }
11    else if ([interestingTags containsObject: elementName]) {
12        currentElementName = elementName;
13        currentText = [[NSMutableString alloc] init];
14    }
15 }
```

Damit erreichen wir zwei Dinge. Handelt es sich um ein `status`-Element, stehen wir damit am Anfang eines neuen Tweet und setzen daher ein `NSMutableDictionary` auf, in dem Name/Wert-Paare für alle interessierten Kindelemente enthalten sind. Handelt es sich dagegen

um eines der „interessanten“ Tags, also `name` oder `text`, merken wir uns den Namen des aktuellen Elements und richten einen `NSMutableString` für dessen Wert ein. Beachten Sie in dem **else if** in Zeile 11, wie die Methode `containsObject:` von `NSSet` es leicht macht, die Liste der interessanten Tag-Namen zu durchsuchen. Das ist der Grund dafür, dass wir es vorher eingerichtet haben.

Die nächste Nachricht, die wir vom Parser erwarten, ist einer bzw. sind mehrere Callbacks mit dem Text eines Elements:

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString *)string
{
    [currentText appendString:string];
}
```

Hier werden die Zeichen an den String `currentText` angehängt, in dem wir den Wert eines Name/Wert-Paars aufbauen. Der Callback sagt Ihnen nicht, zu welchem Element der Text gehört, da Sie das schon aus einem vorherigen Aufruf beim Start des Elements wissen sollten. Beachten Sie allerdings auch, dass wir nicht prüfen, ob das aktuelle Element „interessant“ ist, bevor wir den Text anhängen. Das ist eine pfiffige kleine Optimierung: Allokieren wir einen `currentText` nur für die interessanten Elemente und setzen ihn auf `nil`, wenn wir ein *beliebiges* Element beenden, wird diese Append-Nachricht ins Nirvana geschickt, wenn wir uns nicht innerhalb von interessanten Elementen befinden.

Jetzt wissen wir, was wir am Anfang eines Elements und mit dem Text innerhalb eines Tag-Paars machen wollen, das uns interessiert. Nun lassen Sie uns anschauen, was wir tun, wenn der Parser das Ende eines Elements findet:

NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m

```
- (void)parser:(NSXMLParser *)parser
    didEndElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName {
    if ([elementName isEqualToString:currentElementName]) {
        [currentTweetDict setValue:currentText forKey:
            currentElementName];
    } else if ([elementName isEqualToString:@"status" ]) {
        [tweetsString appendFormat:@"%@: %@\n\n" ,
            [currentTweetDict valueForKey:@"name" ],
            [currentTweetDict valueForKey:@"text" ]];
    }
    [currentText release];
    currentText = nil;
}
```

Am Ende eines Elements erledigen wir zwei Dinge. Wenn es sich um eines der „interessanten“ Tags handelt, wissen wir, dass es nicht mehr Text für dieses Element gibt und wir das Name/Wert-Paar mit `currentElementName` und dem „aufsummierten“ `currentText` nutzen können. Wurde andererseits ein `status`-Element beendet, haben wir alle Metadaten für einen Tweet geparkt und können dessen Werte nutzen. Bei diesem einfachen Beispiel fügen wir einfach den Benutzernamen und die Nachricht des Tweet an den veränderbaren `tweetsString` an. Wir müssen auch jegliche `currentText`-Objekte freigeben und auf `nil` setzen, die wir in `parser:didStartElement:` alloziert haben.

Wenn das Parsen beendet ist, erhalten wir den Callback `parserDidEndDocument:`. Dann müssen wir nur noch den Inhalt der Text-View durch den aufsummierten `tweetsString` ersetzen:

```
NetworkIO/SimpleTwitterClient/Classes/SimpleTwitterClientViewController.m
```

```
- (void)parserDidEndDocument:(NSXMLParser *)parser {
    tweetsView.text = tweetsString;
}
```

Das ist alles, was wir zum Parsen der Public Timeline von Twitter machen müssen. Führen Sie die Anwendung per **BUILD AND RUN** aus und klicken Sie auf den Button zum Aktualisieren. Das Ergebnis sehen Sie in Abbildung 12.9.

Mehr Optionen zum Parsen von XML

Dies ist natürlich ein sehr einfaches Beispiel gewesen, und es gibt ein paar Erweiterungen, die sich anbieten. Wenn Sie zum Beispiel die Twitter-Webservice-API noch umfassender nutzen wollen – wie die Services für die „Friends Timeline“ eines Anwenders oder das Posten eigener Tweets – brauchen Sie die HTTP-Authentifizierung, wie sie in Abschnitt 12.2, *Daten über das Netzwerk lesen*, auf Seite 265 beschrieben wurde.

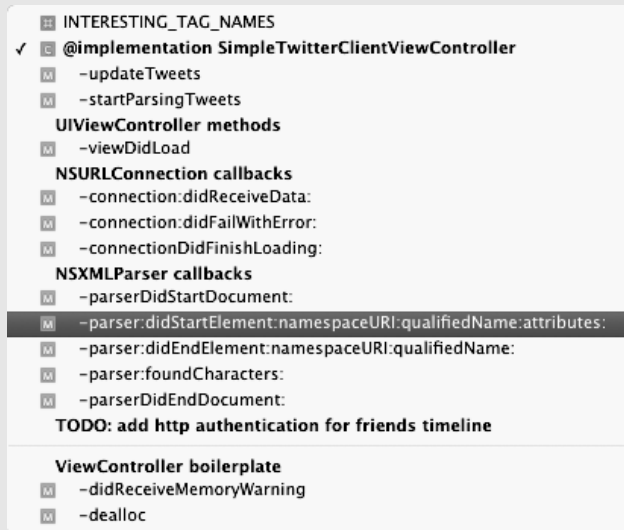
Es ist natürlich möglich, dass Sie beim Parsen komplexer und nachhaltiger vorgehen wollen. So kann es zum Beispiel nützlich sein, sich eher am DOM zu orientieren und die Eltern-Kind-Beziehungen des XML-Codes nachzubilden. Das lässt sich durch Delegation in Objective-C erstaunlich einfach erreichen. Um die Baumstruktur aufzubauen, erzeugen Sie eine Elementklasse, die selber parst – das informelle Delegate-Protokoll `NSXMLParser` also implementiert. Sie erzeugen ein Root-Element und lassen es der Delegate des Parsers sein.

Code ordentlich halten – mit #pragma mark

Wenn Sie eine einzelne Klasse haben, die mehrere Protokolle implementiert und zudem Methoden aus ihren Superklassen überschreibt, kann das Ganze schnell unübersichtlich werden. Um Ihre Methoden klar und ordentlich zu halten, können Sie die Direktive **#pragma mark** nutzen. Jeglicher Text hinter **#pragma mark** wird zu einem Menüeintrag in der Liste der Methoden und Funktionen im Xcode-Editor. Hier ein Beispiel:

#pragma mark NSURLConnection Callbacks

Wir haben ein paar dieser Zeilen in SimpleTwitterClientViewController.m untergebracht, um die Methoden besser organisieren zu können. Dies ist das Ergebnis:



Beachten Sie, dass wir auch einen „TODO“-Eintrag haben. Alle einzeiligen Kommentare der Form `// TODO: Text` werden ebenfalls diesem Menü hinzugefügt. Um die Liste aufgeräumt darzustellen, können Sie auch einen einzelnen Bindestrich als Text des **#pragma mark** nutzen, um damit eine Trennlinie zu erzeugen.



Abbildung 12.9: Die Public Timeline von Twitter mit dem NSXMLParser anzeigen

Das Element kümmert sich dann um die verschiedenen Callbacks. Erhält es einen für ein neues Element im XML, erzeugt es ein neues Elementobjekt und macht dieses neue Element zum Delegate des Parsers. Wenn der Parser umgekehrt das Ende eines XML-Elements erreicht, gibt das Elementobjekt den Delegate an sein Elternelement zurück. Das Root-Element erhält dann schließlich den Callback-Aufruf `parserDidEndDocument:` und ist damit das Elternelement eines Objektbaums, der den Inhalt des XML-Codes repräsentiert. Mehr Details zu diesem rekursiven Vorgehen finden Sie in *Event-Driven XML Programming Guide for Cocoa* [App08b] von Apple.

Es gibt noch eine weitere Option, die man sich anschauen sollte – zwei Projekte bei Google Code. TouchXML ist eine kleine, übersichtliche Alternative zur XML-Unterstützung von Mac OS X, das eine Reihe von Klassen hat, die auf dem iPhone OS nicht verfügbar sind. Ausgehend von der Open Source-Bibliothek libxml2, wird das Parsen eines NSData oder NSString unterstützt, es gibt eine eingeschränkte Unterstützung von XPath und eine Reihe von Navigationsmethoden, um Geschwisterelemente, Kindelemente und anderes anzusteuern. Ein weiteres Projekt,

TouchJSON, ist ein Parser und Generator für das JSON-Format, das von einigen Webservices genutzt wird. Beide Projekte gehören zur Google-Sammlung Touch Code mit Open Source-Projekten für das iPhone OS. Sie finden sie unter <http://code.google.com/p/touchcode/>.

12.5 Mails aus Ihrer Anwendung schicken

Bisher haben wir einige der am häufigsten genutzten Anwendungen eines Netzwerks behandelt: das Browsen im Web, das Herunterladen von Daten von Webservern und das Verarbeiten von Webservice-Daten. Unsere letzte Aktivität wird in diesem Kapitel die Nutzung von E-Mails sein. Jedes iPhone und jeder iPod touch bringt schon eine anständige E-Mail-Anwendung mit, und im iPhone OS 3.0 haben Entwickler Zugriff auf dessen Standard-UI, um E-Mails zu verschicken.

Das optionale Message UI-Framework besteht aus nur einer Klasse und einem Delegate-Protokoll. Unser Beispiel wird dementsprechend einfach sein. Erstellen Sie eine View-basierte Anwendung mit dem Namen InAppMailer. Fügen Sie dem Projekt das MessageUI.framework hinzu, ergänzen Sie in InAppMailViewController.h die Zeile `#import <MessageUI/MessageUI.h>` und tragen Sie dann das Protokoll `<MFMailComposeViewControllerDelegate>` in der **@interface**-Deklaration ein. Da Sie schon gerade in der Header-Datei sind, deklarieren Sie auch noch ein `IBOutlet` für eine `UITextView` namens `mailHistoryView`, machen es zu einer **@property** und deklarieren eine `IBAction` mit dem Namen `composeMailTapped`.

Das GUI wird einfach sein: Öffnen Sie `InAppMailViewController.xib` und fügen Sie eine Text-View (verbunden mit dem Outlet `mailHistoryView` des File's Owner) und einen Button mit dem Text „Mail erstellen“ hinzu, dessen Event Touch Up Inside mit der Action `composeMailTapped` des File's Owner verbunden ist. Sie haben jetzt einen Button, um die View zum Schreiben einer Mail anzuzeigen, und eine Text-View, um die Ergebnisse auszugeben.

Um eine E-Mail zu erstellen, erzeugen Sie eine Instanz des `MFMailComposeViewController`, der eine View verwaltet, die wie die View „Neue E-Mail“ der Mailanwendung aussieht (siehe Abbildung 12.10). Der View-Controller kümmert sich um alle Events, während die View zum Schreiben der Mail angezeigt wird – zum Beispiel das Anzeigen der Tastatur, wenn der Anwender sich in einem eingabebereiten Feld befindet, oder die Ausgabe eines Pickers, wenn der Anwender auf die Zeile `ACCOUNTS` tippt, um einen seiner Accounts auszuwählen.



Abbildung 12.10: Mails erstellen

Normalerweise erstellen Sie den `MFMailComposeViewController` per Code und zeigen ihn modal an. Also wird folgender Code ausgeführt, wenn der Anwender auf den Button tippt:

`NetworkIO/InAppMailer/Classes/InAppMailerViewController.m`

```
-(IBAction) composeMailTapped {
    if (![MFMailComposeViewController canSendMail]) {
        UIAlertView *cantMailAlert = [[UIAlertView alloc]
            initWithTitle:@"Mail nicht möglich"
            message:@"Dieses Gerät kann keine E-Mail verschicken"
            delegate:NULL
            cancelButtonTitle:@"OK"
            otherButtonTitles:NULL];
        [cantMailAlert show];
        [cantMailAlert release];
        return;
    }
    MFMailComposeViewController *mailController =
        [[[MFMailComposeViewController alloc] init] autorelease];
    [mailController setMessageBody:@"Meine App kann mailen" isHTML:NO];
    mailController.mailComposeDelegate = self;
    [self presentViewController:mailController animated:YES];
}
```

Zuerst wird hier die Klassenmethode `canSendMail` aufgerufen, um herauszufinden, ob das iPhone überhaupt so eingerichtet wurde, dass Mails verschickt werden können. Ist das nicht der Fall, wird eine Fehlermeldung ausgegeben und zurückgesprungen.

Wenn das iPhone Mails verschicken kann, erzeugen Sie eine Instanz des `MFMailComposeViewController` und befüllen optional schon diverse Felder der Nachricht. Sie können den Betreff, die Empfänger, CC- und BCC-Empfänger und den Nachrichtentext (als reinen Text oder als HTML) vorgeben, aber auch schon Anhänge hinzufügen. In diesem Beispiel nutzen wir `setMessageBody:isHTML:`, um den Nachrichtenrumpf mit `Meine App` kann mailen vorzubelegen. Beachten Sie, dass die View zum Erstellen einer Mail angezeigt wird und der Anwender daher beliebige Felder ändern kann, die Sie befüllt haben – diese Klasse erlaubt es Ihnen nicht, den Benutzer etwas abschicken zu lassen, ohne dass er es weiß. Der Controller besitzt zudem eine Eigenschaft `mailComposeDelegate`, die Sie setzen sollten, um darüber benachrichtigt zu werden, wenn der Anwender mit dem Erstellen der Mail fertig ist. Sind Felder vorbefüllt und der Delegate gesetzt, können Sie die View zum Erstellen der Mail mit `presentModalViewController` auf den Bildschirm bringen.

Das Protokoll `MFMailComposeViewControllerDelegate` definiert eine einzelne Callback-Methode: `mailComposeController:didFinishWithResult:error:`. Diese zeigt an, dass der Anwender mit dem Erstellen der Mail fertig ist. Der zweite Parameter ist einer der vier enumerierten Werte von `MFMailComposeResult`: `MFMailComposeResultSent` wird genutzt, wenn der Anwender eine Nachricht erstellt und verschickt hat, `MFMailComposeResultSaved`, wenn er auf `ABBRECHEN` getippt, sich dann aber dafür entschieden hat, die Nachricht im Entwürfe-Ordner der Mail-App abzuspeichern, `MFMailComposeResultCancelled`, wenn der Anwender abgebrochen und nicht gespeichert hat, und `MFMailComposeResultFailed`. Mehr Details im Fehlerfall erhalten Sie dann über den Parameter `error`.

In unserer Beispiel-App protokollieren wir nur das Ergebnis und die Zeit in der Text-View:

NetworkIO/InAppMailer/Classes/InAppMailerViewController.m

```
- (void)mailComposeController:(MFMailComposeViewController*)controller
    didFinishWithResult:(MFMailComposeResult)result
    error:(NSError*)error {
    if (error) {
        UIAlertView *cantMailAlert = [[UIAlertView alloc]
            initWithTitle:@"Mail-Fehler"
            message: [error localizedDescription]
            delegate:NULL
```

```

        cancelButtonTitle:@"OK"
        otherButtonTitles:NULL];
        [cantMailAlert show];
        [cantMailAlert release];
    }
    NSString *resultString;
    switch (result) {
        case MFMailComposeResultSent:
            resultString = @"Mail gesendet" ; break;
        case MFMailComposeResultSaved:
            resultString = @"Main gespeichert" ; break;
        case MFMailComposeResultCancelled:
            resultString = @"Mail abgebrochen" ; break;
        case MFMailComposeResultFailed:
            resultString = @"Fehler" ; break;
    }
    mailHistoryView.text = [NSString stringWithFormat:
        @"%@%@ um %@\n" , mailHistoryView.text,
        resultString, [NSDate date]];
    [controller dismissModalViewControllerAnimated:YES];
    [controller release];
}

```

Wenn wir diesen Callback erhalten, hat der MFMailComposeViewController seine Arbeit getan, daher rufen wir `dismissModalViewControllerAnimated` auf, um ihn vom Bildschirm zu holen und dann freizugeben.

Sie werden feststellen, dass es noch eine Reihe von Dingen gibt, die Sie vielleicht gern mit der fertigen E-Mail machen würden, aber nicht können. Die Felder der E-Mail sind nur über die weiter oben vorgestellten Setter-Methoden erreichbar. Sie können also nicht per Code darauf zugreifen und die Adresse, den Text oder andere Elemente der Nachricht auslesen. Sie werden auch merken, dass Sie das Verschicken oder Speichern der Nachricht nicht per Code beeinflussen können. Das MessageUI-Framework ist dazu gedacht, dem Benutzer die Kontrolle zu überlassen – und das „unbemerkte“ Verschicken von E-Mail oder der Zugriff auf eine Nachricht ist nun wirklich eine Frage der Privatsphäre.

Kapitel 13

Peer-to-Peer-Netzwerke

iPhones und iPod touches gibt es überall, und alle besitzen sie verschiedene drahtlose Verbindungsoptionen. In diesem Kapitel werden wir uns anschauen, wie Anwendungen auf iPhones die Dienste erkennen können, die sie benötigen, und wie sie sich per Wifi oder Bluetooth mit anderen iPhones oder Laptops in der Nähe verbinden können. Zunächst werden wir uns mit dem Bonjour-Protokoll befassen und uns dann das mit dem iPhone 3.0 neu dazugekommene Game Kit Framework vornehmen. Das Game Kit bietet zwei wichtige neue Kommunikationstechnologien für Entwickler an: eine drahtlose Bluetooth-Verbindung und ein Voice Chat.

13.1 Ad-hoc-Netzwerk-Services mit Bonjour

In Kapitel 12, *Verbindung zum Internet*, auf Seite 259 haben wir uns mit URLs verbinden lassen, die entweder eindeutig zu einer Anwendung gehören (zum Beispiel für einen öffentlichen Webservice) oder vom Anwender selbst eingegeben wurden. Aber es gibt auch Fälle, in denen sich der Anwender gar nicht mit den Netzwerkdetails befassen müssen soll. Insbesondere bei LAN-basierten Anwendungen, beispielsweise beim Finden von Freunden für einen Chat oder beim Drucken von Dokumenten, ist der Benutzer lediglich an der Funktionalität interessiert – er interessiert sich nur dafür, was passiert, nicht wie es passiert.

Mit dem iPhone OS sind die Anwendungsbereiche für Bonjour noch umfassender. iPhone und iPod touch kommen ständig in die Nähe von LANs und verlassen sie auch wieder. Statt den Anwender nun immer wieder neue Hostnamen oder Netzadressen eingeben zu lassen, ist es viel angenehmer, das Gerät selbst die notwendigen Dienste finden zu lassen.

Diese service-orientierte Ausrichtung ist die Philosophie von *Bonjour*. In Bonjour ist sich der Anwender der Hostnamen, Ports oder Pfade häufig gar nicht bewusst. Stattdessen stellen Bonjour-Anwendungen Netzwerkdienste bereit: Drucker, Chat-Partner, iTunes-Musiksammlungen und so weiter.

Um zu zeigen, wie das funktioniert, bauen wir darauf auf, dass Ihr Mac seinen Apache Webserver für Bonjour einrichten kann. Wir werden dies nutzen, um mit dem iPhone Sites im LAN anzusteuern, ohne eine Adresse angeben zu müssen.

Apache für Bonjour einrichten

Zunächst müssen wir aber Ihre Apache-Websites für einen Zugriff per Bonjour einrichten. In Mac OS X 10.5 (Leopard) ist Bonjour standardmäßig in Ihren Apache-Konfigurationen deaktiviert. Sie müssen es also einschalten, sofern Sie nicht von älteren Versionen des Mac OS X ein Update durchgeführt haben (denn dort war es standardmäßig aktiv). Schauen Sie sich die Datei `/etc/apache2/other/bonjour.conf` an. Die Standarddatei von Leopard, die keinen Zugriff von Bonjour erlaubt, sieht so aus:

```
<IfModule bonjour_module>
    RegisterUserSite customized-users
</IfModule>
```

Wir können diese Datei bearbeiten, um das Haupt-Webverzeichnis Ihres Rechners, einzelne Benutzerverzeichnisse oder beides freizugeben. Für die Haupt-Website (also `http://localhost/`) fügen Sie die Direktive `<RegisterDefaultSite>` hinzu. Sie können auch Benutzerverzeichnisse (zum Beispiel `http://localhost/~cadamson`) mit der Direktive `<RegisterUserSite>` freigeben, wobei entweder ein Benutzername oder der Text `all-users` übergeben wird. Um sowohl die Haupt-Website als auch alle Benutzer-Websites freizugeben, sollte Ihre Datei `bonjour.conf` so aussehen:

```
<IfModule bonjour_module>
    # RegisterUserSite customized-users
    RegisterDefaultSite
    RegisterUserSite all-users
</IfModule>
```

Starten Sie danach den Apache neu – entweder durch das Deaktivieren und erneute Aktivieren der Webfreigabe in den Freigabeeinstellungen der Systemeinstellungen oder durch das Ausführen von `sudo apachectl restart` an der Befehlszeile.

Um sicherzustellen, dass Ihre Sites für Bonjour erreichbar sind, starten Sie Safari und lassen sich die Lesezeichen anzeigen. In der Liste finden Sie einen Eintrag für Bonjour. Klicken Sie darauf, sollten Sie Ihren Computernamen und alle Anwender angezeigt bekommen, für die Sie die Bonjour-Webfreigabe aktiviert haben (siehe Abbildung 13.1).

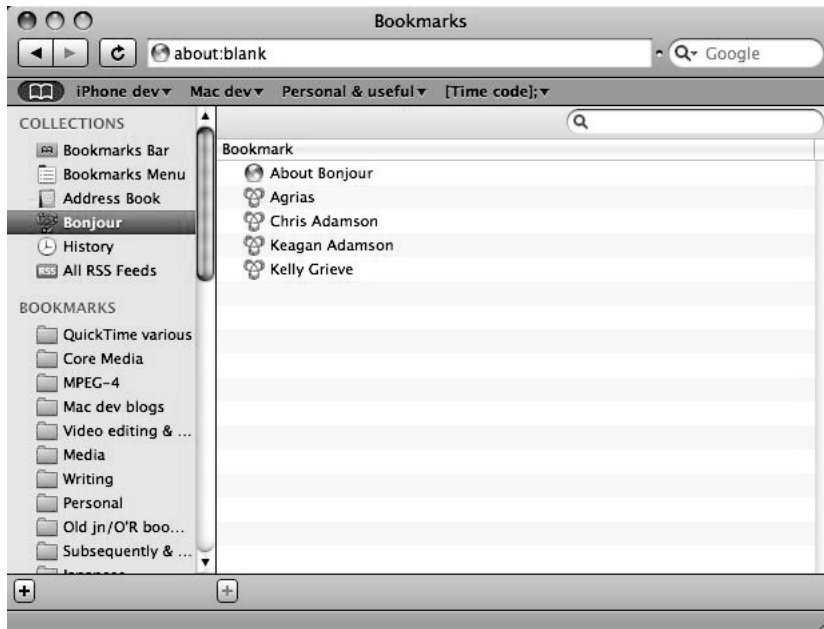


Abbildung 13.1: Bonjour-Apache-Webfreigaben mit Safari anzeigen

Können Sie Ihre Haupt-Site (also Ihren Computernamen, wie er in den Freigabeeinstellungen definiert ist – in diesem Beispiel „Truhe“) und alle aktiven Benutzer sehen, können Sie mit Ihrem iPhone ebenfalls per Bonjour surfen.

13.2 Bonjour Service Discovery

Um mit Bonjour Dienste zu finden, die einen interessieren, stellt das iPhone die OS Cocoa- und Core Foundation-APIs `NSNetServices` und `CFNetServices` bereit. Die Cocoa-Variante, auf die wir uns hier konzentrieren werden, ist sehr dynamisch und recht leicht zu verwenden, da Sie sich mit vielen der Low-Level-Netzwerkdetails von Bonjour nicht auseinandersetzen müssen. Um nach einem Dienst zu suchen, müssen Sie nur einen `NSNetServiceBrowser` erstellen (den wir einer Instanzvariablen `bonjourBrowser` zuweisen werden) und die zu durchsuchende Domain (bei einem leeren String suchen Sie einfach im LAN), einen

„Servicetyp“ sowie einen Delegate mitgeben. Der Delegate kümmert sich dann um die Callbacks aus dem Service Discovery Process.

NetworkIO/BonjourWebBrowser/Classes/RootViewController.m

```

1 -(void) startSearchingForWebServers {
2     bonjourBrowser = [[NSNetServiceBrowser alloc] init];
3     [bonjourBrowser setDelegate: self];
4     [bonjourBrowser searchForServicesOfType:@"_http._tcp" inDomain:@""];
5 }

```

Die einzige Zeile, die sich hier nicht selbst erklärt, ist die vierte mit dem String "_http._tcp", um die Dienste zu definieren, nach denen gesucht werden soll. Dieser String besteht aus zwei Teilen – einem zum Definieren der Semantik des Diensts (hier _http) und einem Transportmechanismus (_tcp), getrennt durch einen Punkt. Bekannte Diensttypen sind auf der Website <http://www.dns-sd.org/ServiceTypes.html> dokumentiert, die zum Zeitpunkt der Entstehung dieses Buchs mehr als 400 Bonjour-Typen enthält – unter anderem daap für das Digital Audio Access Protocol von iTunes, ipp für das Internet Printing Protocol, nfs für das alte Network File System und viele mehr. In dieser Liste finden Sie auch http mit der folgenden Beschreibung:

```

World Wide Web HTML-over-HTTP
Tim Berners-Lee <timbl at W3.org>
Protocol description: RFC 2616
Defined TXT keys: u=<username> p=<password> path=<path to document>
(see txtrecords.html#http)
NOTE: The meaning of this service type, though called just "http",
actually denotes something more precise than just "any data transported
using HTTP".
The DNS-SD service type "http" should only be used to advertise content
that:
* is served over HTTP,
* can be displayed by "typical" web browser client software, and
* is intended primarily to be viewed by a human user.
...

```

Mit anderen Worten, http ist der Servicetyp, den ein typischer Webserver Webbrowsern per Bonjour in der Regel anbieten sollte. Diesen Typ nutzen wir eben auch, um die vom Apache angebotenen Websites zu finden.

Rufen Sie die Methode `searchForServicesOfType:inDomain:` für den `NSNetServiceBrowser` auf, erhält Ihr Delegate Callbacks, die ihn über den Status der Suche informieren. Wenn Ihre Suche keinen Erfolg hat – vielleicht weil Sie einen ungültigen Diensttyp genutzt haben (Sie müssen vor den Typnamen und das Protokoll einen Unterstrich setzen und dürfen auch nicht den Punkt vergessen, der beides trennt) –, erhalten

Sie direkt den Fehler-Callback `netServiceBrowser:didNotSearch:`. Konnte die Suche jedoch erfolgreich beginnen, erhalten Sie den Callback `netServiceBrowserWillSearch:`, den Sie nutzen können, um dem Benutzer anzuzeigen, dass eine länger dauernde Suche begonnen hat (zum Beispiel durch einen Aktivitätsindikator).

Immer dann, wenn ein passender Dienst gefunden wurde, erhalten Sie den Callback `netServiceBrowser:didFindService:moreComing:`, dessen Argumente aus einem `NSNetService` mit einer Beschreibung des Diensts und einem `BOOL` bestehen, mit dem der `NetServiceBrowser` angibt, dass er auf weitere Dienste wartet. Normalerweise nutzt ein Delegate diesen Callback, um eine Liste verfügbarer Dienste aufzubauen. In der Beispielanwendung werden Dienste des Typs `NetworkIO/BonjourWeb Browser` genutzt, um eine `UITableView` mit einer Liste verfügbarer Dienste zu erstellen (siehe Abbildung 13.2). Zusammen mit den Homepages der Anwender in Ihrem LAN finden Sie eventuell noch andere Dienste – Scieszka ist in dieser Abbildung ein Laserdrucker, der über eine web-basierte Konfiguration verfügt und über Bonjour erreichbar ist.



Abbildung 13.2: Bonjour-Apache-Webfreigaben mit dem `NSNetServiceBrowser` anzeigen

Aber was passiert, wenn ein Dienst verschwindet? In diesem Fall erhalten Sie den Callback `netServiceBrowser:didRemoveService:moreComing:`, der wie die Version `didFindService` funktioniert, nur dass er das Verschwinden eines Diensts meldet. Im Beispielcode entfernen wir dann den angegebenen Dienst aus der Liste. Sie können das ausprobieren, indem Sie die App starten, in den Systemeinstellungen auf FREIGABEN gehen und die Webfreigabe abschalten. Dann verschwinden alle erkannten Webservices aus der Liste im Simulator oder auf dem iPhone.

Jetzt haben Sie einen Dienst, aber was machen Sie mit ihm? Nur den *Service Record* zu finden, wie wir es hier getan haben, reicht zum Anwenden des Diensts nicht unbedingt aus. Es gibt noch einen eigenen *Resolution*-Schritt, der alle Details zum Dienst liefert und Host, Port und andere Felder des `NSNetService` füllt, die beim ersten Erkennen des Diensts nicht initialisiert sind.

Der Beispielcode prüft in `viewDidLoad:`, ob der Dienst schon aus einem früheren Aufruf mit Leben gefüllt wurde. Sie erkennen das daran, dass `hostName` oder `port` des Diensts schon sinnvolle Werte enthalten. Wenn nicht, müssen Sie die *Resolution* aufrufen.

NetworkIO/BonjourWebBrowser/Classes/WebPageViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    // Resolution bei Bedarf starten, ansonsten Pfad holen und
    // die Seite anzeigen.
    // Unter http://developer.apple.com/qa/qa2004/qa1389.html
    // finden Sie Informationen, warum das notwendig ist.
    if ([netService hostName] != nil) {
        [self loadPageFromService];
    } else {
        [resolutionActivityIndicator startAnimating];
        [netService setDelegate: self];
        [netService resolveWithTimeout: RESOLUTION_TIMEOUT];
    }
}
```

Wie beim Service Discovery ist die Service Resolution der Cocoa-API ein sehr dynamischer Prozess. Die Resolution wird ausgelöst, indem man einen Delegate für den Dienst definiert und `resolveWithTimeout:` aufruft, das als Argument ein `NSTimeInterval` in Sekunden mitbekommt. Danach muss sich Ihr Code nur um die Callbacks kümmern, die während des Resolution-Prozesses gerufen werden.

Wenn die Resolution fehlschlägt, erhalten Sie den Callback `netService:didNotResolve:`. Im Erfolgsfall erhalten Sie `netServiceDidResolveAddress:`, danach enthält das `NSNetService`-Objekt alle Details,

die Sie zur Kommunikation mit einem Service benötigen: einen Host, einen Port und eine Sammlung von dienstspezifischen Name/Wert-Paaren im TXT-Record.

Ist die Resolution für den Dienst durchgeführt worden, können Sie seinen Host, den Port und weitere Metadaten auslesen. Der Beispielcode nutzt den entsprechenden Dienst, um eine URL aufzubauen, daraus einen NSURLRequest zu erstellen und ihn in eine UIWebView zu laden (die in Abschnitt 12.1, *Ein Browser in zehn Minuten mit der UIWebView*, auf Seite 259 vorgestellt wurde).

NetworkIO/BonjourWebBrowser/Classes/WebPageViewController.m

```

1  -(void) loadPageFromService {
2      // Pfad aus dem TXT-Record holen
3      NSDictionary *txtRecordDictionary =
4          [NSNetService dictionaryFromTXTRecordData:
5              [netService TXTRecordData]];
6      NSData *pathData =
7          (NSData*) [txtRecordDictionary objectForKey: @"path" ];
8      NSString *path = [[NSString alloc] initWithData: pathData
9                          encoding:NSUTF8StringEncoding];
10
11     // In http://www.dns-sd.org/txtrecords.html#http sind Regeln
12     // für das Ermitteln der URL aus den Dienstdaten beschrieben
13
14     // URL aus Host, Port und Pfad aufbauen
15     NSString *urlString = [[NSString alloc]
16                             initWithFormat: @"http://%d/%@" ,
17                             [netService hostName],
18                             [netService port],
19                             path];
20     NSURL *url = [[NSURL alloc] initWithString: urlString];
21     urlLabel.text = urlString;
22     self.title = [netService name];
23     NSURLRequest *request = [[NSURLRequest alloc] initWithURL:url];
24     [webView loadRequest: request];
25     // Aktivitätsindikator ausblenden, auch wenn WebView entweder
26     // entweder fertig ist oder fehlerhaft ist, durch das
27     // Delegate von UIWebView
28     [resolutionActivityIndicator stopAnimating];
29
30     [request release];
31     [url release];
32     [urlString release];
33     [path release];
34     // pathData und txtRecordDictionary werden nicht freigegeben,
35     // da sie nicht in "Besitz" genommen wurden
36 }

```

Es kann hilfreich sein, sich daran zu erinnern, was wir vom NSNet-Service benötigen, um die Webseite zu laden. Um eine URL zu erstellen, brauchen wir einen Host, einen Port und einen Pfad. Host und Port sind

einfach, da sie über die Methoden `hostName` und `port` ermittelt werden können. Der Pfad ist schon schwieriger. Für die Webseite des Rechners kann der Pfad ein leerer String sein, während er bei den Benutzerseiten so etwas wie `~cadamson` sein kann. Im Diensttyp `http` ist definiert, dass der TXT-Record den Pfad in einem Name/Wert-Paar speichert, wobei der Schlüssel dem String `path` entspricht. Unser Beispielcode kann also eine sinnvolle URL erstellen, indem er diese Elemente aus dem aufgelösten `NSNetService` zieht. In den Zeilen 3 bis 9 wird der Pfad als Byte-Block geholt und in einen `NSString` umgewandelt. Dann wird in den Zeilen 15 bis 20 eine URL erzeugt, aus der Sie einen `NSURLRequest` bauen und diesen dann an eine `UIWebView` übergeben können. Die gefundene und gerenderte Webseite sehen Sie in Abbildung 13.3.¹

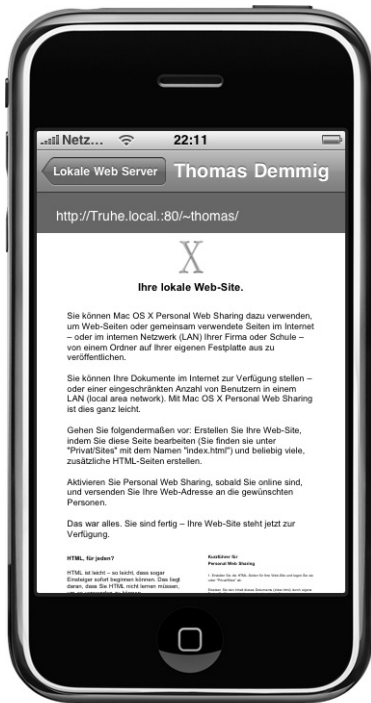


Abbildung 13.3: Anzeige einer Webseite, die über Bonjour gefunden wurde

Sobald Bonjour den Dienst gefunden und aufgelöst hat, ist die Arbeit erledigt. In diesem Beispiel wird die *Nutzung* des Diensts (das Laden der Webseite) vollständig durch die `UIWebView` abgehandelt. Bei anderen

¹ Wir haben `/Users/cadamson/Sites/index.html` das „(cadamson)“ hinzugefügt, um sicherzugehen, dass wir die richtige Seite gefunden haben. Ihre eigene Site mag anders aussehen.

Anwendungen richten Sie vielleicht Ihre eigene Verbindung zum Dienst ein. Ist er so definiert, dass er HTTP nutzt, könnten Sie das URL Loading System nutzen, das in Abschnitt 12.2, *Daten über das Netzwerk lesen*, auf Seite 265 beschrieben ist. Bei einem eigenen Protokoll müssten Sie eine Socket-Verbindung öffnen, die auf dem Host und Port aus dem aufgelösten Dienst und vielleicht noch auf anderen, protokoll-spezifischen Metadaten basiert, die sich im TXT-Record finden. Aber sobald Bonjour einen Dienst aufgelöst hat, arbeiten Sie auf jeden Fall wieder in der guten alten Netzwerk-I/O.

Dieses Beispiel hat sich nur um die Clientseite von Bonjour gekümmert – unter der Annahme, dass iPhones eher Clients als Server sind. Aber im iPhone OS findet sich der gesamte Bonjour-Stack, daher können Sie Ihr Gerät auch als Server nutzen und es über Bonjours Publication-APIs bei den Clients bekannt machen. Wenn Sie sich mit einem anderen Gerät mit iPhone OS verbinden wollen, können Sie aber auch auf das auf Bonjour basierende Game Kit Framework zurückgreifen, das im nächsten Abschnitt vorgestellt wird, um als Server oder Peer zu dienen.

Ihr Cocoa-Code, der die NSNetServices-APIs nutzt, kann ebenfalls unter Mac OS X kompiliert und ausgeführt werden, Sie können also den Code auf beiden Plattformen nutzen. Apple hat auch Bonjour-Implementierungen für Windows und Linux bereitgestellt und bietet sogar eine Java-API an – Sie haben also eine große Auswahl, wenn Sie die Plattformen nutzen wollen, um Dienste für Ihre iPhone-Apps bereitzustellen. Um mehr über Bonjour zu lernen, hat der Autor von Zeroconf und Fachlektor des Buchs, das Sie gerade lesen, das ultimative Werk darüber geschrieben: *Zero Configuration Networking, The Definitive Guide* (O'Reilly). Es behandelt Bonjour und seine APIs umfassend – von der obersten bis zur untersten Ebene des Standards.

13.3 Game Kit

Viele Anwendungen können von der Bluetooth-Ad-hoc-Verbindung des iPhone profitieren – stellen Sie sich zum Beispiel den Austausch elektronischer Visitenkarten oder Dokumente vor, selbst wenn Sie sich nicht in einem lokalen Wifi-Netz befinden. Und die Voice Chat-Features, die mit dem iPhone OS 3.0 hinzugekommen sind, funktionieren in jeder Art von Netzwerk, nicht nur über das im Nahbereich funktionierende Bluetooth, und können daher für viele Kommunikations-Apps genutzt werden. Denn trotz des Namens sind die Features des Game Kit für alle möglichen Arten von Anwendungen nützlich, nicht nur für Spiele.

Das Game Kit wird durch das optionale Framework `GameKit.framework` bereitgestellt, das aus nur drei Klassen (`GKPeerPickerController`, `GKSession` und `GKVoiceChatService`) und drei formalen Protokollen besteht (`GKPeerPickerControllerDelegate`, `GKSessionDelegate` und `GKVoiceChatClient`).

Es ist wichtig, sich darüber im Klaren zu sein, dass die vom Game Kit bereitgestellten Features zu Bluetooth und Voice Chat vollständig unabhängig voneinander sind. Der Voice Chat kann über jede Netzwerkverbindung durchgeführt werden – durchaus auch über eine vom Game Kit bereitgestellte Bluetooth-Verbindung, aber ebenso über eine drahtlose Verbindung von einer der auf niedrigerer Ebene arbeitenden Netzwerk-APIs, wie `CFNetwork` oder sogar `BSD Sockets`. Sie können sogar beide Arten von Netzwerken in einer einzelnen Anwendung unterstützen.

Sie brauchen eventuell zwei Geräte!

Während dieses Buch entsteht, gibt es in der iPhone-Familie eine unterschiedlich große Unterstützung der Netzwerkfähigkeiten des Game Kit. Das ursprüngliche iPhone-Bluetooth arbeitet nicht mit Game Kit zusammen, und die erste Generation des iPod touch besitzt nicht einmal Bluetooth. Damit bleiben das iPhone 3G, das 3GS und die zweite Generation des iPod touch die einzigen Geräte, die diese Klassen verwenden können.

Leider unterstützt der Simulator im iPhone 3.0 SDK kein Game Kit-Netzwerk über Bluetooth, selbst auf Macs mit Bluetooth. Stattdessen führt er das Game Kit-Protokoll über die Internetverbindung des Rechners aus, entweder über Wifi oder Ethernet. Somit brauchen Sie entweder zwei iPhones, die das Game Kit unterstützen, oder zwei Macs, auf denen der Simulator läuft, da man bei Netzwerkverbindungen nun einmal mehrere Teilnehmer benötigt.*

* Tatsächlich wurde der Beispielcode in diesem Kapitel im iPhone Lab auf der WWDC 2009 entwickelt, da das die einzige Möglichkeit war, zwei Game Kit-kompatible Geräte zu verbinden.

Obwohl die Netzwerk-Features des Game Kit Bonjour nutzen, um Dienste über Bluetooth anzubieten, zu finden und aufzulösen, verwenden sie nicht die in Abschnitt 13.1, *Ad-hoc-Netzwerk-Services mit Bonjour*, auf Seite 293 behandelte Bonjour-API. Stattdessen kapselt eine Klasse `GKSession` sowohl das Service Discovery als auch das Senden und Empfangen von

Daten über Bluetooth. Aber ein bisschen Bonjour ist trotzdem noch darin – wenn Sie Ihren Dienst anbieten, nutzen Sie immer noch die gleiche Art von Service-ID-String wie bei einer normalen Bonjour-Anwendung.

13.4 Ein Spiel mit Bluetooth-Unterstützung aufsetzen

Lassen Sie uns zunächst die Bluetooth-Netzwerkclassen betrachten. In den nächsten Abschnitten werden wir ein einfaches Peer-to-Peer-Spiel aufbauen, das sich über Bluetooth verbindet und kommuniziert. Das Spiel ist unsäglich einfach, aber selbst auf diesem Niveau werden wir einige der Fragen beantworten können, die beim Entwickeln eines Netzwerkspiels auftauchen.

Unser Spiel wird den Namen P2PTapWar erhalten, und das Konzept ist wirklich ganz einfach. Sobald zwei Spieler miteinander verbunden sind, muss jeder so schnell wie möglich auf eine Box auf dem Bildschirm tippen. Jedes Tippen zählt als ein Punkt und wird dem Gegenspieler automatisch mitgeteilt. Derjenige, der zuerst 50 Mal getippt hat, gewinnt.

Dieses Beispiel nutzt das „Single-View Application“-Template. Die einzige Klasse, mit der wir arbeiten werden, ist P2PTapWarViewController. Die View für diese Klasse, definiert in P2PTapWarViewController.xib, sehen Sie in Abbildung 13.4. Sie enthält eine Navigationsleiste für den Titel und einen Button FINDEN (wobei wir in dieser Anwendung gar nicht wirklich navigieren), ein paar Labels für die Ergebnisse und eine große graue Box, auf die unsere Anwender wie wild tippen werden.

Zunächst wechseln Sie also zu P2PTapWarViewController.h und erstellen IBOutlet-Eigenschaften für die UILabel playerTapCountLabel und opponentTapCountLabel sowie für den UIBarButtonItem startQuitButton. Sie müssen auch zwei IBActions definieren: handleStartQuitTapped und handleTapViewTapped. Danach können Sie P2PTapWarViewController.xib im Interface Builder öffnen und die Outlets verbinden. Bei den Actions müssen Sie den Connections Inspector für das Bar Button Item anzeigen ([⌘]-[2]) und dessen selector mit der Methode handleStartQuitTapped des File's Owner verbinden. Für die Tap-View gibt es zunächst keine Actions, da UIViews keine Benutzereingaben akzeptieren und daher auch keine Actions erzeugen können. Nutzen Sie den Identity Inspector ([⌘]-[4]), um die Klasse in UIControl zu ändern und dann das Event Touch Down mit dem File's Owner-Objekt handleTapViewTapped zu verbinden.

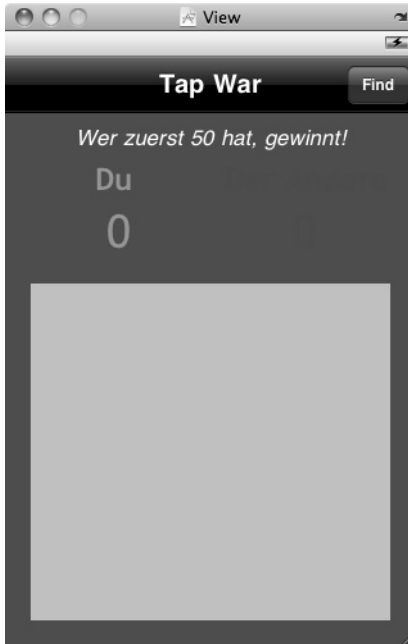


Abbildung 13.4: View für P2PTapwarViewController im Interface Builder

13.5 Erstellen eines Peer Pickers

Die Netzwerkklassen des Game Kit stellen zwei Dinge zur Verfügung: eine Abstraktion einer drahtlosen Bluetooth-Session und ein Peer Picker-UI, um andere Geräte über Bluetooth zu finden und sich mit ihnen zu verbinden. Da der Picker uns dabei helfen wird, einen Verbindungspartner (Peer) zu finden und eine Session einzurichten, wollen wir damit beginnen und die zur Session gehörenden Details unterwegs beschreiben.

Der Peer Picker wird durch die Klasse GKPeerPickerController kontrolliert und ist ziemlich übersichtlich: Es gibt Instanzmethoden, um den Picker per `show` zu zeigen und per `dismiss` auszublenden, eine Eigenschaft, die definiert, was für Verbindungen angeboten werden sollen (Bluetooth oder andere) und einen `delegate`. Um den Picker anzuzeigen, wenn der Button FINDEN angetippt wird, müssen wir nur Folgendes machen:

```
NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m
```

```
-(IBAction) handleStartQuitTapped {
    if (! opponentID) {
        actingAsHost = YES;
        GKPeerPickerController *peerPickerController =
            [[GKPeerPickerController alloc] init];
```

```

        peerPickerController.delegate = self;
        peerPickerController.connectionTypesMask =
            GKPeerPickerControllerConnectionTypeNearby;
        [peerPickerController show];
    }
}

```

Diese Methode zeigt den Picker nur dann, wenn `opponentID`, eine Instanzvariable vom Typ `NSString`, die Sie in der Header-Datei definieren müssen, nicht `nil` ist. Peers werden im Game Kit durch `peerID`-Strings identifiziert, und da wir für dieses Spiel nur einen benötigen, können wir die An- oder Abwesenheit dieser Instanzvariablen als Kennzeichen dafür nehmen, ob wir schon einen Gegner gefunden haben.

Gibt es noch keinen Gegner, setzen wir eine Flag-Variable `actingAsHost` (die gleich noch erklärt wird) auf einen temporären Wert und allokieren dann den `GKPeerPickerController`. Der Picker braucht ein `delegate`, um den Verbindungsstatus bekannt geben zu können. Das kann dieser View-Controller selbst sein, wenn wir in der Header-Datei deklariert haben, dass wir das Protokoll `GKPeerPickerControllerDelegate` implementieren. Wir werden dessen verschiedene Methoden gleich noch implementieren.

Als Nächstes legen wir über die Eigenschaft `connectionTypesMask` fest, welche Art von Verbindungen der Peer Picker anbieten soll. Dabei wird ein Bit-Feld mit möglichen Werten übergeben. Game Kit definiert zwei: `GKPeerPickerControllerConnectionTypeNearby` steht für eine Bluetooth-Verbindung, während `GKPeerPickerControllerConnectionTypeOnline` alle anderen Arten von drahtlosen Verbindungen repräsentiert. Wenn Sie beide Werte mit dem `|`-Operator kombinieren, erhält der Anwender als Erstes eine Auswahlmöglichkeit zwischen beiden Arten von Verbindungen. Allerdings stellt das Game Kit nur eine Bluetooth-API bereit. Entscheidet sich der Anwender also für „Online“, wird der Picker wieder ausgeblendet, und Ihr Delegate erhält einen Callback für `peerPickerController:didSelectConnectionType:`. Das wäre für Sie dann das Signal, eine eigene Netzwerkverbindung einzurichten – sehr wahrscheinlich eine über die Low-Level-Netzwerk-APIs wie `CFNetwork` oder `BSD Sockets`. In diesem Beispiel bieten wir nur eine Bluetooth-Verbindung an, so dass der Anwender direkt zu einem GUI geleitet wird, in dem die Bluetooth-Peers aufgeführt sind. Wenn schließlich alle Objekte eingerichtet sind, rufen wir die Methode `show` auf, um den Picker anzuzeigen (siehe Abbildung 13.5 mit einem einzelnen erkannten Peer).



Abbildung 13.5: GKPeerPickerController mit Bluetooth-Peers

13.6 Einen Peer Picker-Delegate bereitstellen

Ein Großteil der Interaktion der Anwendung mit einem Peer Picker findet sich in Ihrer Implementierung des Protokolls `GKPeerPickerControllerDelegate` wieder. Der Delegate ist dafür verantwortlich, dem Picker ein Session-Objekt bereitzustellen und zu reagieren, wenn sich ein Peer verbindet. Er kann auch darauf reagieren, wenn der Picker abgebrochen wird, ohne dass ein Peer gewählt wurde. Nutzen Sie `GKPeerPickerControllerConnectionTypeOnline`, um über den Picker auch Nicht-Bluetooth-Verbindungen anzubieten, werden Sie über diese Wahl durch einen Callback von `peerPickerController:didSelectConnectionType:` informiert. Dann müssen Sie diese Verbindung selbst einrichten.

Bei unserer reinen Bluetooth-Anwendung muss unser Delegate vor allem die Methode `peerPickerController:sessionForConnectionType:` implementieren. Diese wird aufgerufen, wenn der Picker angezeigt werden soll und ein `GKSession`-Objekt benötigt. Das Session-Objekt implementiert dabei die Bluetooth-Verbindung zwischen den verschiedenen Geräten. Wir brauchen eine Session, um uns mit einem Gegner zu verbinden, also deklarieren wir `gkSession` als Instanzvariable in `P2PTapWarViewController.h` und instantiieren sie erst bei Bedarf – wenn der Picker danach fragt.

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```

-(GKSession*) peerPickerController: (GKPeerPickerController*) controller
    sessionForConnectionType: (GKPeerPickerConnectionType) type {
    if (!gkSession) {
        gkSession = [[GKSession alloc]
            initWithSessionID:AMIPHD_P2P_SESSION_ID
            displayName:nil
            sessionMode:GKSessionModePeer];
        gkSession.delegate = self;
    }
    return gkSession;
}

```

`peerPickerController:sessionForConnectionType:` übergibt den Picker, der die Session anfordert, zusammen mit einer der Typkonstanten (wobei wir davon ausgehen können, dass es `GKPeerPickerConnectionType-Nearby` ist, da wir den Picker so eingerichtet haben, dass er nur diese nutzt). Das `GKSession`-Objekt hat einen Initializer mit drei Parametern, von denen jeder sorgfältig bedacht werden muss:

- `initWithSessionID:` Die `sessionID` ist ein `NSString`, der das Protokoll, das Ihre Anwendung zur Kommunikation nutzt, eindeutig identifiziert. Durch die Verwendung eindeutiger IDs kann eine Anwendung vermeiden, sich mit einem Gerät zu verbinden, das das Game Kit vielleicht für ein ganz anderes (und inkompatibles) Kommunikationsprotokoll nutzt. Apple empfiehlt, eine Bonjour-Service-ID zu nutzen – eine von den maximal 14 Zeichen langen IDs, die Sie in Abschnitt 13.2, *Bonjour Service Discovery*, auf Seite 295 kennengelernt haben. Wenn Sie Ihre eigenen Kommunikationsprotokolle definieren wollen, wie auch wir das hier tun, sollten Sie eine Bonjour-Service-ID auswählen und sie bei der Bonjour-Service-Registry unter <http://dns-sd.org/> registrieren. Für dieses Beispiel haben wir die Service-ID `amiphd-p2p` registriert² und sie per `#dene` in der Header-Datei als `AMIPHD_P2P_SESSION_ID` hinterlegt. Wenn Sie dagegen für dieses Argument `nil` angeben, wird aus der Kennung Ihres Anwendungs-Bundles eine Service-ID generiert.
- `displayName:` Dies ist der Name, der den Peers präsentiert wird. Dabei sollte es sich um einen anzeigbaren, von Menschen lesbaren String handeln. Wenn Sie also ein Spiel schreiben würden, bei dem der Anwender einen Namen wählen könnte, würden Sie diesen hier nutzen. Übergeben Sie für dieses Argument `nil` (wie wir es getan haben), wird der Name des Geräts verwendet.

² Dieser String basiert auf dem internen Codenamen dieses Buchs und hat keine weitere Bedeutung.

- `sessionMode`: Es gibt drei verschiedene Session-Modi, die Sie deklarieren können. Ein Server nutzt `GKSessionModeServer`, um sich selbst im Netzwerk zu präsentieren, aber nicht nach anderen Diensten zu suchen. Clients verwenden `GKSessionModeClient`, um zu zeigen, dass ihre Session nach Diensten sucht, aber keine eigenen bereitstellt. Und ein Peer, der durch `GKSessionModePeer` definiert wird, agiert gleichzeitig als Server und Client – er bietet selbst einen Dienst an, sucht aber auch gleichzeitig nach anderen. Der Peer ist bei der Netzwerk-API des Game Kit der häufigste Anwendungsfall und genau der, den auch wir hier nutzen werden.

Beim Einrichten der `GKSession` stellen wir ebenfalls einen `delegate` bereit, der asynchron durch Events aus der Session versorgt wird, wie zum Beispiel das Verbinden mit und Trennen von Peers. Da wir wollen, dass der View-Controller als Delegate dient, müssen wir der Liste der implementierten Protokolle in der `@interface`-Deklaration der Header-Datei auch noch `GKSessionDelegate` hinzufügen.

Haben wir einmal eine Session bereitgestellt – möglichst eine mit einem sinnvollen Delegate –, gibt es eigentlich nichts mehr, was der Picker-Delegate tun muss. Der Picker ist nur für das Einrichten einer Verbindung verantwortlich, während sämtliche Netzwerkaktivitäten in der `GKSession` und deren Delegate stattfindet. Tatsächlich muss der Picker-Delegate in `peerPickerController:didConnectPeer:toSession:` nur noch `[picker dismiss];` aufrufen, um das UI abzubauen. Der Picker baut sich selbst ab, wenn er abgebrochen wird, auch wenn Sie `peerPickerControllerDidCancel:` implementieren können, um über dieses Event informiert zu werden.

13.7 Spiellogik im Netz

Als Nächstes werden wir jetzt mit der `GKSession` arbeiten, um Daten an einen Peer zu senden und mit dem `GKSessionDelegate` das Verbinden, Trennen und das Empfangen von Daten zu handeln. Aber vorher müssen wir uns eine wichtige Frage stellen: Wie werden wir die Verbindung nutzen, um die Spieldaten zu übermitteln?

Nun, zunächst ist die Antwort sehr einfach: Die Klasse `GKSession` besitzt zwei Methoden für das Verschicken von Daten – `sendData:toPeers:withDataMode:error:` und `sendDataToAllPeers:withDataMode:error:`. Aber *was* werden wir versenden?

Beim Verwalten des Spielstatus über eine Netzwerkverbindung ergibt sich eine Reihe interessanter Probleme. Wenn jeder Peer ein Modell des Spielstatus verwaltet – wie halten Sie alle synchron? Wie viel Zeit und Bandbreite wird dafür benötigt? Was passiert, wenn Sie es nicht tun?

Unser Spiel scheint eine einfache Anforderung zu haben. Es soll sich merken, wie oft jeder Spieler auf seine Box getippt hat. Aber dahinter steckt mehr. Wir müssen auch den Start des Spiels koordinieren – Sie können nicht einfach einen Spieler einen Gegner wählen und mit dem Tippen beginnen lassen, während der Peer sich immer noch überlegt, ob er die Verbindung überhaupt akzeptiert. Und Sie müssen das Ende des Spiels koordinieren – gewinnt ein Spieler, muss dem anderen gesagt werden, dass er nicht mehr weitertippen muss.

All das hat Auswirkungen auf das Netzwerkprotokoll. Wir verschicken unterschiedliche Nachrichtentypen, und das Protokoll sollte stabil genug sein, damit umgehen zu können. Es ist vielleicht hilfreich, ein Protokoll zu entwerfen, das zukünftig auch durch neue Nachrichtentypen erweitert werden kann.

Wir haben einen flexiblen, wenn auch ineffizienten Ansatz für unser einfaches Spiel gewählt. Jede Nachricht besteht aus einer Reihe von Schlüssel/Wert-Paaren, die durch einen `NSKeyedArchiver` in ein `NSData`-Objekt verpackt werden (siehe auch Abschnitt 8.8, *Eigenschaftslisten und NSCoder*, auf Seite 178). Wir werden in der Header-Datei eine Reihe von Nachrichtenschlüsseln definieren, und die Peers, die Nachrichten empfangen, werden nach diesen Schlüsseln Ausschau halten.

`NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.h`

```
#define START_GAME_KEY @"startgame"
#define END_GAME_KEY @"endgame"
#define TAP_COUNT_KEY @"taps"
```

In dieser einfachen Version wird eine Seite `START_GAME_KEY` in einer Nachricht nutzen, um das Spiel zu starten. Immer dann, wenn einer der Peers ein Tap-Event erhält, schickt er `TAP_COUNT_KEY` mit dem aktuellen Tippzähler als Wert. Ist die Anzahl an Tipps erreicht, die man zum Gewinnen benötigt, kann auch noch `END_GAME_KEY` mitgeschickt werden, um die Gegenseite darüber zu informieren, dass das Spiel zu Ende ist.

Wenn Sie sich dazu entschieden hätten, einen Timer statt einer maximalen Anzahl an Tipps zu verwenden, könnten Sie einen weiteren Nachrichtenschlüssel nutzen, um Zeit-Updates zu verschicken. In solch einem Fall würden Sie einen Peer auswählen, dessen versendete Zeitwerte von den anderen als „offizielle“ Zeit angesehen werden. Würde

jeder Peer seinen eigenen Timer nutzen, könnte es durchaus passieren, dass die einzelnen Uhren gerade so weit auseinanderliegen, dass es ein Problem mit der Konsistenz des Spielstatus gibt.

Wir müssen uns Gedanken über den *Modus* machen, den das Game Kit nutzt. Nachrichten können „zuverlässig“ oder „unzuverlässig“ versendet werden. Unzuverlässige Nachrichten werden einmal verschickt, ohne dass darauf geachtet wird, ob dies erfolgreich war, und auch ohne weitere Versuche. Zuverlässige Nachrichten werden wiederholt versendet, bis sie den Empfänger erfolgreich erreicht haben. Erfahrene Netzwerkentwickler werden dies als den Unterschied zwischen UDP und TCP erkennen.

Da wir den Status über das Netzwerk verteilt halten wollen, denken Sie vielleicht, dass wir jedes Tippen mit einer zuverlässigen Nachricht versenden sollten. Aber dieses Spiel verzeiht durchaus Nachrichtenverluste – im Interesse einer besseren Performance. Statt eine Nachricht zu schicken, die nur „Gegner hat getippt“ besagt, können wir eine Nachricht mit dem aktuellen „Tippzähler“ versenden. Kommt diese nicht erfolgreich an, macht das normalerweise nicht viel aus, da wir einen Sekundenbruchteil später beim nächsten Tippen eine weitere Nachricht verschicken, die auch wieder alle notwendigen Daten enthält (also den Tippzähler).

Toleriert man in komplexeren Spielen Verluste bei der Netzwerkübertragung, muss man auf *Dead Reckoning* zurückgreifen – das Abschätzen des Spielstatus aus dem aktuellen Status. Modelliert Ihre App sich bewegende Objekte und geht ein Update von einem Netzwerk-Peer verloren, können Sie die Objekte anhand der Spielphysik weiterbewegen lassen und eine Korrektur vornehmen, wenn Sie wieder neue Daten erhalten.

Allgemein sollten Sie versuchen, so viel wie möglich über „unzuverlässige“ Kanäle zu übertragen, und die zuverlässige Übertragung nur für Dinge nutzen, die wirklich wichtig sind. So könnte man zum Beispiel eine verlorene Tippzähler-Nachricht verkraften, aber wir wollen keine Nachrichten verpassen, die den Start oder das Ende eines Spiels anzeigen. Diese sollten daher zuverlässig übermittelt werden.

13.8 Mit der GKSession kommunizieren

Nachdem wir uns nun eine Strategie für das Verschicken von Daten per Bluetooth überlegt haben, können wir unser Protokoll mit den Kommunikationsmethoden des Game Kit implementieren. Wir wollen Statusänderungen von Peers verarbeiten (wenn sich zum Beispiel der Gegner

verbindet oder wieder abmeldet), Daten an den Gegner schicken und welche von ihm empfangen.

Daten senden

Wir müssen an einen Peer immer dann Daten schicken, wenn die Tap-View angetippt wird. Also werden wir unseren Event-Handler `handleTapViewTapped` implementieren:

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(IBAction) handleTapViewTapped {
    playerTapCount++;
    [self updateTapCountLabels];
    // Haben wir gerade gewonnen?
    BOOL playerWins = playerTapCount >= WINNING_TAP_COUNT;
    // Tippzähler an den Peer senden
    NSMutableData *message = [[NSMutableData alloc] init];
    NSKeyedArchiver *archiver =
        [[NSKeyedArchiver alloc] initWithMutableData:message];
    [archiver encodeInt:playerTapCount forKey: TAP_COUNT_KEY];
    if (playerWins)
        [archiver encodeBool:YES forKey:END_GAME_KEY];
    [archiver finishEncoding];
    GKSendDataMode sendMode =
        playerWins ? GKSendDataReliable : GKSendDataUnreliable;
    [gkSession sendDataToAllPeers: message withDataMode:
                                     sendMode error:NULL];

    [archiver release];
    [message release];
    // Spiel auch lokal beenden
    if (playerWins)
        [self endGame];
}
```

Da werden ganz offensichtlich ein paar interne Spielmethoden aufgerufen, die wir noch gar nicht geschrieben haben, so zum Beispiel das lokale Aktualisieren des Zählers mit `updateTapCountLabels`. Der kritische Teil der Methode beginnt allerdings erst dahinter: Es wird ein `NSKeyedArchiver` erstellt, um ein `NSMutableData` mit Schlüssel/Wert-Paaren für unsere Nachricht zu bestücken. Dem wird der aktualisierte Tippzähler hinzugefügt, und wenn er dem zum Gewinnen notwendigen Wert entspricht, auch der `END_GAME_KEY`. Wenn der Schlüssel `END_GAME_KEY` enthalten ist, rufen wir dann die Methode `sendDataToAllPeers:withDataMode:error:` von `GKSession` im zuverlässigen Modus auf. Ohne diesen Schlüssel würden wir den unzuverlässigen Modus nutzen. Schließlich kommt noch ein bisschen lokale Logik, um das Spiel mit den noch zu schreibenden Methoden `endHostedGame` und `endJoinedGame` bei Bedarf lokal zu beenden.

Das war das Versenden, aber es gibt noch einiges, um das wir uns kümmern müssen – so das Empfangen von Nachrichten und das Starten des Spiels. Diese Aufgaben werden nicht durch unsere Anwendung ausgelöst, sondern durch die Delegate-Methoden durchgeführt, die asynchrone Events der Session verarbeiten.

Statusänderungen

Lassen Sie uns mit `session:didReceiveConnectionRequestFromPeer:` beginnen, das aufgerufen wird, wenn ein Teilnehmer eine Verbindungsanfrage erhält. Ist die `GKSession` über den Peer Picker verbunden, erhält diesen Callback nur der Spieler, der gefragt wird, ob er am Spiel teilnehmen will, nicht derjenige, der den Gegner im Picker ausgewählt hat. Damit haben wir die Chance, den anfragenden Spieler zum *Host* zu machen – eine Entscheidung, durch die nur einer der Teilnehmer das Spiel auch starten kann.

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(void)session:(GKSession *)session
    didReceiveConnectionRequestFromPeer:(NSString *)peerID {
    actingAsHost = NO; }
```

Wenn dieser Spieler die Anfrage akzeptiert, erhalten die Delegates auf beiden Seiten einen Callback für `session:peer:didChangeState:` mit dem Status `GKPeerStateConnected`. Es gibt noch eine Reihe weiterer Statuswerte, aber jetzt wollen wir nur die Logik zum Starten des Spiels implementieren, wenn sich ein Peer verbindet:

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
- (void)session:(GKSession *)session peer:(NSString *)peerID
    didChangeState:(GKPeerConnectionState)state {
    switch (state)
    {
        case GKPeerStateConnected:
            [session setDataReceiveHandler: self withContext: nil];
            opponentID = peerID;
            actingAsHost ? [self hostGame] : [self joinGame];
            break;
    }
}
```

Sobald eine Verbindung empfangen wird, ruft diese Methode als Erstes für die `GKSession` die Methode `setDataReceiveHandler:withContext:` auf. Das ist entscheidend, denn damit erhält die Session ein Objekt, das dazu in der Lage ist, Daten über das Netzwerk zu empfangen. Das Handler-Objekt ist nicht durch ein formales Protokoll festgelegt, stattdessen muss es eine Callback-Methode mit der folgenden Signatur implementieren:

```
- (void) receiveData:(NSData *)data fromPeer:(NSString *)peer
    inSession: (GKSession *)session context:(void *)context;
```

setDataReceiveHandler:context: erwartet ebenfalls einen context, der an die Methode receiveData:fromPeer:inSession:context übergeben wird. Als **void*** kann diese Kontextreferenz eine beliebige Art von Zeiger sein, einschließlich aller Objective-C-Objekte. Wir brauchen für dieses Spiel kein Kontextobjekt, daher setzen wir es auf **nil**.

Als Nächstes merkt sich unser Statusänderungs-Handler die Peer-ID des Gegners in der Instanzvariablen **opponentID** und startet dann entweder ein Spiel oder tritt ihm bei – abhängig davon, ob dieser Spieler der Host ist. Beide Methoden müssen den lokalen Status und die GUIs aktualisieren, aber nur der Host muss die Nachricht „Spiel beginnen“ über die Verbindung schicken. Dies sind die Methoden **hostGame** und **joinGame**, zusammen mit den Hilfsmethoden **initGame** und **updateTapCountLabels**, die von beiden aufgerufen werden:

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
- (void) updateTapCountLabels {
    playerTapCountLabel.text =
        [NSString stringWithFormat:@"%d" , playerTapCount];
    opponentTapCountLabel.text =
        [NSString stringWithFormat:@"%d" , opponentTapCount]; }

-(void) initGame {
    playerTapCount = 0;
    opponentTapCount = 0;
}

-(void) hostGame {
    [self initGame];
    NSMutableData *message = [[NSMutableData alloc] init];
    NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
        initWithWritingWithMutableData:message];
    [archiver encodeBool:YES forKey:START_GAME_KEY];
    [archiver finishEncoding];
    NSError *sendErr = nil;
    [gkSession sendDataToAllPeers: message
        withDataMode:GKSendDataReliable error:&sendErr];
    if (sendErr)
        NSLog(@"send greeting failed: %@", sendErr);
    // Zustand des startQuitButton ändern
    startQuitButton.title = @"Beenden" ;
    [message release];
    [archiver release];
    [self updateTapCountLabels];
}

-(void) joinGame {
    [self initGame];
    startQuitButton.title = @"Beenden" ;
    [self updateTapCountLabels];
}
```

In `startGame` können Sie wieder sehen, wie wir einen `NSKeyedArchiver` nutzen, um eine Nachricht in einem `NSMutableData` aufzubauen, das als Subklasse von `NSData` mit der Methode `sendDataToAllPeers:withDataMode:error:` der `GKSession` genutzt werden kann.

Daten empfangen

Nachdem wir uns nun um die Statusänderungen der Gegner gekümmert haben,³ verbleibt die Aufgabe, die Daten zu verarbeiten, die wir von einem Peer erhalten. Wir haben die ausgehenden Daten mit einem `NSKeyedArchiver` erzeugt, um sie also beim Empfänger wieder zu entpacken, müssen wir einen `NSKeyedUnarchiver` nutzen.

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(void) receiveData: (NSData*) data fromPeer: (NSString*) peerID
    inSession: (GKSession*) session context: (void*) context {
    NSKeyedUnarchiver *unarchiver =
        [[NSKeyedUnarchiver alloc] initWithReadingWithData:data];
    if ([unarchiver containsValueForKey:TAP_COUNT_KEY]) {
        opponentTapCount = [unarchiver decodeIntForKey:TAP_COUNT_KEY];
        [self updateTapCountLabels];
    }
    if ([unarchiver containsValueForKey:END_GAME_KEY]) {
        [self endGame];
    }
    if ([unarchiver containsValueForKey:START_GAME_KEY]) {
        [self joinGame];
    }
    [unarchiver release];
}
```

Wie Sie sehen, erhält der Unarchiver die Daten, die von der `GKSession` empfangen wurden, und sucht nach den bekannten Schlüsseln. Wenn er `TAP_COUNT_KEY` findet, packt er den Wert aus und aktualisiert die Trefferanzeige, während er bei `END_GAME_KEY` eine Methode zum Beenden des Spiels aufruft, den lokalen Status aufräumt, alle Peers von der `GKSession` trennt und eine Hilfsmethode zum Anzeigen eines „Gewonnen“- oder „Verloren“-Pop-ups nutzt (siehe Abbildung 13.6).

NetworkIO/P2PTapWar/Classes/P2PTapWarViewController.m

```
-(void) showEndGameAlert {
    BOOL playerWins = playerTapCount > opponentTapCount;
    UIAlertView *endGameAlert = [[UIAlertView alloc]
        initWithTitle: playerWins ? @"Gewonnen!" : @"Verloren!"
        message: playerWins ? @"Ihre Finger sind flink!" :
            @"Ihre Finger könnten schneller werden!"
```

³ Eine robustere App würde auch noch einige der anderen Statusänderungen behandeln, wie zum Beispiel ein elegantes Beenden, wenn sich ein Peer abgemeldet hat.

```

        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
[endGameAlert show];
[endGameAlert release];
}
-(void) endGame {
    opponentID = nil;
    startQuitButton.title = @"Finden" ;
    [gkSession disconnectFromAllPeers];
    [self showEndGameAlert];
}

```

Das ist alles, was Sie brauchen, um dieses Peer-to-Peer-Bluetooth-Spiel zu bauen und auszuliefern. Wir fassen zusammen: Mit einem GKPeerPickerController kann der Anwender einen Gegner auswählen. Der Picker hat eine GKSession erhalten, um das lokale „Bluetooth-Netz“ zu nutzen. Durch Delegate-Methoden konnte die Session asynchrone Events (wie das Verbinden mit einem Peer) übermitteln. Bei Auftreten des Events GKPeerStateConnected haben wir das Spiel gestartet, wobei die Session zum Versenden von Daten an den Peer genutzt wurde. Zudem hat die Session einen Handler zum Empfangen von Daten erhalten, der die eintreffenden Nachrichten des Peers verarbeitet.



Abbildung 13.6: Hinweis auf das Spielende für die Peers in einem Netzwerkspiel

13.9 Voice Chat

Neben den lokalen Verbindungen per Bluetooth gibt es beim Game Kit noch ein weiteres Feature – den Peer-to-Peer-Chat. Wie schon erwähnt, sind diese beiden Features komplett unabhängig voneinander: Sie können den Voice Chat mit dem Bluetooth-Netzwerk nutzen, das wir im vorigen Abschnitt aufgesetzt haben, aber auch eine Wifi-Verbindung verwenden. Verschaffen wir uns also einen Überblick über die Funktionsweise des Voice Chat.

Der Voice Chat nutzt nur zwei Klassen. `GKVoiceChatService` steht für einen einzelnen, gemeinsam genutzten Zugriffspunkt für die Voice Chat-Funktionalität. Über die Klassenmethode `defaultVoiceChatService` erhalten Sie eine Referenz auf dieses Singleton, durch die Sie dann mit `startVoiceChatWithParticipantID:error` starten können. Der String `participantID` identifiziert dabei eindeutig das Gegenüber, mit dem Sie chatten wollen.

Sie fragen sich vielleicht, wer diesen String auf einen echten entfernten Peer abbildet, mit dem man chatten möchte. Die Antwort ist – Sie. Sie müssen den Service durch ein Objekt bereitstellen, das das Protokoll `GKVoiceChatClient` implementiert. Dieses Objekt stellt die Netzwerkverbindung bereit, die vom `GKVoiceChatService` genutzt wird, und es bestimmt, wie eine Nachricht an eine bestimmte `participantID` tatsächlich geroutet und ausgeliefert wird. Genauso ruft es beim Empfangen von Sprachdaten über seine Netzwerkverbindung die Methode `receivedData:fromParticipantID:` oder `receivedRealTimeData:fromParticipantID:` des Service, um die Sprachdaten an den Voice Chat Service auszuliefern, so dass die empfangenen Daten dann dekodiert und über den Lautsprecher oder die Kopfhörer ausgegeben werden können.

Wollten Sie auf P2PTapWar noch einen Voice Chat aufsetzen, könnten Sie das recht einfach erreichen. Indem Sie die `participantID` des Voice Chat als Äquivalent zur `peerID` des Bluetooth-Netzwerks betrachten, könnte Ihr `GKVoiceChatClient` die Methode `voiceChatService:sendData:toParticipantID:` implementieren, indem er `sendData:toPeers:withDataMode:error:` der `GKSession` aufruft. Es würde eventuell nur schwieriger sein, das Ihr Protokoll zwischen Sprachdaten und anderen Spieldaten unterscheiden müsste, aber man könnte dafür einfach einen neuen Schlüssel einführen (zum Beispiel `VOICE_CHAT_DATA`). Beim Empfänger würde das `NSData`-Objekt, das die Sprachdaten enthält, vom Daten-Handler ausgepackt und an den `GKVoiceChatService` des Geräts geschickt werden.

Ein paar Details in Bezug auf den Voice Chat Service haben mehr mit der Audioverarbeitung zu tun als mit der Netzwerkkommunikation: Der `GKVoiceChatService` besitzt eine `Level Metering-API`, so dass Sie Ihren Anwendern die Lautstärke-Level der lokalen und der entfernten Lautsprecher anzeigen können. Die Methoden dafür entsprechen weitgehend denen des AV Foundation Framework (siehe Kapitel 16, *Audiodaten abspielen und aufzeichnen*, Seite 355) und den Music Player APIs im Media Player Framework (siehe Kapitel 15, *Zugriff auf die iPod-Bibliothek* auf Seite 329). Haben Sie einmal damit gearbeitet, sollte das Hinzufügen einer Lautstärkeanzeige für Ihren Voice Chat kein Problem sein.

Ein wichtiger Aspekt, den Sie beim Voice Chat berücksichtigen sollten, ist, dass alle Medien-Apps beim iPhone OS ihre Anforderungen und Wünsche im Voraus an das Audiosystem übermitteln sollten. Die APIs dazu werden in Abschnitt 16.7, *Audio-Sessions*, auf Seite 382 behandelt, und bei der Unterstützung des Voice Chat gibt es zwei wichtige Überlegungen. Zum einen müssen Sie deklarieren, dass Ihre Anwendung Audiodaten simultan abspielen und aufnehmen will, damit das System „Hintergrundgeräusche“ (zum Beispiel von der iPod-Anwendung) unterdrückt und die Audioeingabe für Ihre Anwendung reserviert. Dazu teilen Sie der Audio-Session mit, dass Ihre Anwendung die Kategorie `AVAudioSessionCategoryPlayAndRecord` nutzen will. Zudem werden Sie die Audio-Session verwenden wollen, um zu überprüfen, ob die Audioeingabe überhaupt nutzbar ist – iPhones haben zwar immer ein Mikrofon an Bord, neuere iPod touches aber können nur dann sinnvoll etwas aufnehmen, wenn ein passendes Mikro angeschlossen ist. Beide Aufgaben werden in den späteren Kapiteln über Medien beschrieben.

Kapitel 14

Videos abspielen

Das iPhone ist ein tolles Gerät, um Medien abzuspielen. Sein großer, heller, hochauflösender Bildschirm lässt sich erstaunlich gut zum Betrachten von Filmen nutzen, und die Audiounterstützung ist sehr umfassend. Das iPhone spielt nicht nur die verbreitetsten Audioformate ab, es bietet auch eine umfassende Unterstützung für die Arbeit mit Audiodaten auf unterschiedlichsten Ebenen an – vom einfachen Abspielen bis hin zur Low-Level-Verarbeitung von Audiorohdaten.

Wir werden unseren Überblick über das Medien-Handling auf dem iPhone OS mit Videos beginnen. Das ist zugegeben ein wenig untypisch, da der Umgang mit Audiodaten normalerweise einfacher ist als der mit Videos. Das ist der Grund dafür, dass Sie bei neuen Plattformen meist erst die Audiofähigkeiten kennenlernen, bevor Sie sich in die Videoverarbeitung stürzen. Aber beim iPhone ist die Video-Playback-API ausgesprochen einfach, und es stellt sich heraus, dass das Abspielen eines Videos so ziemlich das leichteste ist, was Sie mit dem SDK anstellen können.

14.1 Videos abspielen mit dem `MPMoviePlayerController`

Die Video-API des iPhone SDK ist sehr einfach, aber leider auch sehr eingeschränkt. Sie können nichts anderes mit einem Video anstellen, als es abzuspielen. Auf einem iPhone 3GS können Sie zudem Videos in der Bildbibliothek aufzeichnen (siehe Abschnitt 20.4, *Videos aufnehmen*, auf Seite 437). Aber ansonsten können Sie mit Videos nicht wirklich viel machen.

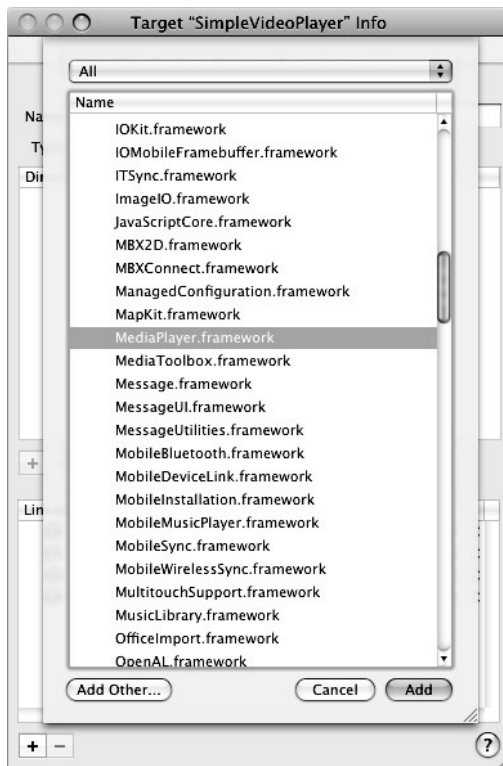
Die Video Playback-API besteht im Prinzip nur aus der Klasse `MPMoviePlayerController`, einem Objekt, mit dem Sie folgendes machen können:

- Ein Video von einer URL laden.
- Das Video per Code abspielen und pausieren.

- Eigenschaften für den Videohintergrund, die Benutzersteuerung und den Skalierungsmodus anpassen (ob der Film an die Größe des Bildschirms angepasst und dabei eventuell auch beschnitten wird).
- Nachrichten erzeugen, durch die interessierter Code über Statusänderungen beim Abspielen benachrichtigt werden kann.

Lassen Sie uns schauen, was wir mit einem kleinen Beispiel erreichen können. Erzeugen Sie in Xcode eine neue View-basierte Anwendung mit dem Namen SimpleVideoPlayer.

Das MediaPlayer-Framework ist nicht Standardbestandteil der Xcode-Templates. Sie müssen es Ihrem Projekt erst hinzufügen. Klappen Sie in der *Groups & Files*-Liste in Xcode die Targets auf und wählen Sie das Symbol der SimpleVideoPlayer-Anwendung. Lassen Sie per INFO-Button in der Toolbar oder - den Inspector anzeigen und wählen Sie den Tab GENERAL. Am unteren Rand dieser Anzeige sehen Sie die aktuell verlinkten Frameworks und darunter einen +-Button, um neue Frameworks hinzuzufügen – wie zum Beispiel das MediaPlayer.framework.¹



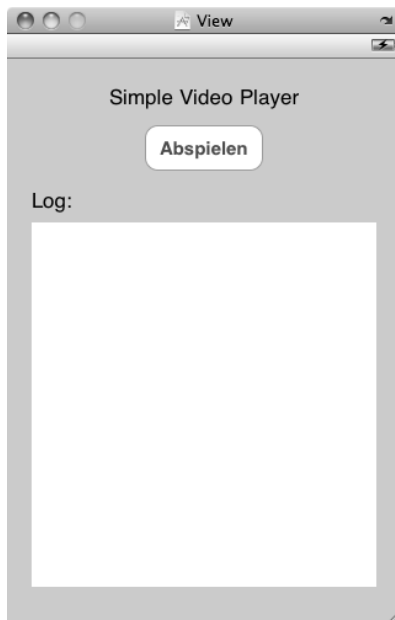
¹ Sie können Frameworks auch hinzufügen, indem Sie mit der rechten Maustaste auf den *Frameworks*-Ordner in *Groups & Files* klicken und dann **ADD EXISTING FRAMEWORKS** wählen.

Dazu müssen Sie die Zeile `#import <MediaPlayer/MediaPlayer.h>` in Ihre Header-Datei `SimpleVideoPlayerViewController.h` einfügen.

Erstellen des GUI

Unser GUI wird erstaunlich einfach sein. Wir stellen einen PLAY-Button bereit, mit dem das Video gestartet wird, und ein Textfeld, um Events zu protokollieren.

Öffnen Sie im IB `SimpleVideoPlayerViewController.xib` und erstellen Sie ein GUI, das so ähnlich wie das folgende aussieht:



Wo werden wir unser Video anzeigen? Um eine View müssen wir uns nicht kümmern. Nachdem der Anwender einmal ein Video mit dem `MPMoviePlayerController` gestartet hat, übernimmt der Controller Ihren Bildschirm, rotiert bei Bedarf automatisch ins Querformat und spielt das Video ab. Es gibt kein View-Objekt, das in eine View eingebettet und im IB verdrahtet werden muss, da das Abspielen von Videos auf dem iPhone eine Alles-oder-nichts-Sache ist.

Das von uns gebaute GUI ist also nur dazu da, das Tippen auf den PLAY-Button zu verarbeiten und Nachrichten in das Textfeld zu schreiben. Daher benötigen wir ein Outlet und eine Action. Wenn Sie schon in `SimpleVideoPlayerViewController.h` sind, fügen Sie auch gleich noch eine Instanzvariable für den `MPVideoController` hinzu.

Der Header sollte jetzt so aussehen:

```
MediaPlayer/SimpleVideoPlayer/Classes/SimpleVideoPlayerViewController.h
```

```
@interface SimpleVideoPlayerViewController : UIViewController {
    MPMoviePlayerController *moviePlayer;
    UITextView *logView;
}
- (IBAction) playVideo;
    @property (retain, nonatomic) IBOutlet UITextView *logView;
@end
```

Mit dem so definierten Outlet und der Action sollten Sie im IB jetzt das Textfeld mit dem Outlet `logView` im File's Owner und das Event Touch Up Inside des Buttons mit der Action `playVideo:` verbinden können. Speichern Sie die `.nib`-Datei danach und beenden Sie den IB. Vergessen Sie nicht, die Eigenschaft `logView` per **@synthesize** in der Implementierungsdatei mit Gettern und Settern zu versehen.

Das Video auswählen

Nachdem wir nun das GUI aufgesetzt haben, müssen wir nur noch ein Video bereitstellen und den Controller erzeugen. `MPMoviePlayerController` besitzt einen Initializer `initWithContentURL:`. Diesem kann man entweder eine URL im `file://`-Stil übergeben und damit auf eine Datei irgendwo in der Dateisystem-Sandbox der Anwendung verweisen, oder eine Netzwerk-URL nutzen (meist eine URL vom Typ `http://`, wie zum Beispiel `http://www.subfurther.com/video/running-start-iphone.m4v`, eine Kopie des Films, der für die Screenshots in diesem Kapitel genutzt wurde). Im herunterladbaren Beispiel wird davon ausgegangen, dass Sie eine einzelne MPEG-4-Datei² namens `movie.m4v` in Ihre Projektdatei auf den Resources-Ordner gezogen haben, um sie dem Anwendungs-Bundle hinzuzufügen. Dann ist das Erzeugen des `MPMoviePlayerController` in der Methode `viewDidLoad` wirklich trivial:

```
MediaPlayer/SimpleVideoPlayer/Classes/SimpleVideoPlayerViewController.m
```

```
NSString *videoPath = [[NSBundle mainBundle]
    pathForResource:@"movie" ofType:@"m4v" ];
if (videoPath == NULL)
    return;
NSURL *videoURL = [NSURL fileURLWithPath: videoPath];
moviePlayer = [[MPMoviePlayerController alloc]
    initWithContentURL:videoURL];
```

2 In Abschnitt 14.3, *Unterstützte Medienformate*, auf Seite 326 erhalten Sie weitere Informationen zu den unterstützten Formaten.



Mit dem so erzeugten und der Eigenschaft `moviePlayer` zugewiesenen Controller ist die Implementierung von `playVideo` ein Einzeiler: `[moviePlayer play];`. Klicken Sie nun auf BUILD AND RUN und tippen Sie dann auf den PLAY-Button, um das Video im Simulator betrachten zu können.³

Streaming-Unterstützung im iPhone OS 3.0

Im iPhone OS 3.0 kann Ihre URL auch auf einen HTTP-Livestream verweisen, ein neues Format, das von Apple entwickelt und bei der Internet Engineering Task Force als neuer Standard eingereicht wurde.* In diesem Format teilt der Server eine große Mediendatei oder eine Livequelle in kleinere Segmente auf, die dann per HTTP an den Client geschickt werden können. In Apples *HTTP Live Streaming Overview* [App09a] finden Sie eine Beschreibung des Formats und ein paar Beispiel-URLs.

Ein Großteil der Arbeit beim HTTP Live Streaming geschieht auf dem Server – ein iPhone-Client muss nur die URL an den `MPMoviePlayerController` übergeben. Es lassen sich übrigens auch Shoutcast-MP3-Audio-Streams über HTTP abspielen, aber da der `MPMoviePlayerController` den gesamten Bildschirm einnimmt, sehen Sie bei dem reinen Audio-Shoutcast-Stream nur ein großes, graues QuickTime-Logo ... keine ideale Anwendererfahrung!

* HTTP Live Streaming (Internet-Draft) [Pan09]

³ Im iPhone SDK 2.2 gibt es einen Fehler: Wenn man ein Video im Simulator abspielt, während ein Bluetooth-Gerät mit Ihrem Mac verbunden ist, erhalten Sie eine Fehlermeldung. Entfernen Sie dann Ihre Bluetooth-Geräte in den Systemeigenschaften oder lassen Sie die App auf dem iPhone laufen, damit dieses Beispiel funktioniert.

Der `MPMoviePlayerController` besitzt eine Handvoll Eigenschaften, die Sie setzen oder auslesen können. Der `scalingMode` erlaubt es Ihnen, festzulegen, ob ein Video, das nicht genau der Bildschirmgröße des iPhone entspricht, so skaliert werden soll, dass es in einer Dimension passt (`MPMovieScalingModeAspectFit`), in beiden Dimensionen mit einem eventuellen Beschneiden (`MPMovieScalingModeAspectFill`), in beiden Dimensionen mit einer eventuellen Änderung des Seitenverhältnisses (`MPMovieScalingModeFill`) oder ob es gar nicht skaliert werden soll (`MPMovieScalingModeNone`). Die andere interessante Eigenschaft ist `movieControlMode`. Dort können Sie einstellen, welche Steuerelemente angezeigt werden sollen, wenn man während der Wiedergabe auf den Bildschirm tippt: alle Steuerelemente für die Timeline und die Lautstärke (`MPMovieControlModeDefault`), nur die für die Lautstärke (`MPMovieControlModeVolumeOnly`) oder gar keine Steuerelemente (`MPMovieControlModeHidden`).

14.2 Nachrichten vom Movie Player empfangen

Der `MPMoviePlayerController` versendet Nachrichten über Statusänderungen und andere Ereignisse, die während des Abspielens eines Videos passieren. Nutzt man diese Nachrichten, könnte eine Player-App zum Beispiel einen Lade-Indikator anzeigen, bis der Film abspielbereit ist, um dann einen `PLAY`-Button anzuzeigen. Oder Sie könnten eine Zwischensequenz in einem Spiel durch ein Video umsetzen, das nach dem Ende des Films zum Spiel zurückkehrt.

Aktuell sind drei Nachrichten definiert:

- `MPMoviePlayerContentPreloadDidFinishNotification` zeigt an, dass das Laden abgeschlossen ist und der Controller nun den Film abspielen kann, oder dass ein Fehler aufgetreten ist.
- `MPMoviePlayerPlaybackDidFinishNotification` weist darauf hin, dass das Abspielen beendet ist.
- `MPMoviePlayerScalingModeDidChangeNotification` wird dann verschickt, wenn der Anwender das Control zum Anpassen der Skalierung genutzt hat.

Um eine dieser Nachrichten zu erhalten, verwenden Sie die `NSNotification-API`, wobei Sie die Standardinstanz von `NSNotificationCenter` zusammen mit dem Namen der Nachricht und einem Selektor (einem Methodennamen) übergeben, um Callbacks zu erhalten, wenn die Nachricht verschickt wird. Sie können sich also in `viewDidLoad` für eine „Abspielen beendet“-Nachricht registrieren:

MediaPlayer/SimpleVideoPlayer/Classes/SimpleVideoPlayerViewController.m

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector: @selector (playbackFinished:)
    name:@"MPMoviePlayerPlaybackDidFinishNotification"
    object:nil];
```

Um Nachrichten an die Text-View auszugeben, erstellen wir eine einfache Methode, die die Text-View mit dem aktuellen Text, dem Zeitstempel und einer Nachricht befüllt:

MediaPlayer/SimpleVideoPlayer/Classes/SimpleVideoPlayerViewController.m

```
- (void) appendTextToLogView: (NSString*) text {
    logView.text = [NSString stringWithFormat: @"%@%@: %@\n" ,
        logView.text, [NSDate date], text]; }
```

Wenn die Nachricht dann die Methode `playbackFinished` per Callback ruft, protokollieren wir einfach den String Fertig:

MediaPlayer/SimpleVideoPlayer/Classes/SimpleVideoPlayerViewController.m

```
- (void) playbackFinished: (NSNotification*) notification {
    [self appendTextToLogView:@"Fertig" ]; }
```

Mit einer ähnlichen Log-Nachricht in `playVideo` sieht die Haupt-View nach dem Abspielen eines Videos ungefähr so aus:



14.3 Unterstützte Medienformate

Wenn Sie Audiodaten oder Videos abspielen wollen, müssen Sie wissen, welche Medienformate das iPhone unterstützt und wie man Daten produziert, die auf dem Gerät laufen.⁴

Es gibt einen großen Unterschied zwischen *Audio-/Video-Containerformaten* und deren Inhalt. Manche Formate, wie zum Beispiel die .mov-Dateien von QuickTime, nehmen so gut wie jeden Audio- oder Video-Codec auf: Ein H.264-Video zusammen mit MP3 innerhalb einer .mov-Datei ist nicht verboten (und wird auf dem iPhone abgespielt). Andere Formate sind auf eine Art von Inhalt beschränkt – so enthalten .mp3s nur MP3-Audiodaten.

Hier eine Übersicht über die wichtigsten Mediencontainer und -inhalte, die vom iPhone unterstützt werden:

Audioformat	Inhalt
.aif	PCM (Big-Endian Integer Samples); AIFC-Format (auch .aif) kann auch mit IMA4, μ -law, A-law und anderen umgehen
.wav	PCM (Little-Endian Integer Samples), μ -law, A-law
.aac, .m4a, .mp4, .3gp, .3g2	AAC
.mp3	MPEG-1 Layer 3 (MP3).
.caf	alle vorigen, dazu Internet Low-Bitrate Codec (iLBC), Apple Lossless (ALAC), IMA4, Adaptive Multi-Rate (AMR) und andere

A/V-Format	Inhalt
.mp4, .m4v, .3gp, .3g2	MPEG-4 Part 2 Simple Video, H.264 Baseline Profile Video; AAC Audio
.mov	alle vorigen Video-Codecs, alle vorigen Audio-Codecs

⁴ Die verfügbaren Formate sind auf den kundenorientierten Support-Webseiten für das iPhone aufgeführt, zum Beispiel unter <http://www.apple.com/iphone/specs.html>, aber es lohnt sich, sie kurz zusammenzufassen.

Bei den Containern mag das Core Audio File (.caf) das unbekannteste Format sein, aber es ist auch das flexibelste. Wie ein QuickTime-Video (.mov) akzeptiert es *jeden* Audio-Codec als Inhalt. Zudem gibt es Time-to-Sample-Tabellen, so dass das Springen in einer Datei schneller von-statten geht als bei rein Stream-basierten Formaten wie .mp3. Apple empfiehlt das CAF, um Sounds in Ihrer App mitzuliefern.

Einige der unterstützte Codecs haben Einschränkungen bezüglich Bit-rate, Optionen, Größe oder anderer Parameter, daher raten wir Ihnen, einen Blick in den *iPhone Application Programming Guide* [App09c] zu werfen, wenn Sie es genau wissen wollen. Unter den von Core Audio unterstützen Formaten können Sie mithilfe des Befehlszeilentools `/usr/bin/afconvert` zwischen den Formaten und Codecs konvertieren. Zudem führt dessen Hilfetext die unterstützten Formate und Inhalte auf – geben Sie einfach den folgenden Befehl in `Terminal.app` oder `xterm` ein:

```

⇒ afconvert --help
⇒ Usage:
afconvert [option...] input_file [output_file]
Options: (may appear before or after arguments)
    { -f | --file } file_format:
        '3gpp' = 3GP Audio (.3gp)
                data_formats: 'aac ' 'samr'
        '3gp2' = 3GPP2 Audio (.3g2)
                data_formats: 'aac ' 'samr' ...

```

Wenn Sie schon ein erfahrener Kompressionist sind, können Sie Ihre Ergebnisse anhand dieser Spezifikation optimieren, aber der durchschnittliche Entwickler hat hier zu viele Gelegenheiten, Fehler zu machen. Man kann auch den Movie to iPhone-Exporter von QuickTime nutzen, der ein iPhone-kompatibles Video erzeugt, ohne dass man sich dabei mit Kompressionsoptionen herumschlagen muss. Viele Leute gehen davon aus, dass man QuickTime Pro kaufen müsse, um die Exportfunktionalität von QuickTime nutzen zu können, aber das entsperrt die Export-Features nur im QuickTime-Browser-Plug-in und in der Player-Anwendung. Sie können den QuickTime-Exporter auch in anderen Anwendungen nutzen, wie zum Beispiel in Final Cut Express und Final Cut Pro (über den Menüpunkt `EXPORT USING QUICKTIME CONVERSION`) oder iMovie (fügen Sie Ihr Video in ein neues Projekt ein und wählen Sie dann `BEREITSTELLEN MIT QUICKTIME EXPORTIEREN`), ohne QuickTime Pro kaufen zu müssen. In diesen Anwendungen stellt das Dialogfenster des QuickTime-Exports ein Pop-up-Menü bereit, in dem man das Format wählen kann. Eines davon ist für das iPhone gedacht.⁵



Wenn Sie ein QuickTime- oder QTKit-Entwickler sind, können Sie diesen Export mit iPhone Export Component auch in einer eigenen Mac-Anwendung vornehmen. In Apples *Technical Note TN2188: Exporting Movies for iPod, Apple TV and iPhone* [App09d] finden Sie die Einzelheiten dazu.

5 Tatsächlich gibt es sogar zwei. Der Exporter „Film - iPhone (Mobiltelefon)“ erzeugt eine kleinere, bandbreitenschonendere Datei, die über das langsamere EDGE-Netzwerk verteilt werden kann.

Kapitel 15

Zugriff auf die iPod-Bibliothek

Wenn ein Anwender Ihre Applikation herunterlädt und installiert, ist es sehr wahrscheinlich, dass er auch schon iTunes dazu genutzt hat, die bevorzugten Podcasts und Filme, seine Lieblingsmusik sowie andere Medien auf sein iPhone oder seinen iPod touch zu laden. In früheren Versionen des iPhone OS lagen diese Medien vollständig außerhalb des Zugriffsbereichs Ihrer Anwendung. Mit dem iPhone SDK 3.0 hat Ihre Anwendung aber einen eingeschränkten Zugang zur Musikbibliothek. Sie kann dabei die Musik des Anwenders abspielen, den Inhalt der Bibliothek durchforsten und sogar den Hintergrund-iPod-Player kontrollieren.

In diesem Kapitel werden wir uns die neuen Features anschauen, die das Media Player-Framework bringt. Wir werden mit der iPod-Anwendung interagieren, während sie aktiv ist, und in der Musikbibliothek des Anwenders nach Liedern, Podcasts und Hörbüchern suchen, um sie abzuspielen.

15.1 Den iPod-Player überwachen

Wir werden unsere Betrachtung des Media Player-Framework damit beginnen, den iPod-Player zu überwachen. Der mitgelieferte Media Player ist etwas Besonderes – anders als Anwendungen, die mit dem öffentlichen SDK geschrieben wurden, kann er im Hintergrund laufen. Sie können also in der iPod-Anwendung Audiodaten abspielen lassen, die Anwendung beenden oder sogar den Bildschirm sperren und trotzdem weiter Musik hören.

Das Musik-GUI erstellen

Die iPod-Anwendung

iPhone und iPod touch nutzen unterschiedliche Namen für ihren Media Player. Auf dem iPhone gibt es eine einzelne iPod-Anwendung, mit der Sie sowohl Audio- als auch Videodaten durchstöbern und abspielen können. Auf dem iPod touch gibt es dafür separate Musik- und Videoanwendungen. Abgesehen von diesem Unterschied ist die Funktionalität aus Entwicklersicht identisch: Das von `[MPMusicPlayerController iPodMusicPlayer]` zurückgegebene Objekt bietet auf beiden Plattformen das Gleiche an. Wenn wir uns also auf die „iPod-Anwendung“ beziehen, meinen wir damit den mitgelieferten Media Player, unabhängig davon, wie er sich dem Endanwender präsentiert.

Als Beispiel wollen wir eine umklappbare Hilfsanwendung schreiben, deren Vorderseite das aktuelle Element der iPod-Anwendung und ein paar einfache Steuerelemente zeigt. Später werden wir dann in diesem Kapitel die Rückseite nutzen, um eine Benutzerschnittstelle zum Auswählen von Elementen aus der Bibliothek zu schaffen, die dann abgespielt werden. Beachten Sie, dass der Simulator keinen Musik-Player besitzt. Wenn Sie dieses Beispiel ausführen wollen, müssen Sie es also für ein echtes Gerät bauen und es darauf installieren.

Erstellen Sie eine Utility-Anwendung mit Xcode und geben Sie ihr den Namen `MusicLibraryClient`. Damit werden View-Controller-Klassen für die Haupt-View und für die Rückseite erzeugt, und wir erhalten View-Klassen, die wir nicht weiter anpassen müssen.

Öffnen Sie die Datei `MainView.xib` im IB (nicht `MainWindow.xib`) und legen Sie die GUI-Elemente wie in Abbildung 15.1 an. Dabei handelt es sich um folgende Komponenten:

- Ein Label für die aktuelle Spielzeit in einer sehr großen Schrift (wir haben Trebuchet in 48 Punkt verwendet).
- Ein Button zum Abspielen und Pausieren. Der Text ist „OK“, im herunterladbaren Beispiel haben wir einen Button verwendet, der das Bild `PlayButton.png` für den normalen und `PauseButton.png` für den gedrückten Status nutzt.

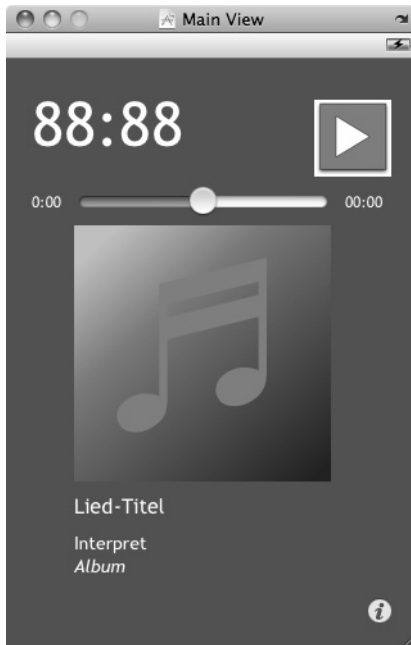


Abbildung 15.1: Layout für die Haupt-View von MusicLibraryClient

- Ein Slider, um sich im Lied vorwärts und zurück bewegen. Dies wird im Allgemeinen als *Scrubbing* bezeichnet. Der Wert des Sliders wird die aktuelle Position in der Audiodatei in Sekunden angegeben, da dies das Format der Media Player-Bibliothek für solche Eigenschaften ist. Fügen Sie an beiden Enden des Sliders kleine Labels hinzu, um die Startzeit (immer 0:00) und die Laufzeit des Lieds auszugeben.
- Eine UIImageView für das Cover zum Lied. Da manche iPod-Bibliothekselemente kein Coverbild besitzen, ist es sinnvoll, ein Ersatzbild zu haben, das man in diesem Fall nutzen kann und das hier auch als erstes Bild angezeigt wird.
- Labels für den Titel, den Künstler und das Album. Bei Podcasts und Hörbüchern werden hier etwas andere Inhalte zu finden sein: Episode/Künstler/Podcast beziehungsweise Titel/Autor/Buch.
- Ein INFO-Button, um auf die Rückseite zu wechseln. Er ist schon da und auch korrekt verdrahtet.

Sie müssen nun IBOutlet's für die Komponenten deklarieren, deren Werte sich zur Laufzeit ändern werden. Deklarieren Sie die folgenden Instanzvariablen, machen Sie sie durch **@property** und **@synthesize** zu Eigenschaften und verdrahten Sie sie wie üblich:

MediaLibrary/MusicLibraryClient/Classes/MainViewController.h

```
UILabel *currentTimeLabel;
UISlider *currentTimeSlider;
UIButton *playPauseButton;
UILabel *currentItemDurationLabel;
UILabel *currentItemTitleLabel;
UILabel *currentItemArtistLabel;
UILabel *currentItemAlbumLabel;
UIImageView *currentItemArtworkView;
```

Da wir ein Framework nutzen, das nicht zum Standard gehört, müssen wir es unserem Projekt hinzufügen. Ergänzen Sie in der Project-Ansicht im Frameworks-Ordner MediaPlayer.framework und fügen Sie dem MainViewController.h ein `#import <MediaPlayer/MediaPlayer.h>` hinzu.

Den aktuellen Player-Status ermitteln

Um diese verschiedenen Felder zu befüllen, holen wir uns ein Controller-Objekt, das die Hintergrundanwendung verwaltet. Diese Klasse nennt sich `MPMusicPlayerController` und kann zwei Arten von Instanzen hervorbringen. Der iPod Music Player ist ein Objekt, das mit dem Media Player des Systems interagiert, wobei Sie Informationen darüber erhalten können, was gerade gespielt wird, und es möglich ist, die Wiedergabe zu starten, zu pausieren und zu stoppen. Zudem können Sie eine neue Liste mit abzuspielenden Objekten angeben. Die andere Art Player ist der *Application Music Player*. Der hat die gleiche Funktionalität wie der iPod Player, ist aber komplett unabhängig von ihm – mit diesem Objekt beeinflussen Sie nicht den Status des System Media Player. Sie können diesen Player verwenden, wenn Sie Audiodaten aus der iPod-Bibliothek des Anwenders nutzen wollen, um sie dann aber in Ihrer eigenen Anwendung abzuspielen – zum Beispiel für Hintergrundmusik in einem Spiel. Nutzen Sie den Application Player, kann der iPod Player parallel dazu laufen. Das ist vermutlich nicht unbedingt gewünscht, da dann potenziell zwei Lieder gleichzeitig gespielt werden. Sie könnten auch das iPod Player-Objekt holen und ihn stoppen, aber es ist besser, die in Abschnitt 16.7, *Audio-Sessions*, auf Seite 382 beschriebenen Techniken anzuwenden, um dem System mitzuteilen, wie Sie die verschiedenen Audioquellen kombinieren wollen.

Sie erhalten den Player Ihrer Wahl über eine Klassenmethode – entweder per `iPodMusicPlayer` oder `applicationMusicPlayer`. Mit dem Player können Sie dann eine Reihe von Eigenschaften nutzen, um den Abspielmodus oder -zustand zu verwalten, wobei die nützlichsten die selbsterklärenden `currentPlaybackTime`, `volume`, `repeatMode` und `shuffleMode` sind. Es gibt zudem einen nur lesbaren `playbackState`, dessen verschiedene enumerierte `MPMusicPlaybackState`-Werte dazu genutzt werden können, den Zustand unseres Play-/Pause-Buttons zu setzen:

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
- (void) updatePlayState {
    MPMusicPlayerController *iPodController =
        [MPMusicPlayerController iPodMusicPlayer];
    playPauseButton.selected =
        (iPodController.playbackState == MPMusicPlaybackStatePlaying);
}
```

Wir wollen ebenfalls Informationen über das gerade gespielte Stück erhalten. Die Eigenschaft `nowPlayingItem` enthält ein `MPMediaItem`, das beschreibt, was aktuell gespielt oder im Pausemodus gehalten wird oder demnächst folgen soll. Das `MPMediaItem` repräsentiert ein abspielbares Objekt – ein Lied, eine Podcast-Episode, einen Abschnitt eines Hörbuchs und so weiter. Es besitzt nur zwei öffentliche Methoden, aber eine davon, `valueForProperty:`, liefert uns alles, was wir über das aktuell laufende Objekt wissen müssen (und später auch Informationen über Objekte aus der Medienbibliothek). Es gibt eine lange Liste von vordefinierten Schlüsseln, mit denen wir Metadateneigenschaften auslesen können. Diese lassen sich in drei Gruppen unterteilen:¹

- *Allgemeine Medienobjekteigenschaften:* Alle Eigenschaften, die jedes Medienobjekt besitzt: `MPMediaItemPropertyTitle`, `MPMediaItemPropertyArtist`, `MPMediaItemPropertyPlaybackDuration` und noch deutlich mehr, als hier sinnvoll aufzuzählen wäre.
- *Podcast-Objekteigenschaften:* Metadainformationen für Podcasts. Der aktuell einzige Schlüssel ist `MPMediaItemPropertyPodcastTitle`.
- *Benutzerdefinierte Objekteigenschaften:* Eigenschaften, die sich mehr auf die Nutzung des Medienobjekts als auf seine intrinsischen Eigenschaften beziehen: `MPMediaItemPropertyPlayCount`, `MPMediaItemPropertyLastPlayedDate` und so weiter.

¹ In der Dokumentation des `MPMediaItem` finden Sie eine vollständige Liste und die für jede Eigenschaft zurückgegebenen Datentypen.

Das Auslesen der Eigenschaften des aktuell gespielten Objekts ermöglicht es uns, einen Großteil der Labels zu befüllen, die wir auf der View anzeigen, zudem können wir das Cover darstellen. Für Letzteres sollten wir einen Plan B haben, wenn der Anwender in iTunes kein Cover zugewiesen hat. Haben Sie ein leeres Album-PNG in Ihren Projektressourcen untergebracht, deklarieren Sie eine Instanzvariable `UIImage *emptyAlbumImage`; im Header, definieren per **#define** einen `EMPTY_ALBUM_FILE_NAME` und erzeugen das `UIImage` ziemlich zu Anfang Ihrer Methode `viewDidLoad`.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
emptyAlbumImage = [[UIImage alloc] initWithContentsOfFile:
    [[NSBundle mainBundle]
        pathForResource:EMPTY_ALBUM_FILE_NAME ofType:@"png" ]];
```

Damit können wir nun eine Methode `updateCurrentiPodItemMetadata` schreiben, um das aktuell laufende Objekt zu ermitteln und die meisten UI-Komponenten mit den verschiedenen Eigenschaften zu befüllen. Läuft gerade nichts, können wir sie auch leeren.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
1 -(void) updateCurrentiPodItemMetadata {
2     MPMusicPlayerController *iPodController
3         [MPMusicPlayerController iPodMusicPlayer];
4     MPMediaItem *nowPlayingItem= [iPodController nowPlayingItem];
5     if (nowPlayingItem == nil) {
6         currentTimeSlider.maximumValue = 1.0;
7         currentTimeSlider.value = 0;
8         currentItemDurationLabel.text = @"00:00";
9         currentItemTitleLabel.text = @"(kein Stück)";
10        currentItemArtistLabel.text = nil;
11        currentItemAlbumLabel.text = nil;
12        currentItemArtworkView.image = emptyAlbumImage;
13    } else {
14        NSNumber *durationNumber = [nowPlayingItem
15            valueForKeyProperty:MPMediaItemPropertyPlaybackDuration];
16        currentTimeSlider.maximumValue =
17            [durationNumber floatValue];
18        currentItemTitleLabel.text = [nowPlayingItem
19            valueForKeyProperty:MPMediaItemPropertyTitle];
20        currentItemArtistLabel.text = [nowPlayingItem
21            valueForKeyProperty:MPMediaItemPropertyArtist];
22        currentItemAlbumLabel.text = [nowPlayingItem
23            valueForKeyProperty:MPMediaItemPropertyAlbumTitle];
24        MPMediaItemArtwork *coverArt = [nowPlayingItem
25            valueForKeyProperty:MPMediaItemPropertyArtwork];
26        if (coverArt)
27            currentItemArtworkView.image = [coverArt
28                initWithSize: currentItemArtworkView.frame.size];
```



```

29         else
30             currentItemArtworkView.image = emptyAlbumImage;
31     }
32 }
33

```

Viele dieser Eigenschaftswerte sind `NSStrings`, wodurch es leicht ist, sie den Labels des GUI als Text zuzuweisen. Eine Ausnahme bildet die Spiel­länge (Zeile 14), bei der es sich um eine `NSNumber` handelt, die für die Sekunden als `NSTimeInterval` steht (ein **double**). Beachten Sie auch das Auslesen des Covers, des Podcast-Logos oder was sonst als Bild zugewiesen ist, in Zeile 24.² Dieser Eigenschaftswert ist ein Objekt vom Typ `MPMediaItemArtwork`, aus dem Sie über `imageWithSize:` ein skaliertes `UIImage` erhalten können. Hier bitten wir um ein Bild in der Größe der `UIImageView`. Ist diese Eigenschaft `nil`, nutzen wir stattdessen `emptyAlbumArt`.

Arbeiten mit Zeiteigenschaften

Wir haben bisher noch nicht den aktuellen Zeitpunkt im Musikstück angepasst. Es ist kein Problem, die aktuelle Zeit über die Methode `currentTime` des `MPMusicPlayerController` auszulesen. Damit erhalten wir die Anzahl an Sekunden als `NSTimeInterval` (ein **double**), die seit dem Anfang des Tracks vergangen sind. Der Trick ist, wie wir sie dem Anwender darstellen. Er möchte schließlich keine große Gleitkommazahl zu sehen bekommen.

Mit ein bisschen Formatierungszauber können wir einen `NSString` erhalten, der die Uhrzeit anzeigt – aufgefüllt auf eine festgelegte Ziffernzahl, so dass Sekunden kleiner als 10 eine führende 0 erhalten. Den formatierten Wert für die Laufzeit erhalten Sie durch die folgende zusätzliche Zeile in der Methode `updateCurrentiPodItemMetadata` im **else**-Block:

```

currentItemDurationLabel.text =
    [NSString stringWithFormat: @"%02d:%02d" ,
     [durationNumber intValue] / 60,
     [durationNumber intValue] % 60];

```

Wichtiger ist noch, dass Sie jetzt die aktuelle Position im Track anpassen können, indem Sie die Eigenschaft `currentTime` aus dem `MPMusicPlayerController` auslesen

```

- (void) updateCurrentiPodItemTime {
    MPMusicPlayerController *iPodController =
        [MPMusicPlayerController iPodMusicPlayer];
    MPMediaItem *nowPlayingItem= [iPodController nowPlayingItem];
}

```

² Diese Eigenschaft unterstützt keine „erweiterten Podcasts“, die das Bild während des Abspielens der Episode anpassen. Sie erhalten immer das gleiche für eine Episode spezifische Bild, unabhängig von der aktuellen Abspielposition.

```

    if (nowPlayingItem == nil) {
        currentTimeLabel.text = @"00:00" ;
    } else {
        double currentTime = iPodController.currentPlaybackTime;
        currentTimeLabel.text = [NSString stringWithFormat: @"%02d:%02d" ,
            (int) currentTime/60,
            (int) currentTime%60];
        currentTimeSlider.value = (float) currentTime;
    }
}

```

Wenn die Musik beim Aufruf der Methode gerade abgespielt wird, ist diese Anzeige natürlich nur für einen kurzen Moment korrekt. Wir müssen sie also regelmäßig aktualisieren lassen. Wie bei der Uhr in Kapitel 9, *Einstellungen*, auf Seite 181 können wir einen NSTimer nutzen, um `updateCurrentiPodItemTime` regelmäßig aufzurufen (wenn Sie noch einmal nachlesen wollen, wie das funktioniert, werfen Sie einen Blick in Abschnitt 9.4, *Abstecher: Sekündliches Aktualisieren der Uhr*, auf Seite 191). Deklarieren Sie `NSTimer *currentTimeUpdateTimer`; in der `.h`-Datei und richten Sie sie in der Methode `viewDidLoad` der `.m`-Datei ein.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```

// Timer zum Aktualisieren der Uhr
currentTimeUpdateTimer = [NSTimer scheduledTimerWithTimeInterval:0.1
    target:self selector:@selector(updateCurrentiPodItemTime)
    userInfo:NULL repeats:YES];

```



Abbildung 15.2: Das aktuell gespielte iPod-Objekt anzeigen

Als Letztes müssen in `viewDidLoad` einmalig unsere drei Methoden zum Aktualisieren des GUI aufgerufen werden, so dass die View beim Laden den Abspielstatus, die aktuelle Spielzeit und die Metadaten des aktuellen Objekts von der iPod-Anwendung erhält:

```
MediaLibrary/MusicLibraryClient/Classes/MainViewController.m
```

```
[self updatePlayState];
[self updateCurrentiPodItemMetadata];
[self updateCurrentiPodItemTime];
```

Jetzt haben wir alles beisammen, was wir brauchen, um anzuzeigen, was gerade im System-Musik-Player vor sich geht. Starten Sie iPod (oder Musik) und spielen Sie Musik, einen Podcast oder ein Hörbuch ab. Währenddessen bauen Sie Ihre Anwendung und lassen sie auf dem iPhone laufen. Sie sehen dann das aktuelle Objekt in der Haupt-View (siehe Abbildung 15.2). Wir müssen zwar noch die Kontrolle über das Abspielen durch unsere Anwendung erlangen, aber Sie können schon die Fernsteuerung an den Kopfhörern nutzen.

Benachrichtigungen vom Player erhalten

Bisher ist die einzige Metadatenkomponente, die regelmäßig aktualisiert wird, die momentane Position im abgespielten Objekt. Das erreichen wir durch den `NSTimer`. Wenn Ihr Lied endet und der Media Player zum nächsten Stück wechselt, ist ein Großteil der Informationen auf dem UI veraltet. Das Gleiche passiert, wenn Sie zum nächsten Lied springen, indem Sie an Ihrer Kopfhörerfernsteuerung doppelt klicken oder per Dreifachklick zurückspringen.

Glücklicherweise gibt es einen systemweiten Dienst für das Benachrichtigen von Anwendungen über asynchrone Events – das *Notification Center*. Anwendungen können Benachrichtigungen an das Notification Center schicken, von wo sie weiter an alle registrierten *Observers* gehen. Der `MPMusicPlayerController` stellt zwei Benachrichtigungen bereit, die durch Konstanten vom Typ `NSString` repräsentiert werden:

- `MPMusicPlayerControllerPlaybackStateDidChangeNotification` zeigt an, dass sich der aktuelle Abspielstatus (abspielend, pausierend, gestoppt, vorwärts springend und so weiter) geändert hat.
- `MPMusicPlayerControllerNowPlayingItemDidChangeNotification` benachrichtigt Observer darüber, dass sich das aktuelle Objekt geändert hat – entweder durch eine Aktion des Anwenders (um vorwärts oder rückwärts zu springen) oder durch das Ende eines Stücks und den Wechsel zum nächsten in der Queue.

Für das erste Event können wir einfach unsere Methode `updatePlayState` aufrufen, um den Zustand des Play-/Pause-Buttons zu ändern. Beim zweiten Event müssen wir alle Metadatenkomponenten aktualisieren.

Sie fügen einen Observer für eine bestimmte Benachrichtigung hinzu, indem Sie sich die Standardinstanz des `NSNotificationCenter` holen und `addObserver:selector:name:object:` aufrufen. Die Parameter für diese Methode sind das Objekt, das die Callbacks erhalten soll, ein Methodenselektor, der Benachrichtigungsname und ein optionales „Sender“-Argument, das die Benachrichtigungen auf diejenigen beschränkt, die von einem bestimmten Objekt kommen. Wir erzeugen die Observer in `viewDidLoad`.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(playbackStateChanged:)
    name:@"MPMusicPlayerControllerPlaybackStateDidChangeNotification"
    object:nil];
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(nowPlayingItemChanged:)
    name:@"MPMusicPlayerControllerNowPlayingItemDidChangeNotification"
    object:nil];
[[MPMusicPlayerController iPodMusicPlayer]
    beginGeneratingPlaybackNotifications];
```

Jetzt müssen wir die Callback-Methoden schreiben, die sich um die Benachrichtigungen kümmern. Diese Methoden benötigen ein `NSNotification`-Argument, in dem sich der Name der Benachrichtigung, das absendende Objekt und (für manche Benachrichtigungen) ein „User-Info“-Dictionary mit zusätzlichen Informationen befindet. In diesem Fall brauchen wir das alles nicht und können einfach unsere Aktualisierungsmethoden aufrufen.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
-(void) playbackStateChanged: (NSNotification*) notification {
    [self updatePlayState];
}

-(void) nowPlayingItemChanged: (NSNotification*) notification {
    [self updatePlayState];
    [self updateCurrentiPodItemMetadata];
    [self updateCurrentiPodItemTime];
}
```

Mit dieser Ergänzung können Sie jetzt mithilfe Ihrer Fernsteuerung am Kopfhörer vor- und rückwärts springen und sehen, wie sich die Metadaten in der Haupt-View ändern. Das ist noch beeindruckender, wenn

Sie eine Wiedergabeliste mit Liedern von unterschiedlichen Alben nutzen, wie zum Beispiel eine Genius-Wiedergabeliste. Probieren Sie es einmal aus.

15.2 Die iPod-Anwendung steuern

Sie müssen kein passiver Client der iPod-Anwendung sein. Der `MPMusicPlayerController` stellt eine ganze Reihe von Methoden zur Verfügung, um die iPod-Anwendung zu steuern: `play`, `pause`, `stop`, `skipToNextItem`, `skipToBeginning` und so weiter. Auf einige der Abspielmoduseigenschaften kann man auch schreibend zugreifen, wie zum Beispiel auf `repeatMode`, `shuffleMode` und `currentPlaybackTime`.

Damit wird das Implementieren unseres Play-/Pause-Buttons sehr einfach. Deklarieren Sie `handlePlayPauseTapped` als `IBAction`, verbinden Sie den Play-/Pause-Button damit und implementieren Sie die Methode wie folgt:

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
-(IBAction) handlePlayPauseTapped {
    MPMusicPlayerController *iPodController =
        [MPMusicPlayerController iPodMusicPlayer];
    if (iPodController.playbackState == MPMusicPlaybackStatePlaying) {
        [iPodController pause];
    } else {
        [iPodController play];
    }
}
```

Beachten Sie, dass dieser Handler den Zustand des Buttons nicht beeinflusst. Das muss er auch nicht. Wenn die iPod-Anwendung mit dem Abspielen beginnt, schickt sie eine Benachrichtigung über die Statusänderung. Diese wird vom Observer empfangen, den wir im vorigen Abschnitt registriert haben und der den Button-Zustand bei Bedarf anpasst.

Die aktuelle Position im iPod Player setzen

Wir können auch die aktuelle Position im momentan abgespielten Stück setzen und damit ein nettes „Scrubbing“-Verhalten erreichen, statt ungenau und langsam Vorlauf- und Rücklauf-Buttons nutzen zu müssen (die Sie natürlich auch über die Methoden `beginSeekingForward`, `beginSeekingBackward` und `endSeeking` des `MPMusicPlayerController` implementieren können) – der Anwender verschiebt den Slider und setzt damit die aktuelle Position.

Wir müssen berücksichtigen, dass der Wert des Sliders immer wieder durch den Aufruf von `updateCurrentiPodItemTime` angepasst wird, was mehrmals pro Sekunde durch `NSTimer` geschieht. Während also der Anwender eine Scrub-Geste ausführt, müssen wir dieses Verhalten unterbinden.

Wir definieren zunächst in der Header-Datei die Instanzvariable `BOOL userIsScrubbing`; und setzen ihren Wert in `viewDidLoad` auf `NO`. Wir können den Wert dieser Statusvariablen setzen, wenn der Anwender auf den Slider tippt und wenn er ihn wieder „loslässt“. Deklarieren Sie also zwei `IBActions`, die sich um `userIsScrubbing` kümmern, und implementieren Sie sie wie folgt:

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
-(IBAction) handleScrubberTouchDown {
    userIsScrubbing = YES;
}

-(IBAction) handleScrubberTouchUp {
    userIsScrubbing = NO; }
```

Als Nächstes verbinden Sie das Event „Touch Down“ mit `handleScrubberTouchDown` und die beiden Events „Touch Up Inside“ und „Touch Up Outside“ mit `handleScrubberTouchUp`. Schließlich passen Sie noch die letzte Zeile von `updateCurrentiPodItemTime` so an, dass der Slider nur dann aktualisiert wird, wenn der Anwender ihn nicht gerade verschiebt:

```
if (! userIsScrubbing)
    currentTimeSlider.value = (float) currentTime;
```

Jetzt muss nicht mehr um den Slider gekämpft werden, indem Timer und Anwender gleichzeitig versuchen, unterschiedliche Werte zu setzen. Als Letztes weisen wir den `MPMusicPlayerController` an, als Reaktion auf das Verschieben des Sliders die aktuelle Position zu setzen. Zum Glück generiert der Slider beim Verschieben auch ein Event „Value Changed“. Deklarieren Sie in der Header-Datei `-(IBAction)handleScrub`; und verbinden Sie es im IB mit dem Event „Value Changed“. Die Implementierung ist einfach – setzen Sie schlicht die Eigenschaft `currentPlaybackTime` des Players auf den Wert des Sliders.

MediaLibrary/MusicLibraryClient/Classes/MainViewController.m

```
-(IBAction) handleScrub {
    MPMusicPlayerController *iPodController =
        [MPMusicPlayerController iPodMusicPlayer];
    iPodController.currentPlaybackTime = currentTimeSlider.value;
}
```

Jetzt sollten Sie in Liedern vor- und zurückspringen können, selbst wenn sie gerade laufen.³ Der Slider schickt durch die Verschiebegeräte kontinuierlich Events vom Typ `Value` aus, so dass Sie sofort die Ergebnisse hören können und sehen, wie sich die aktuelle Position verändert.

15.3 Die iPod-Bibliothek verwenden

Bisher haben wir uns darauf verlassen, dass der Anwender in der iPod- oder Musikanwendung schon Musik abspielt, mit der wir arbeiten können. Der Player kann aber auch von uns Lieder (oder andere Stücke) erhalten, die er abspielen soll. Dazu müssen wir aber das Media Player-Framework nutzen, um herauszufinden, was der Anwender so in seiner Medienbibliothek hat.

Das Media Player-Framework bietet zwei Wege an, Objekte in der Medienbibliothek des Anwenders zu finden, die als `MPMediaItems` repräsentiert werden (so wie das vom `MPMusicPlayerController` zurückgelieferte aktuelle Objekt):

- Die Klasse `MPMediaQuery` ermöglicht es Ihnen, in der Bibliothek nach verschiedenen Kriterien zu suchen, zum Beispiel nach Art (Lied, Podcast-Episode, Hörbuch), Titel, Interpret, Albumtitel und so weiter.
- Die Klasse `MPMediaPickerController` bietet ein GUI an, das es Ihnen ermöglicht, in der Medienbibliothek zu stöbern. Abhängig vom gewählten Modus (Alben, Interpreten, Podcasts und so weiter) entspricht die Darstellung fast genau der, die Sie aus der iPod- oder Musikanwendung kennen.

Um diese APIs kennenzulernen, werden wir die Rückseite des `MusicLibraryClient` nutzen, um Lieder zum Abspielen zu finden und diese dann an die eigentliche Music Player-Anwendung zu schicken, wenn der Anwender die Rückseite mit dem FERTIG-Button wieder umklappt.

³ Zumindest in den Betaversionen von 3.0 ist das „Scrubben“ im iPod Player etwas sprunghafter als bei einer lokalen Datei, die man mit dem `AVAudioPlayer` abspielt (siehe Abschnitt 16.6, *Audiodaten mit dem AVFramework abspielen*, auf Seite 377).

Grenzen des Media Player-Framework

Auch wenn das Media Player-Framework eine ganze Menge Funktionalität zu bieten hat, gibt es deutliche Grenzen beim Suchen in der Bibliothek und beim Nutzen dieses Suchergebnisses. Hier sind ein paar wichtige aufgezählt:

- Kein Zugriff auf Videos: Videos, wie zum Beispiel Filme, TV-Folgen und Video-Podcast können nicht über die Abfrage-API oder den Picker gefunden werden, und Sie können sie auch nicht von Ihrer Anwendung aus beeinflussen, da die iPod-Anwendung das Abspielen solcher Objekte beendet, wenn man sie verlässt.
- Kein Zugriff auf Audiodaten: Die Klasse `MPMediaItem` ermöglicht Ihnen nicht den Zugriff auf die Audiodateien selbst, ihre komprimierten Daten oder den dekodierten Audiostream.
- Kein schreibender Zugriff: Sie können Audiometadaten nicht bearbeiten, löschen oder ergänzen.
- Kein Zugriff auf beliebige Metadaten: Metadatenstandards wie ID3 für MP3-Dateien und User Data Atoms in AAC ermöglichen beliebige, selbst definierte Metadaten, aber das Media Player-Framework ermöglicht nur den Zugriff auf die üblichen Metadaten.

Aufbau der Rückseite

Dem `MPMusicPlayerController` – egal ob er den System-Media Player (`[MPMusicPlayerController iPodMusicPlayer];`) oder einen für Ihre Anwendung allozierten (`[MPMusicPlayerController applicationMediaPlayer];`) repräsentiert – kann man neue Audioobjekte zum Abspielen in Form einer *Queue* übergeben. Sie können dem Player eine `MPMediaItemCollection` schicken, die Sie aus einer Abfrage erhalten oder selbst aufbauen, oder Sie nutzen eine `MPMediaQuery`, die Sie ausführen und dann abspielen lassen. Der Anwender wählt die Ergebnisse dieser Abfragen aus, woraus dann ein Array mit den gewählten Elementen entsteht. Diese Elemente werden auf dem Bildschirm als `UITableView` angezeigt. Wenn der Anwender schließlich auf FERTIG tippt, schicken wir die Elemente an den nativen iPod-Player. Kehren wir dann zur Haupt-View zurück, sollten wir sehen, wie das erste vom Anwender gewählte Lied abgespielt wird.



Abbildung 15.3: Abfragen der iPod-Bibliothek mit einem UISearch-DisplayController

Das Projekt-Template mit der Utility Application hat schon eine ganz einfache Rückseite für uns vorbereitet, bestehend aus den Klassen `FlipsideView` und `FlipsideViewController` sowie einer Datei `FlipsideView.xib`: Die Funktionalität zum Umklappen ist bereits korrekt verdrahtet, so dass wir direkt damit beginnen können, unser GUI zu bauen. Öffnen Sie die `FlipsideView.xib` im Interface Builder, finden Sie dort die Standardelemente vor: eine Navigationsleiste mit einem FERTIG-Button und ein Navigationselement namens „Title“. Das Folgende fügen wir noch hinzu:

- Eine mehrteilige Steuerung (*Segmented Control*) direkt unter der Navigationsleiste mit den drei Segmenten Titel, Interpret und Album.
- Darunter ein Suchfeld (Search Bar) und ein zugehöriger Search Display Controller. Das ist ein neues und bemerkenswertes Mitglied der IB-Bibliothek im iPhone OS 3.0. Früher konnten Sie ein Suchfeld hinzufügen, mussten aber das Verhalten komplett selbst verdrahten. Dieses neue Feature ergänzt Ihre View nicht

nur um eine UISearchBar, sondern es erstellt auch noch einen UISearchDisplayController, mit dem die Search Bar schon verbunden ist. Wenn das Suchfeld den Fokus erhält, blendet es parallel zur Tastatur eine Tabelle mit Suchergebnissen über der aktuellen View ein. In Abbildung 15.3 sehen Sie, wie das in unserem vollständigen Beispiel aussieht – mit den Suchergebnissen in der Tabelle und der Tastatur über dem Rest der View.⁴

- Füllen Sie den Rest der rückseitigen View mit einer UITableView für die vom Anwender zu füllende Liste der Lieder.

Wie üblich müssen wir ein paar IBOutlets deklarieren. Fügen Sie FlipsideViewController.h die folgenden Instanzvariablen hinzu und deklarieren Sie sie dann mit @property und @synthesize als Eigenschaften:

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.h

```
UISegmentedControl *searchTypeControl;
UISearchDisplayController *searchController;
UITableView *queueTable;
```

Wir müssen auch die drei Protokolle deklarieren, die Sie nutzen wollen: Delegates für Tabelle und Suchfeld sowie die Data Source für die Tabelle:

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.h

```
@interface FlipsideViewController : UIViewController <UITableViewDataSource,
    UITableViewDelegate, UISearchBarDelegate> {
```

Verbinden Sie im IB die Outlets vom File's Owner mit den entsprechenden Elementen im Nib. Sie werden auch delegate und dataSource der Tabelle mit File's Owner verbinden wollen. Beachten Sie übrigens, dass der UISearchDisplayController bereits selbst vier Verbindungen zu File's Owner hergestellt hat: delegate, searchContentsController, searchResultsDataSource und searchResultsDelegate. Offensichtlich werden wir noch einiges tun müssen, um diesen Rollen zu Diensten sein zu können, aber zumindest die Verbindungen sind schon hergestellt.

Ändern Sie schließlich noch den Titel der Navigationsleiste auf „Neue Abfrage“. Wenn Sie mit dem Erstellen des GUI fertig sind, sollte es so aussehen wie das in Abbildung 15.4.

⁴ Für diesen Screenshot haben wir allerdings dafür gesorgt, dass schon beim Eintippen mit der Suche begonnen wird, während die endgültige App erst dann mit der Suche beginnt, wenn der Anwender auf SUCHEN tippt.



Abbildung 15.4: Layout des GUI für das Erstellen von Medien-Queues

Die Medienbibliothekssuche implementieren

In unserem Such-GUI sollte der Anwender Text ins Suchfeld eingeben und die Medienbibliothek damit nach Liedern, Interpreten oder Alben mit diesem Text in den entsprechenden Metadateneigenschaften durchsuchen. Die Ergebnisse sollten dann in der eingeblendeten Tabelle des Search Controller angezeigt werden. Wählt der Anwender ein Lied aus, fügen wir es einer Queue mit Liedern hinzu, die dann letztlich an die Media Player Application geschickt werden. Wir werden zwei Arrays als Instanzvariablen deklarieren müssen: ein Array mit dem aktuellen Suchergebnis und ein veränderbares Array, in das die Auswahl des Benutzers eingefügt wird.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.h

```
NSArray *searchResults;
NSMutableArray *newPlaybackQueue;
```

Zudem leeren wir die Queue immer dann, wenn wir zur View dieser Rückseite kommen, da eine vorige Queue schon an den System-Media Player geschickt worden ist. Das können Sie in `viewDidLoad` erledigen.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
[newPlaybackQueue release];
newPlaybackQueue = [[NSMutableArray alloc] init];
```

Diese Arrays werden von den Data Sources beider Tabellen genutzt. Tatsächlich haben wir hier eine recht interessante Situation: Unser View-Controller ist gleichzeitig Delegate und Data Source für zwei Tabellen – die Tabelle mit den Suchergebnissen, die vom `UISearchBarDelegate` bereitgestellt wird, und die `Queue`-Tabelle, die wir selbst in die View eingefügt haben. Aber wie Sie sehen werden, ist das kein Problem.

Um die Suche zu ermöglichen, müssen wir einige der vom Protokoll `UISearchBarDelegate` deklarierten Delegate-Methoden implementieren. Sie können sich dazu entscheiden, durch das Implementieren von `searchBar:textDidChange:` bei jeder neuen Tasteneingabe eine Suche anzustoßen oder erst dann loszulegen, wenn der Such-Button angeklippt wird – mit `searchBarSearchButtonClicked:`. Beim Schreiben dieses Kapitels haben wir festgestellt, dass die Performance beim Suchen in einer Bibliothek mit gerade einmal 500 Liedern auf einem iPhone der ersten Generation nicht wirklich toll ist, daher empfehlen wir, den Such-Button zu nutzen und erst dann die Suche anzustoßen. Sie werden auch die Delegate-Methode implementieren wollen, die beim Abbruch der Suche durch den Anwender aufgerufen wird.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
- (void)searchBarSearchButtonClicked:(UISearchBar *)searchBar {
    [self updateSearchResults];
    [searchController.searchResultsController reloadData];
}

- (void) searchBarCancelButtonClicked: (UISearchBar *) searchBar {
    searchResults = nil;
}
```

Das ist offensichtlich eine einfache Implementierung von `searchBarSearchButtonClicked:`, da sie die eigentliche Abfrage an eine noch zu schreibende Methode `updateSearchResults` abgibt. Der Grund dafür ist, dass Sie dann eine neue Suche mit den gleichen Daten durchführen wollen, wenn der Anwender auf die Steuerleiste (`Segmented Control`) tippt, um den Abfragetyp zu ändern.

Lassen Sie uns nun schauen, wie die Abfrage ausgeführt wird. Das `Media Player-Framework` stellt eine Klasse `MPMediaQuery` bereit, um die Medienbibliothek abzufragen. Dabei gibt es eine Reihe von „vorgefertigten“ Abfragen, die als Klassenmethoden bereitgestellt werden, wie zum Beispiel `songsQuery` und `podcastsQuery`, die alle Elemente eines bestimmten Typs zurückgeben.

Zu einer dieser vorgefertigten Abfragen oder auch zu einer leeren Abfrage können Sie *Filterprädikate* hinzufügen, um die Suchergebnisse einzuschränken. Die meisten für den Benutzer sichtbaren Eigenschaften, wie zum Beispiel Liedtitel und Interpretennamen, können so als Suchkriterium genutzt werden (schauen Sie sich die Dokumentation zu `MPMediaItem` an, um herauszufinden, welche Eigenschaften als **filterable** gekennzeichnet sind.)

Um die Medienbibliothek zu durchsuchen, beginnen wir mit einer `MPMediaQuery` für alle Lieder und filtern dann die heraus, die den im Suchfeld eingegebenen Text enthalten – entweder im Titel oder im Interpreten bzw. Album-Namen. Das können wir erreichen, indem wir abhängig vom Wert des Segmented Control per **switch** vorgehen und ein `MPMediaPropertyPredicate` erstellen, das wir der Abfrage hinzufügen. Mit diesem ergänzten Prädikat können wir dann die passenden Objekte als `NSArray` aus der Eigenschaft `items` der Abfrage übernehmen.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
- (void) updateSearchResults {
    NSString *searchText = searchController.searchBar.text;
    MPMediaQuery *query = [MPMediaQuery songsQuery];
    switch (searchTypeControl.selectedSegmentIndex) {
        case 0: {
            MPMediaPropertyPredicate *titlePredicate =
                [MPMediaPropertyPredicate predicateWithValue:searchText
                 forProperty:MPMediaItemPropertyTitle
                 comparisonType: MPMediaPredicateComparisonContains];
            [query addFilterPredicate:titlePredicate];
            break;
        }

        case 1: {
            MPMediaPropertyPredicate *artistPredicate =
                [MPMediaPropertyPredicate predicateWithValue:searchText
                 forProperty:MPMediaItemPropertyArtist
                 comparisonType: MPMediaPredicateComparisonContains];
            [query addFilterPredicate:artistPredicate];
            break;
        }

        case 2: {
            MPMediaPropertyPredicate *albumPredicate =
                [MPMediaPropertyPredicate predicateWithValue:searchText
                 forProperty:MPMediaItemPropertyAlbumTitle
                 comparisonType: MPMediaPredicateComparisonContains];
            [query addFilterPredicate:albumPredicate];
            break;
        }

        default: {
            // Unbekannter Segmenttyp – einfach alles zurückgeben
            query = [MPMediaQuery songsQuery];
        }
    }
}
```

```

[searchResults release];
searchResults = query.items;
[searchResults retain];
}

```

Wie Sie sehen können, erstellen Sie ein `MPMediaPropertyPredicate` mit Klassenmethoden, die einen Filterwert und einen optionalen `comparisonType` enthalten. Letzterer kann entweder ein `MPMediaPredicateComparisonEqualTo` oder ein `MPMediaPredicateComparisonContains` sein. Textuelle Vergleiche ignorieren Groß- und Kleinschreibung, und es gibt auch keinen „legalen“ Weg, das zu ändern.

Beachten Sie, dass die Ergebnisse unsortiert geliefert werden. Sie können eine der diversen Sortiermethoden von `NSArray` nutzen oder die Eigenschaft `groupingType` der Abfrage setzen und dann die Eigenschaft `collections` auslesen, die die Ergebnisse als `Array` aus `MPMediaItemCollections` zurückgibt – gruppiert nach dem angegebenen Gruppierungstyp. Um dieses Beispiel einfach zu halten, lassen wir die Suchergebnisse unsortiert.

Unsere Methode `updateSearchResults` befüllt also `searchResults` mit einem `Array` aus `MPMediaItems`, die dem in die Search Bar eingegebenen String entsprechen. Wir wollen diese Suche erneut ausführen, wenn der Anwender durch Antippen des Segmented Control den Suchtyp ändert (also zum Beispiel von der Liedsuche zur Interpreten- oder Albumsuche wechselt). Das ist einfach: Deklarieren Sie in `FlipsideViewController.h` die Methode `-(IBAction) searchTypeChanged;`, verbinden Sie das Event „Value Changed“ des Segmented Control mit dieser Action und schreiben Sie eine Implementierung, die einfach unser `updateSearchResults` aufruft und die Tabelle des Search Controller auffrischt.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```

-(IBAction) searchTypeChanged {
    [self updateSearchResults];
    if (searchController.active == YES) {
        [searchController.searchResultsTableView reloadData];
    }
}

```

Die Tabellen für die Suchergebnisse und die Medien-Queue implementieren

Wir haben jetzt die Suche durchgeführt, aber die Ergebnisse liegen noch im `Array searchResults`. Wir müssen die Methoden von `UITableViewDelegate` und `UITableViewDataSource` implementieren, um die Daten dem Anwender zu präsentieren.

Wie schon erwähnt, ist der `FlipsideViewController` als Delegate mit zwei verschiedenen Tabellen verbunden – der zum `UISearchDisplayController` gehörenden Tabelle mit den Suchergebnissen und unserer eigenen Tabelle, die wir für die Queue nutzen wollen. Diese Queue baut der Anwender auf, indem er Suchergebnisse auswählt. Unsere Delegate-Methoden müssen also darauf achten, welche Tabelle sie aufruft.

Lassen Sie uns mit der Delegate-Methode `tableView:didSelectRowAtIndexPath:` beginnen. Wenn ein Anwender auf ein Element in der Tabelle mit den Suchergebnissen tippt, wollen wir das ausgewählte `MPMediaItem` holen und es in das Array `newPlaybackQueue` einfügen. Dann aktualisieren wir die Queue-Tabelle und werfen das Such-GUI.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
-(void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    if (tableView != queueTable) {
        // Element aus dem Suchergebnis zur Queue hinzufügen
        [newPlaybackQueue addObject:
         [searchResults objectAtIndex:indexPath.row]];
        [queueTable reloadData];
        [tableView deselectRowAtIndexPath:indexPath animated:NO];
        [searchController setActive:NO animated:YES];
        searchResults = nil;
    } else {
        [tableView deselectRowAtIndexPath:indexPath animated:NO];
    }
}
```

Beachten Sie, dass das Auswählen einer Zeile in der Queue-Tabelle zu keiner Reaktion führt.

Damit ist die wichtigste Aktion erledigt: ein ausgewähltes Suchergebnis in das Array mit der neuen Abspiel-Queue zu übertragen. Jetzt müssen wir noch die anderen wichtigen Tabellenmethoden bereitstellen – die Anzahl an Abschnitten und die Größe der Tabellen.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
-(NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

-(NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
    if (tableView == queueTable) {
        return [newPlaybackQueue count];
    } else {
        return [searchResults count];
    }
}
```

Mit einer ähnlichen Strategie können wir auch einen Abschnitts-Header für die Queue-Tabelle bereitstellen, um die Anzahl an Elementen anzuzeigen, die der Anwender hinzugefügt hat.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
-(NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section {
    if (tableView == queueTable) {
        return [NSString stringWithFormat:
            @"Queue (%d Lieder)" , [newPlaybackQueue count]];
    } else {
        return nil;
    }
}
```

Schließlich müssen wir noch die Felder bereitstellen. Die von Xcode schon vorbereitete einfache Implementierung für andere Tabellen (wie in Abschnitt 5.4, *Tabellenfelder*, auf Seite 90 gezeigt) reicht hier aus. Ändern Sie einfach den Stil in `UITableViewCellStyleSubtitle` und setzen Sie dann `textLabel` und `detailTextLabel` auf Eigenschaftswerte, die Sie aus dem „aktuellen“ `MPMediaItem` aus dem passenden Array ausgelesen haben (`newPlaybackQueue` für die `queueTable`, `searchResults` für die Tabelle des Such-Controllers).

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
-(UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell" ;
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleSubtitle
            reuseIdentifier:CellIdentifier] autorelease];
    }

    // Konfigurieren des Feldes
    MPMediaItem *anItem = nil;
    if (tableView == queueTable) {
        anItem = [newPlaybackQueue objectAtIndex:indexPath.row];
    } else {
        anItem = [searchResults objectAtIndex:indexPath.row];
    }
    cell.textLabel.text =
        [anItem valueForKeyProperty:MPMediaItemPropertyTitle];
    cell.detailTextLabel.text =
        [anItem valueForKeyProperty:MPMediaItemPropertyArtist];
    return cell;
}
```

Mit diesen Tabellenmethoden können Sie nun Suchergebnisse auswählen und eine Queue aufbauen (siehe Abbildung 15.5).



Abbildung 15.5: Eine Abspiel-Queue aus MPMediaItems erstellen

Eine Queue abspielen

Als Letztes nehmen wir jetzt noch die Queue mit den vom Anwender gewählten Liedern und schicken sie an die eingebaute Player-Anwendung. Das Template hat den FERTIG-Button schon mit der Methode `done` verbunden, die zurück zur Haupt-View schaltet, und wir können hier auch unseren Code unterbringen, um die Queue zu befüllen.

Wie wir schon vor Beginn des Aufbaus des rückseitigen GUI besprochen haben, besitzt der `MPMusicPlayerController` eine ganze Reihe von Methoden, denen man eine Queue mit abzuspielenden Objekten übergeben kann. Da wir die abzuspielenden Elemente schon gesammelt haben, rufen wir `setQueueWithItemCollection:` auf und übergeben eine `MPMediaItemCollection`, die wir aus unserem Array erstellt haben.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```
-(IBAction)done {
    if ([newPlaybackQueue count]) {
        MPMusicPlayerController *iPodController =
            [MPMusicPlayerController iPodMusicPlayer];
        [iPodController stop];
    }
}
```

```

MPMediaItemCollection *queueCollection =
    [MPMediaItemCollection collectionWithItems:
        newPlaybackQueue];
[iPodController setQueueWithItemCollection: queueCollection];
[iPodController play];
}
// Vom Xcode-Template vorausgefüllt.
[self.delegate flipsideViewControllerDidFinish:self];
}

```

15.4 In der iPod-Bibliothek stöbern

Im vorigen Abschnitt haben wir unsere eigene Suchoberfläche aufgebaut, um nach Elementen zu suchen, die wir der Abspiel-Queue hinzufügen wollten. Neben der MPMediaQuery stellt uns das Media Player-Framework eine weitere Möglichkeit bereit, Elemente in der Medienbibliothek zu finden: den MPMediaPickerController.

Dieser Picker ist ein UIViewController, dessen View abhängig von Ihrer Konfiguration fast genau so aussieht wie die der nativen iPod- oder Musikanwendung. Sie initialisieren sie mit einem Bit-Feld der anzuzeigenden MPMediaTypes. Um zum Beispiel ein Such-GUI zu erstellen, das nur Podcasts und Hörbücher anzeigt, würden Sie den Picker wie folgt erstellen:

```

MPMediaPickerController *pickerController =
    [[MPMediaPickerController alloc]
        initWithMediaTypes: MPMediaTypePodcast |
                           MPMediaTypeAudioBook];

```

Damit wird ein Tabbed GUI mit zwei Tabs erzeugt, jeweils für Podcasts und Hörbücher, die beide stark denen der eingebauten Anwendungen gleichen. Der größte Unterschied resultiert daraus, ob Sie dem Picker erlauben, eine Mehrfachauswahl zuzulassen. Setzen Sie diese Option auf **YES**, bekommen die Elemente ein +-Zeichen verpasst, damit der Anwender sie der Auswahl-Queue hinzufügen kann.

Lassen Sie uns eine Suchoption für das rückseitige GUI erzeugen. Fügen Sie einen Tab-Leisten-Button auf der rechten Seite des Navigationselements hinzu und benennen Sie ihn `Browse`. Deklarieren Sie in `FlipsideViewController.h` die Methode `-(IBAction) handleBrowseTapped`; und implementieren Sie sie wie folgt:

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```

-(IBAction) handleBrowseTapped {
    MPMediaPickerController *pickerController =
        [[MPMediaPickerController alloc]
         initWithMediaTypes: MPMediaTypeMusic];

    pickerController.prompt = @"Lieder zur Queue hinzufügen" ;
    pickerController.allowsPickingMultipleItems = YES;
    pickerController.delegate = self;

    [self presentViewController:pickerController animated:YES];
    [pickerController release];
}

```

Wir erzeugen zunächst den `MPMediaPickerController` und initialisieren ihn, um die Musik aus der Benutzerbibliothek anzuzeigen. Neben der Angabe einer verständlichen Beschreibung und der Entscheidung für oder gegen eine Mehrfachauswahl stellen wir einen Delegate bereit, der das Protokoll `MPMediaPickerControllerDelegate` implementiert. Da dies der `FlipsideViewController` selbst sein soll, müssen Sie darauf achten, dass Sie in der Header-Datei `MPMediaPickerControllerDelegate` der Liste der implementierten Protokolle hinzufügen. Mit dem so gesetzten Delegate können wir den Picker-Controller in einer modalen View anzeigen, die von unten her auf den Bildschirm geschoben wird (siehe Abbildung 15.6).

Tippt der Anwender auf FERTIG oder ABBRECHEN (muss nicht unbedingt vorhanden sein – abhängig von den Medientypen und der Mehrfachauswahleinstellung des Controllers), ruft der `MPMediaPickerController` die Delegate-Methode `mediaPicker:didPickMediaItems:` oder `mediaPickerDidCancel:` auf. Für unsere Zwecke müssen wir nur ausgewählte Elemente in die `newPlaybackQueue` einfügen und den modalen View-Controller verwerfen.

MediaLibrary/MusicLibraryClient/Classes/FlipsideViewController.m

```

-(void)mediaPicker: (MPMediaPickerController *)mediaPicker
    didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection {

    [self dismissModalViewControllerAnimated:YES];
    [newPlaybackQueue addObjectsFromArray:mediaItemCollection.items];
    [queueTable reloadData];
}

```



Abbildung 15.6: Die Medien-Bibliothek mit dem MPMediaPicker-Controller durchstöbern

```
-(void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker {
    [self dismissModalViewControllerAnimated:YES];
}
```

Damit erhalten Ihre Anwender die gleiche Kontrolle über die Musik wie in der iPod-Anwendung: sie können in der Musikbibliothek stöbern, suchen, eines oder mehrere Elemente zum Abspielen auswählen und sie auch abspielen – komplett mit Anzeige der Metadaten und Scrubbing. Was Sie damit anstellen, ist Ihre Sache. Statt nur einen anders aussehenden Player anzubieten, können Sie zum Beispiel auch die Lieblingsmusik des Anwenders (vielleicht basierend auf Eigenschaften wie `MPMediaItemPropertyPlayCount` oder `MPMediaItemPropertyRating`) als Hintergrundmusik in Ihrer Anwendung laufen lassen. Oder erstellen Sie eine Social Application, die die Bibliothek analysiert und sie mit der von anderen Anwendern vergleicht.⁵ Kombinieren Sie die Analyse der Musikbibliothek mit der Location-API, um nahe liegende Plattenläden und Veranstaltungsorte zu finden, an denen man die gewünschte Musik spielt und verkauft. Oder Sie schalten bestimmte Features Ihrer Anwendung nur dann frei, wenn der Anwender Interpreten und Lieder auf seinem iPhone hat, die Sie mögen.

⁵ Natürlich nur mit der Erlaubnis der Anwender, denn das „Freigeben“ von Anwenderdaten hat weitreichende Auswirkungen auf die Privatsphäre.

Kapitel 16

Audiodaten abspielen und aufzeichnen

Gestaltet sich die Videounterstützung im iPhone 3.0 recht zurückhaltend, ist die Audiounterstützung dagegen erstaunlich umfassend. Das Core Audio Framework bietet verschiedene Abstraktionsschichten an – von einer High-Level-API für Objective-C, die das Aufsetzen eines einfachen Audiorekorders oder eines Players, der mit Dateien in Ihrem *Documents*-Verzeichnis arbeitet, sehr einfach macht, bis hin zu Low-Level-C-APIs für das Streamen von Audio, durch die Sie direkt mit Raw-Audio-Samples arbeiten können.

Auch hier gilt wieder die 80-20-Regel: Ein Großteil der Entwickler wird eine kleine, aber entscheidende Untermenge dieses Framework nutzen. Wir werden uns in diesem Kapitel größtenteils auf die Objective-C-Klassen konzentrieren, die für die wichtigsten Aufgaben genutzt werden: das Aufzeichnen und Abspielen lokaler Dateien.

16.1 Einen AVAudioRecorder erstellen

In Zeiten vor dem iPhone SDK 3.0 mussten Entwickler, die Audiodaten aufzeichnen wollten, mit der Audio Queue Services-API arbeiten (die später noch in Abschnitt 17.3, *Überblick über Core Audio*, auf Seite 397 beschrieben wird), einer prozeduralen C-API, die über Streams arbeitet. Sie erhielten Callbacks mit neu erfassten Audio-Samples und mussten diese mit einer anderen C-API für das Schreiben von Audiodateien abspeichern. Viele Leute haben sich darüber beschwert, dass das für solche eine einfache Aufgabe viel zu kompliziert sei.

Als Reaktion darauf bietet das iPhone SDK 3.0 nun eine viel einfacher zu verwendende Option an: die Klasse `AVAudioRecorder` als Teil des `AVFoundation`-Framework. Diese Klasse ermöglicht es Ihnen, ein Recorder-Objekt mit einer Ziel-URL, die das Ziel der Aufnahme sein soll, und ein paar Einstellungen zu erzeugen und dann einfach mit dem Aufzeichnen der Audiodaten zu beginnen. Dazu muss man lediglich Methoden wie `record`, `pause` und `stop` aufrufen. Natürlich ist es dann doch nicht so einfach – wie immer bei der Audioverarbeitung. Denn wir werden sehen, was wir bei den Optionen für die Audiokomprimierung berücksichtigen müssen, und zudem gibt es nur wenige Kombinationen aus Einstellungen, die tatsächlich funktionieren, aber für die meisten Leute ist dieser Weg trotzdem viel einfacher als der frühere.

Die Beispielanwendung aufsetzen

Um mit den Aufzeichnungs- und Wiedergabefähigkeiten der `AVFoundation` zu spielen, werden wir eine Beispielanwendung bauen, die Audiodaten in Dateien aufzeichnen kann (über das Mikrofon des Telefons oder des Headsets) und diese Dateien dann auch wieder abspielt. Wir werden dem Anwender eine Reihe von möglichen Audioformaten zur Auswahl geben, so dass wir die Vor- und Nachteile der einzelnen Formate kennenlernen können.

Die Beispielanwendung, die wir vorbereitet haben, ist eine der am weitesten ausgearbeiteten in diesem Buch. Wir beschreiben ihre wichtigsten Elemente, aber da wir den Fokus auf der Audioverarbeitung halten und keinen vierseitigen Ausflug in den Interface Builder unternehmen wollen, ist dies ein Fall, in dem es wirklich sinnvoller ist, den herunterladbaren Beispielcode zu nutzen, in dem wir `AudioRecorderPlayerStarter` als Basisprojekt mit dem fertig gebauten und verdrahteten GUI anbieten. Wenn Sie diese Anwendung tatsächlich von Grund auf selbst bauen wollen, sollten Sie bedenken, dass wir davon ausgehen, dass Sie alle `#imports`, Eigenschaften, Hilfsmethoden, `IBActions` und `IBOutlets`, die wir nicht explizit beschrieben haben, selber erzeugen können.

Für unser Beispiel werden wir die Tab-basierte Anwendung `AudioRecorderPlayer` mit zwei Tabs erstellen – einen für das Aufzeichnen und einen für das Wiedergeben der aufgezeichneten Dateien. Das bedeutet, dass wir zwei View-Controller `RecordViewController` und `PlayViewController` schreiben und sie als Klassen für die beiden View-Controller-Objekte einsetzen werden, die das Template für Sie in `MainView.xib` angelegt hat. Im `Tab Bar`-Template befindet sich die erste View standardmäßig in `MainView.xib`; im herunterladbaren Beispiel

haben wir uns jedoch dazu entschieden, eine `RecordView.xib` zu erstellen und den View-Controller des ersten Tabs daraus laden zu lassen. Auch haben wir das Refactoring-Tool genutzt, um den `FirstViewController` in `RecordViewController` umzubenennen.

In Abbildung 16.1 sehen Sie, was wir für die Haupt-View in `RecordView.xib` eingestellt haben. Die Schlüsselemente sind ein `filenameField`, um einen Dateinamen einzugeben, unter dem die Aufzeichnung gespeichert werden soll, ein `UISegmentedControlFormatSegments`, in dem die Wahl zwischen PCM- und Encoded-Formaten getroffen werden kann, ein `formatButton`, um modale Views mit Details zu den Formaten anzuzeigen, ein `currentTimeLabel`, ein deaktivierter `recordPauseButton` und ein `stopButton`. Darunter finden sich zwei Views für den `leftLevelMeter` und den `rightLevelMeter`, die später durch eine eigene Klasse implementiert werden. Natürlich muss all das als `IBOutlet`s deklariert und im Interface Builder verbunden sein.

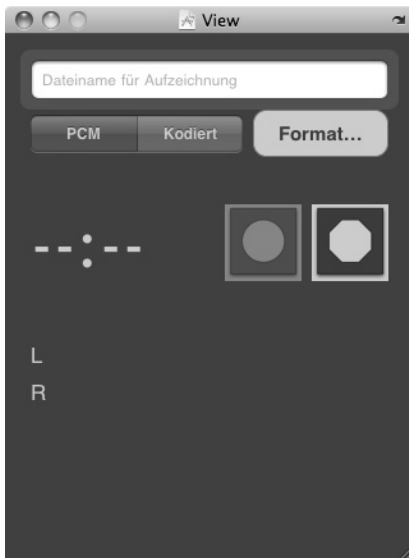


Abbildung 16.1: Layout für den ersten Tab des `AudioRecorderPlayer`

Die Klasse `AVAudioRecorder` findet sich im optionalen `AVFoundation` Framework, daher müssen Sie sie Ihrem Frameworks-Ordner hinzufügen und in Ihrer Header-Datei die Zeile `#import <AVFoundation/AVFoundation.h>;` eintragen. Die Klasse besitzt nur eine echte Initializer-Methode `initWithURL:settings:error:`, der eine Datei (als `NSURL`) übergeben wird, in die die Klasse schreibt, ein `NSDictionary` mit Formateinstellungen und einen Zeiger auf ein `NSError`-Objekt, um Fehler erfassen zu können, auf-

grund derer das Recorder-Objekt nicht erzeugt werden konnte. Wir brauchen sowohl einen Ort, an den die Daten geschrieben werden können, als auch die Einstellungswerte, um das Recorder-Objekt erzeugen zu können. Wir können den Recorder entweder erzeugen, wenn der Anwender auf den RECORD-Button tippt, oder immer dann, wenn sich der Name beziehungsweise Ort der Datei oder die Einstellungen ändern. Lassen Sie uns die zweite Variante nutzen: Erhalten wir beim Einrichten des Recorder einen Fehler gemeldet (zum Beispiel aufgrund ungültiger Formateinstellungen), können wir frühzeitig darauf aufmerksam machen, so dass der Anwender nicht erst beim Versuch, aufzuzeichnen, aufgehalten wird. Wir rufen also immer dann, wenn etwas am Textfeld oder an den Formateinstellungen geändert wurde, die Methode `createAVAudioRecorder` auf, um ein neues Objekt `audioRecorder` zu erzeugen – unsere Instanz eines `AVAudioRecorder`.

Lassen Sie uns mit dem Textfeld beginnen. Setzen Sie seinen Delegate auf den `RecordViewController` und stellen Sie die normale Implementierung von `textFieldShouldReturn` bereit, durch die die virtuelle Tastatur mithilfe von `[textField resignFirstResponder];` freigegeben wird. Zudem wollen wir `createAVAudioRecorder` aufrufen, wenn der Anwender mit dem Bearbeiten fertig ist. Auch können wir das Feld bereits überprüfen. Es gibt eine Reihe von Einschränkungen bei Audiodateiformaten, von denen wir einige in den nächsten beiden Abschnitten: *Abschnitt 16.2, Unkomprimierte Audioformate* auf Seite 359 und *Abschnitt 16.3, Kodierte Audioformate* auf Seite 364, behandeln werden. So müssen zum Beispiel unkomprimierte PCM-Daten in einer `.wav`-Datei im Format Little Endian gespeichert sein, während dieselben Daten in einer `.aif`-Datei im Format Big Endian vorliegen müssen. Und der *einzig*e gültige Inhalt in einer `.mp3`-Datei sind MPEG-1-Layer-3-Daten. Wir ersparen uns eine Menge Ärger, wenn wir das Core Audio-Format (`.caf`) als Standard nutzen, da diesem der Inhalt egal ist. Alle Core Audio-Komponenten können den Inhalt einer `.caf`-Datei abspielen. Aus diesem und noch anderen Gründen ist es auch das empfohlene Format für Audiodaten auf dem iPhone, und wir werden dies durch unsere Standardwahl `.caf` honorieren, wenn keine Dateierweiterung angegeben wurde.

MediaPlayback/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(void)textFieldDidEndEditing:(UITextField *)textField {
    [textField resignFirstResponder];
    // Prüfen, ob es eine Dateierweiterung gibt
    if ([[textField.text pathExtension] length] == 0)
        textField.text =
            [NSString stringWithFormat: @"%@.caf" , textField.text];
    [self createAVAudioRecorder]; }
```


Um die NSURL zu erhalten, die zum Erstellen des AVAudioRecorder notwendig ist, brauchen wir den Pfad auf das Verzeichnis Documents. Hier eine Hilfsmethode, die eine Variable `_documentsPath` erst dann instantiiert, wenn sie benötigt wird:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(NSString*) documentsPath {
    if (!_documentsPath) {
        NSArray *searchPaths =
            NSSearchPathForDirectoriesInDomains
            (NSDocumentDirectory, NSUserDomainMask, YES);
        _documentsPath = [searchPaths objectAtIndex: 0];
        [_documentsPath retain];
    }
    return _documentsPath;
}
```

Damit können wir mit unserer Methode `createAVAudioRecorder` beginnen. Wir erstellen eine NSURL, die den angegebenen Dateinamen im Verzeichnis Documents repräsentiert.

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(NSError*) createAVAudioRecorder {
    [audioRecorder release];
    audioRecorder = nil;

    NSString *destinationString = [[self documentsPath]
        stringByAppendingPathComponent:filenameField.text];
    NSURL *destinationURL = [NSURL fileURLWithPath: destinationString];
```

Jetzt wollen wir uns das zweite Argument `settings` der Methode `initWithURL:settings:error:` der Klasse `AVAudioRecorder` anschauen. Die Dokumentation beschreibt es als `NSDictionary` mit Schlüssel/Wert-Paaren mit vier Schlüsselsets: allgemeine Einstellungen zum Audioformat, zu Linear PCM, zum Encoder und zur Samplerate.

16.2 Unkomprimierte Audioformate

Wir wollen mit der linearen Pulse Code Modulation (PCM) beginnen. Das meinen wir normalerweise mit „unkomprimierte Audiodaten“. Um sie zu verstehen, sollten wir uns erst einmal überlegen, wie digitale Audioverarbeitung funktioniert. Schall besteht aus oszillierenden Druckwellen, die sich durch ein Medium bewegen und dabei in einem Spektrum von Frequenzen liegen, die ein Mensch bemerken kann. Wenn Sie sprechen, erzeugt Ihr Kehlkopf Schallwellen, die sich durch die Luft bewegen und das Trommelfell eines anderen zum Schwingen bringen, womit dieser den Schall hören kann.

Analoge Soundsysteme, wie zum Beispiel das klassische Radio, Telefone und Schallplattenspieler, übertragen diesen Schall als elektrische Signale. Die Vibrationen an einem Mikrofon werden als elektrisches Signal über einen Draht übermittelt, das wiederum einen Lautsprecher zum Schwingen bringt, um den Schall zu reproduzieren. Bei digitaler Audioverarbeitung stellen wir eine Schallwelle dar, indem wir die Amplitude des Signals Tausende Male pro Sekunde abtasten und die Signalstärke als numerischen Wert umsetzen. Mit *Pulse Code Modulation* meinen wir nun die Umsetzung einer Schallwelle in eine Reihe von Abtastwerten, wobei *Linear PCM* bedeutet, dass die Abtastwerte direkt proportional zur Signalamplitude sind (im Gegensatz zu einer anderen Relation, wie zum Beispiel einem logarithmischen Verhältnis). So wird zum Beispiel Audio in „CD-Qualität“ mit 44.100 Samples pro Sekunde erfasst¹ und in 16-Bit-Werten gespeichert.

Linear („samplen“)PCM hat den Vorteil, dass sie universell einsetzbar ist – jedes digitale Audiosystem basiert darauf – und dass es die höchstmögliche Qualität besitzt. Erzeugen oder bearbeiten Sie Audiodaten, werden Sie in Ihrem gesamten Ablauf mit PCM arbeiten und sie erst am Ende in ein komprimiertes Format konvertieren wollen. PCM klingt also toll, aber die Daten werden wirklich richtig groß.

Bei unserer Recorder-Anwendung wollen wir zunächst mit dem Aufzeichnen in PCM beginnen. Im GUI haben wir ein Segmented Control, um zwischen „PCM“ und „Encoded“ zu wechseln, und einen FORMAT-Button. Dahinter steckt die Idee, aufgrund der vielen verschiedenen Einstellungen für die Audioformate eine komplett neue View anzuzeigen, in der der Anwender seine Konfiguration für das gewählte Format vornehmen kann. Wir erstellen getrennte Views mit den dazugehörigen View-Controllern für jede Art von Format: `PCMSettingsViewController` und `EncodingSettingsViewController`. Erzeugen Sie diese leeren Klassen, verpassen Sie dem `RecordViewController` zwei `IBOutlet`s, wechseln Sie dann in `RecordView.xib` und ziehen Sie die beiden View-Controller-Objekte in das .nib-Dokument. Im Identity Inspector ändern Sie dann ihre Klassen in die von Ihnen erzeugten Klassen. Jetzt sollten Sie die Outlets mit diesen neuen VC-Objekten verbinden können. Erstellen Sie zudem .nib-Dateien für die Views – `PCMSettingsView.xib` und `EncodingSettingsView.xib` – und verwenden Sie den Properties Inspector, um jeden View-Controller mit seinem View-Nib zu verbinden.

1 44.100 ist keine magische Zahl. Das *Nyquist-Shannon-Abtasttheorem* besagt, dass wir Frequenzen reproduzieren können, indem wir mit einer Rate abtasten, die doppelt so hoch ist wie die höchste Frequenz, die wir reproduzieren müssen. Da das menschliche Gehör im Allgemeinen bei etwa 20.000 Hz seine obere Grenze hat, müssen wir bei einer doppelt so hohen Rate abtasten.

Als Erstes müssen wir jetzt also die entsprechende Konfigurations-View anzeigen, wenn der Anwender auf **FORMAT** tippt. Das ist nicht schwer, wenn wir es im Event-Handler machen, der mit diesem Button verbunden ist:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(IBAction) handleFormatButtonTapped {
    if (formatSegments.selectedSegmentIndex == 0) {
        [self presentModalViewController:pcmSettingsViewController
            animated:YES];
    } else {
        [self presentModalViewController:encodingSettingsViewController
            animated:YES];
    }
}
```

Hat der Anwender PCM gewählt und tippt er nun auf den **FORMAT**-Button, wird er zum `pcmSettingsViewController` geleitet, der die PCM-Konfigurations-View einblendet, die in `PCMSettingsView.xib` definiert ist und mehr oder weniger wie in Abbildung 16.2 aussieht.

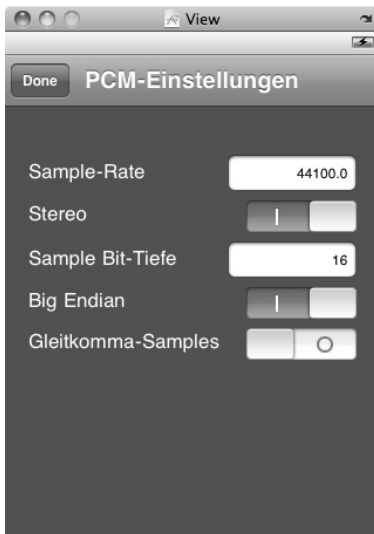


Abbildung 16.2: Layout für die PCM-Konfigurations-View

Die fünf Felder in dieser View ergeben sich aus den Einstellungen, die in der Dokumentation zum `AVAudioRecorder` definiert sind. Die ersten beiden sind allgemeine Einstellungen: die Abtastrate und die Anzahl der Kanäle (die wir durch einen „Stereo“-Schalter umsetzen). Die Schlüssel für diese Einstellungen sind `AVSampleRateKey` und `AVNumberOfChannelsKey`. Eine Stereounterstützung ist nett, allerdings sind beide Standardeingabegeräte für das iPhone – das Telefonmikro und das am Headset – nur zum Aufzeichnen von Monodaten geeignet.

Die nächsten drei Einstellungen sind PCM-spezifisch. Bei der „Sample bit depth“ kann der `AVLinearPCMBitDepthKey` einen von vier möglichen Werten haben: 8, 16, 24 oder 32. 16 ist eine gute Standardwahl, da diese intern im „kanonischen“ Audioformat von Core Audio verwendet wird. Der nächste Switch wird für den `AVLinearPCMBigEndianKey` genutzt, der festlegt, wie Werte größer als 8 Bit zu interpretieren sind. Wie schon erwähnt, können WAV-Dateien nur Little Endian, AIFFs nur Big Endian und CAFs beides nutzen. Eine ausgefeiltere Anwendung setzt den Wert dieses Switchs vielleicht schon abhängig vom ausgewählten Dateityp. Schließlich gibt es noch einen Switch für `AVLinearPCMBIsFloatKey`, mit dem angegeben wird, ob die Samples Integer- oder Gleitkommawerte sind. Integer PCM wird häufiger genutzt und ist auf dem iPhone OS das Standardformat.

Im PCM-Konfigurations-View-Controller brauchen wir also Eigenschaften für die fünf UI-Controls: `sampleRateField`, `stereoSwitch`, `sampleDepthField`, `bigEndianSwitch` und `floatingSamplesSwitch`. Sie werden auch eine `IBOutlet`-Eigenschaft für den `RecordViewController` nutzen wollen, dessen Verbindung Sie im IB herstellen können.

Diese letzte Eigenschaft brauchen wir, wenn der Anwender auf den FERTIG-Button tippt.² Sie müssen dann den `RecordViewController` anweisen, einen `AVAudioRecorder` zu erzeugen, der auf den geänderten Werten in diesem modalen View-Controller basiert.

MediaPlayer/AudioRecorderPlayer/Classes/PCMSettingsViewController.m

```
-(IBAction) handleDoneTapped {
    [self dismissModalViewControllerAnimated:YES];
    [recController createAVAudioRecorder];
}
```

Zurück im `RecordViewController`, können wir mit der Implementierung von `createAVAudioRecorder` fortfahren und dabei die Werte aus diesem `PCMSettingsViewController` verwenden:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
1 NSMutableDictionary *recordSettings =
2   [[NSMutableDictionary alloc] initWithCapacity:10];
```

2 Wenn Sie eine Navigationsleiste und einen Tab-Leisten-Button nutzen, wie wir es hier getan haben, haben Sie eine `selector`-Action im Connections Inspector des IB. Ziehen Sie diese bei gedrückter `[Ctrl]`-Taste auf die Methode `handleDoneTapped` des File's Owner, wandelt sich die Verbindung von „selector“ in „handleDoneTapped.“ Das entspricht der Standard-Flipside-View, die vom *Utility Application*-Template von Xcode bereitgestellt wird.

```

3  if (formatSegments.selectedSegmentIndex == 0) {
4      // PCM-Einstellungen müssen neu geladen werden, um die Felder
      // zu lesen
5      pcmSettingsViewController.view;
6      [recordSettings setObject:
7      [NSNumber numberWithInt: kAudioFormatLinearPCM] forKey:
8          AVFormatIDKey]
9      float sampleRate =
10         [pcmSettingsViewController.sampleRateField.text floatValue];
11      [recordSettings setObject:
12      [NSNumber numberWithFloat:sampleRate] forKey: AVSampleRateKey];
13      [recordSettings setObject:
14      [NSNumber numberWithInt:
15          (pcmSettingsViewController.stereoSwitch.on ? 2 : 1)]
16          forKey:AVNumberOfChannelsKey];
17      int bitDepth =
18         [pcmSettingsViewController.sampleDepthField.text intValue];
19      [recordSettings setObject:
20      [NSNumber numberWithInt:bitDepth] forKey:AVLinearPCMBitDepthKey];
21      [recordSettings setObject:
22      [NSNumber numberWithBool:
23          pcmSettingsViewController.bigEndianSwitch.on]
24          forKey:AVLinearPCMBIsBigEndianKey];
25      [recordSettings setObject:
26      [NSNumber numberWithBool:
27          pcmSettingsViewController.floatingSamplesSwitch.on]
28          forKey:AVLinearPCMBIsFloatKey];
29  }

```

Bevor wir uns den Schlüssel/Wert-Paaren zuwenden, schauen Sie sich Zeile 5 an. Diese Zeile kümmert sich um ein kleines Problem beim Laden des Nib: Wir wollen Werte aus der View für die PCM-Konfiguration übernehmen, wobei es sich natürlich durchaus auch um die Standardwerte handeln kann, die im IB definiert worden waren. Aber wenn der Anwender diese Konfigurations-View nie angezeigt hat (zum Beispiel rufen wir `createAVAudioRecorder` auf, weil der Anwender den Dateinamen geändert hat), ist keiner Eigenschaft ein Wert zugewiesen, da die View noch gar nicht geladen ist. In Zeile 5 wird nun auf die View referenziert – und damit wird sie auch geladen, und die Eigenschaften werden verbunden.

Als Nächstes stellen wir ein Schlüssel/Wert-Paar für die Formateinstellungen bereit. In unserem Fall ist das Format offensichtlich PCM, und wir nutzen die Konstante `kAudioFormatLinearPCM`, die in der Datei `CoreAudioTypes.h` definiert und durch ein `NSNumber` umhüllt ist – wie der Wert für den `AVFormatIDKey`.

Auch wenn sich im Rest dieses `if`-Blocks noch eine Menge Code befindet, handelt es sich dabei vor allem um das Konvertieren der Typen aus den UI-Widgets in die `NSNumber`s, die als Werte im `NSDictionary` für das

Argument `settings` benötigt werden. Wenn zum Beispiel der `stereo-Switch` den Wert `on` hat, setzen wir den `AVNumberOfChannelsKey` auf 2, sonst auf 1. Sie finden noch weitere Informationen über zulässige Werte für die einzelnen Schlüssel in der Header-Datei `AVAudioSettings.h`, die die Einstellungsschlüssel definiert. Bei den Textfeldern mit numerischem Inhalt konvertieren wir den Text in ein `int` oder `float` und verpacken sie in eine `NSNumber`, während die `UISwitch`-Werte über `numberWithBool` in `NSNumber`s konvertiert werden.

16.3 Kodierte Audioformate

Wir haben die Recorder-Einstellungen für PCM behandelt, jetzt wollen wir uns der kodierten Formate annehmen. Dabei handelt es sich um die verschiedenen Formen der Audiokompression, durch die wir Audio-dateien so verkleinern können, dass man sie sinnvoll über Netzwerke verschicken oder auf kleinen Geräten speichern kann ... wobei die meisten kodierten Formate verlustbehaftet („lossy“) sind, die dekodierten Audiodaten also technisch gesehen nicht identisch mit den Ursprungsdaten sind. Viele Formate sind in `CoreAudioTypes.h` definiert, aber nicht alle werden vom iPhone unterstützt, und nicht alle lassen sich zum Aufzeichnen nutzen. Laut Apple lassen sich die folgenden Formate mit dem `AVAudioRecorder` nutzen:

- Linear PCM (`kAudioFormatLinearPCM`): Wie im vorigen Abschnitt gezeigt.
- AAC (`kAudioFormatMPEG4AAC`): Der MPEG-4-Audio-Codec, der vor allem durch iTunes und den iPod Verbreitung gefunden hat. Während diese Zeilen entstehen, können nur der iPod touch der zweiten Generation und das iPhone 3GS in diesem Format aufzeichnen.
- ALAC (`kAudioFormatAppleLossless`): Ein Encoder, der mit iTunes 4.5 eingeführt wurde und der die Größe der Dateien um etwa 50% reduziert – dabei aber verlustfrei arbeitet.
- IMA4 (`kAudioFormatAppleIMA4`): In der Windows-Welt auch bekannt als ADPCM. Dies ist eine einfache und nicht sehr rechenintensive 4:1-Kompression für 16-Bit-Audio.
- μ -law und A-law (`kAudioFormatULaw` und `kAudioFormatALaw`): Sehr alte Codecs, die ursprünglich für den Telefoniebereich entwickelt wurden.
- Internet Low-Bitrate Codec (`kAudioFormatiLBC`): Ein Codec, der für Voice over IP, das Streamen von Audio und andere Netzwerk-Aufgaben entworfen wurde. Eine der besonderen Eigenschaften des

iLBC ist, dass er mit verlorenen Frames besser umgehen kann als viele andere Audio-Codecs, wodurch er besonders gut für nicht verlässliche Übertragungsprotokolle wie UDP geeignet ist.

Im unserem Beispiel haben wir einen zweiten View-Controller `EncodingSettingsViewController`, mit dem wir eine View verwalten, in der der Anwender eines dieser Formate auswählen und die dazugehörigen Einstellungen vornehmen kann. Beachten Sie, dass wir uns hier nicht darum kümmern wollen, die Grenzen und Einschränkungen jedes dieser Formate zu berücksichtigen. So gibt es zum Beispiel bei vielen dieser Formate nur eine begrenzte Menge an möglichen Bitraten. Wenn Sie diese kodierten Formate in Ihren eigenen Anwendungen nutzen, sollten Sie sich die Details des gewählten Formats durchlesen und darauf achten, was Sie mit ihnen machen dürfen.

Das Interface Builder-Layout für die Datei `EncodingSettingsView.xib` sehen Sie in Abbildung 16.3. Die erste Komponente ist eine `Segmented View` mit der Möglichkeit, eines der unterstützten Formate auszuwählen. Diese sind auf ein `enum` in unserer `EncodingSettingsViewController.h` abgebildet, die die Formate für das `Segmented Control` aufzählt:

MediaPlayback/AudioRecorderPlayer/Classes/EncodingSettingsViewController.h

```
enum ENCODED_FORMAT_SEGMENT_VALUES {
    ENCODED_FORMAT_AAC = 0,
    ENCODED_FORMAT_ALAC,
    ENCODED_FORMAT_IMA4,
    ENCODED_FORMAT_ILBC,
    ENCODED_FORMAT_ULAW
};
```

Danach finden sich `Abtastrate` und `Stereoeinstellung`, genauso wie in der `PCM-Konfiguration` – Sie können die Felder eigentlich auch aus dieser View hierhin herüberziehen. Die letzten drei Einstellungen sind für Schlüssel gedacht, die nur von kodierten Formaten verwendet werden. Der `AVEncoderBitRateKey` definiert die kodierte Bitrate in Hertz. Sie haben diese Einstellung vermutlich schon in iTunes verwendet, wenn Sie sich überlegt haben, ob Sie Ihre CDs mit 128, 160, 192 oder mehr KBit/s rippen. Als Nächstes enthält der `AVEncoderBitDepthHintKey` einen Wert zwischen 8 und 32, analog zu `AVLinearPCM-BitDepthKey`. Schließlich gibt es einen `AVEncoderAudioQualityKey`, der einen der vier Werte aus `AVAudioSettings.h` enthält.

Deklarieren Sie wie zuvor Eigenschaften für jedes dieser Felder und eine für den `RecordViewController`, wobei der `Fertig-Button` wieder so

verbunden ist, dass er den modalen View-Controller freigibt und `[rec-Controller createAVAudioRecorder];` aufruft.

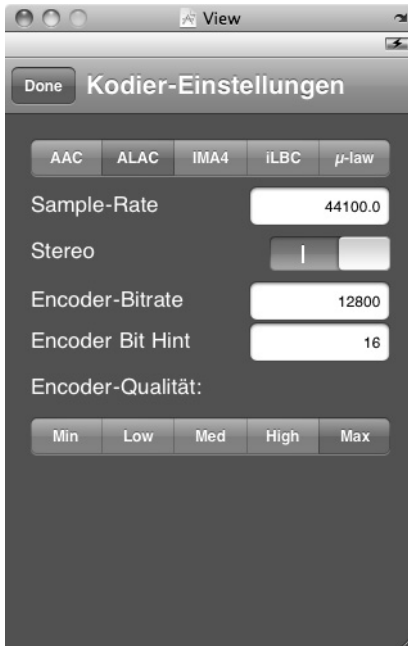


Abbildung 16.3: Layout für die Konfigurations-View für kodierte Audiodaten

Zurück in `RecordViewController.m`, machen wir mit dem **else**-Block weiter, der ein `NSDictionary` für die Recorder-Einstellungen erzeugt, wenn wir ein kodiertes Format verwenden:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
else {
    // Konfigurations-View laden, damit die Felder gelesen werden können
    encodingSettingsViewController.view;
    NSNumber *formatObject;
    switch ([encodingSettingsViewController.formatSegments
            selectedSegmentIndex]) {
        case (ENCODED_FORMAT_AAC):
            formatObject =
                [NSNumber numberWithInt: kAudioFormatMPEG4AAC];
            break;
        case (ENCODED_FORMAT_ALAC):
            formatObject =
                [NSNumber numberWithInt: kAudioFormatAppleLossless];
            break;
        case (ENCODED_FORMAT_IMA4):
            formatObject =
                [NSNumber numberWithInt: kAudioFormatAppleIMA4];
```



```

        break;
    case (ENCODED_FORMAT_ILBC):
        formatObject =
            [NSNumber numberWithInt: kAudioFormatILBC];
        break;
    case (ENCODED_FORMAT_ULAW):
        formatObject =
            [NSNumber numberWithInt: kAudioFormatULAW];
        break;
    default:
        formatObject =
            [NSNumber numberWithInt: kAudioFormatLinearPCM];
}
[recordSettings setObject:formatObject forKey: AVFormatIDKey];
float sampleRate =
    [encodingSettingsViewController.sampleRateField.text floatValue];
[recordSettings setObject: [NSNumber numberWithInt:sampleRate]
    forKey:AVSampleRateKey];
[recordSettings setObject:
    [NSNumber numberWithInt:
        (encodingSettingsViewController.stereoSwitch.on ? 2 : 1)]
    forKey:AVNumberOfChannelsKey];
int encoderBitrate =
    [encodingSettingsViewController.encoderBitrateField.text intValue];
[recordSettings setObject:[NSNumber numberWithInt:encoderBitrate]
    forKey:AVEncoderBitRateKey];
int bitDepthHint =
    [encodingSettingsViewController.encoderBitHint.text intValue];
[recordSettings setObject:
    [NSNumber numberWithInt:bitDepthHint]
    forKey:AVEncoderBitDepthHintKey];
int encoderQuality;
switch ([encodingSettingsViewController.qualitySegments
    selectedSegmentIndex]) {
    case (0) : encoderQuality = AVAudioQualityMin; break;
    case (1) : encoderQuality = AVAudioQualityLow; break;
    case (2) : encoderQuality = AVAudioQualityMedium; break;
    case (3) : encoderQuality = AVAudioQualityHigh; break;
    case (4) : encoderQuality = AVAudioQualityMax; break;
}
[recordSettings setObject:
    [NSNumber numberWithInt: encoderQuality]
    forKey: AVEncoderAudioQualityKey];
}

```

Das ist eine ganze Menge Code, aber auch hier passiert eigentlich gar nicht so viel. Der ausgewählte Wert aus dem Format-UISegmented-Control wird ausgelesen, die dazugehörige Konstante wird geholt und in einer NSNumber verpackt. Die Behandlung der Abtastrate und der Anzahl der Kanäle wurde direkt aus dem weiter oben beschriebenen PCMSettingsViewController übernommen, und die anderen Einstellungen werden ebenfalls analog dazu behandelt, wobei numerische Strings in NSNumbers umgewandelt werden.

16.4 Den AVAudioRecorder verwenden

Wir haben einen Platz, um die Audiodaten aufzuzeichnen, und wir kennen das Format, in dem aufgezeichnet werden soll – jetzt können wir den AVAudioRecorder erzeugen. Der letzte Parameter des Initializer `initWithURL:settings:error:` ist ein Zeiger auf einen NSError, der mit einem Fehlerobjekt gefüllt wird, wenn das Recorder-Objekt nicht mit der angegebenen URL und dem Einstellungs-Dictionary initialisiert werden kann. Wir werden dieses Fehlerobjekt nutzen, um dem Anwender eine Warnung auszugeben, wenn die Initialisierung fehlschlägt.

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```

NSError *recorderSetupError = nil;
audioRecorder = [[AVAudioRecorder alloc] initWithURL:destinationURL
                                                    settings:recordSettings error:&recorderSetupError];
[recordSettings release];

if (recorderSetupError) {
    UIAlertView *cantRecordAlert =
        [[UIAlertView alloc] initWithTitle: @"Kann nicht aufzeichnen"
                                         message: [recorderSetupError localizedDescription]
                                         delegate: nil
                                         cancelButtonTitle:@"OK"
                                         otherButtonTitles:nil];
    [cantRecordAlert show];
    [cantRecordAlert release];
    return recorderSetupError;
}
[audioRecorder prepareToRecord];
recordPauseButton.enabled = YES;
audioRecorder.delegate = self;

```

Nach dieser Vorbereitung brauchen wir nun nur `alloc` und `initWithURL:settings:error:`, um das Recorder-Objekt zu erstellen. War das erfolgreich, können wir `prepareToRecord` aufrufen, um die notwendigen Ressourcen zum Aufzeichnen zu allozieren – dadurch wird eine eventuelle Pause vermieden, wenn diese Aufgaben erst beim Aufruf von `record` ausgeführt werden. Wir setzen zudem den View-Controller als `delegate` des Recorder, so dass wir Aktivitäts-Callbacks von dort erhalten (siehe das Protokoll `AVAudioRecorderDelegate`). Wenn `recorderSetupError` andererseits nicht `nil` ist, zeigen wir eine `UIAlertView` und brechen ab.



Joe fragt...

Was bedeutet 1718449215 denn nun schon wieder?

Mehr als vielleicht andere Frameworks auf dem iPhone baut Core Audio auf den Vier-Zeichen-Codes (4ccs) zur Datenerkennung und für Antwortcodes auf. Erhalten Sie beim Erzeugen des AVAudioRecorder einen Fehler, ist es sehr wahrscheinlich, dass die domain des Fehlers **NSOSStatusErrorDomain** und der code ein 4-Byte-**int** wie 1718449215 ist. Diese Werte werden als vier **char** C-Strings erzeugt, wie Sie bei der Analyse der Header-Dateien von Core Audio sehen können.

Es gibt eine Reihe von Techniken, den Wert wieder in einen lesbaren String umzuwandeln. Die String-Ersetzungsfolge %s behandelt ihr Argument als null-terminierten char*, so dass Sie Fehlercodes wie folgt protokollieren können:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
int errorCode = CFSwapInt32HostToBig ([recorderSetupError
                                     code]);
NSLog(@"Error: %@ [%4.4s]" ,
      [recorderSetupError localizedDescription],
      (char*)&errorCode);
```

Damit wird die lokalisierte Fehlermeldung des NSError und eine String-Repräsentation des 4cc ausgegeben, wobei die Byte-Reihenfolge beachtet (die 4ccs sind alle Big Endian) und das Format des C-Strings auf genau vier Zeichen gesetzt wird. Die Ausgabe sieht dann zum Beispiel so aus:

```
Error: Operation could not be completed. (OSStatus
      error 1718449215.) [fmt?])
```

Eine andere Technik, um 4ccs zu erhalten, ist die Rechneranwendung. Wechseln Sie in die Programmiererdarstellung (-), klicken Sie auf die Buttons 10 und ASCII und fügen Sie Ihre Rückgabezahl ein.

Was ist aber nun fmt? Sie haben jetzt Text, nach dem Sie in den Header-Dateien von Core Audio suchen können. Diesen finden Sie in AudioFile.h – es ist die Konstante kAudioFileUnsupportedDataFormatError.

Wenn der AVAudioRecorder erfolgreich erzeugt wurde, ist es sehr einfach, ihn auch zu verwenden. Seine drei wichtigsten Methoden sind `record`, `pause` und `stop`. Der Unterschied zwischen den letzten beiden ist, dass Sie bei `pause` eigentlich weitermachen wollen, wozu Sie später erneut `record` aufrufen, während `stop` die Aufnahme beendet, die Daten in die Datei schreibt und sie schließt. Ein weiterer Aufruf von `record` nach einem `stop` würde die Datei wieder überschreiben. Mit diesem Wissen ist es nun wirklich simpel, die Hilfsmethoden zum Aufzeichnen, Pausieren und Stoppen zu schreiben:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(void) startRecording {
    [audioRecorder record];
    recordPauseButton.selected = YES;
    formatButton.enabled = NO;
}
-(void) pauseRecording {
    [audioRecorder pause];
    recordPauseButton.selected = NO;
}
-(void) stopRecording {
    [audioRecorder stop];
}
```

Dann rufen Sie diese Hilfsmethoden aus den Event-Handlern der Buttons auf:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(IBAction) handleRecordPauseTapped {
    NSLog(@"handleRecordPauseTapped" );
    if (audioRecorder.recording) {
        [self pauseRecording];
    } else {
        [self startRecording];
    }
}

-(IBAction) handleStopTapped {
    NSLog(@"handleStopTapped" );
    if (audioRecorder.recording) {
        [self stopRecording];
    }
}
```

Sie müssen sich auch noch um die Delegate-Callbacks des AVAudioRecorder kümmern. Zwei davon drehen sich um das Ende der Aufnahme (normal oder irregulär). Wir können diese Nachricht nutzen, um das GUI für die nächste Aufnahme zurückzusetzen.

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

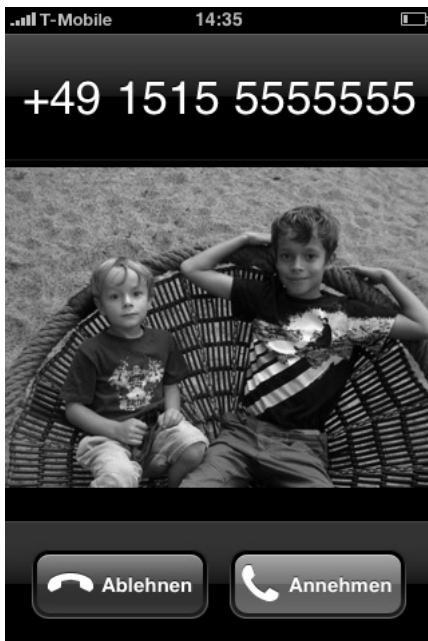
```

- (void)audioRecorderDidFinishRecording:(AVAudioRecorder *)recorder
    successfully:(BOOL)flag {
    NSLog(@"audioRecorderDidFinishRecording:successfully:");
    recordPauseButton.selected = NO;
    recordPauseButton.enabled = NO;
    formatButton.enabled = YES;
    [audioRecorder release];
    audioRecorder = nil;
    filenameField.text = @"" ;
}

- (void)audioRecorderEncodeErrorDidOccur:(AVAudioRecorder *)recorder
    error:(NSError *)error {
    recordPauseButton.selected = NO;
    recordPauseButton.enabled = NO;
    [audioRecorder release];
    audioRecorder = nil;
    filenameField.text = @"" ;
}

```

Die beiden anderen Delegate-Methoden erfordern in unserer Anwendung keine besondere Aufmerksamkeit, sind aber trotzdem erwähnenswert. `audioRecorderBeginInterruption:` zeigt an, dass die Anwendung unterbrochen wird, üblicherweise durch einen Alarm oder ein eingehendes Telefongespräch. In beiden Fällen kann der Nutzer als Reaktion auf die Unterbrechung zu einer anderen Anwendung wechseln, um zum Beispiel den Anruf entgegenzunehmen:



Nimmt der Anwender den Anruf an, erhält Ihr Anwendungs-Delegate einen Aufruf von `applicationWillTerminate:` und wird kurz danach beendet. Weist der Anwender allerdings den Anruf ab, erhält der Recorder-Delegate den Callback `audioRecorderEndInterruption:`, und Ihre Anwendung wird weitergeführt.

Das Einzige, was unsere Anwendung bei einer Unterbrechung wirklich tun muss, ist ein Pausieren der Aufzeichnung. Netterweise pausiert der `AVAudioRecorder` bei Unterbrechungen schon ganz allein.³

16.5 Aufzeichnungsinformationen anzeigen

Im Beispiel haben wir zwei Elemente implementiert, die dem Anwender eine Rückmeldung geben, während er aufzeichnet. Das erste ist ein aktuelles Zeitlabel. Dies entspricht der Zeitanzeige beim Musik-Player in Abschnitt 15.1, *Den iPod-Player überwachen*, auf Seite 329. Kurz gesagt erzeugen Sie einen `NSTimer`, der regelmäßig eine Aktualisierungsmethode aufruft, in der Sie die Eigenschaft `currentTime` (ein `NSTimeInterval`) aus dem `AVAudioRecorder`-Objekt auslesen und einen formatierten String nutzen, um eine MM:SS-Anzeige für das Label zu erzeugen. Das lässt sich schnell erledigen, Sie können einfach den Code aus dem anderen Projekt kopieren und müssen nur noch die Referenz auf den `iPodController` durch den `audioRecorder` ersetzen. Es ist sinnvoll, auch noch zu prüfen, ob der `audioRecorder` den Wert `nil` hat (also nicht erzeugt werden konnte), und in dem Fall das `currentTimeLabel` zu leeren.

Die andere für den Anwender interessante Rückmeldung ist die Darstellung der Aussteuerung des Recorder-Objekts. Es gibt im `AVAudioRecorder` eine Reihe von Methoden, die wir dazu nutzen können. Dabei werden sogar zwei Aussteuerungswerte bereitgestellt: der *Average Power Level* ist normalerweise das, was wir als aktuelle Lautstärke ansehen, während der *Peak Power Level* die größte Leistung in einer kurzen Periode darstellt (meist der letzten Sekunde). Sie haben vermutlich beide Angaben schon bei Audiogeräten gesehen – die durchschnittliche Leistung ist der recht „sprunghafte“ Wert, der mit der Lautstärke der Lautsprecher schwankt, während es sich bei der Spitzenleistung um einen Wert handelt, der fast immer größer als die durchschnittliche Leistung ist und die größte Lautstärke in der letzten Sekunde (oder so) darstellt.

3 Allerdings stoppt der Recorder im iPhone SDK 3.0, auch wenn die Dokumentation etwas anderes behauptet.

Wir wollen nun solch eine Aussteuerungsanzeige für unseren Recorder erstellen. Dafür müssen Sie eine neue Subklasse von `UIView` erzeugen und ihr den Namen `LevelMeterView` geben. Nutzen Sie im IB den Identity Inspector, um die Klasse der beiden flachen, aber breiten Views (neben den Labels „L“ und „R“) auf diese neue Klasse zu setzen.

Wie wir in Kapitel 19, *Zeichen in eigenen Views*, auf Seite 415 noch zeigen, werden wir diese Aussteuerungsanzeige selbst „zeichnen“. Um das Ganze einfach zu halten, nutzen wir nur einen weißen Balken, dessen Breite proportional zur durchschnittlichen Leistung ist, und eine zwei Pixel breite rote Linie für die Spitzenleistung. Ihre Header für diese Klasse werden Instanzvariablen für die durchschnittliche und die Spitzenleistung brauchen, aber auch `CGRects` für die Bereiche, in denen wir zeichnen wollen. Die einzige notwendige Methode ist die zum Setzen der Leistung.

MediaPlayer/AudioRecorderPlayer/Classes/LevelMeterView.h

```
@interface LevelMeterView : UIView {
    float power;
    float peak;
    CGColorRef levelColor;
    CGColorRef peakColor;
    CGRect levelRect;
    CGRect peakRect;
}
-(void) setPower: (float) pow peak: (float) pk;
@end
```

Wir werden im View-Controller mit dem Code zum Aufrufen von `setPower:peak:` beginnen und uns dann um das Zeichnen kümmern. Nach dem Erstellen des `AVAudioRecorder` nutzen Sie seine Eigenschaft `settings`, um herauszufinden, ob er in Stereo oder Mono aufzeichnet. Wenn es sich nur um einen Kanal handelt, deaktivieren wir die View für den rechten Kanal und nutzen den linken für die Monodarstellung. Außerdem teilen wir dem Recorder hier mit, dass wir die Aussteuerungsanzeige zeigen wollen, indem wir die Eigenschaft `meteringEnabled` setzen.

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
audioRecorder.meteringEnabled = YES;
if ([[audioRecorder.settings
    objectForKey:AVNumberOfChannelsKey] intValue] > 1) {
    leftLevelLabel = @"L" ;
    rightLevelLabel.hidden = NO;
    rightLevelMeter.hidden = NO;
} else {
    leftLevelLabel = @"M" ;
    rightLevelLabel.hidden = YES;
    rightLevelMeter.hidden = YES;
}
```

In unserer Methode `updateAudioDisplay`, die regelmäßig durch den Timer aufgerufen wird, um die aktuelle Aufnahmezeit anzuzeigen, fügen wir Code hinzu, um die Aussteuerungswerte vom Recorder zu holen und sie in unseren selbstgebastelten Views zu setzen. Sie erhalten die Werte über die Instanzmethoden `averagePowerForChannel:` und `peakPowerForChannel:`, aber bevor Sie sie nutzen, müssen Sie noch `updateMeters` aufrufen, um die beiden Werte in allen Kanälen zu aktualisieren.

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
[audioRecorder updateMeters];
[leftLevelMeter setPower: [audioRecorder averagePowerForChannel:0]
                        peak: [audioRecorder peakPowerForChannel: 0]];
if (! rightLevelMeter.hidden) {
    [rightLevelMeter setPower: [audioRecorder averagePowerForChannel:1]
                        peak: [audioRecorder peakPowerForChannel: 1]];
}
```

Aber was sind das für Werte? Wir müssen das verstehen, bevor wir eine sinnvolle Aussteuerungsanzeige ausgeben können. Sie erwarten vielleicht, dass sie im Bereich zwischen 0.0 und 1.0 liegen, so wie die Werte der Eigenschaft `volume` des `MPMusicPlayerController`, aber wenn Sie sie per `NSLog()` ausgeben, werden Sie feststellen, dass sie im Bereich von -160.0 für Stille bis 0.0 für die Maximalleistung liegen. Dazu ist die Skala eindeutig nicht linear – wenn es überhaupt etwas zu hören gibt, sind die Werte schnell im Bereich von etwa -30.0 bis 0.0.

Die vom `AVAudioRecorder` zurückgegebenen Werte werden in *Dezibel* gemessen. Dezibel ist eine logarithmische Maßeinheit, die relativ zu einem Referenzwert bestimmt wird. Da Dezibel logarithmisch zur Basis 10 sind, bedeutet eine Differenz von 10 bei der Leistung einen Unterschied einer ganzen Größenordnung. Es klingt aber nicht zehn Mal lauter, da das menschliche Gehör ebenfalls logarithmisch funktioniert, daher ist Dezibel hier genau die richtige Maßeinheit.

Um eine grafische Ausgabe zu haben, wollen wir die logarithmische Skala aber wieder in eine lineare umwandeln, so dass die visuelle Änderung bei der Anzeige zu dem passt, was wir hören. Wir können die logarithmischen Werte von -160.0 bis 0.0 in eine lineare Skala von 0.0 bis 1.0 mithilfe der Formel $\text{widthPercentage} = 10^{0.5\text{level}}$ konvertieren. Dann multiplizieren wir diese **float** mit der Breite der View, um herauszubekommen, wie breit der Balken sein soll. Dadurch führt 0.0 zu einem leeren Balken, 1.0 zu einem vollen und die Werte dazwischen zu einem entsprechend proportional breiten Balken.

Wir initialisieren also zunächst die Farben und die festen Elemente der zu zeichnenden Rechtecke. Das machen wir sinnvollerweise in der Methode `initWithCoder:` der `LevelMeterView`. Diese wird aufgerufen, wenn die View aus dem Nib geladen wird:

MediaPlayer/AudioRecorderPlayer/Classes/LevelMeterView.m

```
-(id) initWithCoder: (NSCoder*) decoder {
    if (self =[super initWithCoder: decoder]) {
        // Initialisierungscode
        levelColor = [UIColor whiteColor].CGColor;
        levelRect.origin.x=0;
        levelRect.origin.y=0;
        peakColor = [UIColor redColor].CGColor;
        peakRect.size.width=2;
        peakRect.origin.y = 0;
    }
    return self;
}
```

Als Nächstes implementieren wir die Methode `setPower:peak:`, die der Updater aufruft. Hier müssen Sie sich nur die Werte merken und ein Neuzeichnen anfordern.

MediaPlayer/AudioRecorderPlayer/Classes/LevelMeterView.m

```
-(void) setPower: (float) pow peak: (float)pk {
    power = pow;
    peak = pk;
    // Neuzeichnen anfordern
    [self setNeedsDisplay];
}
```

Der Aufruf von `setNeedsDisplay` führt schließlich zu einem Aufruf von `drawRect:`, und hier bauen wir unseren Code zum Zeichnen der Anzeige ein. Wir müssen nur die aktuelle View leeren, die Größe von `levelRect` berechnen und es füllen und dann das Gleiche für das `peakRect` durchführen.

MediaPlayer/AudioRecorderPlayer/Classes/LevelMeterView.m

```
1 - (void)drawRect:(CGRect)rect {
2     // Zeichnen
3     CGContextRef context = UIGraphicsGetCurrentContext();
4
5     // View leeren
6     CGColorRef undrawColor = self.backgroundColor.CGColor;
7     CGContextSetFillColorWithColor (context, undrawColor);
8     CGContextFillRect (context, rect);
9
10    // Größe berechnen
11    levelRect.size.height = rect.size.height;
12    levelRect.size.width = pow (10, (0.05 * power)) * rect.size.width;
```

```

13
14     // Mit Farbe füllen
15     CGContextSetFillColorWithColor(context, levelColor);
16     CGContextFillRect(context, levelRect);
17
18     // Spitzenleistung als 2 Pixel breiten Balken zeichnen
19     CGContextSetFillColorWithColor(context, peakColor);
20     peakRect.size.height = rect.size.height;
21     peakRect.origin.x = pow (10, (0. * 05 peak)) * rect.size.width;
22     if (peakRect.origin.x >= (rect.size.width -2))
23         peakRect.origin.x = rect.size.width -2;
24     CGContextFillRect(context, peakRect);
25 }

```

Der entscheidende Teil dieses Codes ist der mit den Berechnungen zur Aussteuerung. In Zeile 12 berechnen wir eine prozentuale Weite für die durchschnittliche Leistung, multiplizieren diese mit der Breite der View und bekommen so die Breite des `levelRect`, das wir dann mit weißer Farbe füllen. Für die Ausgabe der Spitzenleistung verwenden wir in Zeile 21 die gleiche Formel, allerdings nutzen wir in diesem Fall den ermittelten Wert als x-Wert für die linke Seite von `peakRect`, dessen Breite immer 2 ist.



Abbildung 16.4: Audiodaten mit dem `AudioRecorderPlayer` aufzeichnen

Mit der so eingerichteten Aussteuerungsanzeige können Sie nun versuchen, mit der Anwendung Audiodaten aufzuzeichnen. Beobachten Sie mal beim Aufzeichnen die Veränderung der Anzeige als Reaktion auf die Lautstärke Ihrer Stimme. Die meisten Mikrofone können allerdings nur einen Kanal aufzeichnen, daher sehen Sie wahrscheinlich nur die Aussteuerung des linken Kanals (siehe Abbildung 16.4).

Im Simulator⁴ finden Sie Ihre aufgezeichneten Dateien in `~/Library/Application Support/iPhone Simulator/User/Applications/` – sortieren Sie sie nach Datum, um herauszufinden, welcher der Ordner (die als Namen nur eine wüste hexadezimale ID tragen) Ihre Anwendung ist, und schauen Sie dann in den *Documents*-Ordner. Oder Sie machen gleich mit dem nächsten Abschnitt weiter, in dem wir einen Player-Tab hinzufügen, um unsere aufgezeichneten Dateien abzuspielen.

16.6 Audiodaten mit dem AVFramework abspielen

Das AVFoundation-Framework bietet auch eine praktische Objective-C-Klasse `AVAudioPlayer` an, um Audiodaten abzuspielen. Diese Klasse ist sogar schon im iPhone OS 2.2 dazugekommen, daher ist sie im Gegensatz zum erst in 3.0 verfügbaren `AVAudioRecorder` sogar abwärtskompatibel.

Schauen Sie sich die Dokumentation an, finden Sie viele Ähnlichkeiten zum `AVAudioRecorder`. Es gibt die einfachen Methoden `play`, `pause` und `stop`, aber auch die Eigenschaften `duration`, `numberOfChannels` und `currentTime`. Letztere lässt sich sogar verändern, so dass Sie damit in einer Datei springen können (so wie die `currentTime` des `MPMusicPlayerController`, den wir in Abschnitt 15.2, *Die iPod-Anwendung steuern*, auf Seite 339 beschrieben haben). Es gibt sogar eine Aussteuerungs-API, die identisch mit der des `AVAudioRecorder` ist.

Beim Erstellen eines Players für unsere aufgezeichneten Dateien besteht ein Großteil der Arbeit darin, sich Code aus dem Recorder-Tab und aus der Player-App für die Musikbibliothek zu kopieren. Da der `AudioRecorderPlayer` alle Aufzeichnungen im Verzeichnis *Documents* ablegt, soll der Anwender in unserem Player eine Datei aus diesem Verzeichnis auswählen und abspielen können, wobei er die aktuelle Position angezeigt bekommt, springen kann und auch der Lautstärkepegel dargestellt wird ... alles das, was Sie schon kennen.

⁴ Beachten Sie, dass der `AVAudioRecorder` im Simulator anscheinend nicht mit Geräten zusammenarbeitet, die über FireWire angeschlossen sind, wie zum Beispiel das alte, externe iSight. Über die Ton-Systemeinstellungen können Sie Ihren Audioeingang auf ein eingebautes Mikrofon oder ein USB-Gerät umschalten.

Erstellen Sie eine Klasse `PlayViewController` (mit `NEW FILE` und dem Template `UIViewController`) und eine Datei `PlayView.xib` für das GUI. Öffnen Sie `MainWindow.xib` im IB und suchen Sie nach dem Controller des zweiten Tab-Elements. Setzen Sie in dessen *Attributes Inspector* den NIB Name auf `PlayView` und im *Identity Inspector* die Klasse auf `PlayViewController`. Jetzt wird Ihr neuer View-Controller verwendet und mit Ihrer neuen View verbunden.

Als Nächstes können Sie die Datei `PlayView.xib` im IB öffnen, um das Abspiel-GUI zu entwerfen. Nachdem Sie die Klasse des File's Owner auf `PlayViewController` gesetzt haben, fügen Sie weitere Komponenten aus der Library hinzu, um eine Benutzeroberfläche wie die in Abbildung 16.5 zu erhalten.



Abbildung 16.5: Layout für den zweiten Tab des `AudioRecorderPlayer`

Die `UIPickerView` erschlägt einen zwar ein wenig, aber sie lässt sich dafür sehr leicht per Code ansprechen. Auch lohnt es sich, darauf hinzuweisen, dass Sie im Interface Builder Elemente zwischen Nibs hin- und herziehen können. Im Screenshot haben wir die Aussteuerungsanzeigen und ihre Labels aus der Datei `RecordView.xib` direkt in diese View gezogen und sie dann `LevelMeterViews` genannt. (Beachten Sie, dass sie auch einen transparenten Hintergrund besitzen, weshalb wir für den Screenshot den linken Kanal gewählt haben – sonst wären sie unsichtbar.)

Sie müssen erneut IBOutlets für `filePicker`, `playPauseButton`, `stopButton`, `currentTimeLabel`, `currentTimeSlider`, `durationLabel`, `leftLevelMeter`, `rightLevelLabel`, `rightLevelMeter` und `volumeSlider` erzeugen. Denken Sie auch daran, `dataSource` und `delegate` des Pickers mit `File's Owner` zu verbinden.

Jetzt wollen wir uns um den Picker zur Dateiauswahl kümmern. Sie müssen die Header-Datei bearbeiten, um zu deklarieren, dass Sie die Methoden von `UIPickerViewDelegate` und `UIPickerViewDataSource` implementieren, die beide auf ein `NSArray*` `filenames` zurückgreifen. Dieses Array können Sie in `viewDidLoad` einrichten, wo wir auch die Instanzvariable `_documentsPath` befüllen.

MediaPlayer/AudioRecorderPlayer/Classes/PlayViewController.m

```
[_documentsPath release];
NSArray *searchPaths =
    NSSearchPathForDirectoriesInDomains
        (NSDocumentDirectory, NSUserDomainMask, YES);
_documentsPath = [searchPaths objectAtIndex: 0];
[_documentsPath retain];
[filenames release];
filenames = [[NSFileManager defaultManager]
    directoryContentsAtPath:_documentsPath];
[filenames retain];
```

Mit diesem Array mit Dateinamen können wir die Data Source und den Delegate des Pickers problemlos implementieren:

MediaPlayer/AudioRecorderPlayer/Classes/PlayViewController.m

```
#pragma mark UIPickerViewDataSource methods
- (NSInteger) pickerView: (UIPickerView*) pickerView
    numberOfRowsInComponent: (NSInteger) component {
    return [filenames count];
}

- (NSInteger) numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}

#pragma mark UIPickerViewDelegate methods
- (NSString *) pickerView:(UIPickerView *) pickerView
    titleForRow: (NSInteger) row
    forComponent: (NSInteger) component {
    return [filenames objectAtIndex: row];
}
```

Immer wenn der Anwender nun eine andere Datei auswählt, versuchen wir, ein neues `AVAudioFilePlayer`-Objekt mit dieser Datei zu erzeugen. Das machen wir in der noch zu schreibenden Hilfsmethode `createAVAudioPlayer`:

MediaPlayer/AudioRecorderPlayer/Classes/PlayViewController.m

```

-(void)pickerView:(UIPickerView *)pickerView didSelectRow:(NSInteger)row
    inComponent:(NSInteger)component {
    // Stoppen, wenn noch abgespielt wird
    [audioPlayer stop];
    playPauseButton.selected = NO;
    [self createAVAudioPlayer];
}

```

Um einen AVAudioPlayer zu erzeugen, haben wir zwei Möglichkeiten. Es gibt eine Methode `initWithData:error:`, die ein NSData-Objekt erwartet, das sich auf Audiodaten im Speicher bezieht. Das kann nützlich sein, wenn Sie Audiodaten in einen Speicherpuffer geladen haben. Für uns ist hier aber der Initializer `initWithContentsOfURL:error:` sinnvoller. Die Methode erwartet zwar eine NSURL, aber sie kann Audiodaten nur aus URLs des Typs `file://` laden – der AVAudioPlayer spielt keine RTSP-Streams, Shoutcast-http-Streams oder irgendwelche anderen Audiodaten aus dem Netz ab. Wenn Sie einen streamenden Audio-Player bauen wollen, müssen Sie die Frameworks nutzen, die in Abschnitt 17.3, *Überblick über Core Audio*, auf Seite 397 vorgestellt werden.

Dies ist die vollständige Implementierung von `createAVAudioPlayer`:

MediaPlayer/AudioRecorderPlayer/Classes/PlayViewController.m

```

1  -(NSError*) createAVAudioPlayer {
2      [audioPlayer release];
3      audioPlayer = nil;
4      currentTimeSlider.value = 0;
5
6      NSString *filename = [filenames objectAtIndex:
7          [filePicker selectedRowInComponent:0]];
8      NSString *playbackPath =
9          [_documentsPath stringByAppendingPathComponent: filename];
10     NSURL *playbackURL = [NSURL fileURLWithPath: playbackPath];
11     NSError *playerSetupError = nil;
12     audioPlayer = [[AVAudioPlayer alloc]
13         initWithContentsOfURL:playbackURL error:&playerSetupError];
14
15     if (playerSetupError) {
16         NSString *errorTitle =
17             [NSString stringWithFormat:@"%@@ nicht abspielbar:" ,
18                 filename];
19         UIAlertView *cantPlayAlert =
20             [[UIAlertView alloc] initWithTitle: errorTitle
21                 message: [playerSetupError localizedDescription]
22                 delegate:nil
23                 cancelButtonTitle:@"OK"
24                 otherButtonTitles:nil];
25         [cantPlayAlert show];
26         [cantPlayAlert release];

```

```

27         audioPlayer = nil;
28         durationLabel.text = @"--:--" ;
29         return playerSetupError;
30     }
31     audioPlayer.delegate = self;
32     audioPlayer.meteringEnabled = YES;
33     audioPlayer.volume = volumeSlider.value;
34     currentTimeSlider.maximumValue = audioPlayer.duration;
35     durationLabel.text = [NSString stringWithFormat:@"%02d:%02d" ,
36                         (int) audioPlayer.duration/60,
37                         (int) audioPlayer.duration%60];
38     return playerSetupError;
39 }

```

Die Methode fügt den aktuellen Dateinamen an den `_documentsPath` an, wandelt den Pfad in eine `NSURL` um und erzeugt den `AVAudioPlayer` in den Zeilen 12 und 13, wobei sie eine Warnung ausgibt, wenn ein Fehler aufgetreten ist.⁵

Wie Sie sehen können, fehlt jetzt nur noch die Behandlung der Zeit- und der Aussteuerungsanzeige – alles Dinge, die Sie bereits vorher gemacht haben:

- Die aktuelle Abspielzeit wird mit einem `NSTimer` aktualisiert, der regelmäßig eine Methode `updateAudioDisplay` zum Anpassen des `currentTimeLabel` aufruft und dabei die Eigenschaft `currentTime` des Players nutzt – umformatiert im Stil `MM:SS`. Sie haben das schon im Recorder-Tab gesehen, im Kapitel über die Musikbibliothek und bei der Uhr in Kapitel 9, *Einstellungen*, auf Seite 181.
- `updateAudioDisplay` kann auch den `currentTimeSlider` aktualisieren. Sie haben einen Slider wie den in Abschnitt 15.1, *Den iPod-Player überwachen*, auf Seite 329 erstellt und können daher hier auch genau so vorgehen: regelmäßiges Aktualisieren des Werts aus der Player-Eigenschaft `currentTime` sowie Anpassen dieser Eigenschaft, wenn Sie ein Event `Value Changed` vom Slider erhalten (Sie werden auch einen Wert `isScrubbing` nutzen wollen, um `updateAudioDisplay` mitzuteilen, dass es den Slider-Wert nicht automatisch aktualisieren soll, solange der Slider verschoben wird).

⁵ Sie werden häufig einen Fehler erhalten, wenn Sie im Simulator auf diesen Tab wechseln, da die erste Datei, die der Picker anzeigt, `.DS_Store` sein wird – was keine Audiodatei ist. Sie könnten diese App dadurch verbessern, dass Sie das Array `filenames` so filtern, dass nur Dateien mit für Audiodateien passenden Erweiterungen angezeigt werden, wie zum Beispiel `.caf` und `.aif`.

- Wie schon erwähnt, funktioniert die Aussteuerungsanzeige genau so wie auf dem Recording-Tab, der in Abschnitt 16.5, *Aufzeichnungsinformationen anzeigen*, auf Seite 372 beschrieben ist. Die API im AVAudioPlayer entspricht der in AVAudioRecorder.
- Der AVAudioPlayerDelegate definiert Methoden, die denen in AVAudioRecorderDelegate sehr ähnlich sind: `audioPlayerDidFinishPlaying:successfully:` und `audioPlayerDecodeErrorDidOccur:error:` können genutzt werden, um das GUI zu aktualisieren, wenn das Abspielen entweder normal oder aufgrund eines Fehlers endet. `audioPlayerBeginInterruption:` und `audioPlayerEndInterruption:` informieren Sie über Unterbrechungen durch eingehende Anrufe oder andere Systemereignisse, die dazu führen können, dass der Anwender Ihre App beendet.

Durch das Anwenden der beschriebenen Techniken sowie das Kopieren und Anpassen des Codes können Sie den `PlayViewController` recht problemlos fertigstellen. Sie können natürlich auch einfach auf den herunterladbaren Beispielcode zurückgreifen.

16.7 Audio-Sessions

Jede Anwendung, die mit Audiodaten arbeitet, muss auch eine *Audio-Session* erzeugen. Dabei handelt es sich um ein Objekt, das darstellt, wie Ihre Anwendung mit dem restlichen Audiosystem interagiert.

Es ist wichtig, dass Sie eine *Audiokategorie* erzeugen, die beschreibt, wie Ihre Anwendung Audiodaten nutzt und wie sie mit anderen Audiodaten im System interagiert (oder auch nicht). So sollten sich zum Beispiel Anwendungen ohne eigene Tonerzeugung nicht darum scheren, ob der Anwender im Hintergrund Musik laufen hat. Aber ein Spiel mit eigenem Soundtrack, eine App zum Mixen von Musik, ein Client für Webradios und andere Anwendungen wollen durchaus sicherstellen, dass nur „ihre“ Musik zu hören ist. Und in anderen Fällen kann es auch erwünscht sein, Ihre Audiodaten mit denen der iPod-Anwendung zu kombinieren.

Die `AVAudioSession` stellt eine Objective-C-Klasse dar, mit der mit Audio-Sessions gearbeitet werden kann. Wir werden mit der Methode `setCategory:error:` beginnen. Hier legen Sie fest, was Sie audioteknisch mit dem System machen wollen, und die Methode sollte in Ihrer Anwendung sehr früh aufgerufen werden. Bei unserer Anwendung reicht es, sie in der Methode `applicationDidFinishLaunching` des `AudioRecorderPlayerAppDelegate` aufzurufen.

MediaPlayer/AudioRecorderPlayer/Classes/AudioRecorderPlayerAppDelegate.m

```

AVAudioSession *audioSession = [AVAudioSession sharedInstance];
NSError *audioSessionError = nil;
[audioSession setCategory: AVAudioSessionCategoryPlayAndRecord
                      error: &audioSessionError];

if (audioSessionError)
    NSLog(@"Error setting audio category: %@",
          [audioSessionError localizedDescription]);

```

Die einzige Möglichkeit, ein AVAudioSession-Objekt zu erhalten, ist der Weg über die Klassenmethode sharedInstance. Mit diesem Objekt deklarieren wir unsere Kategorie als AVAudioSessionCategoryPlayAndRecord – die Anwendung spielt also Audiodaten ab und nimmt auch welche auf. Das hat den Effekt, dass die iPod-App oder andere Hintergrundmusik „stillgelegt“ und die Audioeingabe-Hardware für unsere Anwendung reserviert wird.

In AVAudioSession sind fünf Kategorien definiert, von denen jede beschreibt, wie sich die Anwendung (audiotechnisch) verhält, wie die Auswirkungen auf das Audiosystem des iPhone sind, welcher Zugriff auf die Ein- und Ausgabe-Hardware gewährt wird, wie die Musik mit der der iPod-App oder anderer Hintergrundmusik gemischt wird und ob dem Stummschalter „gehört“ wird.

Kategorie	Zugriff auf Eingabe-Hardware				Zugriff auf Ausgabe-Hardware				Mischen mit iPod				Stummschaltung berücksichtigen			
AVAudioSessionCategoryAmbient	Nein	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja	Ja
AVAudioSessionCategorySoloAmbient	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja	Nein	Ja
AVAudioSessionCategoryPlayback	Nein	Ja	Nein	Nein	Nein	Ja	Nein	Nein	Ja	Nein	Nein	Nein	Ja	Nein	Nein	Nein
AVAudioSessionCategoryRecord	Ja	Nein	Nein	Nein	Nein	Ja	Nein	Nein	Ja	Nein	Nein	Nein	Ja	Nein	Nein	Nein
AVAudioSessionCategoryPlayAndRecord	Ja	Ja	Nein	Nein	Nein	Ja	Ja	Nein	Nein	Ja	Ja	Nein	Nein	Ja	Nein	Nein

Mit der in der Form deklarierten Kategorie wird das Audiosystem so „aktiviert“, dass man Zugriff auf die Ein- und/oder Ausgabe-Hardware hat und dabei eventuell andere Töne im Hintergrund stummgeschaltet werden:

MediaPlayer/AudioRecorderPlayer/Classes/AudioRecorderPlayerAppDelegate.m

```
[audioSession setActive:YES error:&audioSessionError];
```

Die `AVAudioSession` kann ebenfalls dazu genutzt werden, das Audiosystem zu inspizieren. Sie können die Abtastrate der Hardware und den I/O-Puffertakt (I/O Buffer Duration) auslesen und die bevorzugten Werte setzen.⁶ Sie können auch ermitteln, wie viele Kanäle Ein- und Ausgabe-Hardware besitzen und ob die Audioeingabe überhaupt zur Verfügung steht.

Dieser letzte Punkt ist für Anwendungen wichtig, die Audiodaten aufzeichnen wollen, da nicht alle iPhone-OS-Geräte aufnehmen können. Alle iPhones haben mindestens eine Aufnahmequelle – das Headset-Mikrofon, das Telefonmikrofon oder eventuell andere angeschlossene Elemente. Die erste Generation der iPod touches hingegen bietet keinerlei Möglichkeiten, Audiodaten aufzuzeichnen. Und beim iPod touch der zweiten Generation kann man nur dann Audiodaten aufnehmen, wenn ein entsprechendes Mikrofon angesteckt ist, wie zum Beispiel das Standard-iPhone-Headset mit der Fernsteuerung. Da die Verfügbarkeit der Aufnahmemöglichkeit nicht am Modell hängt, sondern durch den Anwender verändert werden kann, wird eine robust entwickelte Anwendung die `AVAudioSession` nutzen, um die Aufnahmemöglichkeiten zur Laufzeit zu prüfen.

Sie können zum Beispiel eine Methode `alertIfNoAudioInput` zum `RecordViewController` hinzufügen und sie beim Antippen des RECORD-Buttons aufrufen:

MediaPlayer/AudioRecorderPlayer/Classes/RecordViewController.m

```
-(BOOL) alertIfNoAudioInput {
    AVAudioSession *session = [AVAudioSession sharedInstance];
    BOOL audioHWAvailable = session.inputIsAvailable;
    if (! audioHWAvailable) {
        UIAlertView *cantRecordAlert =
            [[UIAlertView alloc] initWithTitle: @"Kann nicht aufzeichnen"
            message: @"Keine Audioeingabe vorhanden"
            delegate: nil
```

⁶ Es ist ein bisschen seltsam, dass die I/O Buffer-Information durch die High-Level-Klasse `AVFoundation` verfügbar ist, da sie nur in den Low-Level-APIs von Core Audio genutzt werden, weil dort die Software-Latenz im Audioumfeld so klein ist, dass man sich Gedanken um die Hardware-Latenz machen kann.

```

        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    [cantRecordAlert show];
    [cantRecordAlert release];
}
return audioHWAvalable;
}

```

Mit einer High-Level-API zum Aufzeichnen und Abspielen von Audiodaten in den verschiedensten Formaten bietet Ihnen das AVFoundation-Framework des iPhone OS 3.0 eine Reihe von Möglichkeiten mit einer ziemlich einfachen Objective-C-API, die den Entwickler vor vielen Details und Problemen bewahrt.

Andererseits sind gerade diese Details für manche Entwickler von Interesse. Auch wenn der Umgang mit Audiodaten durch die AVFoundation sehr einfach wird, wenn man nur schlichte Dateien abspielen will, bleiben die kritischeren Themen wie Streaming, Mischen und Verarbeiten außen vor. Es gibt Frameworks auf niedrigerer Ebene, mit denen Sie auch dies machen können – im nächsten Kapitel werden wir einen Blick darauf werfen.

Kapitel 17

Core Audio

In Kapitel 16, *Audiodateien abspielen und aufzeichnen*, auf Seite 355 haben Sie gelernt, wie Sie mit dem Objective-C-Framework AVFoundation Audiodateien in einer Vielzahl von Formaten aufzeichnen und abspielen können. Dieses Framework ist dazu gedacht, Entwicklern eine einfach zu nutzende API für die schlichteren Audioaufgaben bereitzustellen – zum Beispiel wenn man sich nicht mit den Details herumschlagen, sondern einfach nur eine MP3-Datei abspielen will. Dabei wurde auf Mächtigkeit zugunsten von Einfachheit verzichtet. Die AVFoundation kann keinen der folgenden Punkte berücksichtigen:

- Audiodaten aus dem Netzwerk abspielen.
- Mehrere Klangquellen mischen.
- Zugreifen auf die Audio-Daten – sowohl in kodierter als auch in dekodierter Form
- Zugreifen auf die Audiometadaten (abgesehen von der Laufzeit und der Anzahl der Kanäle). Der AVAudioPlayer stellt nicht einmal das Format bereit, in dem die gerade abgespielte Audiodatei vorliegt.
- Aufgezeichnete Audiodaten manipulieren.

Sie haben sehr wahrscheinlich schon iPhone-Anwendungen gesehen, die all das leisten, daher ist klar, dass es irgendwie möglich sein muss. Der Schlüssel liegt in der Verwendung von Low-Level-APIs, die (anders als AVFoundation) schon von Anfang an im iPhone SDK enthalten sind. Die verschiedenen Audio-APIs auf dem iPhone sind alle Teil von Core Audio, einer umfassenden, Stream-basierten Infrastruktur für den

Umgang mit digitalen Audiodaten. Core Audio besteht aus einer Reihe zusammenhängender APIs, die von der Low-Level-Verarbeitung kleiner Puffer mit Audio-Samples bis hin zum High-Level-Klassen des AVFramework zum Abspielen und Aufzeichnen reichen. Indem Sie eine der niedrigeren Abstraktionsebenen wählen, können Sie Ihrer Audioanwendung mehr Funktionalität verpassen.

Wie auch immer – um das zu erreichen, müssen wir Objective-C hinter uns lassen ...

17.1 Prozedurale C-APIs verwenden

Wie schon in Abschnitt 3.1, *Die Softwarearchitektur des iPhone*, auf Seite 26 erwähnt, befindet sich ein Großteil der Medienunterstützung im iPhone OS im Core Media Layer, der auf einer niedrigeren Abstraktionsebene als Cocoa Touch arbeitet. Begeben Sie sich von den Höhen des Cocoa Touch-Framework in die Niederungen des Stacks, werden Sie bemerken, dass die APIs nun aus C-Funktionen und nicht mehr aus Objective-C-Klassen bestehen. Das bedeutet, dass wir C nicht nur für Audioanwendungen, sondern auch noch für viele andere APIs in den Core Services und Core Media Layers nutzen werden, einschließlich Quartz/Core Graphics und Adressbuch. Wir haben diese Layer schon ein paar Mal aufgerufen, zum Beispiel die Core Foundation-Funktion `NSLog()`.

Apple bezeichnet C meist als *prozedurales C* (Procedural C), um es von Objective-C zu unterscheiden. Es gibt eine Reihe von Entwurfsmustern und Programmierrichtlinien in diesen APIs, durch die eine OO-ähnliche Nutzung möglich sein soll. Es ist hilfreich, diese Muster und Regeln zu kennen, bevor Sie tiefer in die Core-APIs einsteigen, da die Konzepte für eine korrekte Nutzung der APIs ausgesprochen wichtig sind.

Opaque Typen und Objekte

Die Core-APIs bieten zwar Objekte an, aber es gibt in C keine Klassen. Stattdessen sind in der Core Foundation *opaque Typen* definiert, die Klassen dergestalt ähneln, dass sie eine Implementierung vor Ihnen verbergen. Allerdings fehlen ein paar andere Eigenschaften von OO-Klassen, wie zum Beispiel der Polymorphismus. So repräsentiert zum Beispiel der opaque Typ `CFString` einen String aus Unicode-Zeichen, vergleichbar mit dem `NSString` in Cocoa. Die internen Datenstrukturen des `CFString` sind für Sie nicht erreichbar, Ihr Code wird sogar nur mit Referenzen auf diesen Typ arbeiten, die als Zeiger `CFStringRef` definiert (und dokumentiert) sind.

Um die Idee des Aufrufens einer Methode für ein Objekt im prozeduralen C nachzubilden, nutzen die Core Foundation-APIs eine Namenskonvention. Funktionen, die mit einem Objekt arbeiten, beginnen mit dem opaken Typ des Objekts und nutzen das Objekt als erstes Argument. So würden Sie zum Beispiel die Länge einer `CFStringRef` mit dem Namen `myCFString` so erhalten:

```
length = CFStringGetLength (myCFString);
```

Funktionen, die Parameter erwarten, stellen diese hinter das Objekt, wie in diesem Beispiel, bei dem das Unicode-Zeichen an Position 3 (also als viertes Zeichen) ausgelesen wird:

```
fourthUniChar = CFStringGetCharacterAtIndex (myCFString, 3);
```

Die `CFStringRef` ist nicht veränderbar – genauso wie `NSString`. Wie Sie vielleicht schon vermuten, gibt es einen weiteren opaken Typ `CFMutableStringRef`, bei dem Sie den Inhalt des Strings verändern dürfen.

Möglicherweise ist Ihnen bereits aufgefallen, dass es eine enge Korrelation zwischen einigen der opaken Typen aus der Core Foundation und ihren Objective-C-Äquivalenten im Cocoa Foundation Framework gibt. Die Korrelation ist sogar so groß, dass in manchen Fällen ein Cocoa-Objekt und ein Core Foundation-Objekt identisch sind. Die Dokumentation für einige der wichtigsten Cocoa-Klassen weist darauf hin, dass sie „gebührenfrei“ (toll free) in ihre Core Foundation-Äquivalente geändert werden können. Damit ist gemeint, dass Sie ohne Zeit- und Speicheraufwand zwischen Cocoa- und Core Foundation-Referenzen casten können. Wenn Sie `myCFString` mit einer Cocoa-Methode nutzen wollen, casten Sie sie einfach wie folgt:

```
NSString *myNSString = (NSString*) myCFString;
```

Genauso können Sie Ihre `NSString`s mit Core Foundation-Funktionen nutzen, indem Sie sie einfach in die andere Richtung casten:

```
CFStringRef myCFString = (CFStringRef) myNSString;
```

Sie müssen allerdings aufpassen, da ähnliche Namen keine Garantie dafür sind, dass sich zwei Typen in der Form casten lassen können. So ist `NSBundle` zum Beispiel nicht „gebührenfrei“ mit dem ähnlich klingenden `CFBundleRef` verbunden, daher können Sie auch nicht zwischen beidem casten. Ihre beste Informationsquelle dazu ist der Dokumentations-Viewer in Xcode, da in der Übersicht für eine Klasse oder einen opaken Typ im Allgemeinen angegeben ist, ob sie/er sich gebührenfrei casten lassen kann.¹

Speicherverwaltung in der Core Foundation

Wenn Sie mit Core Foundation-Objekten arbeiten, nutzen Sie ein Speicherverwaltungssystem, das mehr dem von Cocoa ähnelt als dem `malloc()`- und `free()`-Schema der klassischen C-Programmierung.

In Cocoa erzeugen Sie Objekte mit der Methode `alloc`. In der Core Foundation erwarten die Funktionen, die Objekte erstellen, eine Referenz auf ein `Allocator`-Objekt, das Speicher für Objekte alloziert und dealloziert. Auch wenn Sie dieses Verhalten beeinflussen können, werden Sie es vermutlich nie tun – übergeben Sie einfach `NULL` oder `kCFAllocatorDefault`, um den Standard-Allocator zu erhalten.

Der Allocator ist während der gesamten Lebenszeit eines Objekts mit ihm verbunden – er verschiebt es bei Bedarf im Speicher und gibt diesen frei, wenn das Objekt nicht mehr länger benötigt wird. Das geschieht über einen Referenzzähler, der dem in Cocoa sehr stark ähnelt. Wir hoffen, dass Sie beim Durcharbeiten dieses Buchs die grundlegende Regel über die Speicherverwaltung stets im Kopf behalten, die in Abschnitt 3.1, *Die Softwarearchitektur des iPhone*, auf Seite 26 vorgestellt wurde: Sie besitzen jedes Objekt, das Sie durch eine Methode erhalten haben, die `alloc`, `new` oder `copy` im Namen trägt. Solche Objekte müssen Sie irgendwann wieder per `release` freigeben. Objekte, die Sie auf anderem Weg erhalten haben, dürfen Sie genau nicht per `release` freigeben, sofern Sie nicht durch `retain` ein Besitzer geworden sind.

Die Regeln in der Core Foundation sind sehr ähnlich:

- Sie besitzen jedes Objekt, das Sie über eine Funktion erzeugt haben, die `Create` oder `Copy` in ihrem Namen trägt. Solche Objekte müssen Sie irgendwann per `CFRelease()` wieder freigeben.
- Sie besitzen keine Objekte, die Sie auf anderem Weg erhalten haben (insbesondere über Funktionen, die `Get` im Namen tragen). Diese dürfen Sie nicht per `CFRelease()` freigeben.
- Wenn Sie eine Objektreferenz behalten müssen und sicherstellen wollen, dass sie nicht freigegeben wird, können Sie durch den Aufruf von `CFRetain()` ein Besitzer werden. Als solcher müssen Sie irgendwann wieder `CFRelease()` für das Objekt aufrufen.

1 Apples *Carbon-Cocoa Integration Guide* [App07a] führt ebenfalls im Abschnitt „Interchangeable Data Types“ alle gebührenfrei wechselbaren Beziehungen auf. Der ist offensichtlich für den Mac gedacht, aber alle dort beschriebenen Klassen und opaken Typen existieren auch im iPhone OS.

17.2 Systemklänge abspielen

Mit diesen Konzepten im Hinterkopf wollen wir uns nun die einfachste C-basierte Audio-API anschauen: die System Sound Services.

Diese API dient dazu, einfache Klänge aus dem Speicher abzuspielen, die höchstens 30 Sekunden lang sind. Sie ist dazu gedacht, UI-Klänge wiederzugeben, wie zum Beispiel das Klicken von Tasten oder die Warnsignale von Events, aber auch schlichte Hintergrundgeräusche in Spielen, wie zum Beispiel das Klacken eines Go-Steins, der auf dem Brett abgelegt wird. Die System Sound Services sind nicht geeignet für aufwendigere Audioanwendungen – sie können weder längere Klänge abspielen, noch haben Sie Einfluss auf die Lautstärke und die Stereopositionierung, und sie bietet auch keine Möglichkeit, einen Klang abubrechen, der gerade abgespielt wird. Zudem können Systemklänge nicht komprimiert werden. Sie müssen einen linearen PCM oder IMA/ADPCM in einem Container des Typs `.aif`, `.wav` oder `.caf` nutzen.

Wir werden mit dieser API experimentieren, indem wir eine View-basierte Anwendung namens `SystemSoundsDemo` erzeugen. Sie müssen für dieses Beispiel ein paar kurze Klänge herstellen.² Bei der herunterladbaren Anwendung haben wir den Sprachsynthesizer des Mac genutzt, um die Zahlen 1 bis 9 mit unterschiedlichen Stimmen zu sprechen und jede davon als CAF-Datei `1.caf` bis `9.caf` abzuspeichern. Diese haben wir dann der Resources Group des Projekts hinzugefügt.³

Um die System Sound Services-API zu nutzen, fügen Sie der Resources Group des Projekts `AudioToolbox.framework` hinzu und tragen in `SystemSoundsDemoViewController.h` den Header `#import <AudioToolbox/AudioToolbox.h>` ein.

Für die Benutzerschnittstelle benötigen wir nur eine einzelne Tabelle und ein Label. Auf die Tabelle kann der Benutzer tippen, um Klänge abzuspielen, das Label wird genutzt, um den Status anzuzeigen. Bearbeiten Sie zunächst `SystemSoundsDemoViewController.h` und fügen Sie der `@interface`-Deklaration die Protokolle `<UITableViewDataSource, UITableViewDelegate>` hinzu. Zusätzlich tragen Sie im `@interface`-Block noch `IBOutlet UILabel* statusLabel;` ein. Deklarieren Sie das Label zudem als `@property` und erzeugen Sie Setter und Getter per

2 Ironischerweise bietet die API trotz des Namens System Sound Services weder Zugriff auf die verschiedenen Warnsignale, die durch das System installiert werden, noch auf die Klingeltöne des Anwenders.

3 Da ein Leser gefragt hat – wir haben diese Dateien mit den Aktionen „Nach Text fragen“ und „Text in Audiodatei“ des Automator erstellt und sie dann von AIFF nach CAF mit dem Befehlszeilentool `afconvert` von Core Audio umgewandelt.

@synthesize in der Implementierungsdatei. Öffnen Sie nun im Interface Builder SystemSoundsDemoViewController.xib und fügen Sie die Tabelle und das Label hinzu. Danach verbinden Sie die Outlets dataSource und delegate der Tabelle mit File's Owner und das Label mit dem Outlet statusLabel.

Wenn Sie neun Klänge nutzen, wie wir es für das Beispiel tun, ist es einfach, die Data Source-Methoden für die Tabelle zu implementieren, indem Sie am Anfang des @implementation-Blocks ein paar Konstanten definieren:

```
MediaPlayer/SystemSoundsDemo/Classes/SystemSoundsDemoViewController.m
```

```
NSString *soundNames[] = {
    @"Vibrieren" , @"Eins" , @"Zwei" , @"Drei" , @"Vier" ,
    @"Fünf" , @"Sechs" , @"Sieben" , @"Acht" , @"Neun"
};
NSInteger SOUND_COUNT = 10;
```

Beachten Sie, wie wir die Zeile 0 für das „Vibrieren“ nutzen. Jetzt implementieren Sie die Data source-Methoden der Tabelle, indem Sie tableView:numberOfRowsInSection: den Wert SOUND_COUNT zurückgeben lassen und dem Standardverhalten von tableView:cellForRowAtIndexPath: eine zusätzliche Zeile hinzufügen.

```
MediaPlayer/SystemSoundsDemo/Classes/SystemSoundsDemoViewController.m
```

```
// Einrichten des Felds ...
cell.textLabel.text = soundNames[indexPath.row];
return cell;
```

Systemklänge erstellen

Wenn die Anwendung den Callback tableView:didSelectRowAtIndexPath: erhält, muss sie auf die Auswahl der Zeile reagieren, indem sie eine der CAFs lädt und den Klang abspielt. Wir können einen Systemklang mit der C-Funktion AudioServicesCreateSystemSoundID() erzeugen. Diese Funktion erwartet einen Parameter vom Typ CFURLRef, der eine URL im **file://**-Stil für den Klang übergibt, und die Adresse einer SystemSoundID⁴, die damit gefüllt werden kann. Die Funktion liefert einen **OSStatus** zurück, über den man erkennen kann, ob das Ganze erfolgreich war. Ist der Rückgabewert kAudioServicesNoError, können Sie die SystemSoundID nehmen und an AudioServicesPlaySystemSound() übergeben, um den Klang abzuspielen. Sie können den Systemklang

4 Beachten Sie, dass dieser Typ als 32-Bit-Zeiger definiert ist. Es handelt sich nicht um einen opaken Typ (denn das wäre ein Zeiger auf ein struct mit einem Ref am Ende des Namens). Daher müssen Sie ihn nicht per CFRelease() freigeben – er hat eine eigene Routine zum Aufräumen, zu der wir gleich noch kommen.

auch an die ähnliche Funktion `AudioServicesPlayAlertSound()` übergeben, die den Klang auf einem iPhone zusammen mit einer Vibration abspielt. Auf einem iPod touch ertönt ein Warn-Jingle (und der übergebene Klang wird ignoriert).

So erzeugen wir also einen Systemklang, wobei wir den `indexPath` nutzen, der an `tableView:didSelectRowAtIndexPath:` übergeben wurde:

MediaPlayer/SystemSoundsDemo/Classes/SystemSoundsDemoViewController.m

```
// Systemklang-ID für die ausgewählte Zeile erzeugen
SystemSoundID soundID;

OSStatus err = kAudioServicesNoError;
if (indexPath.row == 0) {
    // Spezialfall: Vibrieren
    soundID = kSystemSoundID_Vibrate;
} else {
    // CAF-Datei finden
    NSString *cafName = [NSString stringWithFormat:@"%d", indexPath.row];
    NSString *cafPath =
        [[NSBundle mainBundle] pathForResource:cafName ofType:@"caf" ];
    NSURL *cafURL = [NSURL fileURLWithPath:cafPath];
    err = AudioServicesCreateSystemSoundID((CFURLRef) cafURL, &soundID);
}
```

Dieser Code enthält eine Reihe von Dingen, auf die wir genauer eingehen möchten. Ist zum Beispiel die ausgewählte Tabellenzeile 0, wird in Code-Zeile 7 eine Konstante `kSystem-SoundID_Vibrate` der System Sound Services genutzt, um zu vibrieren (oder auf dem iPod touch eine Warnung abzuspielen). Ansonsten holen wir uns in Zeile 10 den Dateinamen über die Zeile (zum Beispiel 3.caf), finden die CAF-Datei im Bundle in Zeile 12 und wandeln den Pfad in der nächsten Zeile in eine `NSURL` um.⁵ Schließlich nutzen wir in Zeile 14 die gebührenfreie Umwandlung, um die `NSURL` in eine von `AudioServicesCreateSystemSoundID()` benötigte `CFURLRef` zu konvertieren. Achten Sie darauf, wie wir die Adresse unseres Zeigers `SystemSoundID` übergeben, der mit einer Referenz auf den Systemklang gefüllt wird, wenn der Aufruf erfolgreich ist. Das ist etwas, das wir bei C-Aufrufen immer wieder machen werden.

Systemklänge abspielen, überwachen und entsorgen

Jetzt sind wir im Prinzip dazu bereit, den Klang abzuspielen, aber um ein Detail müssen wir uns noch kümmern: den Lebenszyklus des Klangs. Wenn er abgespielt ist, müssen Sie ihn mit `AudioServices-`

⁵ Beachten Sie, dass Systemklänge nur über `file://`-URLs erzeugt werden können, nicht über `http://`-URLs oder andere Netzwerk-URLs.

`DisposeSystemSoundID()` entsorgen, aber es ist entscheidend, wann Sie es tun. Wollen wir den Klang im Speicher halten, weil wir ihn wiederholt verwenden wollen, können wir ihn zu einer Instanzvariablen machen und in `dealloc` entsorgen. Aber bei dieser App wollen wir den Klang direkt nach dem Abspielen loswerden.

Der Trick ist, zu wissen, wann Sie ihn entsorgen können, da die Anwendung weiterläuft, während er abgespielt wird. Löschen Sie den Klang direkt nach dem Aufruf von `AudioServicesPlaySystemSound()`, werden Sie ihn nicht hören, da er sofort nach dem Beginn des Abspielens verschwunden ist. Sie müssen also wissen, wann der Klang beendet ist, damit Sie ihn aufräumen können.

Dazu können wir einen *Completion Process* registrieren, eine C-Funktion, die aufgerufen wird, wenn der Klang fertig abgespielt ist. Sie registrieren einen Prozess mit `AudioServicesAddSystemSoundCompletion()`. Dies ist der Rest unserer Implementierung von `tableView:didSelectRowAtIndexPath:`, in dem ein Completion Process registriert, der Klang abgespielt und der Anwender benachrichtigt wird, wenn einer der Aufrufe der System Sound Services einen Fehlercode geliefert hat.

MediaPlayback/SystemSoundsDemo/Classes/SystemSoundsDemoViewController.m

```

1  if (err == kAudioServicesNoError) {
2      // Callback für Completion erzeugen
3      err = AudioServicesAddSystemSoundCompletion
4          (soundID,                // Fraglicher Klang
5          NULL,                    // Run Loop (NULL==main)
6          NULL,                    // Run Loop Mode (NULL==default)
7          SystemSoundsDemoCompletionProc, // Callback-Funktion
8          self                      // Daten für Callback
9      );
10     statusLabel.text = @"Spielt" ;
11     AudioServicesPlaySystemSound (soundID);
12 }
13 if (err != kAudioServicesNoError) {
14     CFErrorRef error = CFErrorCreate(NULL, kCFErrorDomainOSStatus, err, NULL);
15     NSString *errorDesc = (NSString*) CFErrorCopyDescription (error);
16     UIAlertView *cantPlayAlert =
17         [[UIAlertView alloc] initWithTitle:@"Abspielen unmöglich:"
18         message: errorDesc
19         delegate:nil
20         cancelButtonTitle:@"OK"
21         otherButtonTitles:nil];
22     [cantPlayAlert show];
23     [cantPlayAlert release];
24     [errorDesc release];
25     CFRelease (error);
26 }

```

Der Completion Process wird in den Zeilen 3 bis 9 registriert. Diese Funktion erwartet fünf Parameter (siehe auch die Dokumentation). `AudioServicesAddSystemSoundCompletion()` benötigt den abzuspielenden Klang, die Run Loop und den zu nutzenden Run Loop Mode (verwenden Sie einfach `NULL` für das Standardverhalten), den Namen der aufzurufenden C-Funktion und schließlich einen Zeiger auf „Benutzerdaten“, für die wir einfach unser View-Controller-Objekt mithilfe von `self` übergeben.

Ist der Completion Process registriert, können wir das Label auf „Spielt“ setzen (Zeile 10) und endlich den Klang in Zeile 11 abspielen.

Ist etwas schiefgegangen – hat also der `OSStatus` einer der Aufrufe der System Sound Services einen anderen Wert als `kAudioServicesNoError` – wird ein Fehlerbehandlungsblock (Zeilen 13 bis 26) aufgerufen, der dem Anwender eine Warnmeldung ausgibt. In Zeile 14 wird aus dem zurückgegebenen `OSStatus`-Wert eine `CFErrorRef` erzeugt, in Zeile 15 wird der beschreibende Text kopiert und (über eine gebührenfreie Umwandlung) in einen `NSString` umgewandelt, der dann in einer `UIAlertView` angezeigt wird.⁶

Da wir neue Objekte mit `CFErrorCreate()` und `CFErrorCopyDescription()` in den Zeilen 14 und 15 erzeugt haben, müssen wir sie auch wieder freigeben – was wir in den Zeilen 24 und 25 tun.

Jetzt müssen wir nur noch die eigentliche Callback-Funktion bereitstellen. Dieser Code (oder eine C-Forward-Deklaration) sollte weiter oben in der Quelldatei stehen als die `AudioServicesAddSystemSoundCompletion()`, die darauf verweist.

Der Funktions-Header wird durch `AudioServicesSystemSoundCompletionProc()` definiert, den Sie über einen Link in der Dokumentation zu `AudioServicesAddSystemSoundCompletion()` finden können. Die beiden Argumente sind der Systemklang, der fertig abgespielt wurde, und der Zeiger mit den Benutzerdaten. Dies ist eine einfache Implementierung:

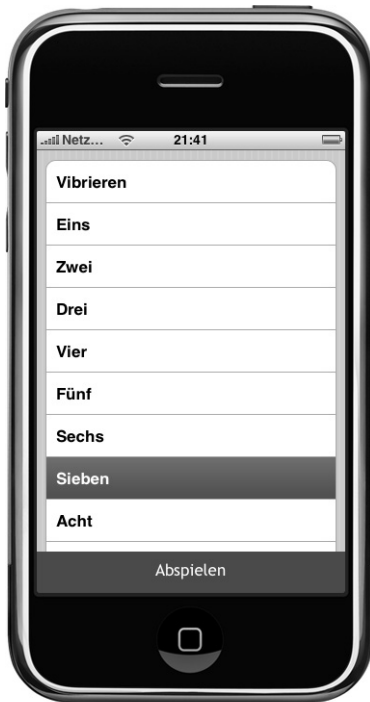
MediaPlayer/SystemSoundsDemo/Classes/SystemSoundsDemoViewController.m

```
void SystemSoundsDemoCompletionProc(
    SystemSoundID soundID,
    void          *clientData)
{
    AudioServicesDisposeSystemSoundID(soundID);
    ((SystemSoundsDemoViewController*)clientData).statusLabel.text=@"Stopp";
};
```

⁶ `CFErrorRef` lässt sich gebührenfrei in einen `NSError` umwandeln. Sie können also auch in einen `NSError` casten und dessen Methoden verwenden.

Wie Sie sehen können, nutzen wir zunächst die C-Funktion `AudioServicesDisposeSystemSoundID()`, um den Systemklang und alle verwendeten Ressourcen zu entsorgen. Dann setzen wir den Text des Labels auf „Stopp“. Auch wenn sich der Code innerhalb von `SystemSoundsDemoViewController.m` befindet, handelt es sich dennoch um eine C-Funktion, die das View-Controller-Objekt oder eine seiner Instanzvariablen nicht kennt. Also casten wir das Benutzerdatenobjekt in eine Instanz des `SystemSoundsDemoViewController` und greifen auf die Eigenschaft `statusLabel` zu, um den Text zu setzen.

Jetzt können Sie endlich Ihre Klänge nutzen. Sie haben Code zum Erzeugen von `SystemSoundIDs` sowie zum Abspielen und zum Verwerfen in einem registrierten Completion Process geschrieben. Mit BUILD AND RUN führen Sie die Anwendung nun im Simulator aus.



Wenn Sie auf den Tabellenfeldern herumtippen, sollten Sie die verschiedenen Systemklänge hören (oder das Vibrieren spüren). Auch wenn die Verwendung von prozeduralem C schwierig zu sein scheint – was nicht ganz falsch ist, wenn Sie nicht viel mit C arbeiten –, ist diese API doch recht einfach, verglichen mit dem Rest von Core Audio. Haben Sie lediglich einen Klang abzuspielen, sollten Sie dieser API den Vorzug vor den umfangreicheren APIs geben, die im nächsten Abschnitt

beschrieben sind. Natürlich wäre es immer noch einfacher, Ihre UI-Klänge mit der Objective-C-AVFoundation abzuspielen, aber die System Sound Services haben den Vorteil, dass sie weniger Ressourcen verbrauchen.

17.3 Überblick über Core Audio

Mit der System Sounds Services-API haben Sie einen ersten Vorge-schmack auf das erhalten, was sich in Core Audio so alles versteckt. Steigen Sie tiefer in dessen APIs ein, erhalten Sie Zugriff auf mehr Funktionalität – allerdings sehen Sie sich auch mehr Komplexität und Anspruch gegenüber.

Mit der etwas schwierigeren C-basierten API und dem notwendigen Wissen über digitale Signalverarbeitung wäre ein tieferer Einstieg in Core Audio ein eigenes Buch wert, daher werden wir in diesem Abschnitt nur einen Überblick über das Framework geben.

Technisch gesehen, ist die Infrastruktur von Core Audio in fünf Frame-works unterteilt: Audio Toolbox, Audio-Unit, AVFoundation und Core Audio (Header, die von mehreren Frameworks genutzt werden)⁷ sowie OpenAL. Darin gibt es eine Reihe von Services, die durch die Audio Toolbox bereitgestellt werden und bei denen es sich eigentlich um eigene kleine Frameworks handelt.

Audio Session Services

Wir haben das Konzept der Audio-Session – ein Ausdruck der Audionut-zung einer App und der Beziehungen zu anderen Audioquellen des Geräts – schon in Abschnitt 16.7, *Audio-Sessions*, auf Seite 382 vorge-stellt. In jenem Abschnitt haben wir mit `AVAudioSession` der AVFounda-tion gearbeitet. Diese Klasse ist eine Objective-C-Abstraktion der C-APIs in diesem Audio Toolbox Service. Mit der Audio-Session-API können Sie eine Audiokategorie definieren und die Eigenschaften der Audio-Hard-ware begutachten, wie zum Beispiel die Latenzzeit der Ein- und Ausgabe-Hardware, die Hardware-Abtastrate und so weiter. In der Audio Toolbox ist dies die API, die auf Unterbrechungen durch andere Anwendungen hinweist, wie zum Beispiel eingehende Anrufe, die die AVFoundation an den `AVAudioPlayer` und den `AVAudioRecorder` weitergibt.

⁷ Ab hier werden wir den Begriff „Core Audio“ nutzen, um uns auf die gesamte Infra-struktur zu beziehen, während „Core Audio Framework“ nur für dieses kleine Framework verwendet wird.

In dieser C-API gibt es noch weitere Funktionen, die aktuell nicht durch die `AVAudioSession` abgebildet werden. Sie können Audio-Sessions zum Beispiel dafür nutzen, den aktuellen Audiopfad zu begutachten, also herausfinden, ob die Audioausgabe an die Lautsprecher oder die Kopfhörer geht. Dazu lesen Sie die `kAudioSessionProperty_AudioRoute` der Session aus. Sie können auch einen Listener für diese Eigenschaft einrichten, so dass Sie einen Callback erhalten, wenn sich der Pfad ändert. So könnten Sie zum Beispiel automatisch einen Audio-Player stoppen, wenn der Anwender seine Kopfhörer entfernt.

Audio Queue Services

Audio Queue Services ist eine High-Level-API (verglichen mit dem Rest von Core Audio) zum Aufzeichnen und/oder Abspielen von Audiodaten von jeder Quelle, die Audio-Samples bereitstellen kann. Dabei werden Puffer mit Audiodaten ausgetauscht. So kann eine Queue zum Abspielen von Audiodaten zum Beispiel wiederholt Ihre Anwendung mit leeren Puffern aufrufen, die Sie dann mit Audiodaten füllen (entweder im komprimierten Format oder als PCM-Samples). Zum Aufzeichnen stellt die Queue Puffer mit vom Mikrofon oder anderen Eingabegeräten empfangenen Daten bereit.

Da die Audio Queue Services nicht zwingend auf das Speichern von Audiodaten in Dateien oder das Abspielen aus Dateien oder dem Speicher beschränkt sind, können Sie diese API für Anwendungen nutzen, die Audiodaten über das Netz transportieren, wie zum Beispiel Webradio-Player oder Chat-Anwendungen. Da Sie die Audiodaten direkt der Queue bereitstellen, können Sie auch verschlüsselte oder per DRM gekapselte Mediendaten unterstützen: Sie würden dann Ihre Quellen im Speicher in Audiodaten umwandeln und diese an die Queue weitergeben.

Audio Queue Services bieten auch eine API zum Messen des Pegels und eine Möglichkeit zum „Offline Rendering“, bei dem Sie Audiodaten in einen Puffer exportieren können, statt sie direkt an die Audio-Hardware zu übergeben.

Audio File Services

Als weiterer Bestandteil der Audio Toolbox vereinfachen die *Audio File Services* die Arbeit mit den verschiedenen Container-Formaten im Audiobereich. So kann eine Anwendung, die mit einem Audiostream arbeitet, diesen Stream in eines der diversen Dateiformate schreiben, die dieses Stream-Format unterstützen, oder daraus lesen, ohne sich um die vielen Unterschieden zwischen .caf, .mp3, .wav, .aif und so weiter kümmern zu müssen.

Wenn man gerade an diesen Details interessiert ist, stellt die Eigenschafts-API der Audio File Services einen Zugriff auf eine ganze Reihe von Metadaten bereit, wobei das Format unwichtig ist. So können Sie zum Beispiel die Metadaten über das Album, den Interpreten oder den Titel aus Musikdateien lesen, aber auch Metadaten nutzen, die für das Bearbeiten von Klängen sinnvoll sind, wie zum Beispiel Abschnitte, Marker und SMPTE-Timecodes.

Audio File Stream Services

Die *Audio File Stream Services*-API hilft Ihnen beim Umgang mit Audio-streams, die nicht unbedingt einen Anfang oder ein Ende haben müssen und damit auch nicht wie eine Datei an einer beliebigen Stelle angesprungen werden können. Dies ist eine weitere API, die Sie für netzwerkbasierende Audio-Apps nutzen können. Lesen Sie Daten aus einem Stream, können die Audio File Stream Services diesen parsen und das Format bestimmen. Zudem können Sie daraus Pakete mit Audiodaten erzeugen, die Sie dann an eine Audio-Queue weitergeben oder selbst verarbeiten.

Audio Conversion Services und Extended Audio Files

Die *Audio Conversion Services* der Audio Toolbox sind eine Gruppe von APIs für das Umwandeln kodierter Daten nach PCM oder umgekehrt (aber nicht zwischen zwei verschiedenen kodierten Formaten, sofern Sie nicht PCM als Zwischenstation nutzen). Sie können genutzt werden, um zwischen verschiedenen PCM-Formaten zu wechseln. Dabei wird mit Datenpuffern im Speicher gearbeitet. Sie müssen diesen Service nicht nutzen, wenn Sie eine Audio-Queue verwenden, da die Queue solche Konvertierungen schon automatisch durchführt.

Die dazugehörenden Extended Audio File Services kombinieren die Audio File und Audio Conversion Services, wodurch Sie Formate konvertieren und gleichzeitig in eine Datei schreiben beziehungsweise daraus lesen können.

Audio-Units und Audio-Unit Graphs

Dies ist ein vollständiges Framework und gleichzeitig das unterste, auf das Sie in der Core Audio-Infrastruktur zugreifen können. Audio-Units sind Software-Objekte, die Audio-Samples verarbeiten und dabei so verkettet werden können, dass man komplexe Verarbeitungsstreams für Audiodaten erhält.

Das wichtigste Element ist dabei die I/O-Unit, die über die Audio Component-API über die Konstante `kAudioUnitSubType_RemoteIO` geholt wird. Die I/O-Unit ist eine Abstraktion der Audioein- und -ausgabe-Hardware. Sie können einen „Renderer Callback“ mit der I/O-Unit verbinden, um Callbacks zu erhalten, über die Audio-Samples bereitgestellt werden, um diese direkt an den Lautsprecher oder die Kopfhörer zu schicken. Anders als der `AVAudioPlayer` und die `AudioQueues`, die Puffer verwenden und daher unausweichlich eine Latenz besitzen, ist die Latenz bei einer direkten Verbindung mit der I/O-Unit sehr klein. Sie können Ihre Audiodaten in weniger als 30 Millisekunden auf die Lautsprecher bringen – genauso schnell, wie der Bildschirm aktualisiert wird –, indem Sie direkt PCM an die I/O-Unit schicken. In die andere Richtung können Sie Audiodaten mit geringer Latenz aufzeichnen, wenn Sie mit der I/O-Unit arbeiten. Eine niedrige Latenz ist wichtig, wenn Sie eine Anwendung schreiben, in der Ihre Nutzer direkt mit Klängen interagieren sollen, zum Beispiel bei virtuellen Instrumenten, die als Reaktion auf eine Berührung, ein Schütteln oder die Stimme sofort verändert werden. Das iPhone SDK 3.0 bringt zusätzlich noch eine *Voice Processing I/O Audio-Unit* mit, die eine *Echo-kompensation* durchführt: Ermöglicht Ihre App einen Zwei-Wege-Voice-Chat, wird so verhindert, dass die Stimme von Benutzer A vom Mikrofon des Benutzers B wieder aufgenommen wird und als „digitales Echo“ erneut bei Benutzer A landet. Die Voice Processing I/O Unit wird von der Game Kit Voice Chat-API genutzt, die in Abschnitt 13.9, *Voice Chat*, auf Seite 316 beschrieben ist.

Neben den I/O-Units stellt das iPhone auch eine Reihe von Mixer-Units bereit, um mehrere Audioquellen zu kombinieren. Dazu gibt es eine iPod Equalizer Unit, um die von „echten“ Musik-Playern angebotenen Verarbeitungsmöglichkeiten bereitzustellen. Mit dem iPhone SDK 3.0 können Sie Ihre eigenen Audio-Units schreiben, die Ihrer Anwendung weitere Verarbeitungsmöglichkeiten anbieten.

Sie können Audio-Units miteinander verbinden, so dass die Ausgabe der einen Unit die Eingabe einer anderen Unit wird. Die Ausgabe-Unit am Ende holt die Audiodaten, indem sie eine `Render`-Methode für jede Unit aufruft, mit deren Eingabe sie verbunden ist, die dann wiederum Daten von allen ihren Verbindungen holt. Diese Verbindungen lassen sich allerdings nicht immer einfach einrichten und verwalten, da sie mit einer sehr eigenen API arbeiten. Die *Audio-Unit Graph*-API vereinfacht die Arbeit mit mehreren Audio-Units. Sie erzeugen einen `AUGraph` und füllen ihn dann mit `AUNodes`, bei denen es sich einfach um Wrapper für Audio-Units handelt. Der Graph vereinfacht die Verbindung und Koordination der Knoten in einem Graphen. Sie verbinden Knoten mit

`AUGraphConnectNodeInput()` und beginnen die Audioverarbeitung mit `AUGraphStart()`.

OpenAL

Das letzte Core Audio-Framework, das wir hier erwähnen, ist gar nicht von Apple. *OpenAL* ist ein Standard-Framework für 3-D-Audio. Sein Design lehnt sich stark an die Grafik-API OpenGL an, und beide sind dafür gedacht, zusammen für die Spieleprogrammierung genutzt zu werden. Auf dem iPhone baut OpenAL auf dem Rest von Core Audio auf, wobei eine 3-D-Mixer-Audio-Unit genutzt wird, um die räumliche Zuordnung des Klangs auf die Kopfhörer zu verteilen. Da die OpenAL-Daten an eine Mixer-Unit und dann an die I/O-Unit geschickt werden, werden auch hier sehr niedrige Latenzzeiten erreicht. Das ist vor allem für Spieleentwickler nützlich, die darauf bauen können, dass die Klänge im Spiel synchron zur Bildschirmausgabe abgespielt werden.

Als Third-Party-API sehen die OpenAL-Anweisungen ganz anders aus als der Rest von Core Audio, und es gibt mehr Parallelen zu OpenGL. OpenAL wird von Creative Technology betreut und aktuell (im Jahr 2009) wird die Website (<http://openal.org>) auf eine Site von Creative weitergeleitet, wo Sie die Dokumentation, Beispiele und ein Wiki finden.

Wir hoffen, dass Ihnen dieses Kapitel eine Idee davon vermitteln konnte, welche Audio-Frameworks es auf dem iPhone gibt und welche für eine bestimmte Art von Anwendung nützlich sein können. Weitere Informationen finden Sie in Apples *Core Audio Overview* [App07b]. Dort erhalten Sie auch die Referenz und Programmier-Guides für jedes Framework.

Im Laufe der Zeit hat sich Apple viel Mühe gegeben, die Dinge einfacher zu gestalten. Im iPhone SDK 2.0 musste man zum Abspielen von Audio-dateien noch `Audio-Queues` nutzen, die zwar verglichen mit Core Audio immer noch recht stark abstrahieren (man muss immerhin nicht die Dateien selbst lesen, kodierte Pakete nach PCM wandeln und diese dann an die I/O-Unit weitergeben), für viele Entwickler aber immer noch zu anspruchsvoll waren. Beim iPhone 2.2 kam der `AVAudioPlayer` des `AVFramework` hinzu und in 3.0 der `AVAudioRecorder` und die `AVAudioSession`. Wenn Sie wie ein Großteil der Entwickler einfach nur in einem kodierten Format abspielen oder aufzeichnen wollen, sind diese Klassen genau das Richtige. Haben Sie aber weitergehende Anforderungen, können Sie selbst bestimmen, welche Stufe von Core Audio Sie benötigen – wollen Sie Puffer mit kodierten Daten an eine `Audio-Queue` weitergeben, oder wollen Sie PCM-Daten in `Audio-Units` verarbeiten? –, und genau diese Funktionen nutzen.

Kapitel 18

Events, Gesten und Multi-Touch

Multi-Touch ist eines der Features, das die Leute echt beeindruckt. Die Möglichkeit, Fotos groß zu ziehen und zu drehen, begeistert jeden. Das Tolle ist, dass Sie mit dem UIKit ohne große Probleme selbst solche Interaktionen in Ihrer Anwendung nutzen können. In diesem Kapitel werden wir das Event-System behandeln und zeigen, wie unsere Anwendungen damit arbeiten und auf die Events reagieren können, um eine tolle Usability zu schaffen.

Das UIKit definiert ein anderes Interaktionsmodell als ein Desktop-Modell. Der offensichtlichste Unterschied ist das Tippen statt des Mausklicks. Aber viel interessanter ist die Möglichkeit mehrerer Berührungen gleichzeitig. In einer normalen Desktop-Umgebung gibt es eine Maus und nur einen Event-Typ, mit dem man sich gleichzeitig herumschlagen muss. Auf dem iPhone kann es mehrere Berührungen gleichzeitig geben, wobei dann manche Berührungen wieder beendet werden, während andere hinzukommen. Die Berührungen haben einen Kontext und können die Absichten des Benutzers anzeigen (zum Beispiel das Drehen eines Fotos). Das Event-System auf dem iPhone bietet uns die Chance, erstaunliche Benutzerinteraktionen zu ermöglichen. Lassen Sie uns anschauen, wie das funktioniert.

18.1 Event-Modell

Events sind das Lebenselixier einer Anwendung – um erstklassige Apps zu schreiben, müssen Sie verstehen, wie Ihre Anwendung Events erhält

und was möglich ist, wenn Sie sie erhalten. Dieser Abschnitt dreht sich nur um Events und das iPhone-Eventmodell.

Events kommen über die Responder-Kette aus dem Betriebssystem. Wir haben Responder und ihre Rolle in der Event-Verarbeitung kurz in Kapitel 2, *Hallo iPhone*, auf Seite 11 und an ein paar anderen Stellen angesprochen. Lassen Sie uns das Ganze nun im Kontext der Event-Verteilung behandeln. In Abbildung 18.1 sehen Sie, dass die Events vom OS verarbeitet und dann an die Event-Queue unserer Anwendung geschickt werden. Die Event-Loop holt sich Events aus der Queue und schickt sie dann an die Responder-Kette.

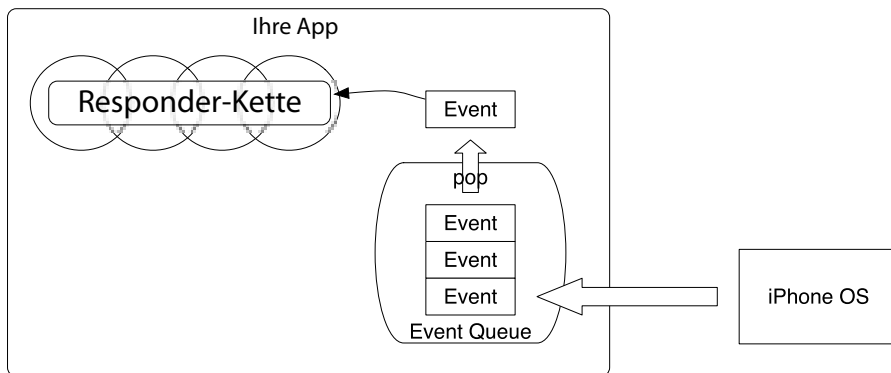


Abbildung 18.1: Event-Verteilungssystem

Die Responder-Kette ist einfach eine Liste mit Objekten, die durch ihre Eigenschaft `nextResponder` miteinander verbunden sind. Wenn ein Responder ein Event erhält, verarbeitet er entweder das Event oder er gibt es an seinen `nextResponder` weiter. Viele Objekte, die Sie zum Aufbau Ihrer Anwendung nutzen, sind Responder. Das Anwendungsobjekt, alle Views (einschließlich Fenstern) und alle UI-Widgets sind Responder. Durch die Verkettung können die Objekt in der Kette auf Events reagieren, die sie verstehen, und die weiterreichen, die sie nicht verstehen.

Wenn Sie viel mit dem UI oder Event-basierter Programmierung gearbeitet haben, ist Ihnen schon die eine oder andere Responder-Kette begegnet. Die Muster-Community nennt dieses Muster „Zuständigkeitskette“. Dabei handelt es sich einfach um eine Möglichkeit, *verarbeitende Objekte* (hier die Responder) so zu organisieren, dass sie *Befehlsobjekte* (in unserem Kontext die Events) konsistent an andere verarbeitende Objekte weiterreichen können, die sie nicht kennen.

Das Anwendungsobjekt ist der First Responder und das Objekt in Ihrer Anwendung, das alle Events empfängt, wenn sie aus der Event-Queue geholt werden. Sobald also das Event die Anwendung erreicht, wird es an den First Responder geschickt und dann in der Responder-Kette weitergereicht, bis es schließlich verarbeitet oder verworfen wird.

Berührt der Finger des Benutzers den Bildschirm über einer View, wird der View die Methode `touchesBegan:withEvent:` geschickt. Ist die View ein Button oder ein anderes Control, verarbeitet sie das Event, indem sie ihre Action aufruft. Ist die View eine Ihrer Views und Sie haben diese Methode nicht implementiert, wird die Standardimplementierung von `UIView` aufgerufen und die Nachricht in der Responder-Kette weitergereicht, bis entweder jemand reagiert oder das Event verworfen wird. Wenn der Anwender den Finger auf dem Bildschirm bewegt, folgt eine Serie von `touchesMoved:withEvent:-`Methoden auf die gleiche Art und Weise. Hebt der Benutzer seine(n) Finger wieder, wird `touchesEnded:withEvent:` geschickt und auch wieder von der Responder-Kette verarbeitet. Unterbricht das System die Anwendung (aufgrund eines System-Events wie einem eingehendem Anruf), wird der Anwendung `touchesCancelled:withEvent:` geschickt, damit sie die Chance hat, die Strukturen aufzuräumen, die sie für die aktuelle Event-Verarbeitung angelegt hat.

In jeder Event-Methode ist das erste Argument eine Reihe von Berührungen, die sich geändert haben, seit die letzte Event-Methode aufgerufen wurde. Das zweite Argument ist das Event, das alle Berührungen (über die Eigenschaft `allTouches`) oder alle Berührungen für eine bestimmte View enthält (über `allTouchesForView:`). Jede Berührung weiß, auf welcher View es losging, und sie kennt die aktuelle und die vorherige Position auf dieser View.

Das Implementieren dieser Event-verarbeitenden Methoden und die Verwendung der Berührungsinformationen ist die entscheidende Grundlage dafür, dass Ihre Anwendung Multi-Touch-Gesten nutzen kann. Im restlichen Kapitel werden wir uns genauer anschauen, wie Ihre Anwendung all die coolen Multi-Touch-Gesten verarbeitet, die die anderen iPhone-Apps auch kennen, wie zum Beispiel das Wischen und das Rotieren. Wir werden eine Reihe von Beispielen zur Behandlung von Events implementieren. Lassen Sie uns mit dem Verfolgen von Berührungen beginnen.

18.2 Berührungen verfolgen

Um Ihren Anwendern anständige Multi-Touch-Gesten präsentieren zu können, müssen Sie verstehen, wie man Berührungen verfolgt. Wenn sich der Finger des Anwenders über den Elementen bewegt, werden Richtung und Geschwindigkeit genutzt, um das Ziel auf interessante Art und Weise zu erkennen. Berühren Sie zum Beispiel ein E-Mail-Feld in der Mailanwendung und verschieben Ihren Finger in diesem Feld, erscheint der Lösch-Button, so dass Sie die E-Mail löschen können. Dieser Event-Typ ist als *Wischen* bekannt. In der Fotoanwendung kommt der Anwender durch *Blättern* zum nächsten oder vorherigen Bild, und durch ein *Zusammenstauchen* und *Dehnen* macht man das Bild kleiner oder größer. Durch das Verfolgen der Berührungen werden die Multi-Touch-Gesten also erst möglich.

Aber jetzt genug über Events – nun wollen wir uns Code anschauen und ein Beispiel erstellen. Erzeugen Sie ein neues View-basiertes Projekt und nennen Sie es Draw. In diesem Beispiel werden wir eine Anwendung bauen, die die Berührungs-Events in einer View verfolgt. Für jeden Aufruf der Methode `touchesMoved:withEvent:` werden wir die Position erfassen und dort einen Punkt zeichnen. Das Beispiel sehen Sie in Abbildung 18.2.



Abbildung 18.2: Punkte aus Berührungen

Damit diese Anwendung funktioniert, verfolgt die View sowohl den Anfang der Berührung als auch die Bewegung. Erzeugen Sie eine neue View-Subklasse und nennen Sie sie `DrawView`. Öffnen Sie die Datei `DrawViewController.xib` im Interface Builder und ändern Sie die Klasse der View in `DrawView`.

Wie bei allen Event-Streams beginnen wir in der Methode `touchesBegan:withEvent:`. Dies ist der Code:

MultiTouch/Draw/Classes/DrawView.m

```

1 - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
2     UITouch *touch = [touches anyObject];
3     CGPoint location = [touch locationInView:self];
4     Dot *dot = [[[Dot alloc] init] autorelease];
5     dot.x = location.x;
6     dot.y = location.y;
7     [self.dots addObject:dot];
8     [self setNeedsDisplay]; }

```

In diesem Code erzeugen wir den ersten Dot für die Abfolge der Events. In Zeile 3 erhalten wir die Position der Berührung in der View, erzeugen einen Dot für diese Position, fügen ihn dann der Liste hinzu und markieren die View schließlich als neu anzuzeigen. Die Klasse `Dot` ist sehr einfach – sie enthält nur die Werte `x` und `y`.

Die Liste der Dots wird dann genutzt, um Kreise (also unsere sichtbaren Punkte) in der View zu zeichnen – einen für jede gemerkte Position. Denken Sie daran, dass dieser Code uns nur zeigen soll, was wir durch das Verfolgen von Events erreichen können, nicht, wie man möglichst effizient auf dem iPhone zeichnet.

Als Nächstes folgt die Methode `touchesMoved:withEvent:`. Dies ist der Code:

MultiTouch/Draw/Classes/DrawView.m

```

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    CGPoint location = [touch locationInView:self];
    Dot *dot = [[[Dot alloc] init] autorelease];
    dot.x = location.x;
    dot.y = location.y;
    [self.dots addObject:dot];
    [self setNeedsDisplay]; }

```

In dieser Methode tun wir mehr oder weniger das Gleiche wie in `touchesBegan:withEvent:`, nur dass diese Methode viel häufiger aufgerufen wird als die vorherige.

Dot ist eine wirklich schlichte Subklasse von NSObject. Sie ist unsere Model-Klasse. Um sie zu erstellen, erzeugen Sie eine neue Subklasse von NSObject und ergänzen zwei Eigenschaften des Typs CGFloat mit den Namen `x` und `y`. Fügen Sie die Instanzvariablen und die `@synthesize`-Anweisungen in der Implementierungsdatei hinzu. Das ist alles, was Sie für die Klasse Dot tun müssen.

18.3 Tippen

Das Tippen ist eine weitere grundlegende Benutzerinteraktion auf dem iPhone. Beim Tippen wird der gleiche Kreislauf von `touchesBegan`, `touchesMoved` und `touchesEnded` durchlaufen. Damit Ihre Anwendung also auf ein- oder mehrfaches Tippen reagiert, implementieren wir auch eine oder mehrere dieser Methoden. Brauchen wir nur ein „doppeltes Tippen“ für einen unserer Responder, können wir `touchesBegan:withEvent:` implementieren. Ist die Eigenschaft `tapCount` gleich 2, rufen wir die Funktionalität für ein „doppeltes Tippen“ auf. Wenn nicht, ignorieren wir das Event.

Interessant wird es, wenn wir wollen, dass ein Responder eine Action als Reaktion auf ein einzelnes Tippen durchführt, während er bei einem doppelten Tippen eine andere Action auslöst. Das System schickt dem Responder das erste Event (einzelnes Tippen) und dann ein zweites Tippen als neues Event. Der Responder erhält Informationen, die so aussehen wie ein einzelnes Tippen und ein folgendes doppeltes Tippen.

Es zeigt sich, dass das gar nicht so schwierig zu lösen ist, aber es kann verwirren, wenn Sie sich solch einer Situation vorher noch nicht gegenübersehen. Der Trick ist, das Ausführen der ersten Tipp-Action so lange zu verzögern, bis Sie wissen, ob es sich wirklich um ein einzelnes Tippen oder um das erste Tippen einer Doppeltippfolge handelt. Auf dem iPhone erreichen wir das mit der Methode `performSelector:withObject:afterDelay:`. Diese Methode steckt die Ausführung einer Methode (ein Selector ist die Datenstruktur, die die ObjC-Runtime nutzt, um Methoden zu repräsentieren) in eine Queue, um sie erst später geschehen zu lassen.

Nachdem wir die Reaktion auf das erste, einzelne Tippen in eine Queue geschoben haben, können wir auf das zweite Tippen warten. Kommt es, brechen wir den Aufruf der ersten Action für das einzelne Tippen über eine weitere Methode `cancelPreviousPerformRequestsWithTarget:` ab. Das klingt vielleicht ein bisschen abstrakt, daher wollen wir uns den Code anschauen. In diesem Beispiel wird die View bei einem einzelnen Tippen rot, bei einem doppelten Tippen aber blau. Ein Teil des Codes finden Sie hier:

MultiTouch/MultiTap/Classes/MultiTapView.m

```

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    if(touch.tapCount == 1) {
        [self performSelector:@selector(turnRed) withObject:nil afterDelay:0.10f];
    }
    if(touch.tapCount == 2) {
        [self turnBlue];
    }
}

```

Wenn das Tippen endet (der Anwender also seinen Finger wieder hochnimmt) und der tapCount den Wert 1 hat, wird die Methode, die beim einzelnen Tippen aufgerufen werden soll, so in der Queue abgelegt, dass sie in 0.1 Sekunden ausgeführt wird. Hat tapCount den Wert 2, wird die Methode für das doppelte Tippen direkt ausgeführt. Wir brechen aber den Aufruf der Einzeltippmethode nicht ab, daher würde mit diesem Code bei einem doppelten Tippen die View erst blau und dann rot. Mit folgendem Code wird das behoben:

MultiTouch/MultiTap/Classes/MultiTapView.m

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch = [touches anyObject];
    if(touch.tapCount == 2) {
        [[self class] cancelPreviousPerformRequestsWithTarget:self
                                                    selector:@selector(turnRed)
                                                    object:nil];
    }
}

```

Wenn das zweite Tippen die Anwendung erreicht, wird die Methode für ein einzelnes Tippen abgebrochen. Die Verzögerung von 0.1 Sekunden gibt dem Anwender die Zeit, die er für das zweite Tippen benötigt. Folgt es zu spät, wird es zu einem weiteren einzelnen Tippen. Mit dieser Technik können wir unseren Respondern unterschiedliche Actions vermitteln. Vielleicht wollen Sie ja ein Control bauen, das bei einem einzelnen Tippen eine Auswahl trifft, bei einem doppelten Tippen aber in den Bearbeitungsmodus springt.

18.4 Multi-Touch-Gesten

Bisher haben wir uns Szenarien angeschaut, bei denen wir nur von einem Finger auf dem Bildschirm ausgegangen sind. Das ist für die meisten Anwendungen durchaus ausreichend, aber es gibt Fälle, in denen mehr als ein Finger notwendig sein kann, um komplexere Interaktionen durchzuführen. Das beste Beispiel dafür ist das „Zusammen-

stauchen“ bei den eingebauten Anwendungen. Man kann sich nur schwer ein besseres Interaktionsmodell vorstellen, das lediglich einen Finger nutzt. Wer will erst noch auf einen Button tippen (selbst wenn dieser erst eingeblendet wird), wenn die Geste mit den zwei Fingern so natürlich ist? Wir sollten in unseren Anwendungen möglichst das gleiche natürliche Interaktionsmodell erreichen. Machen Sie etwas Tolles, von dem der Anwender aber schon weiß, wie er es nutzen kann.

Lassen Sie uns zu der einfachen Zeichenanwendung zurückkehren, die wir in Abschnitt 18.2, *Berührungen verfolgen*, auf Seite 406 gebaut haben, und sie so verändern, dass sie für jede Berührung auf dem Bildschirm Punkte zeichnet. Es ist erstaunlich einfach, diese Anwendung um eine Multi-Touch-Fähigkeit zu erweitern.

Als Erstes müssen wir für die View Multi-Touch aktivieren. Das können wir im Code durch das Setzen der Eigenschaft `multipleTouchEnabled` auf YES erreichen, aber es ist sinnvoller, es direkt in der Konfiguration im IB vorzunehmen. Öffnen Sie `DrawViewController.xib` und wählen Sie Draw View (`DrawViewController.xib` ist Teil des Projekts Draw, das wir schon gebaut haben). Markieren Sie die Checkbox „Multiple Touch“ im Attributes Inspector (`Command-1`). Den Inspector mit der Checkbox sehen Sie in Abbildung 18.3.

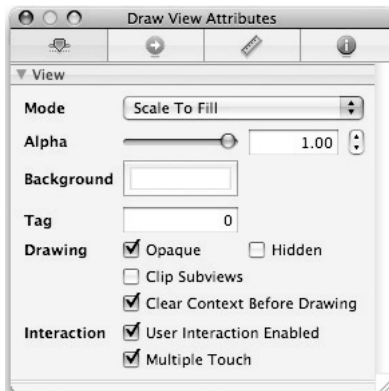


Abbildung 18.3: Aktiviertes Multiple Touch

Nachdem wir die View so angepasst haben, dass sie auch mehrere Berührungen gleichzeitig verfolgen kann, müssen wir das in unserem Code berücksichtigen. Dies ist der Code für die Methode `touchesBegan:withEvent:`. Lassen Sie sie uns genauer anschauen.

MultiTouch/DrawMulti/Classes/DrawView.m

```

1 - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
2     UITouch *anyTouch = [touches anyObject];
3     if(anyTouch.tapCount > 1) {
4         self.dots = nil;
5         [self setNeedsDisplay];
6         return;
7     }
8     BOOL needsDraw = NO;
9     for(UITouch *touch in event.allTouches) {
10        if(UITouchPhaseBegan == touch.phase) {
11            CGPoint location = [touch locationInView:self];
12            Dot *dot = [[[Dot alloc] init] autorelease];
13            dot.x = location.x;
14            dot.y = location.y;
15            [self.dots addObject:dot];
16            needsDraw = YES;
17        }
18    }
19    if(needsDraw) {
20        [self setNeedsDisplay];
21    }
22 }

```

Dieser Code unterscheidet sich gar nicht so sehr von dem vorher besprochenen. Nur beim Hinzufügen von Punkten wird jede Berührung berücksichtigt (beginnend in Zeile 9), die Teil des Events ist, statt jede Berührung im Parameter `touches`. Beachten Sie auch in Zeile 10, dass nur die Berührungen berücksichtigt werden, die gerade erst beginnen. So fügt die Anwendung keine neuen Punkte für bestehende Berührungen hinzu, wenn eine neue Berührung hinzukommt. Als Nächstes schauen wir uns die Methode `touchesMoved:withEvent:` an. Dies ist der Code:

MultiTouch/DrawMulti/Classes/DrawView.m

```

1 - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
2     BOOL needsDraw = NO;
3     for(UITouch *touch in event.allTouches) {
4         if(UITouchPhaseStationary != touch.phase) {
5             CGPoint location = [touch locationInView:self];
6             Dot *dot = [[[Dot alloc] init] autorelease];
7             dot.x = location.x;
8             dot.y = location.y;
9             [self.dots addObject:dot];
10            needsDraw = YES;
11        }
12    }
13    if(needsDraw) {
14        [self setNeedsDisplay];
15    }
16 }

```

Denken Sie daran, dass Berührungen so lange aktiv sind, wie die Finger den Bildschirm berühren. Verschiebt sich eine der Berührungen, wird die Methode `touchesMoved:withEvent:` aufgerufen. Wenn die Events verarbeitet werden, ignoriert die Anwendung in Zeile 4 alle Berührungen, die sich nicht verändert haben. Damit wird verhindert, dass dauernd neue Punkte hinzugefügt werden, auch wenn sich nichts geändert hat.

In dieser View soll auf noch etwas hingewiesen werden: Sie merkt sich keinerlei Informationen über die Events und Berührungen, daher muss auch nichts entfernt oder aufgeräumt werden, wenn die Events abgeschlossen sind. Es gibt also keine Notwendigkeit, `touchesCanceled:withEvent:` zu implementieren.

Mit ein bisschen Mathematik können wir herausfinden, ob sich die Berührungen aufeinander zu oder voneinander weg bewegen, und damit der Anwendung eine Stauchen- oder Strecken-Geste ermöglichen. Durch die Chance, mehrere Berührungen gleichzeitig verarbeiten zu können und zu wissen, ob sie sich bewegen oder nicht, können Sie Ihrer Anwendung überhaupt erst Gesten ermöglichen. Lassen Sie uns ein Beispiel für das Stauchen anschauen.

In Abbildung 18.4 sehen wir eine einfache Anwendung, die den Roboter in einem Layer anzeigt und es Ihnen ermöglicht, mit einer Geste hinein- oder herauszuzoomen. Die beiden grauen Kreise kommen vom Simulator – Sie können Multi-Touch durch das Drücken der Alt-Taste simulieren.



Abbildung 18.4: Vergrößern durch Berühren

Die Geste verfolgt zwei Finger auf dem Display. Wenn diese sich aufeinander zu bewegen, zoomen Sie heraus, kommen sie sich näher, zoomen Sie hinein. Die View muss sich den Abstand zwischen den beiden Berührungspunkten merken. Dies ist der Code für `touchesBegan:withEvent::`

MultiTouch/PinchZoom/Classes/PinchZoomView.m

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
    if(event.allTouches.count == 2) {
        pinchZoom = YES;
        NSArray *touches = [event.allTouches allObjects];
        CGPoint pointOne = [[touches objectAtIndex:0] locationInView:self];
        CGPoint pointTwo = [[touches objectAtIndex:1] locationInView:self];
        previousDistance = sqrt(pow(pointOne.x -pointTwo.x, 2.0f) +
                                pow(pointOne.y -pointTwo.y, 2.0f));
    } else {
        pinchZoom = NO;
    }
}
```

Mit diesem Code stellen wir zunächst sicher, dass es sich um eine Zwei-Finger-Geste handelt. Ist das der Fall, merken wir uns in einer Option, dass es sich um eine Stauchen-/Strecken-Geste handelt, und berechnen den Abstand zwischen den beiden Berührungen. Wir merken uns die Option und den Abstand in Instanzvariablen, so dass wir sie in der Methode `touchesMoved:withEvent:` nutzen können. Hier sehen Sie den entsprechenden Code dort:

MultiTouch/PinchZoom/Classes/PinchZoomView.m

```
1 - (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
2     if(YES == pinchZoom && event.allTouches.count == 2) {
3         NSArray *touches = [event.allTouches allObjects];
4         CGPoint pointOne = [[touches objectAtIndex:0] locationInView:self];
5         CGPoint pointTwo = [[touches objectAtIndex:1] locationInView:self];
6         CGFloat distance = sqrt(pow(pointOne.x -pointTwo.x, 2.0f) +
7                                 pow(pointOne.y -pointTwo.y, 2.0f));
8         zoomFactor += (distance -previousDistance) / previousDistance;
9         zoomFactor = fabs(zoomFactor);
10        previousDistance = distance;
11        self.robotLayer.transform =
12        CATransform3DMakeScale(zoomFactor, zoomFactor, 1.0f);
13    }
14 }
```

Wenn sich einer oder beide Finger auf dem Bildschirm bewegen, erhält die View einen Aufruf der Methode `touchesMoved:withEvent::`. Handelt es sich um eine Stauchen-/Strecken-Geste, berechnen wir den aktuellen Abstand zwischen den beiden Berührungspunkten und ermitteln daraus den Zoomfaktor. Dabei nutzen wir in Zeile 9 nur den absoluten

Wert, um keine negativen Zoomwerte zu erhalten. Ansonsten könnte Ihr Bild gespiegelt werden. Endet eine der Berührungen, setzen wir die Option und den Abstand zurück.

Dies ist der Code dafür:

MultiTouch/PinchZoom/Classes/PinchZoomView.m

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    if(event.allTouches.count != 2) {
        pinchZoom = NO;
        previousDistance = 0.0f;
    }
}
```

Wir setzen die Werte nur dann zurück, wenn wir keine zwei Berührungen mehr haben. Das ist wichtig, den jemand könnte einen dritten Finger auf den Bildschirm setzen und ihn dann wieder entfernen. Wir wollen die Werte nur dann zurücksetzen, wenn keine zwei Finger mehr auf dem Bildschirm sind.

Das Beispiel PinchZoom nutzt einen CALayer, um den Zoomeffekt zu erreichen. Es ist zwar nicht nötig, einen Layer zu verwenden, aber es war die einfachste Möglichkeit, die Prinzipien zu demonstrieren, ohne sich tief in die Interna von Core Graphics vergraben zu müssen. Vergeben Sie uns daher bitte die Verwendung von Layern, auch wenn sie noch nicht erklärt wurden, und glauben Sie uns, dass es viel einfacher ist, das Zoomen hier mit einem Layer umzusetzen. Mehr Details über Layer erfahren Sie in Kapitel 21, *Core Animation*, auf Seite 439.

In diesem Kapitel haben wir das Event-Modell und seine Verwendung in einer Multi-Touch-Umgebung auf dem iPhone kennengelernt. Zudem haben wir gesehen, wie unsere Anwendungen auf mehrere Berührungen reagieren können.

Kapitel 19

Zeichnen in eigenen Views

Bisher haben wir die Views unserer Anwendung immer aus fertigen Komponenten und mithilfe des Interface Builder erstellt. Manchmal werden Sie aber etwas Besonderes bauen wollen – etwas, das Sie nicht mit den normalen Komponenten machen können. In diesem Kapitel werden Sie eigene Benutzerschnittstellen erschaffen, indem Sie die Klasse `UIView` erweitern und Ihr eigenes UI mithilfe der Core Graphics Library zeichnen.

Die Core Graphics Library stellt die Primitive bereit, aus denen Sie Ihre eigenen „Zeichnungen“ zusammenstellen. Insbesondere in diesem Kapitel werden wir uns auf die Vektorzeichenfähigkeiten von Core Graphics konzentrieren. Die bildbasierten Features von Core Graphics folgen in Kapitel 20, *Bilder und Fotos zeichnen*, auf Seite 427.

Ein Wort der Warnung – es dauert lange, das Gewünschte auf den Bildschirm zu bekommen! Wenn Sie keine selbst erzeugten Views benötigen, bleiben Sie besser bei den vorgefertigten Komponenten.

19.1 Zeichenmodell

Das Zeichnen auf dem iPhone geschieht eher auf indirektem Weg. Sie beginnen nicht einfach mit dem Zeichnen, stattdessen müssen Sie Cocoa Touch mitteilen, was zu zeichnen ist. Wenn es dann gerade passt (während der Event-Verarbeitung), trifft Cocoa Touch alle notwendigen Vorbereitungen und weist Ihre View an, sich durch den Aufruf der Methode `drawRect:` zu zeichnen.

Zusätzlich zeichnen iPhone-Anwendungen im Allgemeinen nicht den kompletten Bildschirm neu. Stattdessen werden einzelne Views (oder sogar nur Bereiche der View) als zu aktualisieren markiert. Wir schreiben den Code, der die View aktualisiert, in der Methode `drawRect:`. Während des Event-Zyklus weist Cocoa Touch dann die View durch den Aufruf von `drawRect:` an, sich zu aktualisieren. Abbildung 19.1 zeigt, wie der Zeichenzyklus aussieht.



Abbildung 19.1: Zeichenzyklus

Während jedes Durchlaufs des Event-Zyklus prüft die Zeichnen-Maschinerie, ob Views als zu zeichnen markiert sind. Jede View, die neu gezeichnet werden muss, erhält einen Aufruf der Methode `drawRect:`, nachdem Cocoa Touch dafür gesorgt hat, dass alle notwendigen Vorbereitungen getroffen wurden. Wir werden uns gleich anschauen, wie man eine View als neu zu zeichnen markieren kann.

In der Methode `drawRect:` können Sie beliebige Routinen von Core Graphics nutzen, um all das zu zeichnen, was Sie benötigen. Ihre frisch gezeichneten Pixel werden auf den Bildschirm gebracht, nachdem `drawRect:` fertig ist.

19.2 Vektorgrafik

Lassen Sie uns eine einfache App erstellen, die ein blaues Rechteck erzeugt. Wie Sie aus dem Zeichenzyklus wissen, aktualisieren Sie den Inhalt des Rechtecks, wenn die Methode `drawRect:` der View aufgerufen wird. Dort werden wir also den Code unterbringen, mit dem das Rechteck ausgefüllt wird.

In diesem Beispiel beginnen wir ein neues View-basiertes Projekt mit dem Namen `Filling`. Ist es erzeugt, erstellen Sie eine neue Subklasse von `UIView` mit dem Namen `FillingView`. Im Dialogfeld `NEW FILE` wählen Sie „Objective-C class“ und dann im Pop-up `SUBCLASS OF` den Eintrag „`UIView`“.

Jetzt kommt der interessante Teil. Öffnen Sie `FillingView.m` und wechseln Sie zur Methode `drawRect:`. Sie besitzt den einladenden Kommentar „Drawing code“, der vom Template angelegt wurde und uns dazu auffordert, loszulegen. Es gibt hier natürlich nahezu unbegrenzte Möglichkeiten, aber wir wollen einfach anfangen. Zum Füllen des Rechtecks nutzen wir eine praktische Funktion.

Dies ist der Code:

Drawing/Filling/Classes/FillingView.m

```
-(void)drawRect:(CGRect)rect {
    [[UIColor blueColor] setFill];
    UIRectFill(rect);
}
```

Die erste Codezeile legt Blau als die Farbe fest, mit der das Rechteck gefüllt werden soll (mehr zum Setzen des Grafikkontextstatus, wie zum Beispiel der Füllfarben, folgt gleich). Die zweite Zeile füllt das `CGRect` (also den Parameter `rect`), der der Methode `drawRect:` übergeben wurde.

Die Funktion `UIRectFill()` erwartet ein Rechteck und erzeugt ein Core Graphics-Objekt, das als `CGPath` bezeichnet wird, und füllt es. Wie die Pfade erzeugt werden, wird hier vor uns verborgen (das macht die Sache einfacher). Diese Art von Funktionen wird als *Komfortfunktionen* bezeichnet, da sie uns das Zeichnen erleichtern. Aber wenn Sie mehr tun wollen, als nur ein Rechteck zu füllen, müssen Sie einen Schritt weitergehen und die zugrunde liegenden Pfadobjekte verstehen. Wir werden uns gleich um Pfade kümmern, erst einmal wollen wir uns aber anschauen, was wir tun müssen, um unsere neu erzeugte View in unsere Oberfläche zu bekommen.

Öffnen Sie die Datei `FillingViewController.xib`. Wählen Sie die View aus und ändern Sie deren Klasse durch -. Dann geben Sie in das Feld „Class“ den Wert `FillingView` ein. Klicken Sie auf den Button BUILD AND RUN, sollte nun Ihr gesamter Bildschirm blau sein.

Herzlichen Glückwunsch! Sie haben erfolgreich Ihre erste eigene View gebaut, die auf dem Bildschirm zeichnet. Das Ganze war gar nicht so kompliziert. Nachdem wir nun gesehen haben, wie die grundlegenden Mechanismen funktionieren, um etwas auf den Bildschirm zu bringen, wollen wir uns genauer mit den Pfaden befassen.

19.3 Pfade

Wir nutzen Pfade, um Rechtecke, Kreise und andere Formen zu beschreiben, die wir zeichnen wollen. Pfade können sichtbar gemacht, gefüllt oder sichtbar gemacht und gefüllt werden. Das Sichtbarmachen eines Pfads führt dazu, dass er anhand des Status des Grafikkontexts gezeichnet wird. Beim Füllen wird der Bereich innerhalb des Pfads gefüllt. In Core Graphics gibt es vier Primitive, um Pfade, Punkte, Linien, Bogen und Kurven darzustellen.

Ein Punkt ist eine einzelne Position im zweidimensionalen Raum. Stellen Sie sich ihn nicht als Pixel vor, sondern eher so, wie Sie es früher im Mathematikunterricht getan haben. Ein Punkt hat keinerlei Ausdehnung. Zeichnen Sie einen, werden Sie daher nichts auf dem Bildschirm sehen. Um etwas zu sehen, würden Sie eines der anderen Primitiven nutzen, die aus Punkten erstellt werden.

Eine Linie ist durch zwei Punkte definiert – den Anfangs- und den Endpunkt. Eine Linie kann sichtbar gemacht werden, wobei die Eigenschaften des grafischen Kontexts (wie die Linienbreite oder die Farbe) auf die Linie zwischen den beiden Punkten angewendet werden. Linien umhüllen keine Fläche, daher kann man sie nicht ausfüllen. Stattdessen würden Sie eine Reihe von Linien oder Kurven aneinandersetzen, um eine Form zu erzeugen, dann den Pfad schließen und die eingeschlossene Fläche füllen.

Ein Bogen ist durch einen einzelnen Punkt (das Zentrum), einen Radius, einen Anfangs- und einen Endwinkel definiert. Um einen Kreis zu erhalten, nutzen Sie einen Bogen mit einem Anfangswinkel von 0 und einem Endwinkel von 2π . Da ein Bogen ein Pfad ist und einen Bereich auszeichnet, kann er gefüllt oder sichtbar gemacht werden (oder beides gleichzeitig). Füllen Sie einen Pfad, der nicht geschlossen ist (jeder Pfad, der nicht explizit geschlossen ist, ist offen), schließt die Füllroutine ihn für Sie.

Kurven werden schließlich durch eine Bézierkurve repräsentiert. Sie definieren eine Kurve durch vier Punkte: Zwei Punkte beschreiben Anfang und Ende, während die beiden anderen die Tangente der Kurve an jedem Punkt definieren. Eine Kurve umschließt wie eine Linie keine Fläche, aber da eine Bézierkurve auch eine Kurve ist, wird ein Aufruf der Füllroutine den Pfad schließen (indem eine Linie vom Anfangs- zum Endpunkt gezeichnet wird) und so eine Fläche erstellen.

Pfade können genutzt werden, um ausgesprochen aufwendige Grafiken zu erstellen. Sie können mit Core Graphics so gut wie alles zeichnen, was Sie auch mit fortgeschrittenen Vektorzeichenprogrammen erstellen können.

Das einfache Füllen eines Rechtecks in einer Zeile, das wir im vorigen Beispiel gesehen haben, lässt sich ganz leicht nutzen, aber wie viele andere einfach zu nutzende Dinge verlieren wir eine Menge Flexibilität. Wir werden dieses Beispiel selbst einmal aufbauen, um zu zeigen, was hinter diesem einfachen Funktionsaufruf steckt, um die Pfad-Primitiven besser zu verstehen.

Dies ist der Code für eine neue Methode `drawRect:`, die ein 100-x-100-Rechteck am Ursprung 10,10 füllt. Wir haben ein komplett neues Projekt aufgesetzt und eine neue View erstellt (und die Klasse der View im IB geändert), aber Sie können auch einfach die Methode `drawRect:` in Ihrer bestehenden `FillingView` ändern, wenn Sie möchten. Wir werden den Grafikkontext gleich besprechen, hier konzentrieren wir uns auf den Code für den Pfad.

Drawing/FillingSquare/Classes/FillingView.m

```
- (void)drawRect:(CGRect)rect {
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathMoveToPoint(path, NULL, 10.0f, 10.0f);
    CGPathAddLineToPoint(path, NULL, 100.0f, 10.0f);
    CGPathAddLineToPoint(path, NULL, 100.0f, 100.0f);
    CGPathAddLineToPoint(path, NULL, 10.0f, 100.0f);
    CGPathCloseSubpath(path);
    CGContextRef ctx = UIGraphicsGetCurrentContext();
    CGContextSetFillColorWithColor(ctx, [UIColor blueColor].CGColor);
    CGContextAddPath(ctx, path);
    CGContextFillPath(ctx);
}
```

Wenn Sie diese Anwendung ausführen, sollten Sie so etwas wie in Abbildung 19.2 angezeigt bekommen. Anstatt die gesamte View zu füllen, haben wir nun ein Quadrat.

Nachdem Sie nun gesehen haben, wie Sie einen Pfad mit einer wirklich einfachen Form aufbauen, wollen wir uns einem etwas komplexeren Beispiel zuwenden: einer Balkengrafik mit abgerundeten Enden. In Abbildung 19.3 sehen Sie einen Screenshot der Anwendung.



Abbildung 19.2: Mit blau gefüllte View



Abbildung 19.3: Balkengrafik zeichnen

Lassen Sie uns den Code Stück für Stück anschauen. Dies ist die Methode `drawRect::`

Drawing/BarGraph/Classes/BarGraphView.m

```

1  -(void)drawRect:(CGRect)rect {
2      CGSize size = self.bounds.size;
3      CGFloat width1 = size.width * 0.75f;
4      CGFloat width2 = size.width * 0.35f;
5      CGFloat width3 = size.width * 0.55f;
6
7      CGFloat height = size.height * 0.2f;
8
9      CGRect one = CGRectMake(0.0f, height + 5.0f,
10                             width1, height -10.0f);
11      CGRect oneText = CGRectMake(10.0f, height + 25.0f,
12                                 width1, height -30.0f);
13      CGRect two = CGRectMake(0.0f, 2.0 * (height + 5.0f),
14                             width2, height -10.0f);
15      CGRect twoText =CGRectMake(10.0f, 2.0 * height + 30.0f,
16                                width2, height -30.0f);
17      CGRect three = CGRectMake(0.0f, 3.0 (height + 5.0f),
18                               width3, height -10.0f);
19      CGRect threeText =CGRectMake(10.0f, 3.0 height + 35.0f,
20                                  width3, height -30.0f);
21
22      CGContextRef ctx = UIGraphicsGetCurrentContext();
23
24      [[UIColor blueColor] setFill];
25      CGPathRef pathOne = [self pathInRect:one];
26      CGContextAddPath(ctx, pathOne);
27      CGPathRelease(pathOne);
28      CGContextFillPath(ctx);
29      [[UIColor blackColor] setFill];
30      [@"Eins" drawInRect:oneText withFont:[UIFont systemFontOfSize:34]];
31      [[UIColor redColor] setFill];
32      CGPathRef pathTwo = [self pathInRect:two];
33      CGContextAddPath(ctx, pathTwo);
34      CGPathRelease(pathTwo);
35      CGContextFillPath(ctx);
36      [[UIColor blackColor] setFill];
37      [@"Zwei" drawInRect:twoText withFont:[UIFont systemFontOfSize:34]];
38      [[UIColor yellowColor] setFill];
39      CGPathRef pathThree = [self pathInRect:three];
40      CGContextAddPath(ctx, pathThree);
41      CGPathRelease(pathThree);
42      CGContextFillPath(ctx);
43      [[UIColor blackColor] setFill];
44      [@"Drei" drawInRect:threeText withFont:[UIFon systemFontOfSize:34]];
45  }

```

Die Methode ist recht lang, aber es passieren eigentlich nur fünf Dinge:

1. In Zeile 3 berechnen wir die Breite des Balkens.
2. In Zeile 7 berechnen wir die Höhe des Balkens.

3. In Zeile 9 erstellen wir ein Rechteck für den zu zeichnenden Balken.
4. In Zeile 25 fügen wir den Pfad dem Kontext hinzu und füllen ihn dann.
5. In Zeile 30 zeichnen wir das Label.

Natürlich wird jeder dieser fünf Schritte für jeden der Balken wiederholt, daher sieht der Code so umfangreich aus. Lassen Sie sich nicht davon beeindrucken. Es geschehen wirklich nur die fünf Dinge. Um nun zu sehen, wie der Pfad erstellt wird, wollen wir uns den folgenden Code anschauen.

Drawing/BarGraph/Classes/BarGraphView.m

```
-(CGPathRef) pathInRect:(CGRect)rect { CGMutablePathRef path =
    CGPathCreateMutable(); CGFloat radius = CGRectGetHeight(rect) / 2.0f;
    CGPathMoveToPoint(path, NULL, CGRectGetMinX(rect),
        CGRectGetMinY(rect));
    CGPathAddLineToPoint(path, NULL, CGRectGetMaxX(rect) -radius,
        CGRectGetMinY(rect));
    CGPathAddArc(path, NULL, CGRectGetMaxX(rect) -radius,
        CGRectGetMinY(rect) + radius,
        radius, -M_PI / 2.0f, M_PI / 2.0f, NO);
    CGPathAddLineToPoint(path, NULL, CGRectGetMinX(rect),
        CGRectGetMaxY(rect));
    CGPathCloseSubpath(path);
    CGPathRef immutablePath = CGPathCreateCopy(path);
    CGPathRelease(path);
    return immutablePath;
}
```

Dieser Code sieht auf den ersten Blick auch wieder ziemlich komplex aus, aber auch er ist es nicht. Wir berechnen zunächst den Radius des abgerundeten Endes des Balkens als halbe Balkenhöhe. Dann bewegen wir uns zur linken oberen Ecke des Balkens, setzen unseren ersten Punkt, bewegen uns um den Halbkreis herum, bis wir zur unteren linken Ecke gelangen, und schließen dann den Pfad. Danach erstellen wir eine unveränderliche Kopie des Pfads und kehren zurück.

Diese Beispiele zeigen Ihnen, was mit Core Graphics möglich ist, aber es gibt noch viel mehr. Ein Großteil der Flexibilität und Mächtigkeit von Core Graphics entsteht durch den Grafikkontext. Lassen Sie uns ihn daher als nächstes anschauen.

19.4 Grafikkontext

Denken Sie mal an die Zeit zurück, als Sie noch Kind waren. Wie haben Sie da gemalt? Wenn Sie Rot wollten, haben Sie den roten Wachsmalstift genommen und Ihre Zeichnung verschönert. Wenn Blau an der Reihe war, haben Sie stattdessen zum blauen Stift gegriffen. Nun, der Grafikkontext in Core Graphics ähnelt in dieser Hinsicht stark Ihrer Hand. Wenn wir die Farben und andere Parameter des Grafikkontexts wechseln, wechseln wir zwischen verschiedenen Wachsmalstiften. Im vorigen Beispiel haben wir die Codezeile `[UIColor blueColor];` unauffällig übersprungen. Jetzt wollen wir uns anschauen, was da wirklich passiert.

Sie können den Wert vieler verschiedener Variablen im Grafikkontext zuweisen, um die Art und Weise, wie gezeichnet wird, zu beeinflussen. Wir haben die Füllfarbe in den vorigen beiden Beispielen gesehen, aber es gibt noch viele andere Variablen, die festlegen, was geschieht, wenn wir einen Pfad füllen oder sichtbar machen. So können wir zum Beispiel die Breite des „Stifts“ auf 5 setzen. Zeichnen wir dann einen sichtbaren Pfad, ist der Bereich, der durch unsere virtuelle Tinte abgedeckt wird, 5 Einheiten breit. Weitere Variablen haben ebenfalls Einfluss auf das Darstellen eines Pfads, wie zum Beispiel die Strichelung, die Zeichenfarbe und so weiter. Lassen Sie uns zum gefüllten Quadrat zurückkehren und den Pfad darstellen, nachdem wir die Strichbreite gesetzt haben. Dies ist der Code, um die Strichweite auf 10 und die Strichfarbe auf Gelb zu setzen:

Drawing/StrokedFilledSquare/Classes/StrokedFilledSquareView.m

```
- (void)drawRect:(CGRect)rect {
    // Pfad erzeugen und aufbauen
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathMoveToPoint(path, NULL, 10.0f, 10.0f);
    CGPathAddLineToPoint(path, NULL, 100.0f, 10.0f);
    CGPathAddLineToPoint(path, NULL, 100.0f, 100.0f);
    CGPathAddLineToPoint(path, NULL, 10.0f, 100.0f);
    CGPathCloseSubpath(path);
    CGContextRef ctx = UIGraphicsGetCurrentContext();
    // Füllparameter setzen und Pfad füllen
    CGContextSetFillColorWithColor(ctx, [UIColor blueColor].CGColor);
    CGContextAddPath(ctx, path);
    CGContextFillPath(ctx);
    // Strichparameter setzen und Strich entlang des Pfads zeichnen
    CGContextSetStrokeColorWithColor(ctx, [UIColor yellowColor].CGColor);
    CGContextSetLineWidth(ctx, 10.0f);
    CGContextAddPath(ctx, path);
    CGContextStrokePath(ctx);
}
```

Es gibt ein paar wichtige Dinge, auf die wir hier hinweisen wollen. So setzen wir alle Werte, an denen wir interessiert sind, vor dem eigentlichen Zeichnen. Denken Sie daran, dass der Grafikkontext den Wachsmalstift definiert. Haben wir einen roten Stift in der Hand, wenn wir eine Linie zeichnen, erhalten wir eine rote Linie. Wollen wir eine gelbe Linie, müssen wir zum gelben Stift greifen. Worauf wir außerdem hinweisen wollen, ist, dass wir den Pfad dem Kontext erneut hinzufügen müssen. Zeichnen oder füllen wir einen Pfad, wird er aus dem Kontext entfernt. Unser neu gezeichnetes Quadrat sehen Sie nun in Abbildung 19.4.

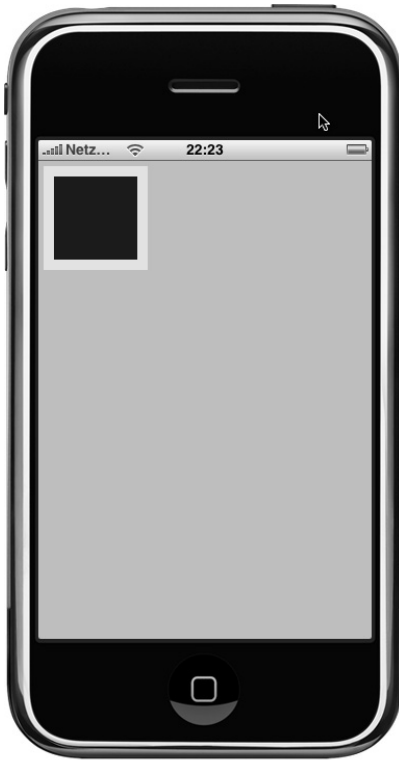


Abbildung 19.4: Gefülltes Quadrat mit Rand

Mit dem Zeichnen von Pfaden und dem Verändern des Grafikkontexts können Sie noch viel mehr anstellen. Wir haben nur an der Oberfläche des Möglichen gekratzt. Viel mehr Details zur Vektorgrafik mit der Core Graphics Library finden Sie in *Programming with Quartz* [GL06] oder der Onlinedokumentation von Apple, *Introduction to Quartz 2D Programming Guide* [App07c].

19.5 Eine neue View anzeigen

Abschließend sollten wir uns noch Gedanken darüber machen, wie wir die View dazu zwingen können, ihren Inhalt neu zu zeichnen. Für jeden neuen Pixel, den wir auf den Bildschirm bringen, zahlen wir mit Performance. Die Pixel, die in einem `drawRect:` mit Farbe gefüllt werden, müssen schließlich auch an die Grafikkarte (und damit auf den Bildschirm) gebracht werden – und das kostet Zeit. Da es teuer ist, neue Pixel auf den Bildschirm zu bringen, achtet das UIKit sorgfältig darauf, nur dann zu zeichnen, wenn es unbedingt notwendig ist. Bei jedem Durchlauf der Event-Loop prüft die Zeichnen-Maschinerie von Cocoa Touch, welche Views neu gezeichnet werden müssen. Standardmäßig muss nichts neu gezeichnet werden, wir müssen also explizit die Views als neu zu zeichnen markieren, bevor etwas passiert. Wenn eine View neu gezeichnet werden muss, konzentrieren sich die Darstellungsroutinen auf diese View, konfigurieren einen Grafikkontext zum Zeichnen, machen ihn zum aktuellen Kontext und schicken dann `drawRect:` an die View.

Sie können eine View, die neu gezeichnet werden muss, markieren, indem Sie die Methode `setNeedsDisplay` oder `setNeedsDisplayInRect:` aufrufen. Die erste Methode markiert die gesamte View als neu zu zeichnen, die zweite hat einen Rechteckparameter, mit dem wir eine Unterregion der View als aktualisierungsbedürftig markieren können. Welche der Methoden Sie auch verwenden – Cocoa Touch wird das entsprechende Rechteck (entweder den gesamten Bereich oder den Unterbereich) an die Methode `drawRect:` übergeben, wenn die View tatsächlich neu gezeichnet werden soll.

Sie rufen `drawRect:` niemals direkt auf. Stattdessen markieren Sie eine View (oder einen Teil davon) als aktualisierungsbedürftig. Cocoa Touch und die Zeichnen-Maschinerie kümmern sich dann darum, einen Grafikkontext für Sie vorzubereiten. Wenn dann der richtige Zeitpunkt für das Zeichnen gekommen ist, ruft Cocoa Touch die Methode `drawRect:` Ihrer View auf.

Fertigen Sie komplexe Zeichnungen mit langwierigen Berechnungen oder Vorbereitungen an, ist es sinnvoll, in Ihrer Methode `drawRect:` den Rechteckparameter zu nutzen, um den Bereich zu begrenzen, in dem Ihre Neuberechnungen und das Neuzeichnen vorgenommen werden müssen. Wenn Ihre Zeichnungen weniger aufwendig sind, kann es durchaus in Ordnung sein, immer die gesamte View neu zu zeichnen. Sie können nur herausfinden, was sinnvoller ist, wenn Sie beide Varianten ausprobieren und dabei die Performance messen. In Kapitel 28, *Performance Tuning*, auf Seite 545 erfahren Sie mehr darüber.

In diesem Kapitel haben wir die Grundlagen des Zeichnens auf dem iPhone behandelt. Darauf können Sie nun aufbauen und richtig coole Anwendungen bauen, die die Vorteile von Core Graphics für sich nutzen.

Kapitel 20

Bilder und Fotos zeichnen

In diesem Kapitel werden wir lernen, wie man auf dem iPhone mit Bildern arbeiten kann. Die Bilder können mit der Anwendung mitgeliefert werden, Teil der Fotobibliothek sein oder mit der iPhone-Kamera aufgenommen werden.

Wo auch immer die Bilder herkommen – wir wollen sie anzeigen, und das ist das Thema dieses Kapitels. In Kapitel 19, *Zeichnen in eigenen Views*, auf Seite 415 haben wir das Zeichenmodell und die Verwendung der Vektorzeichenroutinen besprochen. In diesem Kapitel wollen wir unser Wissen nutzen, um Bilder auszugeben. Wir werden Bilder aus unseren Projekten verwenden, aber auch Fotos aus der Bibliothek holen.

Es gibt viele coole Möglichkeiten, die Fotos in einer Anwendung zu nutzen. Wäre es zum Beispiel nicht toll, eine Anwendung zu haben, die es uns erlaubt, ein Foto unserer Kinder zu machen und diese dann in Wackelköpfe zu verwandeln, wenn wir das iPhone schütteln? Verrückt, aber lustig! Eine andere schöne Möglichkeit zur Integration von Bildern findet sich in der Kontakte-Anwendung. Sie können für eine Person ein Foto aus Ihrer Bibliothek einfügen oder sogar die Kamera nutzen, um ein Bild zu machen. Wie so oft ist das Schwierigste daran die Entscheidung, wie der Anwender das Ganze „erleben“ soll. Der eigentliche Code ist recht einfach. Sie müssen nur ein paar Verbindungen im Interface Builder herstellen und eine Handvoll Methoden implementieren.

Unsere Reise durch dieses Kapitel beginnt mit der Verwendung der `UIImageView`, um unsere Bilder zu zeichnen. Diese Klasse macht es ausgesprochen einfach, Bilder auf den Bildschirm zu bekommen. Danach werden wir uns die Klasse `UIImage` anschauen, um herauszufinden, wie wir damit Bilder ausgeben können. Diese Klasse ist zwar nicht so einfach zu verwenden wie die `UIImageView`, aber wir erhalten mehr Flexibilität bei der Ausgabe. Schließlich werden Sie erfahren, wie Sie mit der Fotobibliothek und der Kamera zusammenarbeiten können, um die Anwendungen für die Benutzer persönlicher zu gestalten.



Joe fragt...

Kann ich jedes beliebige Bildformat verwenden?

Nicht ganz. Auch wenn das iPhone viele verschiedene Bildformate zulässt (JPEG, GIF und so weiter), ist es für PNG-Dateien optimiert. Stellen Sie also sicher, dass Ihre Kunstwerke im PNG-Format abgelegt sind, bevor Sie Ihre Anwendung ausliefern.

20.1 Grundlagen der Bildausgabe

Um Bilder auszugeben, haben Sie zwei Optionen:

- Die `UIImageView` ist die einfachste Methode, ein Bild auf dem Bildschirm anzuzeigen.
- Die zweite Option ist die Verwendung eines Objekts vom Typ `UIImage`, um das Bild zu repräsentieren und es dann in einer View anzuzeigen.

Wir werden mit dem einfachsten Ansatz loslegen und ein Bild in eine `UIImageView` stecken.

Wenn Sie keine komplexen Anforderungen an die Art und Weise der Ausgabe Ihrer Bilder haben, ist die `UIImageView` eine wunderbare Möglichkeit. Sie müssen dazu nur die Eigenschaft `image` auf der Image-View setzen, um Ihr Bild auf dem Bildschirm angezeigt zu bekommen.

Lassen Sie uns eine Beispielanwendung erstellen, die eine `UIImageView` nutzt, um ein Bild aus Ihrem Anwendungs-Bundle auszugeben.

Erstellen Sie ein neues View-basiertes Projekt mit dem Namen Image-View. Dann müssen Sie dem Projekt ein Bild hinzufügen. Nehmen Sie

einfach das Bild `photo.png` aus dem Beispielcode oder ein eigenes. Um eine bestehende Datei Ihrem Projekt hinzuzufügen, markieren Sie die Gruppe, in der Sie es einfügen wollen, klicken mit der rechten Maustaste, wählen **ADD EXISTING FILES** und folgen den Schritten im Wizard. Wir markieren fast immer die Checkbox „Copy items into destination group’s folder (if needed)“ auf der letzten Seite des Wizard, so dass wir eine Kopie statt einer Referenz auf die Datei erhalten.

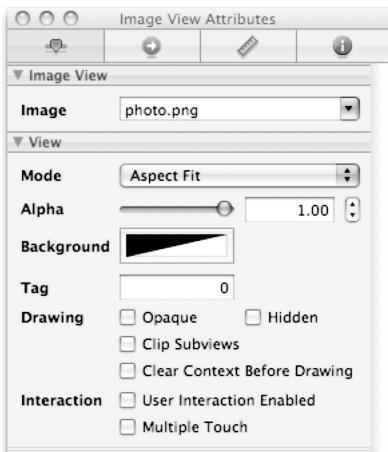


Abbildung 20.1: Ändern des Bildes

Nachdem wir nun ein Bild haben, können wir den Rest der Arbeit für dieses Beispiel im Interface Builder erledigen. Öffnen Sie die Datei `ImageViewViewController.xib`. Wählen Sie aus der Library ($\mathbb{K}$ - $\square$ - $\mathbb{L}$) eine `UIImageView` und ziehen Sie sie auf die View. Passen Sie die Größe an (mit dem Size Inspector – wählen Sie die Image View und tippen Sie dann $\mathbb{K}$ - $\mathbb{3}$), damit sie bequem auf den Bildschirm passt, zum Beispiel 300 x 300 mit dem Ursprung 10,10.

Unsere Image View ist nun fertig, und wir können ein Bild darin unterbringen, das dann ausgegeben wird. Öffnen Sie den Attributes Inspector mit $\mathbb{K}$ - $\mathbb{1}$ und wählen Sie im obersten Feld das Bild aus, das Sie dem Projekt hinzugefügt haben. Unser Inspector sieht danach wie in Abbildung 20.1 gezeigt aus.

Wenn Sie mit Ihrem Projekt so weit sind, beschäftigen Sie sich mal ein wenig mit dem Wert des Pull-down-Felds **MODE**. Dort finden Sie verschiedene Möglichkeiten dazu, wie die `UIImageView` das Bild anzeigt. Sie müssen sich entscheiden, wie das Bild erscheinen soll. Unser Bild mit der Option „Aspect Fit“ sehen Sie in Abbildung 20.2.

Das war einfach – das Bild wird angezeigt, und wir mussten keinerlei Code schreiben. Einfachheit ist gerade der Vorteil der Image View. So bekommen wir Bilder schnell auf unser UI. Jedoch haben wir dabei lediglich eine Einflussmöglichkeit – die unterschiedlichen Modi. Wenn einer dabei ist, der Ihre Anforderungen erfüllt, ist das für Sie gut. Wenn nicht, sollten Sie sich mit UIImage befassen, und genau das wollen wir als Nächstes tun.

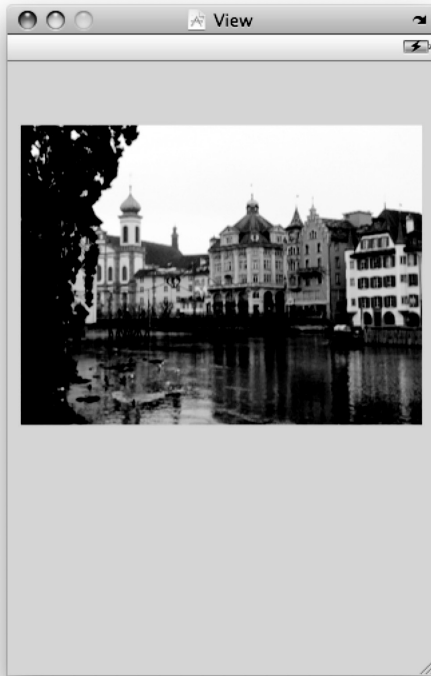


Abbildung 20.2: Bildausgabe mit Aspect Fit

20.2 Die Bildausgabe anpassen

Das UIImage ermöglicht es Ihnen, festzulegen, wie und wo ein Bild ausgegeben wird. Wir erhalten damit die Möglichkeit, die Darstellung des Bilds auf dem Bildschirm drastisch zu ändern.

Lassen Sie uns ein Beispiel erstellen, das die Vorteile dieser Flexibilität nutzt. Erzeugen Sie ein neues View-basiertes Projekt namens Image, fügen Sie eine neue Subklasse von UIView mit dem Namen ImageView hinzu und ändern Sie dann die Klasse der View in ImageViewController-

ter.xib nach UIImageView. Fügen Sie zudem das gleiche Bild hinzu, das Sie im letzten Projekt genutzt haben (wenn Sie kein neues Projekt erzeugen wollen, können Sie immer zum letzten eigenen View-Projekt wechseln und dieses bearbeiten). Nachdem wir nun alles eingerichtet haben, wollen wir die Methode `drawRect:` implementieren, um unser Bild auszugeben.

Images/Image/Classes/ImageView.m

```

1 - (void)drawRect:(CGRect)rect {
2     UIImage *image = [UIImage imageNamed:@"photo.png" ];
3     CGFloat idealSize = 300.0f;
4     CGFloat ratio = 1.0f;
5     CGFloat heightRatio = idealSize / image.size.height;
6     CGFloat widthRatio = idealSize / image.size.width;
7     if(heightRatio < widthRatio) {
8         ratio = heightRatio;
9     } else {
10        ratio = widthRatio;
11    }
12    CGRect imageRect = CGRectMake(10.0f, 10.0f, image.size.width ratio,
13                                   image.size.height ratio);
14    [image drawInRect:imageRect blendMode:kCGBlendModeDifference alpha:1.0f];
15 }

```

In diesem Code implementieren wir im Prinzip den Modus „Aspect Fit“ (einen der Modi, die bei der UIImageView zur Verfügung stehen), aber mit einer kleinen Änderung. Wir zeichnen das Bild mit einem anderen Einblendmodus, nur um einen kleinen Teil der Flexibilität zu zeigen, die wir durch die Verwendung der Klasse `UIImage` erhalten. Lassen Sie uns den Code Schritt für Schritt anschauen. In Zeile 2 sehen wir das Laden des Bilds mit der Methode `imageNamed:`. Diese Methode schaut in den Ressourcen des Projekts nach einem Bild mit dem angegebenen Namen. Ist das Bild geladen, können wir auf seine Größe zugreifen und dann das passende Verhältnis berechnen, um das Seitenverhältnis beizubehalten (in Zeile 7).

Nachdem wir nun das Rechteck kennen, das das Seitenverhältnis widerspiegelt, können wir das Bild zeichnen. Das tun wir mit der Methode `drawInRect:blendMode:alpha:`, wobei wir das Rechteck in Zeile 14 übergeben. Blendmodus und Alpha-Parameter erlauben uns deutlich mehr Kontrolle bei der Anzeige des Bilds als bei der Verwendung der UIImageView. In diesem Fall setzen wir nur den Alpha-Wert auf 1.0. Der Parameter für den Blendmodus ermöglicht uns, zu beeinflussen, wie die Pixel des Bilds mit den Hintergrundpixeln verschmolzen werden. Es gibt viele verschiedene Optionen, und Sie sollten sie ausprobieren, um ein Gefühl dafür zu bekommen, wie Sie die Bilder

verbinden können. In Abbildung 20.3 ist das Bild im Differenzmodus mit einem Alpha-Wert von 1 ausgegeben worden. Wie Sie sehen können, erhalten wir ein paar ziemlich coole Anzeigeeffekte bei der Verwendung der Zeichenmethoden von `UIImage`.



Abbildung 20.3: Bild mit Differenz-Blendmodus

Bilder können auch mit `CGContextDrawImage()` ausgegeben werden, aber die Details des zugrunde liegenden Grafikkontexts zum Zeichnen von Bildern sind ein bisschen komplizierter. Das Standardkoordinatensystem würde die Bilder auf dem Kopf zeichnen, also müssten wir vor dem Zeichnen noch ein wenig Koordinatentransformation betreiben. Es gibt ein Beispielprogramm (namens `CGImage`) beim Code für dieses Buch, das das Bild korrekt zeichnet, wenn Sie an den Details interessiert sind.

20.3 Image Picker

Nachdem Sie nun gesehen haben, wie Sie ein Bild ausgeben, wollen wir uns damit befassen, wie man ein Bild aus der Fotobibliothek des Benutzers holen kann. Das lässt sich nicht nur einfach umsetzen, Ihre Anwendung ist damit auch deutlich „persönlicher“.

Sie interagieren entweder mit der Kamera oder der Fotobibliothek auf dem iPhone über eine Instanz des UIImagePickerControllerController. Der Image Picker ist eine Subklasse des View-Controller. Damit tut er alles, was Sie auch von einem View-Controller erwarten würden. Er kümmert sich um einen ganzen Bildschirm mit Informationen, er hat eine View, die ein- oder ausgeblendet werden kann, und er dient als Controller zwischen den View-Elementen und dem zugrunde liegenden Modell. Der Image Picker ist die einzige Schnittstelle, die wir haben, um auf die Fotos in der Fotobibliothek zugreifen zu können. Glücklicherweise ist das Verwenden des Image Picker so einfach wie das Einrichten eines Delegate und das Implementieren einer Handvoll Methoden. Lassen Sie uns anschauen, was dafür zu tun ist.

Wir haben uns in Kapitel 3, *Grundlagen der iPhone-Entwicklung*, auf Seite 25 das erste Mal mit Delegates befasst, als es um den Anwendungs-Delegate ging. Wie Sie sich erinnern werden, geht es beim Delegieren darum, das Verhalten eines Objekts zu beeinflussen, ohne eine Subklasse davon bilden zu müssen. In diesem Fall trifft der Delegate des Image Picker die Entscheidung, was mit dem ausgewählten Bild passieren soll. Wir implementieren zwei Methoden, um mit dem Image Picker zu interagieren. Die erste übergibt das gewählte Bild, die zweite informiert den Delegate über einen Abbruch des Benutzers.

Lassen Sie uns ein neues Projekt erstellen, das ein Bild aus der Fotobibliothek lädt und es in einer UIImageView anzeigt.

Erstellen Sie ein neues View-basiertes Projekt namens PhotoView. Wir benötigen zwei Outlets im PhotoViewViewController: eines für den Image Picker und eines für die Image View. Passen Sie PhotoViewViewController.h so an, dass die Datei wie folgt aussieht:

Images/PhotoView/Classes/PhotoViewViewController.h

```
@interface PhotoViewViewController : UIViewController
<UIImagePickerControllerDelegate> {
    UIImageView *imageView;
    UIImagePickerController *imagePicker;
}

@property(n nonatomic, retain) IBOutlet UIImageView *imageView;
@property(n nonatomic, retain) IBOutlet UIImagePickerController *imagePicker;

@end
```

Beachten Sie, dass wir diesen View-Controller schon konform zum UIImagePickerControllerDelegate-Protokoll gemacht haben. Wir werden die Methoden dieses Protokolls gleich noch kennenlernen. Jetzt

müssen Sie nur sicherstellen, dass der Header unseren Wunsch widerspiegelt, dass der View-Controller zum Protokoll passt. Als Nächstes wollen wir in den Interface Builder wechseln, um unseren Image Picker zu konfigurieren und einzurichten. Öffnen Sie `PhotoViewViewController.xib` durch einen Doppelklick auf die Datei in Xcode. Öffnen Sie die Library mit `(⌘)-[L]` und suchen Sie nach dem Image Picker Controller. Ziehen Sie eine Instanz des Image Picker auf das Document-Fenster. Ihr Document sollte nun wie in Abbildung 20.4 aussehen.



Abbildung 20.4: Image Picker Controller

Mit dem Image Picker in unserer .nib-Datei müssen wir nun unseren View-Controller damit verbinden. Klicken Sie bei gedrückter `[Ctrl]`-Taste auf den `PhotoViewController` (also den `File's Owner`) und ziehen Sie zum neuen Image Picker. Wenn Sie die Maustaste loslassen, wählen Sie `imagePicker` aus, um die Verbindung zum Image Picker herzustellen. Wir müssen auch noch den Delegate des `UIImagePickerController` mit dem View-Controller verbinden, klicken Sie also bei gedrückter `[Ctrl]`-Taste auf den Image Picker und ziehen Sie zum View-Controller. Lassen Sie die Maustaste los und wählen Sie `delegate`. Sehen Sie zwei Zeilen im Pop-up für `delegate`, wählen Sie einfach die erste.

Als nächstes müssen wir eine Instanz der `UIImageView` hinzufügen und die Verbindung vom `PhotoViewController` zu ihr herstellen. Wechseln Sie zur Library (`(⌘)-[B]-[L]`), markieren die `UIImageView` und ziehen Sie sie in Ihre View. Setzen Sie im Size Inspector (`(⌘)-[3]`) die Breite und Höhe auf 300 und die x- und y-Koordinaten auf 10. Nun ziehen Sie aus dem `PhotoViewController` auf die neue Image View und verbinden das Outlet `imageView` damit.

Nachdem wir die Konfiguration im Interface Builder erledigt haben, müssen wir zu Xcode zurückkehren, um das Protokoll UIImagePickerControllerDelegate auch tatsächlich zu implementieren.

Hier der Code, mit dem der Image Picker angezeigt wird:

Images/PhotoView/Classes/PhotoViewViewController.m

```

1  - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
2      if([[touches anyObject] tapCount] > 1) {
3          // Image Picker anzeigen
4          if([UIImagePickerController isSourceTypeAvailable:
5              UIImagePickerControllerSourceTypeCamera]) {
6              self.imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;
7          } else {
8              self.imagePicker.sourceType =
9                  UIImagePickerControllerSourceTypePhotoLibrary;
10         }
11         self.imagePicker.allowsImageEditing = YES;
12         [self presentViewController:self.imagePicker animated:YES];
13     }
14 }

```

Diese Methode zeigt den Image Picker modal an, wenn der Benutzer mehr als einmal getippt hat (also ein Doppeltipp). Sie erinnern sich: Da sich der View-Controller in der Responder-Kette befindet, erhält er die Möglichkeit, auf Events zu reagieren. Wenn die Methode `touchesEnded:withEvent:` die View erreicht, wird sie an den View-Controller weitergeleitet, und unser Code reagiert darauf, indem er den Image Picker anzeigt.

Es gibt ein paar Dinge, auf die es sich in diesem Code hinzuweisen lohnt. So prüfen wir in Zeile 5 den Kamerateyp, bevor wir den Quelltyp setzen. Es ist wichtig, zu kontrollieren, ob die Kamera zur Verfügung steht (da manche Geräte mit iPhone OS keine Kamera haben), bevor wir versuchen, sie zu nutzen. Als Nächstes erlauben wir in Zeile 11 das Editieren. Wenn diese Option auf YES gesetzt wird, darf der Anwender den Ausschnitt eines Bilds verschieben und hineinzoomen, bevor er den Button AUSWÄHLEN antippt. Ist die Option auf NO gesetzt, kann der Benutzer ein Bild nur auswählen, aber nicht zoomen oder den Ausschnitt verschieben. In Zeile 12 bringt der View-Controller schließlich den Image Picker als modale View auf den Bildschirm.

Neben den beiden Quelltypen, die hier gezeigt werden, können Sie auch `UIImagePickerControllerSourceTypeSavedPhotosAlbum` nutzen, damit der Anwender lediglich aus den gespeicherten Fotos aus der Bibliothek auswählen kann.

In Version 2.2 des iPhone OS SDK gibt es allerdings einen Fehler. Der Simulator gibt an, eine Kamera zu besitzen (wenn Sie eine iSight-Kamera an Ihrem Computer haben), aber kann dann nicht richtig darauf zugreifen, so dass kein neues Foto gemacht werden kann. Um diesen Fehler zu umgehen, können Sie folgenden Code nutzen, der den Quelltyp immer auf UIImagePickerControllerSourceTypePhotoLibrary setzt, wenn die Anwendung im Simulator läuft.

Images/PhotoView/Classes/PhotoViewViewController.m

```
if([[UIDevice currentDevice] name]
    rangeOfString:@"Simulator" ].location != NSNotFound) {
    self.imagePicker.sourceType =
        UIImagePickerControllerSourceTypePhotoLibrary;
}
```

So, nachdem der Image Picker nun angezeigt wird, müssen wir darauf reagieren, wenn der Anwender ein Bild auswählt oder wenn er abbricht. Wie Sie sich vielleicht erinnern, erledigen wir das über die Delegate-Methoden. Dies ist der Code für den Umgang mit einem Abbruch:

Images/PhotoView/Classes/PhotoViewViewController.m

```
- (void)imagePickerControllerDidCancel: (UIImagePickerController *)picker {
    [self.imagePicker dismissModalViewControllerAnimated:YES];
}
```

Wir verwerfen einfach den Image Picker. Als Nächstes schauen wir uns an, was wir mit dem Bild machen, nachdem der Anwender es ausgewählt hat. Hier ist der Code:

Images/PhotoView/Classes/PhotoViewViewController.m

```
1 - (void)imagePickerController:(UIImagePickerController *)
2     picker didFinishPickingMediaWithInfo:(NSDictionary *)info {
3     imageView.image = [info objectForKey:UIImagePickerControllerEditedImage];
4     [self dismissModalViewControllerAnimated:YES];
5 }
```

Wir nutzen hier den Dictionary-Parameter info, um das bearbeitete Foto zu erhalten. Da wir ein Bearbeiten zulassen, kann der Anwender das Bild nach seinen Wünschen beschneiden und skalieren. Wenn er dann auf FERTIG tippt, werden das ursprüngliche Bild, das bearbeitete Bild und der „Begrenzungsrahmen“ zusammen in das Info-Dictionary geschoben. Wir erhalten das bearbeitete Bild mit dem Schlüssel UIImagePickerControllerEditedImage, das ursprüngliche Bild findet sich unter UIImagePickerControllerOriginalImage und der Begrenzungsrahmen über den Schlüssel UIImagePickerControllerCropRect. Die letzten beiden Objekte benötigen wir in unserem einfachen Beispiel nicht, aber wenn Ihre Anwendung sie nutzen kann, sind sie verfügbar.

20.4 Videos aufnehmen

Neben der Möglichkeit, Fotos aus der Bibliothek des Anwenders oder von der Kamera zu übernehmen, kann der Benutzer mit manchen Geräten mit iPhone OS 3.0 und neuer auch Videos aufnehmen. Wir können diese Videos über den gleichen grundlegenden Ablauf in unsere Anwendung übernehmen. Erstellen Sie einen Image Picker, setzen Sie den Quelltyp, geben Sie den Picker aus und übernehmen Sie das Video dann über den Delegate-Callback.

Um zu sehen, ob das Gerät, auf dem Ihre App läuft, Videos aufnehmen kann, fragen Sie den Image Picker nach den Quelltypen über die Methode `availableMediaTypesForSourceType:`. Das zurückgegebene Array enthält die Konstante `kUTTypeMovie`, wenn das Gerät das Aufnehmen von Videos erlaubt.

Unterstützt das Gerät Ihres Anwenders Videos, können Sie auf dieses Video in Ihrer Delegate-Methode zugreifen, indem Sie die Medien-URL aus dem info-Dictionary mit dem Schlüssel `UIImagePickerController-MediaURL` auslesen. Mit dieser URL können Sie alles anstellen, was für Ihre Anwendung mit diesem Video sinnvoll erscheint. In Kapitel 14, *Videos abspielen*, auf Seite 319 finden Sie mehr Informationen über die Verwendung von Videos in Ihrer App.

In diesem Kapitel haben wir gesehen, wie man Bilder anzeigt, die wir als Teil des Anwendungs-Bundles ausliefern. Sie haben auch gelernt, wie man Fotos, Videos und Bilder von der Kamera mit einer iPhone-App verwendet. Die Integration ist einfach – wir müssen nur das Protokoll `UIImagePickerControllerDelegate` implementieren.

Kapitel 21

Core Animation

Jede Animation auf dem iPhone – jede umgeklappte View, jedes Einblenden, Scrollen und so weiter – geschieht über Core Animation. Dass wir noch nicht viel über Core Animation (CA) gesagt haben und dennoch schon so viel erreichen konnten, ist eine seiner tollen Eigenschaften. Es liegt jeglichem UI-Handling auf dem iPhone zugrunde, wird aber fast vollständig durch abstraktere und häufig einfachere APIs abgedeckt. Nicht, dass Core Animation komplex wäre – es gibt einfach nur mehr Knöpfe und Regler, an denen wir spielen können, und damit auch mehr, das wir verstehen und berücksichtigen müssen.

In diesem Kapitel geben wir einen Überblick über CA und gehen dann auf einige speziellere Dinge ein, die auf dem iPhone nur durch Verwendung von Core Animation erreicht werden können.

21.1 Einführung in Core Animation

Layer sind die Grundlage, die alles andere in Core Animation ermöglichen. Stellen Sie sich einen Layer als rechteckige 2-D-Oberfläche vor, die in einer dreidimensionalen Welt „lebt“. Sie können Inhalte wie zum Beispiel Bilder oder PDFs auf dieser Oberfläche platzieren und diese dann bequem verschieben. Für die Performance ist wichtig, dass der Inhalt für diese Bewegung nicht neu gezeichnet werden muss.

Tatsächlich ist das Verschieben eines Layers nur der Anfang. Sie können alles in einem Layer verkleinern. Sie können einen Layer im dreidimensionalen Raum verschieben (so funktioniert die Cover Flow-Ansicht in der iPod-Anwendung). Sie können den Inhalt eines Layers

ein- oder ausblenden. Mit anderen Worten – so gut wie jede Eigenschaft des Layers kann animiert werden. Geometrische Eigenschaften wie *position* und *bounds* sind ebenso beeinflussbar wie Stileigenschaften, beispielsweise wie *opacity* oder *contents*. Alles kann sich verändern.

Es gibt zwei grundlegende Typen von Animationen. Bei *impliziten Animationen* geben Sie eine neue Position für einen Layer vor, wodurch er sich in einer netten, flüssigen Animation dorthin „bewegt“. Wir müssen uns nicht um Timer kümmern, auf Threads achten oder irgendetwas neu zeichnen lassen. Es funktioniert einfach. Wenn Sie eine Eigenschaft setzen oder ändern, die animiert werden kann, lösen Sie eine implizite Animation aus.

Wenn Sie den Effekt, nach dem Sie suchen, nicht durch implizite Animationen erhalten, können Sie die implizit genutzten Animationen einfach durch Ihre eigenen *expliziten* Animationen ersetzen. Damit können wir alle Aspekte der Animation beeinflussen – vom Timing bis hin zu den Schlüsselpunkten entlang der Animation.

Schließlich können Layer auch noch im dreidimensionalen Raum bewegt werden. Core Animation bietet einige wirklich coole (und recht einfach zu nutzende) Features, durch die wir unsere Inhalte im Raum verschieben können. Bevor wir uns damit befassen, wollen wir CA aber noch besser kennenlernen, indem wir uns anschauen, was wir davon bisher schon verwendet haben.

Anstatt Sie in diesem Kapitel nun wieder neue Projekte anlegen zu lassen, werden wir nur die Details behandeln, die die Animationen ermöglichen. Wenn Sie das selbst ausprobieren wollen, können Sie für jeden Abschnitt in diesem Kapitel ein neues Projekt erzeugen. Möchten Sie sich lieber nur den Code anschauen, statt immer von vorne beginnen zu müssen, laden Sie sich den Beispielcode zum Buch herunter.

21.2 UIView animieren

Wir haben schon eine Menge von Core Animation im Zusammenhang mit *UIView* gesehen, aber bisher die Details ignorieren können, weil Cocoa Touch alles abgedeckt hat. Der sichtbare Teil aller *UIView*s ist in Wirklichkeit ein Layer. Die View muss im Prinzip nur die Events verarbeiten und in die Responder-Kette passen. All das Zeichnen, Animieren und die anderen visuellen Features einer View werden direkt oder indirekt durch den Layer behandelt.

Animationsblöcke

Jede Eigenschaft, die man auf einer UIView in einem Animationsblock ändert, wird mit der für diese Eigenschaft definierten impliziten (oder expliziten, siehe weiter unten) Animation gerendert. Das Erstellen von Animationsblöcken geht fix. Die Klassenmethode `+beginAnimations:context:` beginnt einen Block. Wenn Sie dann die Animation starten lassen wollen, rufen Sie `+commitAnimations` auf. Alle Änderungen, die gemacht werden, während der Block aktiv ist, werden dann animiert, statt direkt „gewechselt“ zu werden. Lassen Sie uns mit einem wirklich einfachen Beispiel beginnen. Eine UIImageView mit einem Basketball am oberen Rand ist in Abbildung 21.1 zu sehen. Wir werden das Bild vom oberen Rand des Bildschirms an den unteren und wieder zurück verschieben.



Abbildung 21.1: Image-View mit Basketball

Der Animationsblock beginnt in Zeile 2 mit dem Aufruf von `+beginAnimations:context:`. Beide Argumente dieser Methoden sind mehr oder weniger beliebig und sollen nur dem Delegate helfen, das wir einrichten (mehr dazu gleich).

Dies ist der Code:

CoreAnimation/SimpleMovement/Classes/SimpleMovementViewController.m

```

1 - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
2     [UIView beginAnimations:@"movement" context:nil];
3     CGPoint center = movingBall.center;

```

```

4     if(center.y > 85.0f) {
5         center.y -= 295.0f;
6         movingBall.center = center;
7     } else {
8         center.y += 295.0f;
9         movingBall.center = center;
10    }
11    [UIView commitAnimations];
12 }

```

Wenn sich der Ball am oberen Rand befindet, setzen Sie ihn an den unteren und umgekehrt. Beachten Sie, dass wir die Eigenschaft `center` und nicht den Ursprung des frame setzen. Man kann meist einfacher mit der Eigenschaft `center` arbeiten als mit dem Rahmen, daher nutzen Sie sie möglichst häufig.

Die Zahlen in diesem Beispiel wurden gewählt, weil sie gut aussahen und nicht, weil sie eine spezielle Bedeutung haben. Lassen Sie die Anwendung einmal laufen. Achten Sie darauf, dass sich der Ball vom oberen zum unteren Rand bewegt (die implizite Standardanimation dauert 0,25 Sekunden). Mit einer expliziten Animation könnten wir die Dauer verlängern (wir werden uns eigene Animationen in Abschnitt 21.2, *UIView animieren*, auf Seite 440 anschauen).

Wir können auch den Wechsel zwischen zwei Subviews animieren – mithilfe der Methode `setAnimationTransition:forView:cache:.` Erzeugen wir so eine Transitions-Animation, werden alle Änderungen an der View-Hierarchie nach unseren Vorstellungen animiert. Dies ist der Code für unser Beispiel:

CoreAnimation/SimpleMovement/Classes/SimpleMovementViewController.m

```

1 - (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
2     if(event.allTouches.count > 1) {
3         [UIView beginAnimations:@"switch" context:nil];
4         [UIView setAnimationTransition:UIViewAnimationTransitionCurlUp
5                forView:self.view cache:YES];
6         if(nil == basketBall.superview) {
7             basketBall.center = tennisBall.center;
8             [tennisBall removeFromSuperview];
9             [self.view addSubview:basketBall];
10            movingBall = basketBall;
11        } else {
12            tennisBall.center = basketBall.center;
13            [basketBall removeFromSuperview];
14            [self.view addSubview:tennisBall];
15            movingBall = tennisBall;
16        }
17
18        [UIView commitAnimations];
19    }
20 }

```

In diesem Beispiel starten wir die Transition in Zeile 5, um die View beim Verwerfen „aufzurollen“. Wir können aus einer ganzen Reihe von eingebauten Transition-Animationen wählen. Hier haben wir die „Aufroll“-Animation gewählt, weil sie cool aussieht. Probieren Sie ein paar der anderen Animationen aus, um herauszufinden, welche Ihnen am besten gefällt. Stellen Sie auch sicher, sie auf dem Gerät selber auszutesten, da Transitions auf dem iPhone oft besser aussehen als im Simulator.

Animationen anpassen

Bisher haben wir nur mit Animationen in ihrem Standardzustand experimentiert. Es gibt aber eine Reihe von Optionen, um die impliziten Animationen anzupassen. Ändert man zum Beispiel den Zeitverlauf einer Animation auf einen langsamen Start oder ein langsames Ende, kann der Benutzer damit unauffällig darauf hingewiesen werden, was die Anwendung gerade tut. Diese kleinen Hinweise machen häufig den Unterschied zwischen einem guten UI und einem großartigen UI aus. Lassen Sie unsere Animation mit dem Ball so ändern, dass sie einen anderen Zeitverlauf nutzt. Dies ist der Code:

CoreAnimation/TimingCurve/Classes/TimingCurveViewController.m

```

1 - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
2     [UIView beginAnimations:@"movement" context:nil];
3     [UIView setAnimationCurve:UIViewAnimationCurveEaseIn];
4     [UIView setAnimationDuration:1.0f];
5     [UIView setAnimationRepeatCount:3];
6     [UIView setAnimationRepeatAutoreverses:YES];
7     CGPoint center = basketBall.center;
8     if(center.y > 85.0f) {
9         center.y -= 295.0f;
10        basketBall.center = center;
11    } else {
12        center.y += 295.0f;
13        basketBall.center = center;
14    }
15    [UIView commitAnimations];
16 }
```

Dieser Code sieht vertraut aus, abgesehen von dem zusätzlichen Code zum Anpassen der Animation ab Zeile 3. Lässt man die Animation per „Ease In“ ablaufen, beginnt sie langsam und beschleunigt dann. Durch ein „Ease In“ sieht eine Animation oft realistischer aus, da wir solche Bewegungsmuster auch in der Realität gewohnt sind.

Wir haben zudem die Dauer in der nächsten Zeile auf 1.0 Sekunden gesetzt, was – wie Sie vermutlich erwarten – dazu führt, dass die Animation auch eine Sekunde läuft (statt der normalen 0,25 Sekunden).

Wir nutzen die Zeitangabe oft beim Debuggen. Setzen wir sie auf eine oder zwei Sekunden, sehen wir damit Dinge, die wir in einer Viertelsekunde übersehen hätten.

Als Nächstes setzen wir den Wiederholzähler und Autoreverse. Damit wird der Basketball dreimal hoch- und runterhüpfen (zusammen mit der umgekehrten Animation). Führen Sie die App aus und beobachten Sie den Unterschied durch das Ease In. Das Hüpfen sieht so deutlich natürlicher aus.

Beim Ablauf der Animation ist Ihnen vielleicht aufgefallen, dass der Ball nach dem letzten Wiederholzyklus an das untere Ende des Bildschirms springt. Das liegt an der Natur des Wiederholzyklus. Jede Animation (wir wiederholen drei Mal) beginnt am Anfangspunkt (hier: oben), läuft zum Endpunkt (hier: unten) und wieder zurück (nach oben). Da wir das Zentrum des Ball-Bilds an den unteren Rand verlegt haben und die Animation am oberen Rand des Bildschirms endet, erhalten wir den Sprung.

Beheben lässt sich das durch die Verwendung von Schlüsselbildanimationen (Keyframe Animations) und die Übernahme der Kontrolle über die gesamte Animation. Durch Angabe von sechs Schlüsselbildern – eines für jeden Wechsellpunkt oben und unten – würde der Ball dreimal springen und am unteren Rand nach dem dritten Sprung enden. Wir würden vermutlich auch jeden „oberen“ Punkt etwas niedriger ansetzen als den vorigen. Es gibt in Abschnitt 21.3, *Layer*, auf Seite 447 ein Beispiel für die Schlüsselbildanimation, aber eine detaillierte Behandlung geht über den Rahmen dieses Buchs hinaus. Mehr Informationen finden Sie in *Core Animation for Mac OS X and the iPhone* [Dud08].

Animation-Delegates

Sie können eines Ihrer Objekte zum Delegate der Animation machen. Beginnt oder endet die Animation, wird Ihr Delegate benachrichtigt. Das ist insbesondere dann nützlich, wenn man zum Beispiel am Anfang oder Ende der Animation einen Klang abspielen will. Wir beschreiben die Audioausgabe ab Kapitel 15, *Zugriff auf die iPod-Bibliothek*, auf Seite 329, daher werden wir in unserem Beispiel nur einen Log-Eintrag vornehmen. Wenn Sie sich bereits ausreichend mit Audioprogrammierung befasst haben, können Sie diese Log-Einträge durch eigene Klänge ersetzen.

Sie können den Delegate für Ihre Animationen mit der Methode `setAnimationDelegate:` setzen. Sie müssen auch die Start- und Stoppselektoren angeben, so dass Core Animation weiß, welche Methoden aufzurufen sind. Dabei geben Sie den Startselektor mit `setAnimationWillStartSelector:` und den Stoppselektor mit `setAnimationDidStopSelector:` an.

Lassen Sie uns einen Animations-Delegate in unserem Beispiel mit dem hüpfenden Basketball setzen. Dies ist der aktualisierte Code für die Methode `touchesEnded:withEvent::`

CoreAnimation/AnimationDelegate/Classes/AnimationDelegateViewController.m

```

1 - (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
2     [UIView beginAnimations:@"movement" context:nil];
3     [UIView setAnimationDelegate:self];
4     [UIView setAnimationWillStartSelector:@selector(didStart:context:)];
5     [UIView setAnimationDidStopSelector:@selector(didStop:finished:context:)];
6     [UIView setAnimationCurve:UIViewAnimationCurveEaseIn];
7     [UIView setAnimationDuration:1.0f];
8     [UIView setAnimationRepeatCount:3];
9     [UIView setAnimationRepeatAutoreverses:YES];
10    CGPoint center = basketBall.center;
11    if(center.y > 85.0f) {
12        center.y -= 295.0f;
13        basketBall.center = center;
14    } else {
15        center.y += 295.0f;
16        basketBall.center = center;
17    }
18    [UIView commitAnimations];
19 }

```

In Zeile 3 setzen wir den Delegate für die folgende Animation. In den nächsten beiden Zeilen definieren wir die Selektoren, die aufgerufen werden sollen, wenn die Animation beginnt oder endet. Diese Methoden können beliebige Namen haben, müssen aber die korrekte Signatur besitzen. Der Startselektor sollte zwei Parameter haben: Der erste ist ein `NSString` mit dem Namen, den die Animation in `beginAnimations:context:` von uns erhalten hat. Der zweite ist ein `void`-Zeiger mit dem gleichen Zeiger, der auch an `beginAnimations:context:` übergeben wurde. Der Stoppselektor sollte drei Parameter haben: einen `NSString`, einen `BOOL` und einen `void`-Zeiger. Der erste entspricht wieder dem Namen und der dritte dem `void`-Zeiger, der der Methode `beginAnimations:context:` übergeben wurde. Der zweite Parameter ist ein Schalter, der angibt, ob die Animation fertig ist. Durch das Implementieren dieser Delegate-Methoden haben wir noch mehr Kontrolle über den Ablauf der Animation und das Verhalten am Anfang und am Ende.

Um die vollständige Kontrolle über die Animationen zu erhalten, müssen wir noch ein bisschen tiefer in Core Animation einsteigen. Lassen Sie uns dazu zunächst die Beziehung zwischen dem Layer und der View betrachten.

UIView und CALayer

Layer in Core Animation stellen Delegationsmethoden bereit, mit denen wir das Zeichnen und Animieren beeinflussen können, ohne eine Subklasse vom Layer bilden zu müssen. Mehr Informationen zu Layern und Core Animation im Allgemeinen erhalten Sie in *Core Animation for Mac OS X and the iPhone* [Dud08]. UIView nutzt die Delegation, da die View immer der Delegate ihres Layers ist. Damit können wir Layer-Delegate-Methoden zu unserer View hinzufügen und mehr Kontrolle über den Verlauf einer Animation erhalten. Lassen Sie uns dazu die Standardanimation durch eine eigene ersetzen, um zu sehen, wie das funktioniert. Dies ist der Code:

CoreAnimation/LayerDelegate/Classes/LayerDelegateView.m

```
-(id <CAAction>)actionForLayer:(CALayer *)layer forKey:(NSString *)key {
    CAAnimation *animation = (CAAnimation *)[super actionForLayer:layer
                                                                    forKey:key];
    if([animation respondsToSelector:@selector(setDuration:)]) {
        animation.duration = 2.0f;
    }
    return animation;
}
```

Dieser Code setzt die Dauer für alle Animationen auf 2.0 Sekunden. Wir haben zwar schon andere Möglichkeiten kennengelernt, die Dauer einer Animation festzulegen, aber die Methoden der UIView funktionieren nur in einem Animationsblock. Dieser Code setzt die Dauer jeder Animation auf der View – egal ob wir einen Block konfiguriert haben.¹ Hier können wir auch die Standardanimation durch eine eigene ersetzen. Eine weitergehende Behandlung dieses Bereichs geht über den Rahmen dieses Buchs hinaus. Wenn Sie sich für die Details interessieren, werfen Sie einmal einen Blick in *Core Animation for Mac OS X and the iPhone* [Dud08].

Sie können auch über die Delegationsmethoden (`drawLayer:inContext:` und `displayLayer:`) im Layer zeichnen, aber das ist selten nötig, weil die Methode `drawRect:` der UIView eigentlich so gut wie immer der perfekte Ort für Ihre Zeichnungen ist.

¹ Während der Event-Loop erzeugt das iPhone OS einen Block für uns.

21.3 Layer

Die enge Integration mit dem UIKit ermöglicht es Ihnen häufig, das Gewünschte auch über den etwas einfacheren Weg des UIKit zu erhalten. Aber für 3-D-Transformationen, das Anpassen des Timings einer Animation und andere Veränderungen erhalten Sie eine feinere Kontrolle durch die Layer-basierten APIs.

Wie so oft gewinnt man in einem Bereich, indem man in einem anderen verliert. Layer sind keine Responder, daher reagieren sie nicht auf Events. Selbst wenn Sie die Event-Methoden implementieren (`touchesBegan:withEvent:` und so weiter), würden diese nicht aufgerufen werden. So gut wie immer können Sie aber das Event-Handling durch das Verarbeiten der Events in der View erreichen, die die Layer enthält. Sie können die Methode `hitTest:` nutzen, um den Layer zu finden, den der Anwender angetippt hat, und dann mit dieser Information weiterarbeiten. Denken Sie auf jeden Fall beim Erstellen eines Layer-basierten UI daran, dass Layer keine Responder sind!

Da Layer eigentlich 2-D-Objekte sind, die in einem dreidimensionalen Raum existieren, können wir sie in diesem gesamten Raum bewegen. Dazu nutzen wir die Eigenschaften `zPosition` und `transform`. Lassen Sie uns ein Beispiel für die Eigenschaft `transform` anschauen. Dies ist der Code:

CoreAnimation/ThreeDTransform/Classes/ThreeDTransformViewController.m

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    CAKeyframeAnimation *animation = [CAKeyframeAnimation
        animationWithKeyPath:@"transform" ];
    NSValue *initial = [NSValue valueWithCATransform3D:
        CATransform3DMakeRotation(0.0, 1.0f, -1.0f, 0.0f)];
    NSValue *middle = [NSValue valueWithCATransform3D:
        CATransform3DMakeRotation(M_PI, 1.0f, -1.0f, 0.0f)];
    NSValue *final = [NSValue valueWithCATransform3D:
        CATransform3DMakeRotation(0.0, 1.0f, -1.0f, 0.0f)];
    animation.values = [NSArray arrayWithObjects:initial,
        middle, final, nil];
    animation.duration = 2.0f;
    [self.basketBall addAnimation:animation forKey:@"transform" ];
}
```

In dieser Methode erzeugen wir eine `CAKeyframeAnimation` und fügen drei Transformationen hinzu: `initial` ist der Ausgangspunkt, `middle` die Mitte der Animation und `final` wieder der Ausgangspunkt. Damit dreht sich der Layer im dreidimensionalen Raum um den Vektor (1.0, -1.0, 0.0) um 360 Grad. Den rotierenden Basketball nach etwa einem Drittel des Weges sehen Sie in Abbildung 21.2.



Abbildung 21.2: Image-View mit Basketball

UIViews werden in der Reihenfolge ihrer Position in der Eigenschaft `subviews` ihrer `superview` gezeichnet. `CALayer` sind anhand von zwei Eigenschaften angeordnet: Zuerst kommt die Reihenfolge der Layer im Array `sublayers` ihres `superlayer`, dann folgt der Wert der Eigenschaft `zPosition`. Wenn zwei Geschwister-Layer den gleichen Wert in `zPosition` haben, werden sie anhand der Reihenfolge im Array `sublayers` des übergeordneten Elements sortiert. Ansonsten werden die Layer anhand der Reihenfolge ihrer `zPosition` angezeigt. Lassen Sie uns ein einfaches Beispiel anschauen, das zwei Layer besitzt – einen roten und einen blauen. Wenn die Anwendung startet, wird die `zPosition` des blauen Layers auf 5.0 gesetzt, die des roten auf -5.0. Dies ist der Code für `viewDidLoad`:

CoreAnimation/Depth/Classes/DepthViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.blue = [CALayer layer];
    blue.backgroundColor = [[UIColor blueColor] CGColor];
    blue.bounds = CGRectMake(0.0f, 0.0f, 100.0f, 100.0f);
    blue.position = CGPointMake(160.0f - 20.0f, 240.0f - 20.0f);
    blue.zPosition = 5.0f;
    [self.view.layer addSublayer:blue];
    self.red = [CALayer layer];
    red.backgroundColor = [[UIColor redColor] CGColor];
    red.bounds = CGRectMake(0.0f, 0.0f, 100.0f, 100.0f);
    red.position = CGPointMake(160.0f + 20.0f, 240.0f + 20.0f);
}
```

```

    red.zPosition = -5.0f;
    [self.view.layer addSublayer:red];
}

```

Obwohl der blaue Layer der Liste sublayers zuerst hinzugefügt wurde (und daher auch als Erstes gezeichnet werden sollte), wird er als Letztes gezeichnet, da seine `zPosition` den Wert 5.0 hat, während der rote Layer dort eine -5.0 stehen hat. Um noch deutlicher zu machen, wie die `zPosition` das Zeichnen des Layers beeinflusst, schauen Sie sich diesen Code an:

CoreAnimation/Depth/Classes/DepthViewController.m

```

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{ CGFloat oldBlue = self.blue.zPosition;
  self.blue.zPosition = self.red.zPosition;
  self.red.zPosition = oldBlue;
}

```

Diese einfache Änderung der `zPosition` sorgt dafür, dass die Layer ihre Zeichenreihenfolge ändern. In komplexeren Situationen, in denen die Layer im dreidimensionalen Raum gedreht werden, wird der Effekt durch das Ändern der `zPosition` drastischer sein.

Es gibt noch vieles mehr, was man mit Animationen anstellen kann, um den gewünschten Effekt zu erhalten. Das Core Animation-Buch geht weiter ins Detail (*Core Animation for Mac OS X and the iPhone* [Dud08]).

21.4 OpenGL ES

OpenGL wird auf dem iPhone über einen `CAEAGLLayer`-Layer angesprochen. Bilden Sie eine Subklasse von `UIView` und geben Sie die Klasse `CAEAGLLayer` aus der Methode `+layerClass` zurück. Die Methode `layerClass` liefert standardmäßig `CALayer` zurück und wird aufgerufen, wenn man die View einrichtet. Auf dem iPhone sind alle Views mit einem Layer verbunden (der Layer wird manchmal als *Backing Store* bezeichnet), und die Zeichnungen sowie anderer visueller Inhalt sind dort gepuffert. Die Methode `layerClass` gibt uns die Möglichkeit, unseren eigenen Layer zu spezifizieren, statt den Standard-Layer nutzen zu müssen.

Wenn die Standardoptionen von OpenGL ES für Ihre Zeichnungen ausreichend sind, müssen Sie nichts weiter tun, um eine OpenGL-ES-Oberfläche zu erhalten, mit der Sie arbeiten können. Aber häufig passen diese Standards eben nicht. Um sie anzupassen, müssen Sie nur die Eigenschaft `drawableProperties` des Layers setzen. Eine vollständige

dige Behandlung dessen, was zur korrekten Konfiguration und Initialisierung einer OpenGL-ES-Umgebung notwendig ist, geht über den Rahmen dieses Buches hinaus. Schauen Sie sich dazu am besten die *OpenGL SuperBible* [RSWLH07] an, dort wird das alles wesentlich umfangreicher behandelt.

Lassen Sie uns das Beispiel näher betrachten. Wir werden das übliche rotierende Quadrat nutzen, das aus dem OpenGL-ES-Template von Xcode kommt, um zu erläutern, wie Sie Ihre OpenGL-Umgebung einrichten. Wir werden uns aber nicht mit den Details von OpenGL ES befassen, sondern uns stattdessen auf die Core Animation-Aspekte des Beispiels beschränken.

Schauen wir uns den Code an, der vom Template generiert wird. Beginnen wir mit dem Code zum Konfigurieren des Layers:

CoreAnimation/OpenGL/Classes/EAGLView.m

```

1  - (id)initWithCoder:(NSCoder*)coder {
2      if ((self = [super initWithCoder:coder])) {
3          CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;
4          eaglLayer.opaque = YES;
5          eaglLayer.drawableProperties =
6          [NSDictionary dictionaryWithObjectsAndKeys:
7          [NSNumber numberWithInt:NO],
8          kEAGLDrawablePropertyRetainedBacking, kEAGLColorFormatRGBA8,
9          kEAGLDrawablePropertyColorFormat, nil];
10         context = [[EAGLContext alloc]
11         initWithAPI:kEAGLRenderingAPIOpenGLES1];
12
13         if (!context || ![EAGLContext setCurrentContext:context]) {
14             [self release];
15             return nil;
16         }
17
18         self.animationInterval = 1.0 / 60.0;
19     }
20     return self;
21 }

```

Die Methode `initWithCoder:` wird aufgerufen, wenn diese View aus der `.nib`-Datei geladen wird.

Beachten Sie, dass die Option `opaque` des Layers in Zeile 4 gesetzt wird. Indem Sie Ihre OpenGL-ES-Layer opak machen und Core Animation darüber auch informieren, können Sie die Performance deutlich verbessern. Wenn es möglich ist, sollten Sie also alle Ihre Zeichnungen mit OpenGL ES in opaken Layern vornehmen.

Wir setzen zudem die `drawableProperties` in Zeile 8. Die Konfiguration legt hier die Verwendung des RGBA-8-Farbformats (über die Konstante `kEAGLColorFormatRGBA8`) und kein Retained Backing fest. Es gibt viele, viele Optionen für die Angabe des Farbformats. Angesichts der beschränkten Bandbreite, die auf dem iPhone zur Verfügung steht, ist es allerdings wichtig, die kleinsten Typen zu wählen (einschließlich der Farbformate), die für Ihre Anwendung ausreichen.

Wir haben den API-Level in Zeile 11 definiert. Dieser legt fest, welche Version der API wir nutzen möchten. Für das iPhone OS 2.0 muss der Wert `kEAGLRenderingAPIOpenGLES1` sein, aber die Liste kann in zukünftigen Versionen des iPhone OS noch wachsen. Als Nächstes setzen wir in Zeile 13 den Kontext, durch den die OpenGL-ES-Runtime erfährt, welchen Kontext sie nutzen soll, wenn wir OpenGL-ES-Befehle schicken.

In `layoutSubviews` setzen wir den aktuellen Kontext und erzeugen den Frame-Buffer neu (diese seltsamen OpenGL-Dinge, die Sie im Blue Book nachschauen müssen: *OpenGL SuperBible* [RSWLH07]). Und wenn alles klappt, kann die Anwendung nun ihren ersten Frame zeichnen. `layoutSubviews` wird automatisch für uns aufgerufen, da die View im Fenster liegt. Dies ist der Code:

CoreAnimation/OpenGL/Classes/EAGLView.m

```
-(void)layoutSubviews {
    [EAGLContext setCurrentContext:context];
    [self destroyFramebuffer];
    [self createFramebuffer];
    [self drawView]; }
```

Nachdem `layoutSubviews` ausgeführt wurde, ist die Anwendung mehr oder weniger mit den Aufgaben beim Start fertig. Wenn die App den Start-up-Teil abschließt, dürfen Sie nicht vergessen, dass dem Anwendungs-Delegate die Methode `applicationDidFinishLaunching:` geschickt wird. In dieser Methode sendet der Delegate der View die Methode `startAnimation`. Deren Code findet sich hier:

CoreAnimation/OpenGL/Classes/EAGLView.m

```
-(void)startAnimation {
    self.animationTimer =
    [NSTimer scheduledTimerWithTimeInterval:self.animationInterval
                                     target:self selector:@selector(drawView)
                                     userInfo:nil repeats:YES];
}
```

In dieser Methode erzeugen wir den Timer, der alle `animationInterval` Sekunden ausgelöst wird (in unserem Fall jede 1/60 Sekunde). Jedes Mal, wenn der Timer ausgelöst wird, ruft er die Methode `drawView` auf.

Dies ist ihr Code:

CoreAnimation/OpenGL/Classes/EAGLView.m

```

1  - (void)drawView {
2      const GLfloat squareVertices[] = {
3          -0.5f, -0.5f,
4          0.5f, -0.5f,
5          -0.5f, 0.5f,
6          0.5f, 0.5f,
7      };
8      const GLubyte squareColors[] = {
9          255, 255, 0, 255,
10         0, 255, 255, 255,
11         0, 0, 0, 0,
12         255, 0, 255, 255,
13     };
14
15     [EAGLContext setCurrentContext:context];
16
17     glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
18     glViewport(0, 0, backingWidth, backingHeight);
19
20     glMatrixMode(GL_PROJECTION);
21     glLoadIdentity();
22     glOrthof(-1.0f, 1.0f, -1.5f, 1.5f, -1.0f, 1.0f);
23     glMatrixMode(GL_MODELVIEW);
24     glRotatef(3.0f, 0.0f, 0.0f, 1.0f);
25
26     glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
27     glClear(GL_COLOR_BUFFER_BIT);
28
29     glVertexPointer(2, GL_FLOAT, 0, squareVertices);
30     glEnableClientState(GL_VERTEX_ARRAY);
31     glColorPointer(4, GL_UNSIGNED_BYTE, 0, squareColors);
32     glEnableClientState(GL_COLOR_ARRAY);
33     glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
34
35     glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
36     [context presentRenderbuffer:GL_RENDERBUFFER_OES];
37 }

```

Abgesehen von all dem OpenGL-ES-Code, geschehen in dieser Methode zwei Dinge. Zunächst stellt sie in Zeile 15 sicher, dass der aktuelle Kontext auch der richtige Kontext ist. Dann schickt sie den Render-Buffer in Zeile 36 an den Bildschirm. Der Rest des Codes in dieser Methode ist grundlegender OpenGL-Code, über den Sie mehr im Blue Book erfahren.

Core Animation kann für Ihre Anwendung eine mächtige Erweiterung darstellen. Wenn Sie mehr Kontrolle über Ihre Animationen benötigen oder Dinge im dreidimensionalen Raum bewegen wollen, sind Layer der richtige Weg dafür.

Kapitel 22

Accelerometer

Vor noch nicht allzu langer Zeit bestand die physische Schnittstelle eines Telefons aus zehn Zifferntasten sowie dem * und dem #. Das iPhone zerstört dieses Paradigma durch seinen Touchscreen sehr gründlich, aber es ist genauso wichtig, das unsichtbare *Accelerometer* (ein Beschleunigungssensor) als wichtigen Bestandteil des Geräte-UI zu berücksichtigen. Indem Gravitation und Bewegung gemessen werden, kann das Accelerometer dabei helfen, eine Anwendung besser an den Bedürfnissen der Benutzer auszurichten. Liegt das Gerät auf der Seite, können die Accelerometer-Daten dafür sorgen, dass die App ihr UI dreht. Wird das Gerät geschüttelt, kann eine Anwendung diese Geste nutzen, um ein Undo oder ein Löschen durchzuführen. Und bei vielen Spielen ist ein Kippen und Drehen des Geräts ein ausgesprochen intuitives Kontrollschema, das ebenfalls durch das Accelerometer unterstützt wird.

In diesem Kapitel schauen wir uns genauer an, wie das Accelerometer Ihrer Anwendung Daten bereitstellen kann. Wir beginnen mit den APIs auf höherer Ebene, die Sie über die Ausrichtung des Geräts informieren, dann kümmern wir uns darum, wie das iPhone Ihre Anwendung an die Rohdaten aus dem Accelerometer gelangen lässt und wie Sie etwas damit anstellen können.

Der Simulator kippt nicht

Der iPhone-Simulator bietet keine ernsthafte Simulation des Accelerometers an. Wenn Ihre Anwendung den Simulator nach aktuellen Daten des Accelerometers fragt, passiert nichts. Daher müssen Sie zum Ausführen der Beispielanwendungen in diesem Kapitel in der Lage sein, Ihre Apps auf ein Gerät zu übertragen.

Aber was ist ein Accelerometer? Im Prinzip handelt es sich dabei um ein kleines Gerät, das Beschleunigungen misst. Insbesondere geht es dabei um Gravitation, Bewegung durch den Anwender oder beides. Diese Bewegung wird entlang dreier Achsen gemessen: x, y und z. Wenn Sie sich Ihr iPhone vor sich hingestellt vorstellen, mit dem Home-Button am unteren Ende, ist x die Achse, die von links nach rechts durch Ihren Bildschirm geht (parallel zur Ober- und Unterkante), y geht von unten nach oben (parallel zu den Seiten), und z ist die Achse, die aus dem Bildschirm auf Sie zukommt (senkrecht zur Bildelebene). Diese Anordnung sehen Sie in Abbildung 22.1.

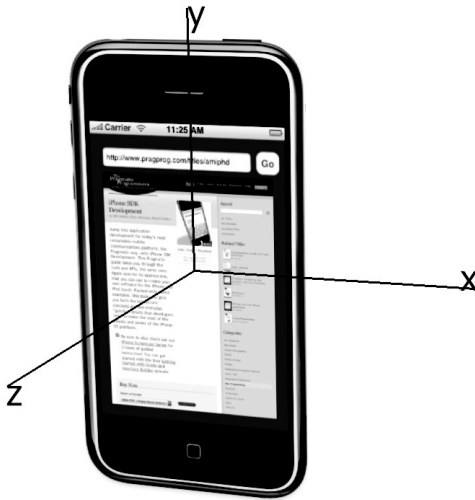


Abbildung 22.1: Darstellung der Accelerometer-Achsen

Vom Accelerometer erhalten Sie immer nur drei Gleitkommawerte für die Kraft entlang dieser Achsen, zusammen mit einem Zeitstempel der Messung. Die Frage ist nun: *Was zum Henker tun Sie damit?*

22.1 Ausrichtung des Geräts ermitteln

Als Erstes wollen wir uns mit den einfachen Fällen befassen, in denen das Accelerometer gar nicht direkt angesprochen wird. So wollen Sie zum Beispiel Ihre Benutzeroberfläche daran orientieren, ob das Gerät gedreht ist. Sie wissen schon aus Kapitel 2, *Hallo iPhone*, auf Seite 11, wie das geht. Dabei lassen Sie sich über eine Rotation informieren, indem Sie `shouldAutorotateToInterfaceOrientation:` so implementieren, dass es YES zurückgibt, wenn die App in die entsprechend vorgeschlagene Ausrichtung gedreht werden kann (eine von `UIInterfaceOrientation`-

`OrientationPortrait`, `UIInterfaceOrientationPortraitUpsideDown`, `UIInterfaceOrientationLandscapeLeft` oder `UIInterfaceOrientationLandscapeRight`). Wenn Sie eine gegebene Rotation unterstützen, erhält der View-Controller eine Reihe von Callbacks, wie zum Beispiel `willRotateToInterfaceOrientation:duration:`, um über den Fortschritt einer Rotationsanimation informiert zu werden. In der Klassenreferenz des `UIViewController` unter „Handling Rotation“ finden Sie eine vollständige Liste der Callback-Methoden und ihrer Bedeutung. Sie können auch Ihre aktuelle Ausrichtung ermitteln, indem Sie die vererbte Eigenschaft `interfaceOrientation` auslesen.

Die APIs des `UIViewController` zur Ausrichtung drehen sich wieder um die Benutzerschnittstelle und insbesondere um eine, die `UIKit` nutzt. Wir haben uns zwar im gesamten Buch auf das `UIKit` konzentriert, aber das ist nicht die einzige Möglichkeit, ein iPhone-GUI zu bauen. So kann zum Beispiel ein Spiel den gesamten Bildschirm per `OpenGL-View` „übernehmen“ und das `UIKit` nur wenig oder gar nicht nutzen. Solch eine Anwendung benötigt dann eine andere Möglichkeit, um an die Ausrichtungsdaten zu gelangen.

Um die Geräteausrichtungsdaten zu erhalten, ohne sich um die Rotation des aktuellen UIs zu kümmern oder überhaupt das `UIKit` zu nutzen, können Sie die Klasse `UIDevice` direkt verwenden. Sie erhalten die systemweit genutzte Instanz über `[UIDevice sharedInstance]` und lesen dann deren Eigenschaft `orientation` aus. Passen Sie aber auf: Sie müssen zunächst die Methode `beginGeneratingDeviceOrientationNotifications` aufrufen, um das Accelerometer auch zu aktivieren. Tun Sie das nicht, enthält die Eigenschaft `orientation` immer den bedeutungslosen Wert 0. Und nach dem Aktivieren des Accelerometers müssen Sie es irgendwo in Ihrer Applikation mit einem Aufruf von `endGeneratingDeviceOrientationNotifications` auch wieder deaktivieren.

22.2 Schüttelgesten über die Responder-Kette erhalten

Im iPhone OS 3.0 bietet das System eine gewisse Unterstützung für das Erkennen von Schüttelgesten für Ihre Anwendung an. Die Klasse `UIResponder`, bei der es sich um eine übergeordnete Klasse von `UIView` handelt, hat drei neue Methoden für die Bewegungserkennung erhalten: `motionBegan:withEvent:`, `motionEnded:withEvent:` und `motionCancelled:withEvent:`. Das erste Argument ist eine enumerierte Konstante des Typs `UIEventSubtype`, deren einziger sinnvoller Wert in 3.0 ein `UIEventSubtypeMotionShake` ist.

Um diese Events zu erhalten, muss Ihre View der First Responder sein (siehe Kapitel 18, *Events, Gesten und Multi-Touch*, auf Seite 403 zu weiteren Informationen über die Event-Verteilung im UIKit und über Responder) und dann mindestens eine der `motion...`-Methoden implementieren. Wir werden später in diesem Kapitel ein Beispiel dafür zeigen, wenn wir eine Accelerometer-basierte Applikation aufbauen. Dort werden wir auch darauf hinweisen, wie Sie die Accelerometer-Rohdaten nutzen können, um eine eigene Schüttelerkennung zu implementieren – eine Technik, die auch abwärtskompatibel zum iPhone OS 2.0 ist.

Im iPhone SDK 3.0 gibt es eine weitere Acceleration-API, die Sie kennen sollten. Nutzen Sie den `NSUndoManager`, um in Ihrer Anwendung Schritte rückgängig zu machen, können Sie auch die Eigenschaft `applicationSupportsShakeToEdit` in der gemeinsam genutzten Instanz von `UIApplication` setzen, um die Undo- und Redo-Buttons des Undo-Managers zu zeigen, wenn der Anwender das Gerät schüttelt.

22.3 Rohdaten des Accelerometers auslesen

Die Ausrichtungs- und die Schüttelerkennung sind zwar nett, aber es gibt eine ganze Reihe interessanter Dinge, die Sie machen können, wenn Sie Zugriff auf die Rohdaten des Accelerometers haben. Sie können Ihre Ausrichtung in drei Dimensionen bestimmen (und müssen nicht annehmen, dass das Gerät aufrecht steht), und Sie können diesen „Kippstatus“ zusätzlich als Benutzerschnittstelle verwenden. Oder Sie filtern die Gravitation aus und verarbeiten einfach die Benutzerbewegungen – das Drehen, Schütteln, Schwingen oder Fallenlassen des Geräts.

Für ein erstes Beispiel wollen wir die Erkennungsmöglichkeit des Accelerometers für die Gravitationskraft nutzen. Im Beispiel `BalanceBall` werden wir einen Ball auf dem Bildschirm herumrollen lassen, indem wir von oben auf das Gerät schauen und es kippen. Wenn Sie es zu einer Seite kippen, wird sich die Gravitationskraft auf den x- und/oder y-Achsen zeigen, und wir werden diese Beschleunigung auf die x- und y-Komponenten der Ballgeschwindigkeit anwenden.

Aus technischen Gründen haben wir uns dazu entschieden, das Utility Application Template von Xcode zu nutzen (das wir in Abschnitt 9.1, *Anzeige einer umklappbaren Einstellungs-View in Ihrer Anwendung*, auf Seite 181 vorgestellt haben). Denn erstens bietet es uns den `INFO-Button`, der die View umblättert, womit wir einen Einstellungsbildschirm auf der Rückseite anbieten können, auf dem wir eine Filterlogik für die Verbesserung der Accelerometer-Daten ein- und ausschalten, und

zweitens erhalten wir eine eigene Subklasse von UIView, die wir nutzen können, um unser Spielfeld zu zeichnen.

Die Utility-Anwendung hat schon einen MainViewController für uns, der sich um die Logik der „Vorderseite“ kümmert. Hier werden wir Position und Geschwindigkeit des Balls verwalten und die View aktualisieren. In dieser Klasse wollen wir über Updates des Accelerometers informiert werden. Danach werden wir fragen, wenn die View geladen ist – also in viewDidLoad.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
// Nach Accelerometer-Events fragen
[[UIAccelerometer sharedAccelerometer] setInterval: 0.03]; // 30 fps
[[UIAccelerometer sharedAccelerometer] setDelegate: self];
```

Die erste Codezeile legt fest, wie oft wir Updates vom Accelerometer erhalten wollen – angegeben als Zeitintervall zwischen den Callbacks in Sekunden. Es gibt keine Garantie, aber innerhalb sinnvoller Bereiche können Sie davon ausgehen, dass Sie mehr oder weniger die Frequenz erhalten, um die Sie gebeten haben. Die Wahl der richtigen Frequenz hängt stark davon ab, was Sie mit den Daten anfangen wollen. Sind Sie nur an der Ausrichtung interessiert, erhalten Sie eine akzeptable Auflösung, wenn Sie 10 oder 20 Mal pro Sekunde abfragen, also ein Intervall von 0.10 bis 0.05 Sekunden nutzen. Bei Spielen oder in anderen Fällen, in denen das Accelerometer eine höhere Auflösung bieten soll, müssen Sie sich näher an der Framerate halten, also ein Intervall von 0.033 (30 Updates pro Sekunde) oder sogar 0.02 (50 Updates pro Sekunde) nutzen.

Die zweite Zeile setzt einen Delegate, um Callbacks mit den Accelerometer-Daten zu erhalten. Sie können in Ihrer Anwendung nur ein UIAccelerometerDelegate nutzen. Sollen mehrere Teile Ihrer App Aktualisierungen vom Accelerometer erhalten, ist die App selbst dafür zuständig, sie zu verteilen. Sobald Sie den Delegate setzen, erhalten Sie Callbacks mit den Daten, bis Sie den Delegate per nil zurücksetzen und damit zeigen, dass Sie das Accelerometer nicht mehr brauchen. In der Ball-Anwendung wollen wir ständig Accelerometer-Daten erhalten, daher setzen wir den Delegate erst in der dealloc-Methode wieder auf nil.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
- (void)dealloc {
    [[UIAccelerometer sharedAccelerometer] setDelegate: nil];
    [super dealloc];
}
```

Das Protokoll `UIAccelerometerDelegate` müssen wir in `MainViewController.h` deklarieren – fügen Sie also `<UIAccelerometerDelegate>` zum `@interface` hinzu und implementieren Sie die Methode `accelerometer:didAccelerate:`. Diese Methode liefert uns das `UIAccelerometer`, das die Daten bereitstellt, und ein Objekt vom Typ `UIAcceleration` mit den Accelerometer-Daten.

Aber was können wir mit den Daten anstellen? Wir haben noch keinen rollenden Ball, auf den wir die Gravitationskraft wirken lassen können, also kümmern wir uns zunächst darum.

Wir verwalten Position und Geschwindigkeit des Balls im View-Controller und lassen die eigene View-Klasse lediglich den Ball zeichnen. Für dieses Beispiel werden wir nur sehr einfache Grafik-Primitiven nutzen, die schon in Kapitel 19, *Zeichnen in eigenen Views*, auf Seite 415 vorgestellt wurden. Zum Zeichnen des Balls müssen wir dessen aktuelles Zeichenrechteck, das als `CGRect` repräsentiert ist, und dessen vorheriges Rechteck kennen, das wir erst löschen werden, bevor wir den Ball an seiner neuen Position zeichnen. Also brauchen wir eine Handvoll `CGRect`s und Methoden, um die Position und die Größe des Balls zu setzen. Dies ist die Header-Datei `MainView.h`:

Accelerometer/BalanceBallFor3.0/Classes/MainView.h

```
@interface MainView : UIView {
    CGRect ballRect;
    CGRect oldBallRect;
}
- (void) setBallX: (CGFloat) newX Y: (CGFloat) newY;
- (void) setBallWidth: (CGFloat) newW height: (CGFloat) newH;
```

Das Implementieren der Methode `setBallWidth:height:` ist trivial:

Accelerometer/BalanceBallFor3.0/Classes/MainView.m

```
-(void) setBallWidth: (CGFloat) newW height: (CGFloat) newH {
    ballRect.size.width = newW;
    ballRect.size.height = newH;
    oldBallRect.size.width = newW;
    oldBallRect.size.height = newH;
}
```

Das Setzen der Ballposition erfordert drei einfache Schritte: Aktualisieren des Ursprungs von `ballRect`, Anfordern einer Aktualisierung der View und Sichern des Rechtecks, das wir gerade gezeichnet haben, als zu löschen-des Rechteck für die nächste Aktualisierung. Löschen und neu zu zeichnen, ist effizienter als das Zeichnen der gesamten View. Zudem verbessern wir die Effizienz noch dadurch, dass wir ein Clipping-Rechteck angeben,

das Quartz den veränderten Teil der View mitteilt, also die einzigen Bereiche, die eventuell neu zu zeichnen sind. Dabei handelt es sich um die Vereinigungsmenge des alten und des neuen Zeichenrechtecks. Dieses Vereinigungsrechteck berechnen wir mit `CGRectUnion()`.

Accelerometer/BalanceBallFor3.0/Classes/MainView.m

```
-(void) setBallX: (CGFloat) newX Y: (CGFloat) newY {
    ballRect.origin.x = newX;
    ballRect.origin.y = newY;

    // View aktualisieren
    CGRect clipRect = CGRectUnion (oldBallRect, ballRect);
    [self setNeedsDisplayInRect:clipRect];

    // oldBallRect aktualisieren
    oldBallRect.origin.x = ballRect.origin.x;
    oldBallRect.origin.y = ballRect.origin.y;
}
```

Der Aufruf `setNeedsDisplayInRect:` führt dazu, dass `drawRect:` aufgerufen wird. Unsere Implementierung dieser Methode zeichnet die View, indem sie den aktuellen Grafikkontext holt, das alte Zeichenrechteck mit der Hintergrundfarbe füllt (die wir im IB auf Grün setzen – Sie mögen da etwas anderes wählen) und dann an der Ballposition einen einfachen weißen Kreis malt.

Accelerometer/BalanceBallFor3.0/Classes/MainView.m

```
-(void)drawRect:(CGRect)rect {
    // Code fuers Zeichnen
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Ball an alter Position entfernen
    CGColorRef undrawColor = self.backgroundColor.CGColor;
    CGContextSetFillColorWithColor (context, undrawColor);
    CGContextFillRect (context, oldBallRect);

    // Ball an neuer Position malen
    CGContextSetGrayFillColor(context, 1.0, 1.0);
    CGContextFillEllipseInRect(context, ballRect);
}
```

Das Zeichnen haben wir nun erledigt, also können wir unsere Aufmerksamkeit wieder dem View-Controller und dem Accelerometer zuwenden. Im Header des `MainViewController` definieren wir Variablen für die Position und die Geschwindigkeit des Balls. Wir wollen uns auch den Timestamp des letzten Accelerometer-Callbacks merken, um zu berechnen, wie viel Zeit seitdem vergangen ist und wie weit sich der Ball daher bewegt hat. Schließlich verbinden wir ein `IBOutlet` mit der `MainView`, so dass wir deren Methode `setBallX:Y:` aufrufen können.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.h

```
CGFloat ballX;
CGFloat ballY;
CGFloat ballVelocityX;
CGFloat ballVelocityY;
NSTimeInterval lastAcceTimestamp;
```

Da wir schon dabei sind, werden wir auch ein paar Konstanten für die Höhe und Breite des Balls (beide 20.0) und eine MAX_ACCEL_PER_SEC definieren, die angibt, wie viele Einheiten pro Sekunde eine Beschleunigung von 1.0 darstellen soll (das ist die Gravitationskraft, wenn das Gerät ganz auf einer Kante steht). Mit all diesen Vorbereitungen können wir nun eine Methode bereitstellen, die den Ball in die Mitte der View platziert. Diese werden wir aufrufen, wenn die Anwendung startet. Zudem müssen wir prüfen können, ob der Ball die View verlassen hat, da wir ihn dann zurücksetzen müssen.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
- (void) resetBall {
    ballVelocityX = 0.0;
    ballVelocityY = 0.0;
    filteredAccelX = 0.0;
    filteredAccelY = 0.0;
    ballX = self.view.frame.size.width / 2;
    ballY = self.view.frame.size.height / 2;
}

- (void) checkBallInPlay {
    if ((ballX + BALL_WIDTH < 0) ||
        (ballY + BALL_HEIGHT < 0) ||
        (ballX > self.view.frame.size.width) ||
        (ballY > self.view.frame.size.height)) {
        NSLog(@"Ball zurücksetzen" );
        [self resetBall];
    }
}
```

In viewDidLoad können Sie jetzt den anfänglichen Status des Spiels durch den Aufruf der Methoden herstellen, mit denen die Ballgröße und die Position gesetzt werden, und die View aktualisieren. Beachten Sie, dass wir die Eigenschaft view nach MainView casten müssen, um setBallWidth:height: und setBallX:Y: ohne Warnung aufrufen zu können.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
[(MainView*) self.view setBallWidth: BALL_WIDTH height:BALL_HEIGHT];
[self resetBall];
[(MainView*) self.view setBallX:ballX Y:ballY];
```

Unsere Simulation eines rollenden Balls hat jetzt einen Ball (komplett mit Darstellung auf dem Bildschirm und einem Modell der Position und Geschwindigkeit), aber er kann bisher nicht rollen. Das liegt daran, dass wir noch nichts mit den Accelerometer-Daten anstellen, die unser Delegate (der View-Controller) empfangen wird. Darum wollen wir uns jetzt endlich kümmern. Immer wenn wir einen Callback mit Accelerometer-Daten erhalten, wenden wir die x- und y-Komponente auf die Geschwindigkeit unseres Balls an, die in Einheiten pro Sekunde gemessen wird. Steht das Gerät senkrecht auf der schmalen Kante, sollte es eine Beschleunigung von 1.0 auf der y-Achse erhalten, steht es auf der breiten Kante, erhält man die 1.0 auf der x-Achse. Durch Neigen erzielt man Beschleunigungen auf beiden Achsen und auch auf z, das wir aber in dieser Anwendung nicht nutzen werden. Wir haben eine Konstante definiert, die angibt, wie viel Beschleunigung eine Sekunde mit 1g in unserer Spielwelt bedeutet, so dass wir `accelerometer:didAccelerate:` implementieren können, um diese Beschleunigung auf die Geschwindigkeit des Balls anzuwenden und mit dieser Geschwindigkeit und der Zeit seit der letzten Aktualisierung auszurechnen, wie weit sich der Ball bewegt hat.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration {
    NSTimeInterval elapsedTime =
        acceleration.timestamp -lastAccelTimestamp;
    ballVelocityX = ballVelocityX +
        (acceleration.x * MAX_ACCEL_PER_SEC * elapsedTime);
    ballVelocityY = ballVelocityY
        (acceleration.y * MAX_ACCEL_PER_SEC * elapsedTime);
    if (lastAccelTimestamp > 0) {
        // Ballposition neu berechnen
        ballX += (ballVelocityX * elapsedTime);
        ballY += (ballVelocityY * elapsedTime);
        // View aktualisieren
        [mainView updateBallX:ballX Y:ballY];
        // Prüfen, ob der Ball noch im Feld ist
        [self checkBallInPlay];
    }
    lastAccelTimestamp = acceleration.timestamp;
}
```

Die Callback-Methode liefert uns ein `UIAcceleration`-Objekt mit vier Eigenschaften: einem Wert vom Typ `NSTimeInterval` für den `timestamp` und den drei Eigenschaften `x`, `y` und `z` vom Typ `UIAccelerationValue` (definiert als `double`). In dieser Implementierung nehmen wir die Beschleunigung für eine Achse und bestimmen, wie stark sich diese auf die Geschwindigkeit in diesem Callback-Intervall auswirkt. Erhalten wir zum Beispiel einen Wert von 1.0 und haben eine ganze Sekunde lang

nicht aktualisiert, addieren oder subtrahieren wir 1.0 Sekunden mal 1.0g mal MAX_ACCEL_PER_SEC zur oder von der Geschwindigkeit. In der Praxis erhalten wir natürlich kleinere Werte viel häufiger. Achten Sie auch darauf, dass wir zwar auf der x-Achse die Werte addieren, sie aber auf der y-Achse subtrahieren. Das liegt daran, dass das Accelerometer die Beschleunigung nach oben als positiv und nach unten als negativ angibt, während im Koordinatensystem des iPhone die y-Werte nach unten hin größer werden. Daher muss eine positive y-Beschleunigung (eine Beschleunigung zum oberen Ende des Bildschirms hin) zu einer negativen Geschwindigkeitskomponente führen, so dass der Ursprung des Balls relativ zum Koordinatensystem nach „unten“ wandert.¹

Damit ist die BalanceBall-Anwendung spielbereit. Wenn Sie die Berechtigung haben, eine Anwendung zu signieren und sie auf Ihr Gerät zu übertragen, legen Sie los! Sie werden feststellen, dass man das iPhone stark kippen muss, um den Ball in die Gegenrichtung zu lenken. Das liegt zum Teil daran, dass wir noch keine Reibung berücksichtigt haben, um den Ball zu verlangsamen, wenn er in Fahrt gekommen ist. Die Anwendung in all ihrer grafischen Schlichtheit sehen Sie in Abbildung 22.2.

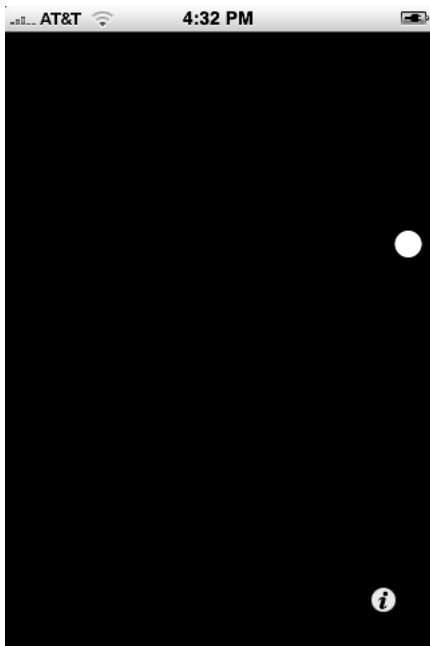


Abbildung 22.2: Einen Ball auf dem Bildschirm mit dem Accelerometer bewegen

¹ Es wäre vielleicht sauberer gewesen, im Zeichnungscode der View eine Transformationsmatrix anzuwenden und das Koordinatensystem damit zu spiegeln.

22.4 Accelerometer-Daten filtern

Bisher haben wir die Rohdaten nur recht simpel eingesetzt, indem wir einen virtuellen Ball durch die gemessenen Beschleunigungen bewegt haben. Aber die hohe Kunst des Accelerometer-Einsatzes ist es, geschickte Entscheidungen bezüglich der durch die Callbacks erhaltenen Daten zu treffen, wie zum Beispiel das Ausblenden fehlerhafter Daten und die Trennung von „Benutzerbewegung“ und Gravitation.

Einfache Accelerometer-Filter

Wie schon beschrieben, nimmt unsere Ball-Anwendung einfach die gemessene Beschleunigung und wendet sie auf die Geschwindigkeit des Balls an – unabhängig davon, ob die Beschleunigung durch Gravitation oder Benutzerbewegung entstanden ist. Viele Apps werden aber nur eine der beiden Kräftequellen auswerten wollen, und oft lässt sich das durch ein Paar einfacher *Filter*-Algorithmen voneinander trennen. Diese einfachen Filter, die auch in der Apple-Dokumentation und in vielen Beispielen genutzt werden, sind ein *Tiefpass*- und ein *Hochpass*-Filter. Beide bauen auf der Tatsache auf, dass die Gravitation immer vorhanden ist.

Der Tiefpass-Filter nimmt einen Satz Accelerometer-Daten und wendet ihn auf vorberechnete Accelerometer-Daten an, wobei er aber nur einen kleinen Teil des neuen Werts nutzt. Damit werden plötzliche Bewegungen herausgefiltert, und nur die langfristigen Beschleunigungen bleiben bestehen, was zur Folge hat, dass man damit unruhige Finger ignorieren und die reinen Gravitationskräfte nutzen kann. Der Hochpass-Filter ist das Gegenteil: Er subtrahiert die berechneten Tiefpass-Filterwerte vom aktuellen Accelerometer-Wert, wodurch langfristig wirkende Kräfte herausfallen und nur die aktuellen Bewegungen übrig bleiben. Daher ist der Hochpass-Filter besser dafür geeignet, die Bewegungen des Anwenders zu erkennen.

Probieren Sie das jetzt an der Ball-Anwendung einmal aus. Wir werden auf die Flipside-View ein Segmented Control packen, damit der Anwender entscheiden kann, welche Art von Filter er nutzen will. Und um die Erkennung schneller Bewegungen noch weiter zu testen, werden wir den Hochpass-Filter nutzen, um das Schütteln des Geräts zu erkennen.

Da wir in Xcode das Utility Application-Template gewählt haben, besitzt Ihre App schon einen INFO-Button auf Ihrer Haupt-View, die Sie auf eine Flipside-View bringt, in der Sie die Einstellungen vornehmen können. Fügen Sie der FlipsideView.xib ein UISegmentedControl hinzu (siehe Abbildung 22.3) und verbinden Sie es mit der Eigenschaft `filterControl`. Es muss sich um eine Eigenschaft handeln, damit man

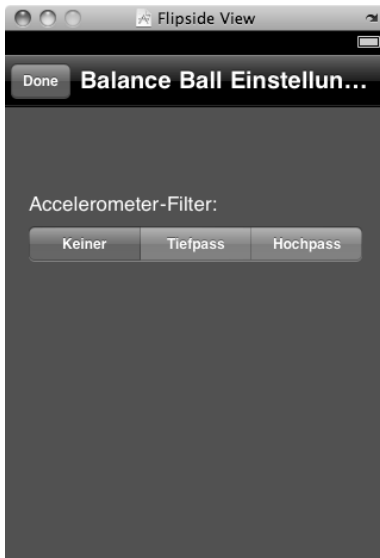


Abbildung 22.3: Einstellungs-UI für den Accelerometer-Filter

deren Wert lesen kann, wenn die Flipside-View verworfen wird. Holen Sie in der Methode `flipsideViewControllerDidFinish` des `MainViewController` den Wert des `filterControl` und nutzen Sie ihn als neuen Wert für `filterPref`, bei dem es sich um eine Instanzvariable handelt, in der der gewählte Filtertyp abgelegt ist.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
-(void)flipsideViewControllerDidFinish:
    (FlipsideViewController *)controller {
    [self dismissModalViewControllerAnimated:YES];
    // Filterwert setzen
    filterPref = controller.filterControl.selectedSegmentIndex;
}
```

Wir werden das Filtern im `MainViewController.m` vornehmen, daher fügen wir innerhalb der **@implementation** ein paar benötigte Konstanten ein – beginnend mit einem **enum** für die möglichen Filterwerte, die vom `Segmented Control` übernommen wurden. Als Nächstes definieren wir einen „Filterfaktor“, der angibt, wie viel Gewicht wir jedem neuen Wert geben, den wir vom Accelerometer erhalten. Mit einem Faktor von 0.1 wird der Tiefpass-Filter dem neuen Wert ein 10%iges Gewicht und den vorher gefilterten Werten ein 90%iges Gewicht geben. Schließlich wollen wir noch eine Konstante haben, die angibt, wie groß eine Beschleunigung sein muss, damit sie als „Schütteln“ gilt. Zweifache Erdbeschleunigung (also 2g) ist dabei stark genug, damit nicht irgendwelche „normalen“ Bewegungen missinterpretiert werden.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
// Filtertypen als Werte des Multi-Switch
enum SORT_TYPES {
    NO_FILTER = 0,
    LOW_PASS_FILTER,
    HIGH_PASS_FILTER
};

// Wie stark wichten wir einen einzelnen Accelerometer-Wert?
float kFilteringFactor = 0.1;

// Welche Beschleunigung zählt als Schütteln (Hochpass-gefiltert)?
CGFloat SHAKE_THRESHOLD = 2.0;
```

Jetzt können wir in der Methode `accelerometer:didAccelerate:`, in der wir uns schon um die Accelerometer-Daten kümmern, auch das Filtern durchführen. Wir benötigen in der Header-Datei deklarierte Instanzvariablen für `filteredAccelX` und `filteredAccelY`, da die Algorithmen für Hochpass- und Tiefpass-Filter die vorher gefilterten Acceleration-Werte brauchen, um den neuen Wert zu berechnen. Hier sehen Sie die einfachen Formeln für die Hochpass- und Tiefpass-Filter auf den x- und y-Achsen:

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```
CGFloat lowPassFilteredX = (acceleration.x * kFilteringFactor) +
(filteredAccelX * (1.0 - kFilteringFactor));
CGFloat lowPassFilteredY = (acceleration.y * kFilteringFactor) +
(filteredAccelY * (1.0 - kFilteringFactor));
CGFloat highPassFilteredX = acceleration.x - lowPassFilteredX;
CGFloat highPassFilteredY = acceleration.y - lowPassFilteredY;
```

Wenn Sie sich erinnern, dass wir `kFilterFactor` als 0.1 definiert haben, können Sie sehen, dass die Tiefpass-Filter im Prinzip nichts anderes sind als: „10% des aktuellen Werts zu 90% des alten Werts addieren“. Damit werden einmalige Ausrutscher stark gedämpft, und es bleiben die Beschleunigungen, die dauerhafter wirken (also die Gravitation). Der Hochpass-Filter nutzt nun den berechneten Tiefpass-Wert und subtrahiert ihn vom aktuellen Accelerometer-Wert.

Jetzt müssen wir nur noch die Wahl des Benutzers bezüglich der gefilterten Beschleunigungswerte nutzen, um die Geschwindigkeit des Balls zu beeinflussen, wobei wir die früheren Berechnungen von `ballVelocityX` und `ballVelocityY` wie folgt ersetzen:

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```

switch (filterPref) {
    case NO_FILTER: {
        filteredAccelX = acceleration.x;
        filteredAccelY = acceleration.y;
        break;
    }
    case LOW_PASS_FILTER: {
        filteredAccelX = lowPassFilteredX;
        filteredAccelY = lowPassFilteredY;
        break;
    }
    case HIGH_PASS_FILTER: {
        filteredAccelX = highPassFilteredX;
        filteredAccelY = highPassFilteredY;
        break;
    }
}

// Beschleunigung auf Geschwindigkeit anwenden
NSTimeInterval elapsedTime = acceleration.timestamp - lastAccelTimestamp;
ballVelocityX = ballVelocityX +
    (filteredAccelX * MAX_ACCEL_PER_SEC * elapsedTime);
ballVelocityY = ballVelocityY +
    (filteredAccelY * MAX_ACCEL_PER_SEC * elapsedTime);

```

Da wir gerade dabei sind, sollten wir auch gleich noch die alten Werte von `filteredAccelX` und `filteredAccelY` sowie die Geschwindigkeit auf null setzen, wenn wir den Ball in die Mitte der View zurücksetzen.

Schüttelerkennung implementieren

Mithilfe der Filter wollen wir auch noch ein Schütteln erkennen. Für Benutzer des iPhone OS 3.0 können wir das dem System überlassen – dieses erkennt das Schütteln und schickt ein Event an unsere Main-View. Das ist praktisch, aber wir wollen im View-Controller auf das Schütteln reagieren, nicht in der View. Also schicken wir das Event mit einem informellen Protokoll an den Controller. Definieren Sie in `MainView.h` eine Eigenschaft `shakeDelegate`:

Accelerometer/BalanceBallFor3.0/Classes/MainView.h

```
@property (nonatomic, retain) id shakeDelegate;
```

Zurück im `MainViewController`, können Sie dieses Delegate mit dem View-Controller belegen. Und wenn Sie schon da sind, sollten Sie die View auch noch zum First Responder machen, um die UIEvents zu empfangen. Machen wir also beides in `viewWillAppear:`.

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```

-(void) viewWillAppear: (BOOL) animated {
    [super viewWillAppear: animated];
    [self.view becomeFirstResponder];
    ((MainView*)self.view).shakeDelegate = self;
}

```

Jetzt müssen wir noch dafür sorgen, dass die View die Schüttel-Events an den Controller weiterreicht. Die View muss First Responder werden können und dann die Callback-Methode `motionBegan:withEvent:` implementieren. Wir rufen einfach die Methode `shakeMotionBegan:` des Delegates auf, wenn es eine solche gibt:

Accelerometer/BalanceBallFor3.0/Classes/MainView.m

```

-(BOOL) canBecomeFirstResponder {
    return YES;
}
-(void) motionBegan: (UIEventSubtype) motion withEvent: (UIEvent *) event {
    [super motionBegan: motion withEvent: event];
    if ((motion == UIEventSubtypeMotionShake) &&
        [self.shakeDelegate respondsToSelector:
         @selector (shakeMotionBegan:)]) {
        [self.shakeDelegate shakeMotionBegan: event];
    }
}

```

Zurück im Controller, implementieren wir den Callback, indem wir einfach das Event protokollieren und die Position des Balls zurücksetzen:

Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m

```

-(void) shakeMotionBegan: (UIEvent*) event {
    NSLog(@"Schüttelbewegung vom System gemeldet");
    [self resetBall];
}

```

Aber was, wenn wir nicht davon ausgehen können, dass der Anwender das iPhone OS 3.0 hat? Um auch ein Schütteln-zum-Zurücksetzen auf dem iPhone 2.0 zu unterstützen, können wir den Hochpass-Filter nutzen, den wir schon eingerichtet haben. Dieser entfernt den größten Teil der Gravitation, und es bleibt hauptsächlich die Bewegung durch den Anwender. Als einfache Implementierung interpretieren wir einfach jede heftige Bewegung auf der x- oder y-Achse (über den Wert von `SHAKE_THRESHOLD` hinausgehend, das wir als 2.0g definiert haben) als Schütteln. Darauf reagieren wir, indem wir den Ball in die Mitte zurücksetzen:

```
Accelerometer/BalanceBallFor3.0/Classes/MainViewController.m
```

```
// Schütteln erkennen
if ((fabsf (filteredAccelX) > SHAKE_THRESHOLD) ||
    (fabsf (filteredAccelY) > SHAKE_THRESHOLD)) {
    [self resetBall];
}
```

Beachten Sie, wie wir die Funktion für den absoluten Wert, `fabsf()`, nutzen, um unabhängig vom Vorzeichen der Beschleunigung nur die Stärke zu berücksichtigen, um herauszufinden, ob das Schütteln ausreichend stark war.

Versuchen Sie einmal, den Ball auf Ihrem Gerät rollen zu lassen und dabei die verschiedenen Filter anzuwenden. Der Unterschied zwischen keinem Filter und dem Tiefpass-Filter ist nur gering, da die Gravitationskraft viel stärker als die Beschleunigung durch Ihr Kippen ist, was nur die Anziehung zum Boden verstärkt. Aber es gibt durchaus einen kleinen Unterschied. Der Hochpass-Filter andererseits ist in diesem Fall nicht sinnvoll nutzbar, da Sie dann mehr Kraft aufwenden müssten als das bisschen Gravitation, die nicht ausgefiltert wird. Der Nebeneffekt des Hochpass-Filters ist trotzdem die Schüttelerkennung. Bewegen Sie Ihr iPhone schnell nach rechts, links, oben oder unten, wird der Ball in die Mitte des Bildschirms zurückgesetzt.

Fortgeschrittenes Filtern der Accelerometer-Daten

Wir haben schon gesagt, dass diese Filter ziemlich simpel sind, und sie haben auch deutliche Grenzen. So gibt es zum Beispiel immer eine Erdanziehung, die durch den Hochpass-Filter schlüpft. Sie können sich also nicht vollständig darauf verlassen, wenn Sie eine Spielsteuerung entwerfen wollen.

Begibt man sich tiefer in die Welt der Filter, riskiert man, zu tief in die digitale Signalverarbeitung einzusteigen, aber wir haben durchaus ein paar Ideen für weitere Forschungen.

Schreiben Sie ein Spiel, das Sie durch Kippen steuern wollen, sollten Sie sich vielleicht den Accelerometer-Code in der Apple-Demo *Touch Fighter II* anschauen, die auf der WWDC 2008 und der iPhone Tech Talk World Tour Ende 2008 präsentiert wurde. Dieser Ego-Weltraum-Shooter merkt sich Kalibrationsdaten, um die anfängliche Ausrichtung und Gravitationskraft zu berücksichtigen. Treibt das Raumschiff des Spielers zu sehr in eine Richtung auf dem Bildschirm ab, wird die Kalibrierung neu durchgeführt, und mit diesen neuen Werten werden die eintreffenden

Accelerometer-Daten bearbeitet, um nur die relevanten Benutzerbewegungen übrig zu behalten. Leider ist der Code zum Zeitpunkt der Entstehung dieses Buchs nur für Teilnehmer dieser Events verfügbar.

Die in diesem Kapitel vorgestellte Schüttel-Logik berücksichtigt nur einen einzelnen, heftigen Ruck, der stark genug ist, 2g Kraft entweder auf die x- oder die y-Achse zu geben. Besser wäre es vielleicht, eine Vor- und Zurückbewegung zu erkennen, die dafür nicht unbedingt so stark sein muss. Eine einfache Version dieser Erkennung würde auf eine ausreichend starke Bewegung in eine Richtung achten und dann darauf warten, ob es innerhalb einer sehr kurzen Zeitspanne (zum Beispiel ein paar hundert Millisekunden) eine Gegenbewegung gibt. Eine ausgefeiltere Schüttelerkennung könnte das Schütteln als wiederholtes Signal entlang einer Achse suchen. Das ließe sich vermutlich mit einer Fourier-Transformation erreichen, die Frequenzen in Funktionen erkennt. Allerdings gibt es im iPhone nicht die Fast Fourier Transform-(FFT-)Funktionen, die normalerweise auf dem Desktop zu finden sind, also müssten Sie Ihre eigene Implementierung bereitstellen. Oder, wenn Sie sich darauf verlassen können, dass Ihre Anwender das iPhone OS 3.0 nutzen, greifen Sie einfach über UIResponder auf die Schüttel-Events des Systems zurück.

Sie können mit dem Gerät auch versuchen, spezielle Bewegungen zu erkennen: Schwünge beim Golf, Zeigebewegungen, Schwerthiebe und so weiter. Diese Art von Bewegungen ist auf einem anderen Gerät, das auch mit Accelerometer-Daten arbeitet, sehr beliebt: der Nintendo Wii. Wir haben ein paar Freunde gefragt, die Spieleentwickler sind, wo sie die entsprechenden Tricks gelernt haben. Ihrer Meinung nach sind die besten Quellen für solche Informationen die Wii-Entwicklerforen, die aber nur für Nintendo-Lizenznehmer erreichbar sind. Wir haben aber trotzdem mit ihrer Hilfe ein paar interessante akademische Forschungsarbeiten zum Thema gefunden, einschließlich eines Artikels der Universität Oldenburg, „Gesture Recognition with a Wii Controller“ [SPHB08], in dem eine Reihe von Techniken beschrieben sind, wie zum Beispiel das Modellieren von Gesten auf Punkte einer Kugel. Ein Gamasutra-Artikel „Where’s the Wiimote? Using Kalman Filtering To Extract Accelerometer Data“ [Ras07] filtert Bewegungsdaten mit einer zeitabhängigen Wahrscheinlichkeitstheorie: Wenn das Gerät eine bestimmte Bewegung hinter sich hat, wohin wird es vermutlich als Nächstes bewegt werden? Beachten Sie, dass die meisten veröffentlichten Artikel über Beschleunigungserkennung stark mathematiklastig sind, gespickt mit

Differentialgleichungen und Matrixmultiplikationen. Das schränkt die Anzahl der Entwickler ein, die die Schlussfolgerungen dieser Artikel verstehen und in Code umsetzen können. Wir hoffen, dass in naher Zukunft ausgefeilte Accelerometer-Bibliotheken erscheinen, die von ganz normalen iPhone-Entwicklern genutzt werden können.

Kapitel 23

Adressbuch

Ihre Anwendung kann Telefonnummern, Adressen und jede andere Information nutzen, die ein Anwender in seinem Adressbuch speichert. Sie verbinden sich mit dem Adressbuch über View-Controller – auf ähnlichem Weg wie mit der Fotobibliothek und der Kamera. Sie werden auch feststellen, dass Sie wieder einmal vor allem mit C-Funktionen arbeiten werden, um auf die ausgewählten Daten zugreifen zu können. In diesem Kapitel werden wir die zwei Frameworks nutzen, die mit dem Adressbuch verbunden sind, um diese zusätzliche Personalisierung in unseren Anwendungen zu ermöglichen.

23.1 Adressbuch-UI

Die Integration des Adressbuchs geschieht über zwei View-Controller, die wir nutzen können, um bestehende Kontakte anzuzeigen. Der erste heißt `ABPeoplePickerNavigationController`, der es ermöglicht, einen Kontakt aus der Datenbank auszuwählen. Der zweite hat den Namen `ABPersonViewController`, und mit ihm können wir einen einzelnen Kontakt anzeigen und bearbeiten. Der People Picker ist, wie der Name schon sagt, ein Navigations-Controller, und der Person-Controller ist ein View-Controller.

Eine Beispielanwendung wird uns dabei helfen, diese beiden Controller besser zu verstehen. Lassen Sie uns eine App erzeugen, mit der wir eine Reihe von Kontakten auswählen, sie alphabetisch aufführen und dann einzeln anzeigen können, um ihre Telefonnummern anzuzeigen. Erzeugen Sie also ein neues navigationsbasiertes Projekt mit dem Namen `Contacts`.

Da wir mit Adressbuchklassen interagieren werden, müssen wir unserem Projekt die Frameworks `AddressBook.framework` und `AddressBookUI.framework` hinzufügen. Klicken Sie dazu mit der rechten Maustaste auf die Gruppe „Frameworks“, wählen Sie `ADD EXISTING FRAMEWORK`, navigieren Sie zum iPhone SDK, wählen Sie die beiden Frameworks aus und klicken Sie dann auf den `ADD`-Button.

Das Integrieren mit dem Adressbuch läuft ähnlich ab wie bei den Controllern der Fotobibliothek in Kapitel 20, *Bilder und Fotos zeichnen*, auf Seite 427. Der `RootViewController` wird Delegationsprotokolle implementieren und dann der Delegate des Adressbuch-Controllers werden. Damit erhält er dann Callbacks von den Adressbuch-Controllern, wenn bestimmte Dinge geschehen (der Benutzer zum Beispiel einen Kontakt auswählt).

Vor allem müssen wir die Protokolle `ABPeoplePickerNavigationControllerDelegate` und `ABPersonViewControllerDelegate` zum `RootViewController` hinzufügen und die notwendigen Methoden implementieren. Zudem benötigen wir zwei Verbindungen: eine, in der die Namen der ausgewählten Kontakte stehen, und eine für deren Kennungen. Öffnen Sie die `.h`-Datei und fügen Sie die Protokolldeklarationen, die Instanzvariable und die Eigenschaft hinzu. Der Code sollte danach in etwa so aussehen:

AddressBook/Contacts/Classes/RootViewController.h

```
#import <UIKit/UIKit.h>
#import <AddressBookUI/AddressBookUI.h>

@interface RootViewController : UITableViewController
    <ABPeoplePickerNavigationControllerDelegate,
    ABPersonViewControllerDelegate> {
    NSMutableArray *contactNames;
    NSMutableArray *contactIDs;
}

@property(n nonatomic, retain) NSMutableArray *contactNames;
@property(n nonatomic, retain) NSMutableArray *contactIDs;

-(IBAction)makeNewEntry;

@end
```

23.2 People Picker-Delegate

Es gibt drei Methoden, die als Teil dieses Protokolls zu implementieren sind. Wir werden sie hier nacheinander vorstellen. Die Methode `peoplePickerNavigationControllerDidCancel`: wird aufgerufen, wenn der Anwender den Abbruch-Button antippt. Da die Auswahl abgebrochen wird, müssen wir in dieser Methode auch nur das aufräumen, was wir eingerichtet haben, um eine Auswahl zu empfangen, und den People Picker-Controller verwerfen.

Dies ist der Code für die Methode:

AddressBook/Contacts/Classes/RootViewController.m

```
-(void)peoplePickerNavigationControllerDidCancel:
(ABPeoplePickerNavigationController *)peoplePicker {
    [peoplePicker dismissModalViewControllerAnimated:YES];
    [peoplePicker autorelease];
}
```

Hier verwerfen wir lediglich den View-Controller und räumen den genutzten Speicher auf. Da es sich um eine Abbruchoperation handelt, wollen wir keine Daten verändern.

Als Nächstes müssen wir die Methode implementieren, die aufgerufen wird, wenn ein Anwender einen Kontakt auswählt. Dabei handelt es sich um die Methode `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:`. Wollen wir die Detailinformationen des Kontakts sehen, muss YES zurückgegeben werden, wollen wir das nicht, geben wir NO zurück. Im letzteren Fall wollen wir meistens auch den Controller verwerfen. In diesem Beispiel wollen wir die Details nicht sehen, daher geben wir NO zurück und rufen die gleichen Methoden auf, die wir auch beim Abbruch genutzt haben. Zudem wollen wir den Namen und die Kennung des Kontakts holen und beide in den entsprechenden Arrays ablegen. Hier ist der Code dazu:

AddressBook/Contacts/Classes/RootViewController.m

```
-(BOOL)peoplePickerNavigationController:
(ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person {
    NSString *name = (NSString *)ABRecordCopyCompositeName(person);
    [self.contactNames addObject:name];
    [self.contactIDs addObject:
        [NSNumber numberWithInt:ABRecordGetRecordID(person)]];
    [peoplePicker dismissModalViewControllerAnimated:YES];
    [peoplePicker autorelease];
    NSIndexPath *path = [NSIndexPath indexPathForRow:self.contactIDs.count -1
        inSection:0];
```

```

        [self.tableView insertRowsAtIndexPaths:
         [NSArray arrayWithObject:path]
         withRowAnimation:UITableViewRowAnimationRight];
        return NO;
    }

```

Wir können den Rückgabewert von `ABRecordCopyCompositeName()` in einen `NSString` casten, denn `CFStringRef` und `NSString` haben das gleiche Speicherlayout. Dieses Konzept ist die *gebührenfreie* Umwandlung zwischen Core Foundation und Foundation. Sie finden mehr darüber in Apples *Carbon-Cocoa Integration Guide* [App06b].

Und schließlich müssen wir `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:` implementieren. Bei unserem Beispiel werden wir in dieser Methode nichts anderes tun, als `NO` zurückzugeben. Mit `NO` als Ergebnis von `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:` und dem Verwerfen des Controllers wird diese Methode niemals aufgerufen werden. Wenn Sie sie implementieren wollen, würden Sie in etwa das Gleiche tun, das Sie auch in der „should continue“-Methode getan haben. Nehmen Sie die Eigenschaft und kopieren Sie ihren Wert in Ihre eigenen Datenstrukturen, um sie in Ihrer App einzusetzen. Der Rückgabewert dieser Methode weist das Adressbuch-Framework an, entweder die Standard-Action durchzuführen (zum Beispiel eine E-Mail zu erzeugen oder eine Telefonnummer anzuwählen) oder nichts zu tun. Geben Sie `YES` zurück, wird die Standard-Action ausgeführt, bei `NO` passiert nichts.

Es gibt eine letzte Methodenimplementierung, bevor wir uns darüber unterhalten können, wie wir unseren People Picker-Controller angezeigt bekommen. Statt die Arrays für die Kontakt-IDs und -Namen in einer Init-Methode einzurichten, implementieren wir die Get-Methoden so, dass sie die Arrays erst bei Bedarf instantiieren:

AddressBook/Contacts/Classes/RootViewController.m

```

-(NSMutableArray *)contactNames {
    if(nil == contactNames) {
        contactNames = [[NSMutableArray alloc] init];
    }
    return contactNames;
}

```

Der entsprechende Code ist auch in der Methode `contactIDs` implementiert.

23.3 Den People Picker erzeugen und konfigurieren

Nachdem wir die Delegate-Methoden implementiert haben, müssen wir jetzt den Person Picker erstellen und anzeigen. Dazu wollen wir einen +-Button auf der rechten Seite der Navigation Bar ergänzen (mehr Informationen zur Navigationsleiste finden Sie in Kapitel 6, *Navigation* ab Seite 115). Wir setzen die Eigenschaft `rightBarButtonItem` des `navigationItem` auf ein neues Bar Button Item. Dies ist der Code dazu:

AddressBook/Contacts/Classes/RootViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.rightBarButtonItem = [[[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self
        action:@selector(add)]
        autorelease];

    self.title = @"Selected Contacts" ;
}
```

Da wir das Systemelement `UIBarButtonItemSystemItemAdd` genutzt haben, erscheint auf der rechten Seite der Navigationsleiste ein +-Button. Zudem wollen wir den Titel dieses View-Controller auf „Selektierte Kontakte“ setzen. Und schließlich sollte `add` den People Picker anzeigen. Hier ist der Code für diese Methode:

AddressBook/Contacts/Classes/RootViewController.m

```
- (void)add {
    ABPeoplePickerNavigationController *peoplePicker =
        [[ABPeoplePickerNavigationController alloc] init];
    peoplePicker.peoplePickerDelegate =
        self; [self presentViewController:peoplePicker animated:YES];
}
```

In dieser Methode erzeugen wir eine Instanz des People Picker, setzen dessen `peoplePickerDelegate` auf `self`, so dass wir die Callbacks erhalten, und zeigen ihn dann als modalen View-Controller an.

23.4 Person-Controller

Als Nächstes soll unsere Anwendung zu den Details eines Kontakts navigieren, wenn der Anwender eine Zeile in der Tabellen-View auswählt. Dazu werden wir eine Instanz des `ABPersonViewController` auf den Stack des Navigations-Controller schieben.

Wie vielleicht schon erwartet, erzeugen wir erneut den Person View-Controller, setzen dessen Delegate auf self und schieben ihn dann auf den Stack des Navigations-Controllers. Wir beginnen mit der Implementierung der einen Methode, die vom Protokoll `ABPersonViewControllerDelegate` gefordert wird. Dies ist der Code:

AddressBook/Contacts/Classes/RootViewController.m

```
- (BOOL)personViewController:
    (ABPersonViewController *)personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)valueID {
    ABPropertyType type = ABPersonGetTypeOfProperty(property);
    switch (type) {
        case kABStringPropertyType: {
            NSString *value = (NSString *)ABRecordCopyValue(person, property);
            NSLog(@"property value = %@", value);
            [value release];
            break;
        }
        case kABMultiStringPropertyType: {
            ABMutableMultiValueRef multi = ABRecordCopyValue(person, property);
            CFIndex index = ABMultiValueGetIndexForIdentifier(multi, valueID);
            CFStringRef value = ABMultiValueCopyValueAtIndex(multi, index);
            NSLog(@"value = %@", (NSString *)value);
            CFRelease(multi);
            CFRelease(value);
            break;
        }
        default:
            break;
    }
    [self.navigationController pushViewControllerAnimated:YES];
    return NO;
}
```

Diese Methode wird aufgerufen, wenn der Anwender eines der Attribute aus der Person-View auswählt. Der Code zeigt einige der API-Aufrufe, die wir für die Adressbuchtypen nutzen können. Als Erstes holen wir den Typ der gewählten Eigenschaft und arbeiten dann abhängig davon weiter. Hier berücksichtigen wir nur zwei Typen, aber es gibt noch viele weitere, aus denen wir wählen können. Wir protokollieren den Wert nur an der Konsole, aber in Ihrem eigenen Code werden Sie sicherlich etwas Spannenderes damit anstellen wollen.

Als Nächstes müssen wir uns anschauen, wie wir den Person View-Controller aufrufen. Wenn der Anwender einen der Kontakte anwählt, die wir in unserer Tabellen-View anzeigen, wollen wir zum Person View-Controller navigieren. Also müssen wir die Methode `tableView:did-`

`SelectRowAtIndexPath:` implementieren, um diesen Controller zu erzeugen und anzuzeigen. Das geschieht mit folgendem Code:

AddressBook/Contacts/Classes/RootViewController.m

```
-(void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    ABAddressBookRef addressBook = ABAddressBookCreate();
    ABRecordRef person = ABAddressBookGetPersonWithRecordID(addressBook,
        [[self.contactIDs objectAtIndex:indexPath.row] intValue]);
    ABPersonViewController *pvc = [[ABPersonViewController alloc] init];
    pvc.personViewDelegate = self;
    pvc.displayedPerson = person;
    [self.navigationController pushViewController:pvc animated:YES];
    CFRelease(addressBook);
    [pvc autorelease];
}
```

In dieser Methode erzeugen wir eine `ABAddressBookRef` und verwenden sie, um die Person zu finden, die durch unsere ID repräsentiert wird. Dann übergeben wir diese Person an die Eigenschaft `displayedPerson` des Person View-Controller, setzen `personViewDelegate` auf `self` und schieben den View-Controller auf den Navigations-Stack. Da wir das person-Objekt nicht kopiert haben, müssen wir es auch nicht freigeben.

Das Adressbuchobjekt ist im Prinzip eine Verbindung zur Adressbuchdatenbank. Müssen Sie neue Einträge erzeugen, beginnen Sie mit einer neuen Instanz von `ABAddressBookRef` und nutzen dann die anderen Adressbuchfunktionen, zum Beispiel `ABPersonCreate()` und `ABAddressBookAddRecord()`. Sie müssen sicherstellen, dass der Zugriff auf eine Instanz von `ABAddressBookRef` immer nur aus einem Thread geschieht, da es ansonsten zu einem undefinierten Verhalten kommen kann.

Nachdem nun unsere Person-View angezeigt wird, können wir einen Wert auswählen. Dieser wird an der Konsole protokolliert, und anschließend wird der Person View-Controller wieder verworfen.

23.5 Neue Kontakte hinzufügen

Wie schon erwähnt, ist die Schnittstelle, über die wir mit der Kontakte-Datenbank kommunizieren, die `ABAddressBookRef`. Ist diese Verbindung einmal hergestellt, können wir Adressbucheinträge hinzufügen, entfernen und bearbeiten. Jeder Thread Ihrer Anwendung, der Zugriff auf die Datenbank haben möchten, braucht auch eine eigene Instanz von `ABAddressBookRef`.

Lassen Sie uns unserem View-Controller eine Methode hinzufügen, mit der die Datenbank um einen Einzelpersoneneintrag ergänzt wird. Der Eintrag wird hartkodiert mit ein paar Daten versehen sein. In „richtigen“ Anwendungen werden Sie die Kontaktinformationen von einem Webservice oder auf anderen Wegen erhalten. Haben Sie die Daten, werden der Prozess und die API aber die gleichen sein.

In unserem Beispiel werden wir ein neues person-Objekt erzeugen, Vor- und Nachname setzen, zwei Telefonnummern hinzufügen und die Privatadresse angeben. Dann werden wir den Personendatensatz der Datenbank hinzufügen und den genutzten Speicher wieder freigeben. Der Prozess hat keine Haken und Ösen, aber wenn Sie noch nie mit C-Code gearbeitet haben, können diese API-Aufrufe etwas einschüchternd wirken. Behalten Sie dabei also zwei Dinge im Hinterkopf: Es gibt eine Einführung zur Arbeit mit der Core Foundation in Abschnitt 17.1, *Prozedurale C-APIs verwenden*, auf Seite 388, und auch wenn das wie eine ganze Menge Code aussieht, geschehen lediglich die fünf beschriebenen Dinge. Lassen Sie uns den Code nun Schritt für Schritt durchgehen. (Um den vollständigen Code auf einmal zu sehen, holen Sie sich das gesamte Projekt und schauen sich die Methode `makeNewEntry` in unserem `RootViewController` an.) Dies ist der Code, mit dem das Adressbuch und der Eintrag erzeugt werden:

AddressBook/Contacts/Classes/RootViewController.m

```
ABAddressBookRef addressBook = ABAddressBookCreate();
ABRecordRef person = ABPersonCreate();
CFErrorRef error = NULL;
```

Trotz des Namens erzeugt ein Aufruf von `ABAddressBookCreate()` keine neue Datenbank, sondern es wird nur eine Verbindung zur bestehenden, gemeinsam genutzten Datenbank geöffnet. `ABPersonCreate()` erzeugt allerdings tatsächlich ein neues Objekt vom Typ `person`, dem wir dann alle Daten hinzufügen werden, um es schließlich unserer Verbindung zur Datenbank zu übergeben. Als nächstes setzen wir den Vor- und Nachnamen unseres neu erzeugten `person`-Objekts:

AddressBook/Contacts/Classes/RootViewController.m

```
// Eigenschaften für Vor- und Nachnamen setzen
ABRecordSetValue(person, kABPersonFirstNameProperty,
                  CFSTR("Wonder" ), &error);
ABRecordSetValue(person, kABPersonLastNameProperty,
                  CFSTR("Widget" ), &error);
if(NULL != error) {
    NSLog(@"an error occurred" );
}
```


Hier nutzen wir über das Makro CFSTR() zwei Core Foundation String-Objekte, um Vor- und Nachnamen zu setzen. Denken Sie daran, dass sich Core Foundation-Objekte gebührenfrei in ihre Foundation-Äquivalente umwandeln lassen. Wenn Sie also lieber NSStrings nutzen, können Sie das über folgenden Cast erreichen:

```
ABRecordSetValue(person, kABPersonFirstNameProperty,
                  (CFStringRef)@"Wonder" , &error);
```

Anschließend fügen wir zwei Telefonnummern hinzu. Telefonnummern bestehen im Adressbuchschemata aus mehreren Werten, daher ist der Code ein bisschen aufwendiger:

AddressBook/Contacts/Classes/RootViewController.m

```
// Telefonnummer
ABMutableMultiValueRef multi =
ABMultiValueCreateMutable(kABMultiStringPropertyType);
ABMultiValueAddValueAndLabel(multi, CFSTR("0123) 4567654" ),
                             kABPersonPhoneMobileLabel, NULL);
ABMultiValueAddValueAndLabel(multi, CFSTR("0123) 8909890" ),
                             kABPersonPhoneWorkFAXLabel, NULL);

// Telefonnummern zu person hinzufügen
ABRecordSetValue(person, kABPersonPhoneProperty, multi, &error);
CFRelease(multi);
if(NULL != error) {
    NSLog(@"an error occurred" );
}
```

Statt die Telefonnummern direkt in das person-Objekt zu schreiben, erzeugen wir zunächst ein Multivalue-Objekt und fügen diesem Objekt die Nummern hinzu. Dann übergeben wir das Multivalue-Objekt an das person-Objekt. Beachten Sie, dass wir hier nur eine minimale Fehlerprüfung vornehmen. In einer ordentlichen Anwendung müssen Sie bessere Prüfungen durchführen.

Das Telefonnummern-Multivalue-Objekt enthält Strings, das Adress-Multivalue-Objekt nutzt ein Dictionary:

AddressBook/Contacts/Classes/RootViewController.m

```
// Privatadresse
CFStringRef keys[4] = {kABPersonAddressStreetKey,
                      kABPersonAddressCityKey,
                      kABPersonAddressZIPKey};
CFStringRef values[4] = {CFSTR("Hauptstr. 22" ),
                        CFSTR("München" ),
                        CFSTR("80424" )};

CFDictionaryRef data =
    CFDictionaryCreate(NULL,
                      (void *)keys,
```

```

        (void *)values,
        3,
        &kCFCopyStringDictionaryKeyCallBacks,
        &kCFTypeDictionaryValueCallBacks);
multi = ABMultiValueCreateMutable(kABDictionaryPropertyType);
ABMultiValueAddValueAndLabel(multi, data, kABHomeLabel, NULL);

// Adresse zu person hinzufügen
ABRecordSetValue(person, kABPersonAddressProperty, multi, &error);
CFRelease(multi);

```

Auf den ersten Blick sieht dieser Code viel komplizierter aus als der bei der Telefonnummer. Aber wenn Sie ihn sich genauer anschauen, sehen Sie, dass eigentlich das Gleiche passiert. Anstatt ein String-Objekt mit `CFSTR()` zu erzeugen, erstellen wir ein Dictionary. Das führt zu mehr Code, aber konzeptionell ist es weiterhin nur ein Objekt. Nach dem Erstellen des Dictionary fügen wir es einer neuen Multivalued-Eigenschaft hinzu. Danach tragen wir diese Eigenschaft in unser `person`-Objekt mit der Adresseigenschaft ein.

Nun haben wir alle Daten im `person`-Objekt. Wir können es dem Adressbuch hinzufügen, die Daten sichern und danach aufräumen.

Dies ist der Code:

AddressBook/Contacts/Classes/RootViewController.m

```

// person-Datensatz hinzufügen, sichern und aufräumen
ABAddressBookAddRecord(addressBook, person, &error);
ABAddressBookSave(addressBook, &error);
CFRelease(person);
CFRelease(addressBook);

```

Viele Menschen verwirrt es, sich bei der Arbeit mit dem Adressbuch mit der C-API von Core Foundation herumschlagen zu müssen. Seien auch Sie daher nicht enttäuscht, wenn Sie nicht auf Anhieb alles verstehen. Es dauert ein bisschen, sich damit vertraut zu machen. Aber mit ein wenig Erfahrung werden Sie schon klarkommen. Wenn Sie eine Einführung in die Programmiersprache C brauchen, ist *A Book on C* [KP88] ein guter Ausgangspunkt.

Herzlichen Glückwunsch, Sie haben das Adressbuch erfolgreich zu einer Anwendung hinzugefügt. Mit diesem Wissen können Sie Ihre Anwendung viel persönlicher gestalten.

Kapitel 24

iPhone Location-API

Durch die Verfügbarkeit Ihrer geografischen Position auf dem iPhone ändern sich Ihre Möglichkeiten beim Reisen. Wenn Sie in einer Großstadt unterwegs sind, hoffen Sie vermutlich, dass irgendwo in der Nähe ein tolles Museum ist. Befinden Sie sich dagegen irgendwo in der Pampa, ist wohl kein Museum in der Nähe, aber vielleicht ein toller Ort zum Fotografieren.

Der Core Location Service gehört zu jedem iPhone, daher weiß unser iPhone, wo wir sind, und kann uns dabei helfen, die besten Museen oder die schönsten Fleckchen zu finden. Statt in einem Buch zu suchen, in dem „tolle Orte“ vorgestellt werden, haben Sie ein Gerät, das sagen kann: „Sie sind hier“ und dann die tollen Orte in Ihrer Nähe für Sie findet.

Wie bei den anderen iPhone-Features ist es wichtig, nicht darüber nachzudenken, wie man am meisten aus Core Location herausholt. Haben Sie einmal entschieden, was Sie mit Location machen wollen, werden Sie feststellen, dass es erstaunlich einfach ist, den Core Location Service auf dem iPhone zu nutzen. Mit ein paar Zeilen Code können wir die Hardware zur Positionsbestimmung ansprechen und Informationen über die Position des Geräts aufrufen. Lassen Sie uns also loslegen.

24.1 Wissen, wo man ist

Sie werden den `CLLocationManager` nutzen, um die Location Services an- und auszuschalten und das für Sie notwendige Service-Level festzulegen. Lassen Sie uns eine Beispielanwendung bauen, die einfach die von Core Location zurückgegebene Position ausgibt.

Erstellen Sie ein neues, View-basiertes Projekt mit dem Namen `LocationDisplay`. Sie müssen dort das Core Location Framework hinzufügen.

Location Manager-Delegate

In einer normalen Anwendung ist das Delegate-Objekt häufig der Controller, der mit dem Modell interagiert, das die Positionsdaten der Anwendung verwaltet. In diesem Beispiel haben wir kein eigenes Modell, daher wird einfach unser View-Controller der Delegate sein. In einer komplexeren Anwendung werden Sie auch etwas Komplexeres damit machen wollen – zum Beispiel einen gewichteten gleitenden Mittelwert der Positionsdaten mit der Zeit. Der Controller, der den gewichteten Durchschnitt der Daten verwaltet, ist da natürlich ein idealer Kandidat für einen Delegate.

Dazu wählen Sie das Target für Ihr Projekt aus und tippen -. Am unteren Ende des Tabs `GENERAL` klicken Sie auf den `+`-Button und wählen das Core Location Framework.

In der Datei `LocationDisplayViewController.h` deklarieren Sie eine Eigenschaft mit dem Namen `locationManager` und dem Typ `CLLocationManager`. Sie müssen zudem den Core Location-Header importieren und die Deklaration treffen, dass diese Klasse das Protokoll `CLLocationManagerDelegate` implementiert.

Im nächsten Schritt erzeugen wir ein neues `CLLocationManager`-Objekt in der Methode `viewDidLoad` des `LocationDisplayViewController` und setzen den Delegate auf den View-Controller. Der Delegate muss kein View-Controller sein, wir könnten jedes Objekt nutzen, solange es das Delegate-Protokoll implementiert, aber der Controller ist häufig einfach praktisch. Nach dem Erzeugen des Managers und dem Setzen seines Delegate rufen wir die Methode `startUpdatingLocation` auf, um die Hardware zur Positionsbestimmung warmlaufen zu lassen und zu schauen, wo sich das iPhone befindet. Hier ist der Code, der notwendig ist, um über Positions-Updates informiert zu werden:

Location/LocationDisplay/Classes/LocationDisplayViewController.m

```

- (void)viewDidLoad {
    [super viewDidLoad];
    self.locationManager = [[[CLLocationManager alloc] init] autorelease];
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
}

```

Nachdem wir den Location Service gestartet und seinen Delegate gesetzt haben, werden wir sämtliche Positions-Updates erhalten, sobald die Delegate-Methode `locationManager:didUpdateToLocation:fromLocation:` aufgerufen wird. Wir könnten diese Methode ratzfatz implementieren, aber zuerst wollen wir ein UI erstellen, das uns die von der Hardware gefundene Position anzeigt. Dafür werden wir zwei Labels und zwei Textfelder nutzen, jeweils eines für den Längen- und den Breitengrad. Das UI ist in Abbildung 24.1 zu sehen.



Abbildung 24.1: Positionskoordinaten

In der Methode `locationManager:didUpdateToLocation:fromLocation:` brauchen wir Zugriff auf diese beiden Labels, daher sollten Sie nicht vergessen, zwei `IBOutlet`s der Header-Datei des View-Controller hinzuzufügen und sie im IB zu verknüpfen. So können wir den text der Textfelder ändern, wenn wir Positions-Updates vom Location Manager bekommen.

Als Nächstes müssen wir uns die Delegate-Methode anschauen, die wir implementieren, um über Positions-Updates informiert zu werden. Dies ist der Code:

Location/LocationDisplay/Classes/LocationDisplayViewController.m

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation {
    latitudeTextField.text = [NSString stringWithFormat:@"%3.5f" ,
        newLocation.coordinate.latitude];
    longitudeTextField.text = [NSString stringWithFormat:@"%3.5f" ,
        newLocation.coordinate.longitude]; }
```

Diese Methode wird vom CLLocationManager aufgerufen, den wir in der Methode viewDidLoad gestartet haben. Nach dem Start des Location Manager schickt er diese Methode mindestens einmal an seinen Delegate (und normalerweise sehr häufig, dazu gleich mehr). In dieser Methode konvertieren wir die beiden Komponenten der Koordinaten in Strings und tragen diese Strings in die Textfelder ein, die wir auf unserer Oberfläche anzeigen. Wenn Sie die Anwendung bauen und im Simulator laufen lassen, werden Sie nur eine einzige Position erhalten – die des Apple Campus in Cupertino in Kalifornien. Ein normaler Mac unterstützt keine Positionsbestimmung, daher kann das auch der Simulator nicht. Wenn Sie Ihre eigene Position exakt ermitteln wollen, müssen Sie die Anwendung auf Ihr iPhone übertragen und dort ausführen.

Nachdem wir nun die Grundlagen der Core Location Services kennengelernt haben, wollen wir uns wieder der Methode viewDidLoad zuwenden und an den Service-Parametern drehen, um die Location Services für unsere Anwendung zu optimieren.

Parameter für den Core Location Service

Wie bei jedem anderen Aspekt der iPhone-Software kostet der Location Service Strom. Wir müssen sorgfältig darauf achten, wie wir den Service konfigurieren, um sicherzustellen, dass wir so wenig Strom wie möglich verwenden und trotzdem die Anforderungen unserer Anwendung erfüllen.

Es gibt zwei Parameter für den Location Manager, die wir nutzen können, um festzulegen, wie viel Strom für uns verbraucht wird. Der erste heißt `desiredAccuracy`; er teilt dem Location Manager mit, wie genau wir die Ergebnisse haben wollen. Es gibt fünf Optionen: von `kCLLocationAccuracyBest` bis `kCLLocationAccuracyThreeKilometers`. Je genauer wir

die Ergebnisse brauchen, desto mehr Strom wird verbraucht. Überlegen Sie also gut, welchen Wert Sie nutzen. Wenn Ihre Anwendung den Leuten dabei helfen soll, das nächste Café zu finden, brauchen Sie eine höhere Genauigkeit, als wenn es nur um den nächsten Nationalpark geht. Nutzen Sie also den größten Wert, der sich in Ihrer Anwendung noch sinnvoll nutzen lässt.

Die zweite Eigenschaft ist `distanceFilter`. Diese ermöglicht uns, die Häufigkeit der Updates festzulegen, die wir vom Location Service erhalten. Wollen Sie in Echtzeit Richtungsanweisungen in einer Stadt geben, muss es ein kleiner Wert sein (zum Beispiel 5 oder 10 Meter). Geht es auch etwas weniger genau, zum Beispiel bei der Information, wie weit Sie mit Ihrem Fahrrad gefahren sind, können die Positions-Updates auch im 100-Meter-Abstand gemeldet werden. Denken Sie daran, auch hier wird wie bei der Eigenschaft `desiredAccuracy` bei mehr Updates auch mehr Strom verbraucht.



Joe fragt...

Wie funktioniert das mit der Positionsbestimmung eigentlich?

Es gibt drei Komponenten beim Location Service auf einem iPhone, beim iPod touch gibt es nur eine. Die Wifi-basierten Location Services gibt es auf beiden Geräten. Dabei wird eine globale Datenbank abgefragt, die die MAC-Adressen von WLAN-Routern auf eine Geoposition abbildet. Das iPhone besitzt zwei weitere Möglichkeiten, den Standort eines Geräts zu bestimmen. Der erste geschieht über die Funkzellen der Mobilfunknetzbetreiber. Viele Zellen (wenn nicht sogar alle) haben eine bekannte Geoposition, die sie auch an interessierte Geräte herausgeben. Über diese Positionen kann das iPhone seine Position basierend auf den Funkzellen der Umgebung per Triangulation bestimmen. Und schließlich besitzt fast jedes iPhone einen integrierten GPS-Empfänger, der seine Position per Triangulation aus den Signalen von drei oder vier GPS-Satelliten ermittelt.

Von diesen drei Möglichkeiten der Positionsbestimmung kostet GPS am meisten Strom, aber als Client der API haben wir keine Möglichkeit, anzugeben, welche Hardware wir nutzen wollen und welche nicht. Unsere Schnittstelle sind einzig die Eigenschaften `distanceFilter` und `desiredAccuracy`.

Das Beste, was Sie bezüglich des Stromhungers machen können, ist bei den Location Services natürlich, sie ganz abzuschalten. Wenn Ihre Anwendung nur selten eine neue Positionsbestimmung braucht, sollten Sie `stopUpdatingLocation` schicken, wenn Sie eine Position erhalten haben, die genau genug ist. Damit kann das iPhone OS alle Hardware-Bestandteile abschalten, die damit zu tun haben. Ein Chip, der Strom sparen soll, ist einer, der abgeschaltet ist.

Noch etwas zum Location Manager – der Anwender kann die Services zur Positionsbestimmung in den Einstellungen deaktivieren. Sie sollten also prüfen, ob die Location Services abgeschaltet sind, bevor Sie `startUpdatingLocation` aufrufen. Das iPhone OS wird den Anwender fragen, ob er die Location Services wieder anschalten will. Kann Ihre App mit abgeschalteten Location Services arbeiten, könnte der Anwender verwirrt sein. Wenn Ihre App hingegen nur mit eingeschalteter Positionsbestimmung funktioniert, wird der Benutzer vielleicht dankbar sein, dass er die Services aus Ihrer Anwendung heraus wieder anschalten kann.

Wie immer prüfen Sie das Verhalten der Anwendung am besten in allen möglichen Fällen, um sicherzustellen, dass die Benutzer eine sinnvolle Oberfläche zur Verfügung haben.

24.2 Positions-Updates

Nachdem wir nun eine Anwendung haben, die vom Location Service aktualisiert wird, wollen wir uns die Informationen anschauen, die wir neben den Längen- und Breitengraden von diesen Updates erhalten können.

Wenn Sie `locationManager:didUpdateToLocation:fromLocation:` implementieren, erhalten Sie nicht nur die aktuell beste bekannte Position, sondern auch die vorherige bekannte Position. Beim ersten Aufruf dieser Methode hat `oldLocation` den Wert `nil`, aber bei allen weiteren Aufrufen erhalten Sie den alten und den neuen Wert. Die alte Position kann nützlich sein, wenn Sie eine einfache Form der Mittelung durchführen wollen. Braucht Ihre App sehr genaue Positionswerte, werden Sie das zumindest teilweise vornehmen wollen. Die Positionsobjekte, die Sie hier erhalten, sollen Thema dieses Abschnitts sein.

Genauigkeit

Sie haben erfahren, wie Sie dem Core Location Service mitteilen, welche Genauigkeit Sie haben wollen, aber nur weil wir eine bestimmte Genauigkeit anfordern, bedeutet das nicht, dass wir sie auch erhalten. Stattdessen wird uns bei jedem Update vom Service die Genauigkeit der gemeldeten Position mitgeteilt. Bestimmt das System die Genauigkeit seiner Positionsangabe, berücksichtigt es dabei die Komponenten, die es zum Ermitteln der Position genutzt hat, und noch eine Reihe weiterer Faktoren. Wir sollten uns immer die Genauigkeit anschauen, um sicherzustellen, dass sie sich im Rahmen dessen bewegt, was für unsere App noch sinnvoll ist.

Es gibt in den Objekten des Typs `CLLocation` zwei Felder zur Genauigkeit. Das erste hat den Namen `horizontalAccuracy`, und wir erfahren darüber, wie sicher sich das System bezüglich des Felds `coordinate` für das Positionsobjekt ist (also die Längen- und Breitenangabe). Das zweite ist die `verticalAccuracy`, die die Genauigkeit des Systems bezüglich der gemeldeten Höhe `altitude` angibt. Beide Eigenschaften liefern einen negativen Wert, wenn das System keine Genauigkeit angeben kann.

Der Location Service ist nur so gut wie die Daten, die er erhalten kann. Befindet sich die MAC-Adresse des WLAN-Netzwerks, auf das das Gerät Zugriff hat, nicht in der Datenbank, oder liefern die Funkzellen in der Umgebung keine Höhendaten, kann das Gerät nicht wissen, wo es ist. In dem Fall müssen wir cleverer vorgehen, um die Daten zu interpretieren. Wie wir gesehen haben, erfahren wir durch die Eigenschaft `verticalAccuracy`, ob wir dem Höhenwert trauen können, und durch `horizontalAccuracy`, wie es bei `coordinate` aussieht. Wenn einer dieser Genauigkeitswerte negativ ist, sollten Sie dem entsprechenden Wert auch nicht trauen. Selbst wenn die Genauigkeitswerte positiv sind, sind sie gelegentlich ungenau. Wenn man sich überlegt, wie die Positionsbestimmung funktioniert, wird auch klar, warum das so ist. Wird nur eine Funkzelle gefunden (ignorieren wir GPS und WLAN hier einmal), kann das Gerät keine Triangulation durchführen und weiß nur, dass es sich im Bereich der einen Funkzelle befinden muss. Selbst wenn zwei Funkzellen erreichbar sind, können die Werte schon ganz schön danebenliegen.

Es ist wichtig, die Genauigkeitswerte des Location Service nicht aus dem Auge zu verlieren, wenn Sie die Positionsinformationen nutzen. Ein Mitteln der Werte über die Zeit hilft, Abweichungen zu dämpfen. Es gibt eine Reihe unterschiedlicher Ansätze, die Sie dabei verfolgen können. Experimentieren Sie damit, um herauszufinden, welcher bei Ihrer Anwendung am besten funktioniert.

Abstand

Ein Positionsobjekt kann den Abstand zwischen sich selbst und einem anderen Positionsobjekt ermitteln, indem es `getDistanceFrom:` aufruft. Diese Berechnung kann in einer ganzen Reihe von Anwendungen genutzt werden.

Lassen Sie uns eine Beispielanwendung bauen, die den gesamten Weg und die durchschnittliche Geschwindigkeit berechnet und auf dem UI ausgibt. Die Benutzeroberfläche mit den entsprechenden Werten sehen Sie in Abbildung 24.2.

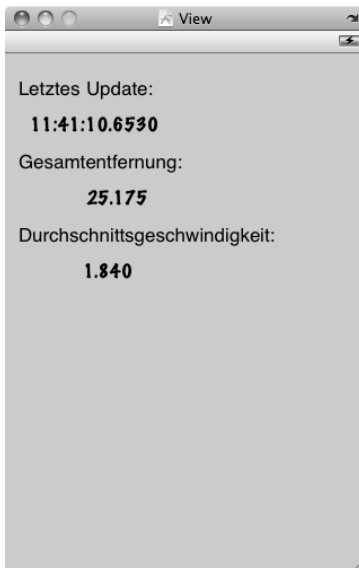


Abbildung 24.2: Abstands-UI

Der Code hinter diesem Beispiel ähnelt stark dem, den wir vorher schon gesehen haben. In `viewDidLoad` starten wir einen `CLLocationManager` und lassen ihn unsere Position bestimmen. Allerdings geben wir dieses Mal etwas genauer an, was wir von ihm erwarten:

Location/Distance/Classes/DistanceViewController.m

```
-(void)viewDidLoad {
    [super viewDidLoad];
    self.locationManager = [[[CLLocationManager alloc] init] autorelease];
    self.locationManager.distanceFilter = 10.0f;
    self.locationManager.delegate = self;
    self.locationManager.desiredAccuracy = kCLLocationAccuracyNearestTenMeters;
    [self.locationManager startUpdatingLocation];
    self.locations = [NSMutableArray arrayWithCapacity:32]; }
```

In dieser Methode setzen wir die Eigenschaft `distanceFilter` auf 10 Meter, weil wir nur Abstandsänderungen von mehr als 10 Metern erhalten wollen. Dann setzen wir die `desiredAccuracy` über die Konstante `kCLLocationAccuracyNearestTenMeters` ebenfalls auf 10 Meter. Diese Kombination der Service-Parameter informiert die Hardware darüber, dass wir wirklich nur eine Genauigkeit von 10 Metern haben wollen. Zudem richten wir ein veränderbares Array ein, das die Positions-Updates speichert. Als Nächstes folgt die Delegate-Methode, die die Positions-Updates verarbeitet. Dies ist der Code:

Location/Distance/Classes/DistanceViewController.m

```
-(void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
        fromLocation:(CLLocation *)oldLocation {
    // Wenn wir eine gültige Position innerhalb von 20 Metern haben,
    // stoppen wir den Service und warten 60 Sekunden
    if(newLocation.horizontalAccuracy > 0.0f &&
        newLocation.horizontalAccuracy < 20.0f) {
        if(self.locations.count > 3) {
            [self.locationManager stopUpdatingLocation];
            [self.locationManager performSelector:@selector(startUpdatingLocation)
                withObject:nil
                afterDelay:60.0f];
        }
        [self.locations addObject:newLocation];
        [self updateDisplay];
    }
}
```

In dieser Methode schauen wir nach einer gültigen Genauigkeit (`horizontalAccuracy` ist also größer als null) und prüfen, ob sie besser als 20 Meter ist. So ist sichergestellt, dass wir nur gültige Daten in unserem Array `locations` haben. Dann achten wir darauf, dass wir mindestens drei gültige Positionen besitzen, um dann den Location Manager abzuschalten. Er soll allerdings nach 60 Sekunden wieder hochfahren. Mit drei Werten vor dem Abschalten ist sichergestellt, dass wir genug Daten haben, um unsere Berechnungen durchzuführen. Durch das Abschalten für 60 Sekunden reduzieren wir den Stromverbrauch, haben aber trotzdem gute Werte für unsere App. Schließlich aktualisieren wir die Benutzerschnittstelle mit den Daten, so dass unsere Anwender sehen können, wie weit sie sich bewegt haben. Hier ist der Code für diese Methode:

Location/Distance/Classes/DistanceViewController.m

```

- (void)updateDisplay {
    CLLocationDistance distance = [self totalDistanceTraveled];
    totalDistance.text = [NSString stringWithFormat:@"%5.3f" , distance];

    NSTimeInterval time = [self timeDelta];
    // Keine Division durch null
    if(time == 0.0f) {
        averageSpeed.text = @"0.000" ;
    } else {
        averageSpeed.text = [NSString stringWithFormat:@"%5.3f" ,
                                                                    distance / time];
    }
    NSDateFormatter *inputFormatter =
        [[[NSDateFormatter alloc] init] autorelease];
    [inputFormatter setDateFormat:@"HH:mm:ss.SSSS" ];
    NSDate *date = [(CLLocation *)[self.locations lastObject] timestamp];
    lastUpdate.text = [inputFormatter stringFromDate:date];
}

```

In dieser Methode führen wir nur die Berechnungen durch, die notwendig sind, um die gesamte Wegstrecke und die Zeit zu bestimmen und daraus die Durchschnittsgeschwindigkeit zu ermitteln. Das Timestamp-Label wird auch noch aktualisiert. Vergessen Sie nicht, dass Sie IBOutlet's für die UILabel's brauchen (die zudem im IB verbunden sind), damit das alles wie gewünscht funktioniert.

Jetzt wollen wir uns anschauen, wie die gesamte Wegstrecke berechnet wird:

Location/Distance/Classes/DistanceViewController.m

```

- (CLLocationDistance)totalDistanceTraveled {
    CGFloat totalDistanceTraveled = 0.0f;
    CLLocation *oldLocation = nil;
    for(CLLocation *location in self.locations) {
        if(nil == oldLocation) {
            oldLocation = location;
            continue;
        }
        totalDistanceTraveled += fabs([location getDistanceFrom:oldLocation]);
        oldLocation = location;
    }
    return totalDistanceTraveled;
}

```

In dieser Methode addieren wir die Abstände zwischen der vorherigen und der neuen Position auf. Wir nutzen dabei den absoluten Wert, da wir uns vor- und zurückbewegen können, aber trotzdem unterwegs sind. Nutzen wir nicht den absoluten Wert, kann letztendlich 0 herauskommen, wenn der Anwender 1 km nach Norden und dann 1 km nach Süden gelaufen ist.

Schließlich haben Sie noch die Berechnungen für die Zeitdifferenz:

Location/Distance/Classes/DistanceViewController.m

```
- (NSTimeInterval)timeDelta {
    NSDate *first =
        [(CLLocation *)[self.locations objectAtIndex:0] timestamp];
    NSDate *last =
        [(CLLocation *)[self.locations lastObject] timestamp];
    return [last timeIntervalSince1970] -[first timeIntervalSince1970];
}
```

Diese Methode berechnet einfach den Zeitunterschied zwischen dem ersten und dem letzten Positionsobjekt.

24.3 Kompass

Mit dem iPhone OS 3.0 hat Apple eine Unterstützung für ein Magnetometer implementiert, was im Prinzip nur ein cooles Wort für einen digitalen Kompass ist. Das neue Feature erfordert eines der neuen iPhone 3GS, aber das Tolle ist, dass Sie hier nicht viel Neues lernen müssen. Als Delegate Ihres Location Manager können Sie die Methode `locationManager:didUpdateHeading:` implementieren, um Richtungsinformationen zu erhalten. Dazu müssen Sie Ihren Location Manager mit dem Aufruf der Methode `startUpdatingHeading` nur anweisen, den Kompass auch zu starten.

Lassen Sie uns in unserem Display auch die Richtung anzeigen. Öffnen Sie die Datei `LocationDisplayViewController.xib` und fügen Sie ein Label und ein Textfeld hinzu. Ändern Sie den Text des Labels in „Richtung“. In der Datei `LocationDisplayViewController.h` fügen Sie ein `IBOutlet` für das neue Textfeld hinzu und geben ihm den Namen `headingTextField`. Ergänzen Sie noch die `@synthesize`-Anweisung und sichern Sie alles. Jetzt verbinden Sie im Interface Builder das neue Outlet mit dem neuen Textfeld. Nun können wir den Kompass einschalten. Dies ist der Code aus dem Beispiel für die Anzeige der Position mit dem zusätzlichen Code, der den Kompass in Zeile 6 aktiviert:

Location/LocationDisplay02/Classes/LocationDisplayViewController.m

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.locationManager = [[[CLLocationManager alloc] init] autorelease];
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingLocation];
    [self.locationManager startUpdatingHeading];
}
```

Durch den einfachen Aufruf von `startUpdatingHeading` wird der Kompass aktiviert, und der Location Manager informiert uns über jede Richtungsänderung durch die Methode `locationManager:didUpdateHeading:`. Erhalten wir diesen Aufruf, aktualisieren wir in der Delegate-Methode das Textfeld:

Location/LocationDisplay02/Classes/LocationDisplayViewController.m

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateHeading:(CLHeading *)newHeading {
    headingTextField.text = [NSString stringWithFormat:@"%3.2f" ,
                                                                    newHeading.trueHeading];
}
```

In Ihrer eigenen Anwendung werden Sie mit den Daten vermutlich noch mehr anstellen wollen. So könnten Sie zum Beispiel eine View abhängig von der Ausrichtung unterschiedlich zeichnen.

Das Magnetometer muss gelegentlich kalibriert werden. Erkennt der Location Manager, dass eine Kalibrierung notwendig ist, schickt er dem Delegate eine `locationManagerShouldDisplayHeadingCalibration:`. Geben Sie YES zurück, öffnet sich ein Pop-up, in dem der Anwender gebeten wird, sein Gerät in einer Acht zu bewegen. Geben Sie NO zurück, wird das Pop-up nicht angezeigt. Beachten Sie dann aber, dass die folgenden Richtungs-Updates nicht mehr kalibriert sind.

Bedenken Sie ebenfalls, dass nur ein paar iPhone OS-Geräte einen Kompass unterstützen. Um herauszufinden, ob das aktuelle Gerät einen besitzt, können Sie die Eigenschaft `headingAvailable` Ihres Location Manager auslesen. Ist der Wert YES, besitzt das Gerät einen Kompass. Sie sollten diese Eigenschaft kontrollieren, bevor Sie versuchen, ihn zu nutzen.

Wie Sie sehen können, gibt es eine Reihe von Möglichkeiten, den Location Manager und den Core Location Service zu nutzen. Es ist wichtig, dass jede Anwendung, die diesen Service nutzt, sorgfältig darauf achtet, nicht mehr Strom zu verbrauchen, als wirklich notwendig ist.

Kapitel 25

Map Kit

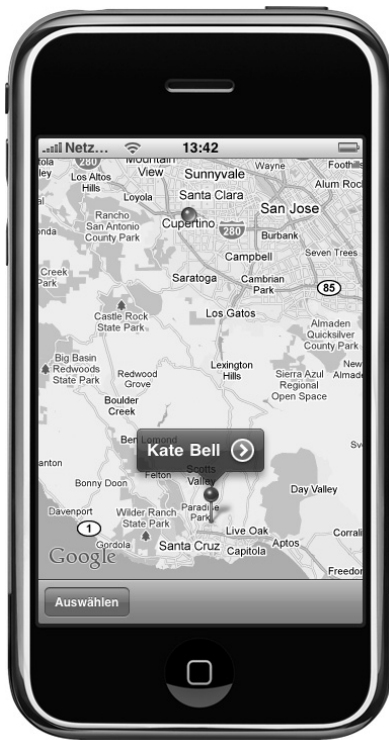
Lokalisierungsdienste sind toll und können unsere Apps um ein paar echt coole Features ergänzen. Weiß unsere App einmal, wo sich das Gerät befindet, ist der nächste logische Schritt meist, diese Position auf einer Karte anzeigen zu lassen. Hier kommt das Map Kit ins Spiel. Damit können wir Positionen auf einer Karte anzeigen. Map Kit stellt die *Tiles* (also die kachelartigen Bilder, die in der Karte genutzt werden) sowie die Möglichkeiten zum Zoomen und Verschieben bereit.

Lassen Sie uns eine Anwendung bauen, die zeigt, was mit dem Map Kit möglich ist.

25.1 Contact Mapper

Contact Mapper ist eine Anwendung, mit der ausgewählte Kontakte auf einer Karte platziert und dann relativ zum aktuellen Standort angezeigt werden können. Nach dem Start zeigt die Karte Ihre Position mit dem netten blauen Button, der für die Kartenanwendung typisch ist. Sie können dann den AUSWÄHLEN-Button antippen, um Ihre Kontakte angezeigt zu bekommen. Nach der Auswahl eines Kontakts wird ein Pin an der Adresse gesetzt, und die Karte zoomt dann so, dass sowohl Ihre Position als auch die des Kontakts angezeigt werden. Der folgende Screenshot wurde von der Anwendung im Simulator gemacht, wobei nur ein einzelner Kontakt hinzugefügt wurde.

Wenn der Anwender auf den Pin klickt, wird das kleine Fensterchen angezeigt (im Screenshot Kate Bell). Klickt man auf den blauen Pfeil, kommt man zur Bearbeitungsseite für den Kontakt. Der blaue Punkt zeigt den aktuellen Standort des Geräts (im Simulator ist das standardmäßig Cupertino in Kalifornien).



Das Map Kit stellt eine Reihe toller Features für diese Anwendung bereit. Der Anwender kann durch Gesten hineinzoomen, einen Finger auf dem Bildschirm bewegen, um die Karte zu verschieben, und sogar „schnippen“, um die Karte ein ganzes Stück auf einmal weiterzuschieben. Alle Tiles von Google Maps werden automatisch geladen und passend gezoomt beziehungsweise skaliert. Map Kit macht das Bauen solcher Anwendungen zu einem Kinderspiel.

25.2 Eine Karte anzeigen

Lassen Sie uns mit dem Aufbau des Contact Mapper beginnen und eine neue View-basierte Anwendung erzeugen. Wir werden hier nicht Schritt für Schritt alle notwendigen Tätigkeiten aufführen, denn es gibt zu viel Map Kit-spezifische Dinge, die wir behandeln müssen. Wenn Sie also ein bisschen Hilfe brauchen, damit die Anwendung funktioniert, holen Sie sich den Code zum Buch und greifen auf den Ordner MapKit zu.

In dieser ersten Version der Anwendung werden wir die Karte anzeigen und die aktuelle Position darstellen. Um Map Kit nutzen zu können, müssen wir die Frameworks `Map Kit.framework` und `CoreLocation.framework` dem Projekt hinzufügen.

Öffnen Sie die Datei `ContactMapperViewController.xib`, so dass wir an der Benutzerschnittstelle arbeiten können.¹ Ist die View nicht schon offen, klicken Sie sie doppelt an. Öffnen Sie die Library ($\text{⌘} - \text{⇧} - \text{L}$) und wählen Sie „Map View“. Sie finden sie in der Gruppe „Data Views“. Hier ein Screenshot der Library mit der ausgewählten Map-View:



Ziehen Sie die Map-View auf die View. Öffnen Sie deren Attributes Inspector ($\text{⌘} - \text{I}$). Legen Sie dort fest, dass die Karte die aktuelle Position des Anwenders anzeigen soll. Wir haben den Kartentyp auf *Map* gesetzt – die anderen Möglichkeiten sind *Satellite* und *Hybrid*. In einer richtigen Anwendung sollten Sie dem Anwender die Möglichkeit geben, zwischen den drei Typen wechseln zu können, sofern es sinnvoll ist. Wir sind im IB erst einmal fertig, speichern Sie also Ihre Arbeit und verbergen Sie das Programm mit $\text{⌘} - \text{H}$.

¹ Um Frameworks hinzuzufügen, wählen Sie das Target „ContactMapper“ (in der Gruppe „Targets“), klicken mit der rechten Maustaste darauf und wählen GET INFO. Im Tab GENERAL klicken Sie auf den +-Button am unteren Ende des Fensters, wählen die beiden Frameworks aus und klicken dann auf den Button ADD.

Jetzt haben wir eigentlich schon eine kartenbasierte Anwendung. Mit BUILD AND RUN wird die Anwendung angezeigt, und Sie sehen einen blauen Punkt an dem Ort, an dem Sie sich befinden. Ist die Karte einmal geladen, können Sie hineinzoomen oder sie verschieben – all das, was Sie von einer Karte auf Ihrem iPhone auch erwarten.

Aber wir wollen ja noch nicht aufhören. Wenn die aktuelle Position gefunden ist, soll die Karte sich dort neu zentrieren und hineinzoomen, so dass man etwas erkennen kann. Die Map-View-Methode `setRegion:animated:` macht genau das, was wir wollen. Der Parameter `region` ist eine C-Struktur, ähnlich wie das `CRect`, das wir zum ersten Mal in Kapitel 19, *Zeichnen in eigenen Views*, auf Seite 415 kennengelernt haben. Die `MKCoordinateRegion` besitzt zwei Elemente, `center` und `span`. Ersteres ist eine `CLLocationCoordinate2D` mit Längen- und Breitengrad des zentralen Punkts der Region. Die Spannweite `span` ist ein `MKCoordinateSpan`, der den Bereich in Längen- und Breitengraden angibt, den die Region umfassen soll. Denken Sie daran, dass der tatsächliche Bereich, der durch den Span definiert wird, davon abhängt, wie weit Sie vom Äquator entfernt sind. Je weiter Sie nach Norden oder Süden kommen, desto enger liegen die Längengrade beieinander.

Wenn die Position des Anwenders aktualisiert wurde, wollen wir die Karte neu darauf zentrieren und den Span auf 0,15 Grad in jede Richtung ändern.

Das Neuzentrieren der Karte geht problemlos, wir erzeugen einfach die neue `MKCoordinateRegion` und teilen der Map-View diese Region mit. Hier sehen Sie den Code dafür:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```
- (void)setCurrentLocation:(CLLocation *)location {
    MKCoordinateRegion region = {{0.0f, 0.0f}, {0.0f, 0.0f}};
    region.center = location.coordinate;
    region.span.longitudeDelta = 0.15f;
    region.span.latitudeDelta = 0.15f;
    [self.mapView setRegion:region animated:YES];
}
```

Wir initialisieren die `region` mit Nullen. Dann setzen wir das Zentrum und den Span unserer Region und teilen der View mit, diese zu verwenden. Setzen wir das Animations-Flag auf YES, wird die Karte die Bewegung zur neuen Region animieren, mit NO wird sie einfach nur dorthin springen. Damit das funktioniert, brauchen wir auch einen Zeiger auf die Map-View – wir müssen also ein `IBOutlet` hinzufügen und die entsprechenden Verbindungen im IB herstellen.

Der interessante Teil dieses Features ist, zu wissen, wann das Update vorgenommen werden soll. Nun, Sie kennen mindestens einen Weg, das zu erreichen. Unser View Controller kann der Delegate eines Location Manager werden und die Karte jedes Mal neu zentrieren, wenn ein Update eintrifft. Mehr Informationen dazu finden Sie in Kapitel 24, *iPhone Location-API*, auf Seite 481. Aber es zeigt sich, dass die Map-View das schon für uns tun kann. Wenn wir den Schalter „Show User Location“ im IB aktivieren oder die Eigenschaft `showsUserLocation` auf YES setzen, schaltet die Map-View den Location Manager an und wird mit der aktuellen Position versorgt. Wir müssen nur noch die passenden Map-View-Delegate-Methoden implementieren. Das werden wir uns gleich anschauen.

25.3 Anmerkungen in einer Karte

Jede Anmerkung in einer Karte besteht aus zwei Elementen. Der Modell-Teil sollte leichtgewichtig sein, so dass Sie viele davon mit einer bestehenden Karte verbinden können und sich nicht um den Speicherverbrauch sorgen müssen. Der andere Teil ist die View. Lassen Sie uns zunächst die Modell-Seite anschauen.

Das Map Kit definiert das Protokoll `MKAnnotation`, besitzt aber keine öffentlichen Implementierungen. Um also der Karte Anmerkungen hinzuzufügen, müssen wir unsere eigene Implementierung dieses Protokolls erstellen. Das Protokoll definiert eine Eigenschaft und zwei optionale Methoden. Die Eigenschaft ist die Position für die Anmerkung, die Methoden sind `title` und `subtitle`. Der Titel wird als Text für das kleine Fensterchen genutzt, wenn der Anwender auf die Anmerkung klickt. Der Subtitel wird in kleinerer Schrift unter dem Titel angezeigt. Dies ist die Header-Datei für unsere Anmerkung:

MapKit/ContactMapper_01/Classes/ContactAnnotation.h

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>
#import <MapKit/MapKit.h>
#import <AddressBook/AddressBook.h>

@interface ContactAnnotation : NSObject <MKAnnotation> {
    CLLocationCoordinate2D _coordinate;
    NSString *_title;
    NSString *_subtitle;
    ABRecordRef _person;
}
```

```

+ (id)annotationWithCoordinate:(CLLocationCoordinate2D)coordinate;
-(id)initWithCoordinate:(CLLocationCoordinate2D)coordinate;

@property (nonatomic, assign) CLLocationCoordinate2D coordinate;
@property (nonatomic, assign) ABRecordRef person;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *subtitle;

@end

```

Da unsere Anmerkung einen unserer Kontakte darstellt, fügen wir für den Kontakt eine Eigenschaft namens `person` hinzu. Wir definieren zudem die Eigenschaften `title` und `subtitle`, statt die im Protokoll definierten Methoden selbst zu definieren. Schließlich haben wir die Methoden `initWithCoordinate:` und `annotationWithCoordinate:`, die das tun, was Sie erwarten: Initialisieren einer `ContactAnnotation` sowie Erzeugen und Zurückgeben einer mit Autorelease versehenen Anmerkung. Hier haben Sie die Implementierung für die `ContactAnnotation`:

MapKit/ContactMapper_01/Classes/ContactAnnotation.m

```

@implementation ContactAnnotation

@synthesize coordinate = _coordinate;
@synthesize title = _title;
@synthesize subtitle = _subtitle;
@synthesize person = _person;

+ (id)annotationWithCoordinate:(CLLocationCoordinate2D)coordinate {
    return [[[self class] alloc]
            initWithCoordinate:coordinate] autorelease]; }

-(id)initWithCoordinate:(CLLocationCoordinate2D)coordinate {
    self = [super init];
    if(nil != self) {
        self.coordinate = coordinate;
    }
    return self; }

@end

```

Da wir unsere Anmerkung nun haben, müssen wir uns anschauen, wie wir sie erzeugen und der Karte hinzufügen können. Erinnern wir uns daran, wie die Anwendung funktionieren soll: Klickt der Anwender auf den Button AUSWÄHLEN, wollen wir den People Picker anzeigen. Nachdem eine Kontaktadresse ausgewählt wurde, wollen wir die Anmerkung hinzufügen. Dazu müssen wir einige Schritte durchführen:

1. Die Frameworks *AddressBook* und *AddressBookUI* dem Projekt hinzufügen.
2. Eine Action-Methode dem View-Controller hinzufügen und sie implementieren, damit sie die Personenauswahl anzeigt.

3. Eine Toolbar am unteren Rand der View hinzufügen und die Größe der Karte anpassen.
4. Den AUSWÄHLEN-Button mit der Action-Methode verbinden.
5. Das Delegate-Protokoll des People Picker implementieren, um die Adresse zu erhalten, wenn sie angeklickt wird.

Lassen Sie uns die Action-Methode anschauen:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```
- (IBAction)choose {
    ABPeoplePickerNavigationController *picker =
    [[ABPeoplePickerNavigationController alloc] init];
    picker.peoplePickerDelegate = self;
    [self presentViewController:picker animated:YES];
    [picker release];
}
```

Nun haben wir die Implementierung und müssen die Deklaration in die Header-Datei einfügen. Dann wechseln wir wieder in den IB, um die Toolbar hinzuzufügen und die Verbindung vom Button zum File's Owner herzustellen.

Mehr Details zum People Picker finden Sie in Kapitel 23, *Adressbuch*, auf Seite 471. Nachdem wir nun der Delegate des People Picker geworden sind, müssen wir das Protokoll der Header-Datei hinzufügen und die Methoden implementieren. Von den drei Methoden werden wir uns nur eine anschauen, nämlich `peoplePickerNavigationController:shouldContinueAfterSelectingPerson:property:identifier:.` Dies ist ihr Code:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```
1 - (BOOL)peoplePickerNavigationController:
2   (ABPeoplePickerNavigationController *)peoplePicker
3     shouldContinueAfterSelectingPerson:(ABRecordRef)person
4                                     property:(ABPropertyID)property
5                                     identifier:(ABMultiValueIdentifier)identifier{
6   if(kABPersonAddressProperty == property) {
7       NSString *fullName = (NSString *)ABRecordCopyCompositeName(person);
8       CLLocationCoordinate2D coordinate = {0.0f, 0.0f};
9       self.newAnnotation =
10         [ContactAnnotation annotationWithCoordinate:coordinate];
11       self.newAnnotation.title = fullName;
12       self.newAnnotation.person = person;
13       [fullName release];
14       ABMultiValueRef addresses =
15         ABRecordCopyValue(person, kABPersonAddressProperty);
16       CFIndex selectedIndex =
17         ABMultiValueGetIndexForIdentifier(addresses, identifier);
```

```

18     CFDictionaryRef address =
19         ABMultiValueCopyValueAtIndex(addresses, selectedAddressIndex);
20     self.newAnnotation.coordinate =
21         [AddressGeocoder locationOfAddress:address];
22     [self dismissModalViewControllerAnimated:YES];
23 }
24 return NO;
25 }

```

Wenn die Kontaktadresse ausgewählt ist, holen wir uns den vollständigen Namen des Kontakts, erzeugen eine Anmerkung, setzen den full-Name als title und die person der Anmerkung auf den gewählten Kontakt. Zudem holen wir uns ab Zeile 14 die Adresse. In Zeile 20 geokodieren wir sie (was wir uns in Abschnitt 25.3, *Anmerkungen in einer Karte*, auf Seite 497 genauer anschauen werden). Nachdem der Anwender eine Adresse ausgewählt hat, werfen wir die Person Picker-View wieder.

Jetzt sind wir fast fertig. Allerdings haben wir die neue Anmerkung zwar schon erzeugt, sie aber noch nicht der Karte hinzugefügt. Das könnten wir auch in der Delegate-Methode des People Picker erledigen. Allerdings erhalten wir eine viel schönere Animation, wenn wir den Code in der Methode `viewDidAppear:` unterbringen:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    if(nil != self.newAnnotation) {
        [self.mapView addAnnotation:self.newAnnotation];
        self.newAnnotation = nil;
    }
    if(self.mapView.annotations.count > 1) {
        [self recenterMap];
    }
}

```

Erst prüfen wir, ob wir eine neue Anmerkung hinzuzufügen haben. Ist das der Fall, tragen wir sie ein und setzen `newAnnotation` auf `nil`. Gibt es mehr als eine Anmerkung, zentrieren wir die Karte durch den Aufruf von `recenterMap` neu. Das geht einfach – wir erzeugen eine neue Region mit einem `span`-Bereich, der alle Anmerkungen abdeckt. Dies ist der Code dazu:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```

- (void)recenterMap {
    NSArray *coordinates =
        [self.mapView valueForKeyPath:@"annotations.coordinate" ];
    CLLocationCoordinate2D maxCoord = {-90.0f, -180.0f};
    CLLocationCoordinate2D minCoord = {90.0f, 180.0f};
}

```

```

for(NSValue *value in coordinates) {
    CLLocationCoordinate2D coord = {0.0f, 0.0f};
    [value getValue:&coord];
    if(coord.longitude > maxCoord.longitude) {
        maxCoord.longitude = coord.longitude;
    }
    if(coord.latitude > maxCoord.latitude) {
        maxCoord.latitude = coord.latitude;
    }
    if(coord.longitude < minCoord.longitude) {
        minCoord.longitude = coord.longitude;
    }
    if(coord.latitude < minCoord.latitude) {
        minCoord.latitude = coord.latitude;
    }
}
MKCoordinateRegion region = {{0.0f, 0.0f}, {0.0f, 0.0f}};
region.center.longitude = (minCoord.longitude + maxCoord.longitude)/2.0;
region.center.latitude = (minCoord.latitude + maxCoord.latitude) / 2.0;
region.span.longitudeDelta = maxCoord.longitude -minCoord.longitude;
region.span.latitudeDelta = maxCoord.latitude -minCoord.latitude;
[self.mapView setRegion:region animated:YES];
}

```

Nachdem wir den kleinsten und den größten Längen- und Breitengrad bestimmt haben, erzeugen wir darauf basierend eine neue Region und weisen die Map-View an, diese anzuzeigen.

Jetzt haben wir eine Anwendung, die fast all das tut, was wir wollen. Klicken Sie auf BUILD AND RUN und wählen Sie einen Ihrer Kontakte aus, um zu sehen, wie der Pin gesetzt und die Karte an Ihre Position und die des gewählten Kontakts angepasst wird.

Aber unser Pin ist rot, wir wollen ihn jedoch in Lila. Die Standardfarbe ist Rot, und da wir nichts Besonderes gemacht haben, bekommen wir eben den Standardwert. Für den lilafarbenen Pin müssen wir das Protokoll `MKMapViewDelegate` implementieren.

Es gibt bei diesem Protokoll eine Reihe von Methoden, von denen die meisten Callbacks sind, die Sie über interessante Events auf der Map-View informieren (zum Beispiel `mapViewWillStartLoadingMap:`). Die für uns interessante Methode ist `mapView:viewForAnnotation:`. Wie bei der Tabellen-View gibt es auch bei der Map-View eine Reihe von Anmerkungs-Views, die wiederverwendet werden können. Unsere Implementierung dieser Methode muss das berücksichtigen und sicherstellen, wenn möglich die gespeicherten Views zu verwenden.

Dies ist der Code:

MapKit/ContactMapper_01/Classes/ContactMapperViewController.m

```
- (MKAnnotationView *)mapView:(MKMapView *)mapView
    viewForAnnotation:(id <MKAnnotation>)annotation {
    MKPinAnnotationView *view = nil;
    if(annotation != mapView.userLocation) {
        view = (MKPinAnnotationView *)
            [mapView dequeueReusableAnnotationViewWithIdentifier:
                @"identifizier" ];
        if(nil == view) {
            view = [[[MKPinAnnotationView alloc]
                initWithAnnotation:annotation
                reuseIdentifier:@"identifizier" ]
                autorelease];
        }
        [view setPinColor:MKPinAnnotationColorPurple];
        [view setShowCallout:YES];
        [view setAnimatesDrop:YES];
    } else {
        CLLocation *location = [[CLLocation alloc]
            initWithLatitude:annotation.coordinate.latitude
            longitude:annotation.coordinate.longitude];
        [self setCurrentLocation:location];
    }
    return view;
}
```

Zunächst prüfen wir, ob die aktuelle Anmerkung die aktuelle Position des Anwenders ist. Wir könnten auch für den Benutzerstandort eine eigene Anmerkungs-View zurückgeben, aber wenn wir nil liefern, wird die Standard-View genutzt. Für die aktuelle Benutzerposition ist sie der blaue Punkt, der diese nette hüpfende Animation zeigt.

Wenn es nicht die aktuelle Position des Anwenders ist, versuchen wir, eine Anmerkungs-View wiederzuverwenden. Gibt es keine, erzeugen wir eine neue. Dann richten wir die View so ein, wie wir sie haben wollen, und geben sie zurück. In unserem Fall setzen wir die Pin-Farbe auf Lila, aktivieren die Callouts und legen fest, dass die Anmerkung mit einer Animation eingefügt werden soll.

Handelt es sich um die aktuelle Position des Benutzers, rufen wir `setCurrentLocation:` auf. Damit bekommen wir die Integration von Map Kit mit einem Location Manager geliefert, ohne die Delegate-Methoden selbst implementieren zu müssen.

TouchXML und der AddressGeocoder

Lassen Sie uns jetzt über den AddressGeocoder reden. Der Geocoder nutzt einen Webservice, um die Adressen in geografische Positionen umzuwandeln. Wir übergeben eine Adresse und erhalten eine XML-Struktur mit der Adresse sowie dem Längen- und Breitengrad. Wie Sie in Kapitel 12, *Verbindung zum Internet*, auf Seite 259 gesehen haben, können wir den NSXMLParser nutzen, um XML zu parsen. Das funktioniert zwar, es kann aber ziemlich aufwendig sein, die gesamte XML-Datei parsen zu müssen, wenn Sie nur ein paar Informationen daraus haben wollen. Um die gewünschten Daten zu erhalten, gibt es verschiedene Möglichkeiten. Im iPhone steht die Open Source-Bibliothek libXML2 zur Verfügung, die wunderbar funktioniert. Aber die API ist komplett C-basiert, wodurch sie ein wenig gewöhnungsbedürftig ist, wenn Sie sonst nur Objective-C nutzen. Wenden wir uns also lieber der Open-Source-Bibliothek TouchXML zu.

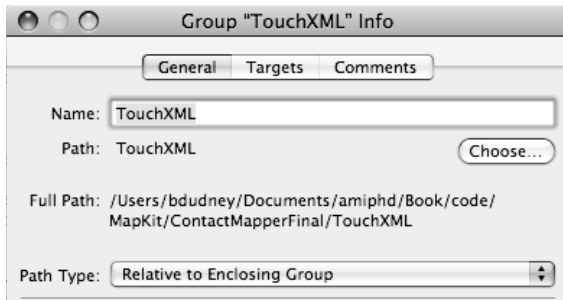
TouchXML stellt einen Objective-C-Wrapper für die libXML2-Bibliothek bereit. Das Projekt finden Sie auf der Google Code-Website unter <http://code.google.com/p/touchcode/wiki/TouchXML>. Version 1.0.6 finden Sie beim Code für dieses Kapitel, Sie müssen sie also nicht herunterladen. Hintergedanke von TouchXML ist, die vereinfachten XML-APIs auf dem iPhone bereitzustellen, die nur auf dem Mac zur Verfügung stehen. Brauchen Sie also eine detaillierte Dokumentation zu einer der Klassen, können Sie das führende C durch NS ersetzen und in der Mac-Dokumentation nachschauen.

TouchXML stellt uns XPath zur Verfügung, wodurch wir viel weniger Code als bei einem Parser brauchen, um die gewünschten Informationen zu extrahieren. Damit TouchXML in Ihrem Projekt funktioniert, müssen Sie ein paar Dinge erledigen:

- Holen Sie sich den Code, entweder über das Projekt zu diesem Buch oder über die Website.
- Fügen Sie den Quellcode für TouchXML und Tidy Ihrem Projekt hinzu.
- Fügen Sie den Header-Pfad für libxml2 Ihrem Header-Suchfad hinzu.

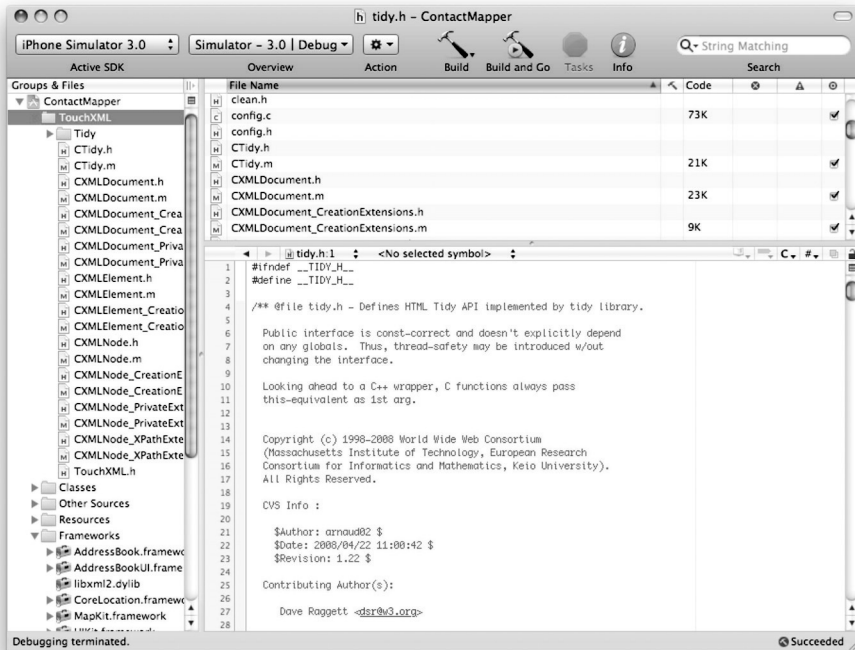
Der Code von TouchXML enthält auch den Code für Tidy, somit haben Sie ihn gleich dabei. Lassen Sie uns anschauen, wie Sie ihn zu Ihrem Projekt hinzufügen, damit Sie ihn statt des Parsers verwenden können.

Zunächst brauchen wir eine neue Gruppe für TouchXML und dann eine darunter für Tidy. Markieren Sie das Projekt, klicken Sie es bei gedrückter `Ctrl`-Taste an und wählen Sie **ADD NEW GROUP**. Geben Sie der Gruppe den Namen „TouchXML“. Markieren Sie sie, klicken Sie sie mit der rechten Maustaste an und wählen Sie **GET INFO**. Auf dem Tab **GENERAL** klicken Sie auf den Button **CHOOSE** neben dem Path-Element. Erzeugen Sie einen neuen Ordner namens *TouchXML*, markieren Sie ihn und klicken Sie auf den **CHOOSE**-Button. Wiederholen Sie das Ganze für die Gruppe Tidy, aber unterhalb der TouchXML-Gruppe. Hier ein Screenshot des Ordners *TouchXML* in unserem Projekt:

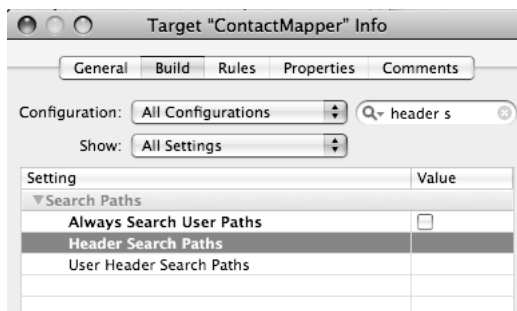


Abhängig davon, woher Sie den Code haben, wird die Verzeichnisstruktur unterschiedlich sein. Holen Sie sich den Code aus den herunterladbaren Beispielen zum Buch, finden Sie alles im Verzeichnis *TouchXML*, auch das Verzeichnis *Tidy*. Holen Sie sich den Quellcode von der Projektseite von TouchXML, finden Sie den TouchXML-Code unter *Common/Source* und *Common/Source/Creation*. Der Sourcecode von Tidy liegt dann unter *Externals/tidy/src* und *Externals/tidy/include*. Sie brauchen alle Dateien, sowohl von TouchXML als auch von Tidy.

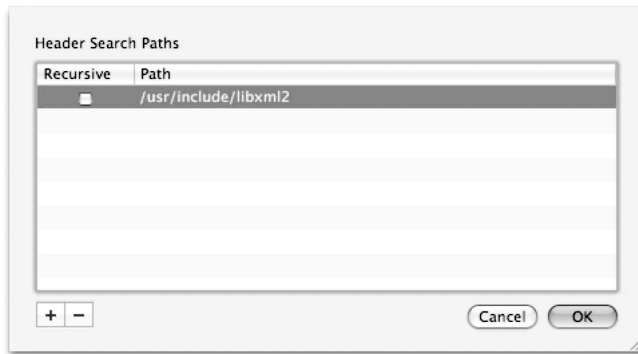
Im Finder markieren Sie alle Dateien für TouchXML und ziehen sie in die Gruppe „TouchXML“ in Xcode. Wenn Sie gefragt werden, stellen Sie sicher, dass Sie die Checkbox „copy files“ markiert haben. Tun Sie das Gleiche für die Tidy-Dateien und die „Tidy“-Gruppe. Wenn Sie fertig sind, sollte Xcode in etwa so aussehen:



Da wir den nun Code haben, müssen wir Xcode mitteilen, wo es die Header von libXML2 findet. Markieren Sie Ihr Target in der „Targets“-Gruppe. Klicken Sie mit der rechten Maustaste oder bei gedrückter **[Ctrl]**-Taste und wählen Sie GET INFO. Auf dem Tab BUILD geben Sie in das Suchfeld header s ein. Sie sollten dann ein Fenster das wie im nächsten Screenshot erhalten.



Klicken Sie doppelt auf die Zeile „Header Search Paths“. Im sich öffnenden Fenster klicken Sie auf den +-Button und geben in der Zeile „Path“ den Wert `/usr/include/libxml2` ein. Wenn Sie fertig sind, sollte das Fenster in etwa so aussehen:



Klicken Sie auf den OK-Button, sollten Sie TouchXML und Tidy als Teil Ihres Projekts nutzen können.

Es gibt auch andere Wege, dieses Ziel zu erreichen – wir könnten aus den Quellen von TouchXML und Tidy eine statische Bibliothek erzeugen, aber das zu erläutern, geht über den Rahmen dieses Buchs hinaus.

Nachdem wir nun TouchXML eingerichtet haben, wollen wir uns die Klasse AddressGeocoder und ihre Verwendung anschauen, um eine Adresse von einem der Geocoding-Services zu erhalten.

Dies ist ein Teil der Implementierung:

MapKit/ContactMapper_02/Classes/AddressGeocoder.m

```

1  + (BOOL)geocodeStreetAddress:(NSString *)street
2         city:(NSString *)city
3         state:(NSString *)
4         state zip:(NSString *)
5         zip country:(NSString *)country
6         intoLocation:(CLLocationCoordinate2D *)location {
7      BOOL success = NO;
8      NSURL *url = [self urlForAddress:street city:city state:state
9                                zip:zip country:country];
10     NSError *error = nil;
11     CXMLElement *doc = [[CXMLElement alloc]
12                        initWithContentsOfURL:url
13                        options:CXMLElementTidyXML error:&error];
14     CXMLElement *element = [doc rootElement];
15     NSDictionary *namespaceMappings =
16         [NSDictionary dictionaryWithObject:@"http://earth.google.com/kml/2.0"
17                                         forKey:@"kml" ];
18     NSArray *status = [element nodesForXPath:@"//kml:Status/kml:code"
19                                namespaceMappings:namespaceMappings
20                                error:&error];
21     if(@"200" isEqualToString:[status objectAtIndex:0] stringValue]) {
22         NSArray *coordElements = [element nodesForXPath:@"//kml:coordinates"
23                                namespaceMappings:namespaceMappings
24                                error:&error];

```

```

25     NSString *coords = [[coordElements objectAtIndex:0] stringValue];
26     NSArray *components = [coords componentsSeparatedByString:@"," ];
27     NSNumberFormatter *formatter = [[NSNumberFormatter alloc] init];
28     NSNumber *longitude =
29         [formatter numberFromString:[components objectAtIndex:0]];
30     NSNumber *latitude =
31         [formatter numberFromString:[components objectAtIndex:1]];
32     location->longitude = [longitude floatValue];
33     location->latitude = [latitude floatValue];
34     [formatter release];
35     success = YES;
36 }
37 return success;
38 }

```

In Zeile 8 wandeln wir die Adresse in eine URL um, die der Geocoding-Service versteht. Diese URL ist natürlich vom verwendeten Service abhängig. Als Nächstes holen wir den XML-Code aus dem Web und wandeln ihn in Zeile 11 in ein CXMLElement um. Im Hintergrund nimmt sich TouchXML die URL, lädt das gesamte XML-Dokument herunter, parst es und verwandelt es in einen DOM-Baum. Das Herunterladen des XML-Codes im Haupt-Thread mag nicht die beste Option für Ihre Anwendung sein, daher sollten Sie sorgfältig überlegen, ob Sie das in einem Hintergrundprozess durchführen sollten. In diesem Beispiel werden wir das nicht tun. Anschließend legen wir den Namensraum fest, der vom XML-Dokument genutzt wird. Dann prüfen wir in Zeile 21 den Status, um sicherzustellen, dass wir eine gültige Antwort erhalten haben. Wenn alles wie geplant ablief, haben wir eine gültige Antwort, die wir ab Zeile 25 in einen Längen- und Breitengrad wandeln können.

Das ist nicht der gesamte Code für den AddressGeocoder. Die vollständige Klasse können Sie im herunterladbaren Beispielcode für das Projekt finden.

Unsere Anwendung ist nun fast fertig. Wir können Kontakte über den People Picker wählen und sie auf einer Karte anzeigen lassen. Als Letztes wollen wir noch implementieren, dass wir die Details eines Kontakts angezeigt bekommen, wenn wir auf einen der Pins klicken. Das werden wir uns jetzt anschauen.

25.4 Eine Anmerkung auswählen

Klickt der Anwender auf eine der Anmerkungen auf der Karte, zeigt die Map-View die Callout-View an. Das Callout kann zwei Accessory-Views besitzen. Auch wenn wir dafür jede beliebige Subklasse von UIView nutzen können, handelt es sich typischerweise um ein UIControl. Nutzen

Sie ein Control, kümmert sich die Map-View um die Events. Schreiben Sie Ihre eigene View, müssen Sie sich selbst mit den Events herum-schlagen.

In der Kartenanwendung wird als Accessory-View ein UIButton genutzt, wobei der Typ auf UIButtonTypeDetailDisclosure gesetzt wurde. Da die Anwender ziemlich an dieses Paradigma gewöhnt sind, werden wir das Gleiche tun. Hier sehen Sie den angepassten Code, der die Anmer-kungs-View erzeugt:

MapKit/ContactMapper_02/Classes/ContactMapperViewController.m

```

1  - (MKAnnotationView *)mapView:(MKMapView *)mapView
2      viewForAnnotation:(id <MKAnnotation>)annotation {
3      MKPinAnnotationView *view = nil;
4      if(annotation != mapView.userLocation) {
5          view = (MKPinAnnotationView *)
6              [mapView dequeueReusableAnnotationViewWithIdentifier: @"identfier" ];
7          if(nil == view) {
8              view = [[MKPinAnnotationView alloc]
9                  initWithAnnotation:annotation reuseIdentifier: @"identfier" ]
10                 autorelease];
11             view.rightCalloutAccessoryView =
12                 [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
13         }
14         [view setPinColor:MKPinAnnotationColorPurple];
15         [view setCanShowCallout:YES];
16         [view setAnimatesDrop:YES];
17     } else {
18         CLLocation *location = [[CLLocation alloc]
19             initWithLatitude:annotation.coordinate.latitude
20             longitude:annotation.coordinate.longitude];
21         [self setCurrentLocation:location];
22         [location release];
23     }
24     return view;
25 }
```

In Zeile 11 setzen wir die rechte Callout Accessory-View als Button. Beachten Sie, dass wir den ganzen Target-/Action-Kram für den Button nicht definieren müssen. Die Map-View kümmert sich schon darum und ruft die Methode `mapView:annotationView:calloutAccessory-ViewControlTapped:` auf, wenn der Button angetippt wird.

Dieser Callback entspricht stark der Methode, die von der Tabellen-View aufgerufen wird, wenn man eine Zeile anklickt. Die Map-View ruft uns, wenn der Anwender auf das Control tippt, so dass wir selbst wei-termachen können. Hier ist die Implementierung:

MapKit/ContactMapper_02/Classes/ContactMapperViewController.m

```

- (void)mapView:(MKMapView *)mapView
  annotationView:(MKAnnotationView *)view
calloutAccessoryControlTapped:(UIControl *)control {
    ContactAnnotation *ann = (ContactAnnotation *)view.annotation;
    ABPersonViewController *personVC =
    [[[ABPersonViewController alloc] init] autorelease];
    UINavigationController *nav =
    [[[UINavigationController alloc]
      initWithRootViewController:personVC]
      autorelease];

    personVC.navigationItem.leftBarButtonItem =
    [[[UIBarButtonItem alloc]
      initWithBarButtonSystemItem:UIBarButtonSystemItemStop
      target:self
      action:@selector(stopEditingPerson)]
      autorelease];
    personVC.displayedPerson = ann.person;
    personVC.personViewDelegate = self;
    [self presentViewController:nav animated:YES];
}

```

Wir erzeugen den Person View-Controller, richten ihn in einem Navigations-Controller ein, so dass wir der Navigationsleiste Buttons hinzufügen können, und geben dann die Person an, die wir angezeigt haben wollen. Direkt vor dem Anzeigen der Person-View richten wir uns selbst als Delegate ein. In unserer Implementierung der Person Viewer Delegate-Methode geben wir einfach NO zurück. Aber in Ihrer Anwendung kann es sinnvoll sein, dem Benutzer die Anwahl einer Nummer oder das Senden einer E-Mail zu ermöglichen.

Das Map Kit bietet eine Reihe erstaunlicher Funktionen an, die unsere Anwendung viel interessanter machen kann als allein mit dem Location Manager.

Kapitel 26

Anwendungsintegration

Die Integration Ihrer Anwendung in andere Apps auf dem System eröffnet dem Benutzer häufig solch erstaunliche Möglichkeiten, dass er das sicherlich zu schätzen weiß. Wenn Ihre Anwendung zum Beispiel Telefonnummern anzeigt, werden Ihre Benutzer sehr dankbar sein, wenn sie diese antippen können und die Telefonanwendung gestartet wird.

In diesem Kapitel werden wir zeigen, wie unsere Anwendungen andere Programme im System starten können. Der eigentliche Codeanteil ist minimal, der interessante Teil ist, was Sie Ihren Anwendern ermöglichen wollen.

Wir werden Ihnen auch zeigen, wie Sie anderen Anwendungen die Erlaubnis erteilen können, Ihr App genau so zu starten, wie Apple es uns erlaubt, Mail oder Safari zu starten. Wir werden das in der zweiten Hälfte dieses Kapitels besprechen. Lassen Sie uns mit einer Anwendung beginnen, die Safari startet und bei Google nach einem Begriff sucht.

26.1 Starten anderer Anwendungen

Anwendungen, die URLs unterstützen, können mit einem Aufruf der Methode `openURL:` der `UIApplication` gestartet werden. Solange die URL wohlgeformt und die Anwendung ordentlich registriert ist, wird sich das iPhone OS um den Rest kümmern. Schauen wir also, wie wir Safari mit diesem Mechanismus starten können.

Erstellen Sie ein neues Projekt. Fügen Sie ein UI hinzu, über das man einen Suchbegriff eingeben kann, und ergänzen Sie den Code, der Safari startet und eine Suche bei Google anstößt. Erzeugen Sie also ein neues View-basiertes Projekt mit dem Namen `Searcher`. Öffnen Sie die

Datei `SearcherViewController.xib` und fügen Sie auf der View ein Textfeld ein. Setzen Sie den Platzhaltertext im Attributes Inspector (-) auf „Abfragebegriff“ und den Textfeld-Delegate auf den File's Owner (ziehen Sie bei gedrückter -Taste vom Textfeld zum File's Owner-Objekt). Das UI ist damit fertig, so dass Sie den Interface Builder beenden können, wenn Sie möchten.

Jetzt müssen wir das Protokoll `UITextFieldDelegate` für unsere Klasse `SearcherViewController` so implementieren, dass wir den Suchbegriff aus dem Textfeld auslesen können. (Mehr Informationen zum Delegate-Protokoll finden Sie in Abschnitt 4.7, *Erstellen des UI*, auf Seite 74.) Öffnen Sie die Datei `SearcherViewController.h` und fügen Sie die Protokolldeklaration hinzu. Dann öffnen Sie die Datei `SearcherViewController.m` und ergänzen eine Implementierung der Methode `textFieldShouldReturn:`, die in etwa so aussieht:

AppIntegration/Searcher/Classes/SearcherViewController.m

```
- (BOOL)textFieldShouldReturn:(UITextField *)field {
    [field resignFirstResponder];
    return YES;
}
```

Wie in Abschnitt 4.7, *Erstellen des UI*, auf Seite 74 gesehen, ist `textFieldShouldReturn:` ein guter Platz, um den First Responder-Status abzugeben, so dass die Tastatur wieder ausgeblendet wird. Die nächste zu implementierende Methode ist `textFieldDidEndEditing:`. Dies ist ihr Code:

AppIntegration/Searcher/Classes/SearcherViewController.m

```
- (void)textFieldDidEndEditing:(UITextField *)field {
    NSString *query = @"iPhone" ;
    if(field.text != nil && [field.text length]) {
        query = field.text;
    }
    NSString *urlString =
        [NSString stringWithFormat:@"http://google.com?q=%@" , query];
    NSURL *url = [NSURL URLWithString:urlString];
    [[UIApplication sharedApplication] openURL:url];
}
```

Wenn es einen Suchbegriff gibt, holen wir ihn uns, ansonsten suchen wir standardmäßig nach „iPhone“. Haben wir den Begriff, erzeugen wir eine URL daraus. Dann holen wir unser Anwendungs-Objekt über die Methode `sharedApplication` und bitten es, die URL über `openURL:` zu öffnen.

Tippt der Anwender auf den RETURN-Button, nachdem er einen Begriff eingegeben hat, wird Safari gestartet und die URL geöffnet, womit die Suchergebnisse angezeigt werden.

Es ist zwar toll, dass Sie damit andere Anwendungen nutzen können, aber Ihre eigene App wird auf diese Weise beendet. Sie sollten sich also genau überlegen, ob die Vorteile für den Anwender diesen Nachteil aufwiegen. Ist das Starten einer neuen App ein logischer Endpunkt in Ihrer Anwendung und der Benutzer wird eher nicht zu ihr zurückkehren, wenn er nicht gerade einen neuen Zyklus beginnen will, ist diese Art von Integration super. Wird der Anwender aber wahrscheinlich nach kurzer Zeit in Ihrer Anwendung dort weitermachen wollen, wo er sie verlassen hat, werden Sie das beim Starten Ihrer Anwendung berücksichtigen müssen.¹

In diesem Beispiel haben wir HTTP-Links gesehen. Cocoa Touch unterstützt aber auch noch diverse andere Schemata: Die Telefon-App nutzt das Schema `tel:`, SMS nutzt `sms:`. Anders als die anderen Apps nutzen Maps, YouTube und der App-Store besonders aufgebaute `http:`-URLs.

Alle Anwendungen, die über eine URL aufgerufen werden können, lassen sich auf die gleiche Art und Weise starten. Der einzige Unterschied liegt im Aufbau der URL. Sie finden die Details über die verschiedenen unterstützten URL-Schemata in den iPhone-Dokumenten, die Sie per Volltextsuche nach *URL Scheme Reference* auffinden.

Nachdem wir nun gesehen haben, wie man eine Anwendung mit einer URL startet, wollen wir auch lernen, wie wir unsere eigenen Apps startbar machen können.

26.2 Selbst integrierbar werden

In diesem Abschnitt werden wir erklären, was Sie tun müssen, damit Ihre Anwendung über eine URL gestartet werden kann. Dabei handelt es sich nur um zwei Schritte:

1. Fügen Sie einen Eintrag Ihrer Datei `Info.plist` hinzu.
2. Implementieren Sie die Methode `application:handleOpenURL:`.

¹ So könnten Sie zum Beispiel den Status Ihrer Anwendung sichern, wenn die andere App gestartet wird, und dann bei der Rückkehr des Anwenders zu diesem Zustand zurückwechseln.

Das ist alles. Es ist also anscheinend sehr einfach, Ihre Anwendung integrierbar zu machen. Lassen Sie uns ein Beispiel betrachten, das über eine URL gestartet werden kann und das den Resource Specifier der URL nutzt, um ein Label auf dem UI zu aktualisieren.

Erzeugen Sie ein weiteres View-basiertes Projekt namens Integrated. Öffnen Sie die Datei `IntegratedViewController.h` und fügen Sie zwei Instanzvariablen hinzu – einen `NSString` namens `message` und ein `IBOutlet` namens `messageLabel` vom Typ `UILabel`. Ergänzen Sie auch noch eine Eigenschaft für `message`, so dass wir sie vom App-Delegate aus setzen können.

Unser Header sieht dann so aus:

AppIntegration/Integrated/Classes/IntegratedViewController.h

```
@interface IntegratedViewController : UIViewController {
    NSString *message;
    IBOutlet UILabel *messageLabel;
}

@property(n nonatomic, retain) NSString *message;

@end
```

Jetzt wollen wir das UI anpassen und die Verbindungen herstellen. Öffnen Sie die Datei `IntegratedViewController.xib`. Ziehen Sie ein `UILabel` auf die View und setzen Sie ihren Text auf „Default“. Verbinden Sie das Outlet `messageLabel` des `IntegratedViewController` mit dem neu erzeugten `UILabel` (der `IntegratedViewController` ist der File's Owner). Das UI ist damit fertig, Sie können den Interface Builder also schließen.

Als Nächstes müssen wir die Methode `setMessage:` im `IntegratedViewController` implementieren, damit das Label beim Setzen der Nachricht aktualisiert wird:

AppIntegration/Integrated/Classes/IntegratedViewController.m

```
- (void)setMessage:(NSString *)newMessage {
    [message release];
    message = [newMessage retain];
    messageLabel.text = message;
}
```

In dieser Methode achten wir auf das Attribut `retain`, das wir in der Eigenschaftsdeklaration gesetzt haben, indem wir die alte `message` freigeben und die neue `message` übernehmen. Danach aktualisieren wir das `messageLabel`. Das ist alles.

Jetzt müssen wir noch die Methode `application:handleOpenURL:` implementieren. Die Implementierung dieser Methode kann sehr einfach oder sehr komplex sein – abhängig davon, was Ihre Anwendung braucht. Aber es gibt ein paar Dinge, die Sie sich sehr gut überlegen sollten:

- Die URL ist eine öffentliche API. Es ist schwierig, sie zu ändern, sobald sie einmal von anderen genutzt wird.
- Was könnte man mit der URL Böses anstellen? Sie müssen sicher sein, dass Sie sich gegen Angriffe schützen – gegen solche, an die Sie gedacht haben, und gegen solche, an die Sie nicht gedacht haben.
- Beschränkt man die Komplexität (und damit auch die Flexibilität) der URL, lässt sie sich leichter durch andere integrieren.

In unserem Beispiel werden wir die URL nur minimal prüfen, um sicherzustellen, dass sie nicht böse ist (nicht dass die App das braucht, es dient nur der Illustration). Und da dies auch nur eine Beispielanwendung ist, gibt es keine echte Komplexität zu berücksichtigen. Unser URL-Schema sowie die Implementierung sind also sehr einfach. Wir haben für unser Schema `integrated://` gewählt und werden den Resource Specifier (den Teil der URL nach dem `://`) als Nachricht nutzen. Dies ist die Implementierung von `application:handleOpenURL::`

AppIntegration/Integrated/Classes/IntegratedAppDelegate.m

```
- (BOOL)application:(UIApplication *)application handleOpenURL:
(NSURL *)url {
    BOOL flag = NO;
    if([[url resourceSpecifier] length] < 25) {
        viewController.message = [[url resourceSpecifier] substringFromIndex:2];
        flag = YES;
    }
    // Ist die Nachricht länger als 25 Zeichen,
    // ist sie vielleicht böse
    return flag;
}
```

In diesem Code prüfen wir zunächst, ob der Resource Specifier weniger als 25 Zeichen hat. Wenn ja, holen wir ihn uns, ansonsten gehen wir davon aus, dass er vielleicht böse ist, und ignorieren ihn. Entspricht die URL unseren Wünschen, holen wir den Resource Specifier und übergeben ihn an die Eigenschaft `message` des `viewController`, wodurch das UI aktualisiert wird. Wir sind schon fast fertig und müssen uns nur noch um eine Sache kümmern: Das iPhone OS muss wissen, dass unsere App URLs vom Typ `integrated` versteht.

Um klarzustellen, dass unsere Anwendung URLs vom Typ `integrated` nutzen kann, müssen wir unserer Datei `Info.plist` einen Array-Schlüssel `CFBundleURLName` hinzufügen. Das Format der `plist`-Datei ist in den iPhone-Dokumenten beschrieben, daher werden wir hier nicht ins Detail gehen. Dies ist der Code, den Sie am Ende der Datei anfügen können:

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.pragprog.amiphd.integrated</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>integrated</string>
    </array>
  </dict>
</array>
```

Wir deklarieren damit, dass unsere Anwendung URLs mit dem Schema „integrated“ versteht. Um unsere App zu starten, brauchen wir eine URL, die mit `integrated://` beginnt. Bauen und Installieren Sie die App also im Simulator (mit dem Button **BUILD AND RUN**). Ist die Anwendung gestartet, stoppen Sie sie, indem Sie auf den Home-Button klicken. Dann starten Sie Safari und geben die URL `integrated://Nachricht` ein. Klicken Sie auf den GO-Button, wird Safari beendet und unsere Anwendung geöffnet, wobei der Text `default` durch `Nachricht` ersetzt wurde. Sehr cool! Sie haben eine Anwendung, die per URL gestartet und damit durch andere Apps auf dem iPhone genutzt werden kann.

Das Registrieren Ihres neuen Schemas sorgt nicht dafür, dass Ihre URLs vom Textsystem erkannt werden. Schicken Sie also eine E-Mail an Ihre Anwender oder eine SMS, wird die URL nicht automatisch erkannt. Stecken Sie sie aber in spitze Klammern (also `<` und `>`), erkennt das System, dass es sich um einen Link handelt. Tippen Ihre Anwender darauf, wird die passende Applikation gestartet. In unserem Beispiel ermöglicht das Hinzufügen von `<integrated://funktioniert>` zu einer E-Mail, dass der Anwender darauf tippen kann, die App gestartet wird und „funktioniert“ anzeigt.

Natürlich können Sie Ihre Links auch in ordentliche HTML-Anker-Tags stecken, zum Beispiel: `<a href="integrated://funktioniert">sagt funktioniert</a>`. Wie Ihre Anwender Links auf Ihre App zu sehen bekommen sollen, entscheidet, wie Sie Links im Text einfügen. Sollen

Ihre Anwender wissen, dass der Link Ihre App startet, ist die erste Option die bessere. Müssen (oder sollen) Ihre Benutzer nicht wissen, dass hinter einem Link Ihre App steckt, ist die zweite Variante vermutlich die bessere.

In diesem Kapitel haben Sie gesehen, wie Sie andere Anwendungen über URLs starten und dabei auch noch Informationen übermitteln können. Sie haben auch gelernt, wie Ihre eigene Anwendung das gleiche Feature anbieten kann, so dass andere Programme Ihre App starten können.

Kapitel 27

Debuggen

Wenn Sie dieses Buch durchgearbeitet und Ihren eigenen Code entwickelt haben, sind wir sicher, dass Sie – wie wir – keine Probleme hatten. Ihr Code lief gleich beim ersten Mal, funktionierte korrekt und war dabei erstaunlich effizient. Der ganze Kram ist schließlich total einfach, oder?

Oder?

Nein, natürlich nicht. Und auch bei uns ging nicht alles auf Anhieb. Jedes Codebeispiel in diesem Buch wurde vom Compiler häufiger zurückgewiesen, als wir zugeben möchten.

Das Entwickeln einer funktionierenden Anwendung, insbesondere auf einer Plattform, die für uns wie auch für die ganze Welt neu ist, ist ein Prozess konstanter Verbesserungen und damit auch eine permanente Fehlerquelle. Das läuft in etwa so ab:

1. Erst lässt sich Ihr Code nicht kompilieren.
2. Dann lässt er sich kompilieren, stürzt aber ab.
3. Dann stürzt er nicht mehr ab, funktioniert aber nicht so, wie er soll.
4. Dann funktioniert er, ist aber sooo langsam.
5. Dann ist er schnell.
6. Und dann ist er richtig gut!

In diesem Kapitel werden wir uns der ersten drei dieser Probleme annehmen – wie Sie die vom iPhone SDK angebotenen Tools nutzen, damit Ihr Code stabil und korrekt ist. Danach können Sie sich Kapitel 28, *Performance Tuning*, auf Seite 545 zuwenden, in dem Sie dafür sorgen, dass Ihr Code auf dem Gerät so schnell wie möglich läuft.

27.1 Fehler beim Build verstehen und beheben

Das erste Problem, dem Sie sich sehr wahrscheinlich gegenübersehen, ist der Compiler-Fehler. Das sind richtig dicke Fehler, denn egal wie gut der Rest Ihres Codes ist – die Teile, die zu Compiler-Fehlern führen, sind so kaputt, dass Sie die Anwendung gar nicht erst starten können, bis die Fehler behoben sind. Aber so ernst das Problem auch sein mag, die Ursache ist häufig trivial. Oft müssen Sie zum Beheben eines Compiler-Fehlers nur einen Tippfehler korrigieren.

Lassen Sie uns einmal absichtlich solch einen Fehler produzieren. Das Beispielprojekt `TrivialCompilerErrors` bietet nur ein GUI mit einer View an, deren Inhalt eine einfache `UITextView` ist, verbunden mit der Variablen `textView` im `TrivialCompilerErrorsViewController`. Wir müssen jetzt in dieser Klasse nur den Text setzen, wenn die View angezeigt wird.

Debugging/TrivialCompilerErrors/Classes/TrivialCompilerErrorsViewController.m

```
- (void)loadView {
    [super loadView];
    textView.txt = @"Hallo Welt" ;
}
```

Wie Sie sehen können, soll die Anwendung nur die Texteigenschaft der `UITextView` auf einen statischen String setzen. Aber wir haben einen Tippfehler gemacht – wir haben die Eigenschaft mit dem Namen `txt` verwendet, der korrekte Name ist aber `text`. Ein Hinweis darauf könnte schon sein, dass Xcode den Namen der Eigenschaft in Schwarz anzeigt. Denn normalerweise werden bekannte Eigenschafts- oder Funktionsnamen einer Klasse durch das Syntax-Highlighting von Xcode in Lila dargestellt. Da muss also schon etwas falsch sein.

Was passiert, wenn Sie diesen Code bauen? Sie erhalten einen Build-Fehler. Sie werden einen kleinen Statusindikator in der unteren rechten Ecke des Projektfensters sehen, und jedes Source-Fenster, das in diesem Projekt geöffnet wird, sieht so aus:



Die Gesamtanzahl an Fehlern wird in dieser Statusbar neben dem weißen „x“ in einem roten Kreis angezeigt. Haben Sie Build-Warnungen, erscheinen sie als weißes „!“ in einem gelben Dreieck. Jeder dieser Warnungs-/Fehlerzähler funktioniert wie ein Hyperlink. Sie können die Anzahl anklicken, um zum Fenster mit den Build Results zu gelangen

(siehe Abbildung 27.1). Innerhalb dieses Fensters können Sie jeden Fehler und jede Warnung im oberen Bereich anklicken, um im unteren Bereich zur fraglichen Zeile zu gelangen, wobei ein kleines Pop-up den Fehler anzeigt.

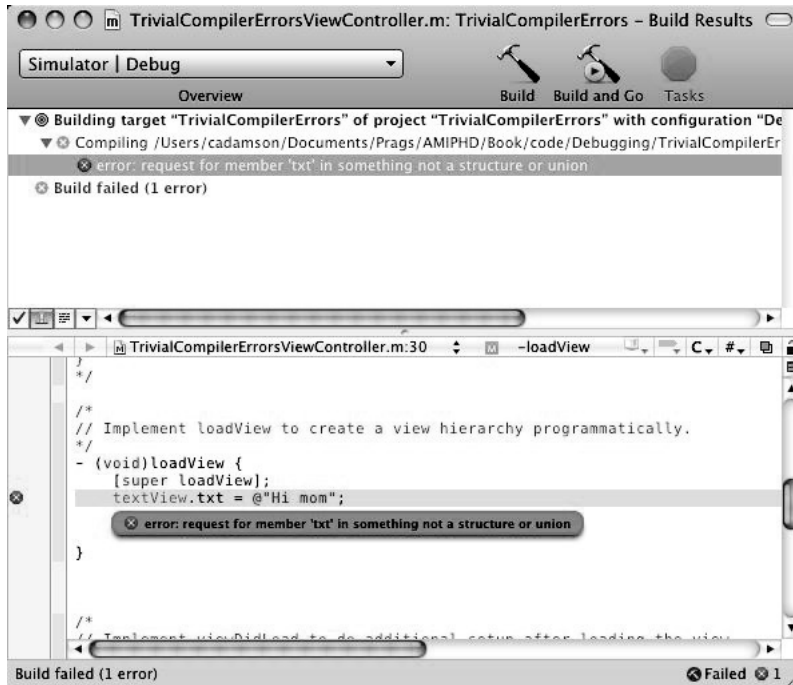


Abbildung 27.1: Build Results-Fenster in Xcode

In diesem Fall sehen Sie den Fehler "error: request for member 'txt' in something not a structure or union".¹ Sie können ihn an jeder Stelle beheben, an der er sichtbar ist – im Build Results-Fenster, im Projektfenster (nachdem Sie die Quelldatei aus der Liste ausgewählt haben), im Bearbeitungsfenster der Quelldatei selbst und so weiter. Ändern Sie einfach txt in text, und lassen Sie das Projekt erneut bauen.

Da wir gerade in solch einem einfachen Projekt sind, wollen wir noch ein paar andere Teile des Codes verdrehen, um häufig vorkommende Fehler zu erzeugen. Neue Objective-C-Programmierer begehen häufig den Fehler, ihre Objekte nicht als Zeiger zu deklarieren. Lassen Sie uns anschauen,

¹ Das ist unbestritten entweder eine furchtbare Fehlermeldung oder ein Argument gegen die Entscheidung, den Punktoperator wiederzuverwenden. Ursprünglich war er in C dazu gedacht, die Member einer Struct kenntlich zu machen, während er in Objective-C für die Eigenschaften eine ganz andere Bedeutung hat. Der Fehler hat nichts mit Structs oder Unions zu tun, aber das können Sie nicht wissen, wenn Sie die Meldung wörtlich nehmen.

wie das aussieht. Gehen Sie zur Datei `TrivialCompilerErrorsViewController.h` und suchen Sie nach der Deklaration der Text-View:

```
IBOutlet UITextView* textView;
```

Wir werden das „verfälschen“, indem wir den Stern `*` entfernen. Damit ist es nicht mehr länger eine Objektreferenz in Objective-C:

```
IBOutlet UITextView textView;
```

Auf diese Weise erhalten wir gleich drei Fehler. Zwei der Fehler (jeweils einer in `TrivialCompilerErrorsAppDelegate.m` und `TrivialCompilerErrorsViewController.m`, da beide `TrivialCompilerErrorsViewController.h` importieren) beschwerten sich über die statisch allozierte Instanz der Objective-C-Klasse `UITextView`, während der dritte darauf aufmerksam macht, dass `struct UITextView` keinen Member namens `text` besitzt, was eigentlich ein Seiteneffekt davon ist, dass wir gar keine Instanz von `UITextView` besitzen und daher auch nicht wissen, was ihre Eigenschaften sind.

Beachten Sie übrigens, dass die Dateien, die Xcode als fehlerhaft anzeigt, gar nicht die eigentliche Ursache des Problems enthalten. Der wirkliche Fehler sitzt in der Header-Datei `TrivialCompilerErrorsViewController.h`, aber der Compiler meldet die Fehler in den Implementierungs-`.m`-Dateien, die diesen Header *importieren*. Wenn Sie Fehler erhalten, ist es wichtig, die nicht so klare Natur einiger der Meldungen zu berücksichtigen. Glücklicherweise ist es in diesem Fall so, dass Xcode beim Anklicken des Fehlers im Build Results-Fenster die falsche Zeile in der Header-Datei hervorhebt.

27.2 Fehler beim Importieren/Linken verstehen und beheben

Ein anderer häufiger Fehler ist, Code zu haben, der zwar korrekt ist, sich aber nicht bauen lässt, weil die Information über das Bauen des Codes falsch ist. Es gibt zwei oft vorkommende Fehler dieser Art:

- Kein Importieren benötigter Header-Dateien.
- Kein Linken von benötigten Frameworks.

Der erste wird jedem vertraut sein, der schon mit irgendeiner Varietät von C gearbeitet hat. Vergessen Sie, eine benötigte Header-Datei per `#import` oder `#include` zu importieren, führt jede Referenz auf ein Objekt, eine Methode oder eine prozedurale Funktion aus dieser Header-Datei zu einem Build-Fehler.

So haben wir zum Beispiel in Kapitel 10, *Die Datenbank SQLite*, auf Seite 205 die Header-Datei der SQLite3-Datenbank importiert, um deren Funktionalität verwenden zu können. Hätten wir vergessen, die Header-Datei per `#include <sqlite3.h>` zu importieren, würden wir Fehler oder Warnungen für jeden versuchten Aufruf von SQLite erhalten mit dem Hinweis, dass entweder die Funktionen undefiniert seien oder wir sie, wie in Abbildung 27.2 gezeigt, implizit definieren würden.

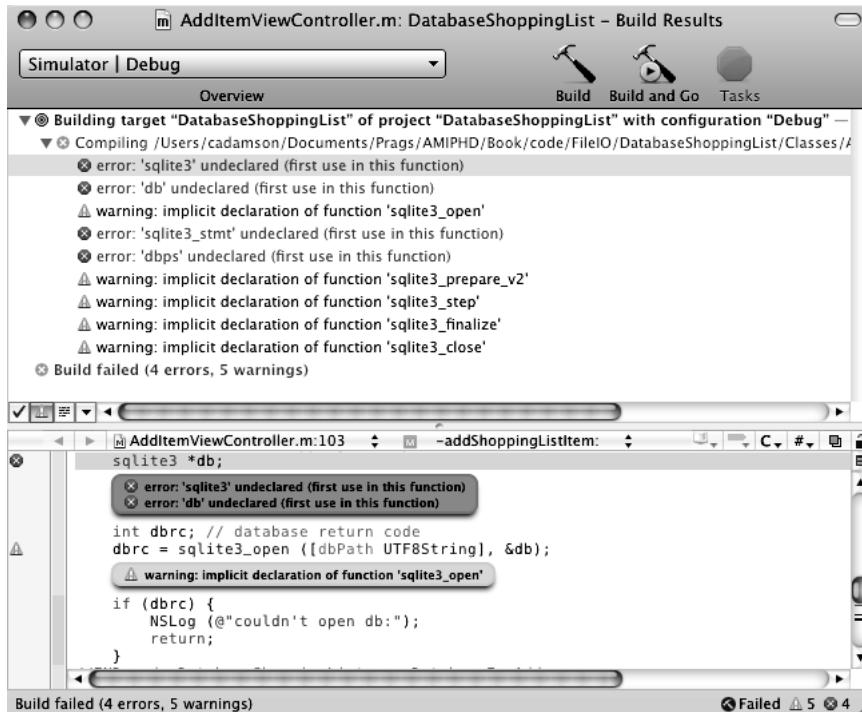


Abbildung 27.2: Build-Ergebnisse mit fehlendem `#include`

Ein ähnliches Problem entsteht, wenn man vergisst, dem Xcode-Projekt ein Framework oder eine Bibliothek hinzuzufügen. Auch wenn die `#include`-, `#import`- oder `@class`-Deklarationen den Compiler erst einmal beruhigen, da sie Klassen, Methoden und/oder Definitionen bereitstellen, braucht der Linker trotzdem noch einen Pfad auf das `.framework-Bundle` oder die `.dylib-Datei`, die den Code des Frameworks oder der Bibliothek tatsächlich enthält. Wenn nicht, erhalten Sie einen Linking-Fehler „Symbol(s) not found“, wie den in Abbildung 27.3.

Wenn Sie diesen Fehler sehen und bemerken, dass Sie die Bibliothek oder das Framework Ihrem Projekt nicht hinzugefügt haben, müssen Sie nur mit der rechten Maustaste auf den *Frameworks*-Ordner im Ab-

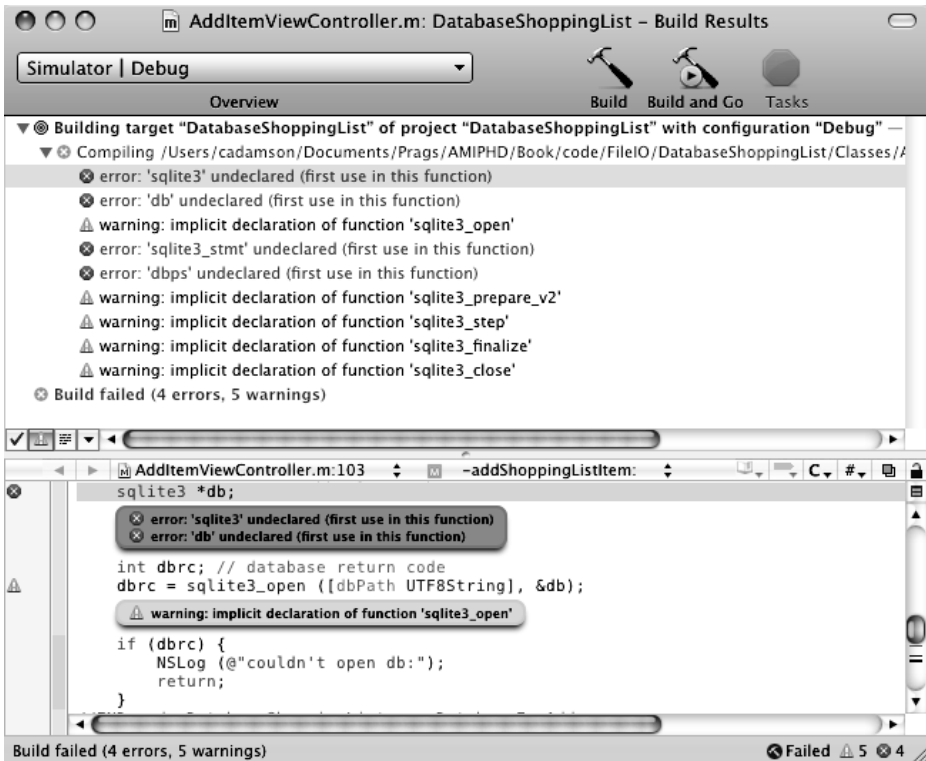


Abbildung 27.3: Build-Ergebnisse mit fehlender Bibliothek

schnitt „Groups & Files“ Ihres Xcode-Projekts klicken, ADD EXISTING FRAMEWORK auswählen und dann zum benötigten Framework-Bundle oder zur Bibliotheksdatei navigieren (achten Sie darauf, dass sich der Pfad innerhalb der SDK-Verzeichnisse befindet, zum Beispiel /Developer/Platforms/iPhoneSimulator.platform oder /Developer/Platforms/iPhoneOS.platform, und Sie nicht im Mac OS X-Verzeichnis /System/Library/Frameworks gelandet sind).

Für den unerfahrenen Entwickler sind beide Fehler schockierend, weil die bloße Anzahl an Fehlern einen erschlagen kann – niemand ist glücklich, wenn ein Build mit mehr als 100 Fehlern und Warnungen abbricht. Aber mit der Zeit werden Sie herausfinden, dass beide Fehlerarten anhand ihrer Build Results leicht erkannt werden können und die Ursache des Fehlers leicht behoben werden kann.

27.3 Die iPhone SDK-Dokumentation verwenden

Schauen Sie sich nochmals das erste Beispiel an, bei dem der Build nicht funktioniert hat. Es geht um den Tippfehler, durch den wir versucht haben, auf die nicht existierende Eigenschaft `txt` einer `UITextView` zuzugreifen. Was ist, wenn Sie wirklich davon ausgegangen sind, dass die Eigenschaft den Namen `txt` besitzt? Oder wenn Sie nicht wussten, dass es sich eigentlich um eine Eigenschaft handelt, und daher versucht haben, eine Methode `setText:` zu nutzen, die es nicht gibt?

In solch einem Fall wäre das der richtige Moment, einen Blick in die Dokumentation zu werfen. Denn vielleicht der beste Weg, Fehler zu vermeiden, ist, zu wissen, was Sie tun.

Wie schon erwähnt, kann das Syntax-Highlighting einen Hinweis geben. Haben Sie eine gültige Methode, Funktion oder Eigenschaft eingetippt, färbt Xcode es entsprechend Ihrer Editor-Einstellungen. Wenn Sie nicht mehr weiterwissen, können Sie auch versuchen, die Code-Completion-Features von Xcode zu nutzen. Haben Sie ein paar Zeichen eines Klassen-, Methoden-, Funktions- oder Eigenschaftsnamen eingegeben, versucht Xcode, ihn automatisch zu vervollständigen, wenn Sie mit der Eingabe stoppen. Ist die vorgeschlagene Vervollständigung nicht das, was Sie meinten, können Sie die nächste mit `C` anzeigen oder per `ESC` eine Liste aller Möglichkeiten anzeigen.

Wenn Sie mehr Informationen benötigen als eine Code-Vervollständigung anbieten kann, stellt Xcode Ihnen die Dokumentation des gesamten iPhone SDK in einem Documentation Viewer-Fenster zur Verfügung. Dieses können Sie über das Hilfemenü oder durch das Tastenkürzel `DE?` erreichen. Wie in Abbildung 27.4 zu sehen, bietet der Documentation Viewer die Dokumentation für Ihre installierten SDKs an – normalerweise Mac OS X, Java und eine oder mehrere Versionen des iPhone SDK sowie für Xcode selbst. Diese „Doc Sets“ werden im schmalen rechten Bereich aufgeführt. Öffnen Sie dieses Fenster das erste Mal, finden Sie vielleicht auch noch einen Button `GET` oder `SUBSCRIBE` neben einem Doc Set. Wenn Sie ausreichend Bandbreite haben, ist das Subskribieren vielleicht die beste Option, denn damit haben Sie immer lokale Kopien der aktuellsten Dokumentation. Ist nicht das aktuelle Doc Set lokal bei Ihnen installiert, werden Sie beim Suchen im Viewer nach dem Anklicken des Suchergebnisses manchmal in Ihrem Standard-Webbrowser landen, in dem eine Dokumentationsseite von Apples Developer-Website angezeigt wird (<http://developer.apple.com/>).

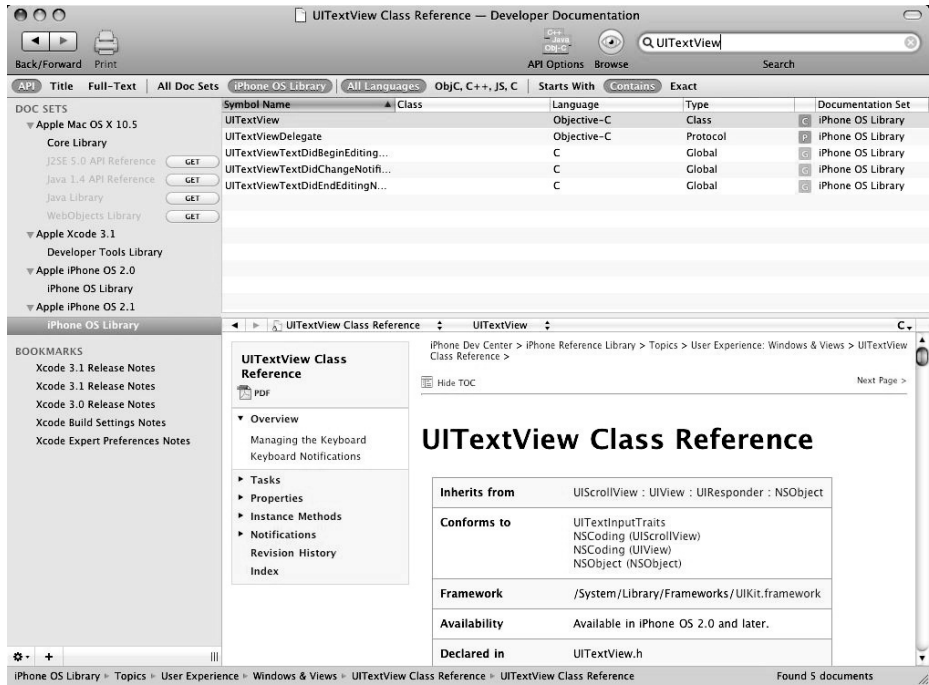


Abbildung 27.4: Documentation Viewer-Fenster von Xcode

Im Fall von Abbildung 27.4 haben wir nach der Dokumentation für die Klasse UITextView gesucht, indem wir ihren Namen in das Suchfeld in der oberen rechten Ecke der Toolbar eingegeben haben. Darunter finden Sie einen Bereich, der das Suchverhalten beeinflusst. Das linke Segment bestimmt, ob Sie nach dem Begriff in APIs suchen wollen (Klassen-, Methoden- oder Funktionsnamen, Konstanten und so weiter), in Dateinamen oder im vollständigen Text der Dokumentation. Das nächste Segment bestimmt, ob Sie die Suche auf alle Doc Sets anwenden wollen oder auf das im linken Bereich ausgewählte. Danach legt ein weiteres Paar Buttons fest, ob die Suche auf alle Sprachen oder nur auf Objective-C, C++, JavaScript und C angewandt werden soll. Schließlich bestimmt der letzte Schalter, ob die Abfrage Dokumente finden soll, die mit Ihrem Suchbegriff beginnen, ihn enthalten oder ihm genau entsprechen.

Die Tabelle im oberen Bereich des Fensters zeigt die Suchergebnisse an. Eine Suche nach APIs mit dem Suchbegriff UITextView findet damit die Klassen UITextView und UITextViewDelegate sowie eine Reihe von Konstanten, die für Benachrichtigungen verwendet werden. Das oberste Ergebnis der Liste wird automatisch ausgewählt und das dazugehörige Dokument im darunterliegenden HTML-Viewer angezeigt. Würden wir also tatsächlich nicht wissen, wie man den Text einer

UITextView setzt, könnten wir die Beschreibung der Klasse bis zum Abschnitt Tasks herunterscrollen. Dort finden wir einen Link auf die Eigenschaft `text`, über den wir zu einer Beschreibung sowie einer Definition der Eigenschaft weiter unten in der Datei gelangen.

Es gibt eine Reihe von Kürzeln, die Sie nutzen können, um aus Ihrem Code schnell auf die Dokumentation zuzugreifen. Ein Doppelklick in den Quellcode bei gedrückter Optionstaste zeigt den Documentation Viewer an, wobei der angeklickte Begriff schon im Suchfenster eingetragen ist (Sie müssen den Text vorher nicht markiert haben – Xcode nutzt den umgebenden Whitespace, um herauszufinden, welchen Begriff Sie angeklickt haben). Das ist sehr praktisch, wenn man nach Dokumentationen zu unbekannten Methoden oder Klassen sucht. Aber was, wenn Sie eine Objektreferenz haben und nicht genau wissen, aus welcher Klasse die Instanz wirklich besteht? In diesem Fall können Sie die Referenz mit der rechten Maustaste (oder mit der linken bei gedrückter `Ctrl`-Taste) anklicken, um ein Kontextmenü aufzurufen. Darin findet sich (unter anderem) eine Reihe von Suchmöglichkeiten. Die vielleicht hilfreichste ist „Jump To Definition“. Damit werden Sie zu der Stelle geleitet, an der das Objekt definiert ist – meist eine Header-Datei –, egal ob in Ihrem Code oder in einem Framework, das Sie importiert haben. Die Definition deklariert die Klasse des Objekts, und dort können Sie auch wieder Option+Doppelklick nutzen, um nach deren Dokumentation zu suchen.

Eine weitere Möglichkeit, die Dokumentation zu nutzen, ist die Verwendung des Research Assistant, einem Fenster, das über den Eintrag „Show Research Assistant“ im Hilfemenü (CD?) angezeigt wird. Dieses Fenster (siehe Abbildung 27.5) sucht nach dem markierten Text und bietet einen Überblick über die Dokumentation dazu an, wenn es sich um eine Klasse, Methode, C-Funktion oder ein anderes dokumentiertes Element handelt. Dazu gehört auch die Definition, eine Zusammenfassung (meist der erste Absatz der Dokumentation), die Verfügbarkeit im SDK, Beispielcode und einiges mehr. Blaue Elemente in diesem Fenster sind Hyperlinks, die Sie zur Dokumentation, zu Beispielcodeprojekten oder Definitionen in Header-Dateien führen. Anstatt den Research Assistant immer nur bei Bedarf zu öffnen, können Sie ihn auch dauerhaft offen lassen, wenn Sie im Code arbeiten und beobachten, wie er sich immer dann aktualisiert, wenn Sie den Cursor über ein Element führen, das er erkennt.

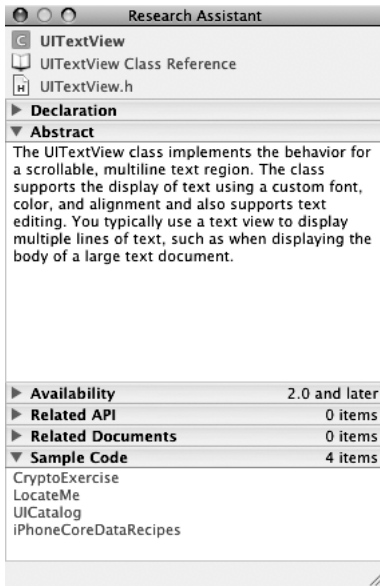


Abbildung 27.5: Research Assistant von Xcode

27.4 Fehler im Interface Builder verstehen und beheben

Bisher haben wir uns ausschließlich um Probleme mit dem Code gekümmert, was für Programmierer ganz natürlich ist. Bei der Entwicklung in Cocoa passieren aber auch schnell mal Fehler im Interface Builder.

Das vielleicht einfachste IB-Problem ist, Änderungen am GUI vorzunehmen (oder an anderen Inhalten, die aus einer .nib-Datei geladen werden), ohne die .nib-Datei im IB zu sichern. Wenn Sie dann gegen die alte Version der .nib-Datei bauen und das Programm laufen lassen, können GUI-Elemente fehlen, oder Sie erhalten nicht alle Benachrichtigungen über UI-Events. Vielleicht sehen Sie auch keine Änderungen am GUI, die Ihr Code eigentlich auslösen sollte, wie zum Beispiel das Füllen von Text-Views. In solchen Fällen müssen Sie nur schnell zum IB zurückwechseln und sich den Schließen-Button des Nib anschauen – hat er einen Punkt in der Mitte, sind die Änderungen nicht gesichert.

Ähnliche Probleme mit dem IB können dadurch verursacht werden, dass Sie Ihre Komponenten falsch miteinander verbinden. Lassen Sie uns zwei sehr häufige Fälle zeigen. Der eine ist, eine View hinzuzufügen und zu vergessen, deren Verbindungen herzustellen. Wenn Sie diese View im Code ansprechen wollen – zum Beispiel um den Text einer UITextView zu setzen –, deklarieren Sie normalerweise eine Referenz darauf in einer

Ihrer Header-Dateien und versehen die Deklaration mit dem Schlüsselwort `IBOutlet`. Im IB ändern Sie dann die Klasse von File's Owner so ab, dass sie der Referenz entspricht. Durch diese beiden Schritte können Sie nun über den IB eine Verbindung von der View zum Outlet herstellen. Vergessen Sie dies aber, wird durch Aufrufe aus Ihrem Code mit der View gar nichts angestellt, da keine Verbindung zwischen der Referenz im Code und der View hergestellt wurde, als die View aus der .nib-Datei geladen wurde. Das Gemeine daran ist, dass dies ein ganz stiller Fehler ist. Der Versuch, Methoden der nicht initialisierten Referenz aufzurufen oder ihre Eigenschaften zu setzen, liefert keinen Fehler und führt auch zu keinem Absturz. Es passiert einfach nur nichts.

Ein ähnliches Problem entsteht, wenn Sie vergessen, die `delegate`-Verbindung einer View mit der Klasse herzustellen, die die `Delegate`-Methode implementiert, oder die `dataSource` einer `UITableView` auf die Klasse zu setzen, die `UITableViewDataSource` implementiert. Sie erhalten dann eine Tabelle, die keine Daten anzeigt oder nicht auf eine Eingabe reagiert, auch wenn Sie sicher sind, dass Sie alle `Delegate`- und `Table Source`-Methoden implementiert haben.

Ein weiterer häufiger IB-Fehler führt glücklicherweise zu einer deutlichen Fehlermeldung. Stellen Sie sich vor, Sie haben einen eigenen View Controller und eine eigene View entwickelt und wollen diese nun im Code nutzbar machen, zum Beispiel indem Sie den View Controller auf den Stack eines `UINavigationController` bringen. Dazu allozieren Sie normalerweise den View Controller und initialisieren ihn mit einer Methode wie `initWithNibName:bundle:`. Ein sehr häufiger Fehler führt dazu, dass Ihre Anwendung beim ersten Laden der View abstürzt.

Wie können Sie den Fehler identifizieren? Öffnen Sie die Debugger-Konsole von Xcode über das Run-Menü (oder über das Tastenkürzel `DBR`) und achten Sie auf die folgende Fehlermeldung, die direkt vor der Ausgabe einer Menge von Callstack-Adressen durch den `gdb`-Debugger erscheint:

```
2008-10-10 15:53:22.149 FilesystemExplorer[7959:20b]
    Terminating app due to
    uncaught exception 'NSInternalInconsistencyException' ,
    reason: '-[UIViewController
loadView] loaded the "FileOverviewView" nib but no view was set.'
```

Was ist hier geschehen? Ganz einfach – Sie haben die eigene View im IB erstellt und vielleicht auch die Klasse von File's Owner angepasst, aber vergessen, die `view` von File's Owner mit der eigenen View in der .nib-Datei zu verbinden. Der View Controller braucht eine gültige View, die er im GUI anzeigen kann. Ohne diese Verbindung findet er aber keine. Der entschei-

dende Schritt beim Herstellen der Verbindung ist in Abbildung 27.6 zu sehen.² Wenn Sie das nicht machen, bevor Sie Ihre Anwendung ausführen, werden Sie sich nach dem Absturz bestimmt daran erinnern.

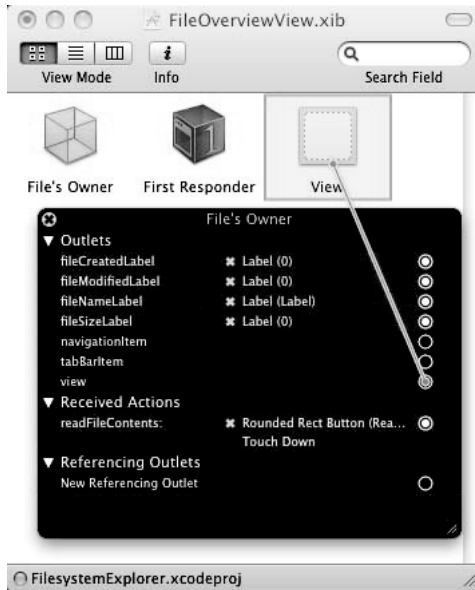


Abbildung 27.6: Einen View Controller mit seiner View verbinden

27.5 Debuggen

Auch wenn nichts abstürzt, werden Sie sich oft genug wundern, warum sich Ihre Anwendung falsch verhält. Der Code lässt sich kompilieren, die Verbindungen im IB sind korrekt, aber die Anwendungslogik ist irgendwie falsch. In dem Fall müssen Sie sich Ihren Code beim Ablauf genauer anschauen und herausfinden, warum er sich so verhält, wie er sich verhält.

Protokollieren an die Standardausgabe

Der Programmabsturz im vorigen Abschnitt hat seinen Fehler an der „Standardausgabe“ (auch bekannt als *stdout*) des Systems protokolliert, die wir uns im Konsolenfenster von Xcode angeschaut haben. Da diese Protokollmeldungen an die Standardausgabe des Systems gehen,

² Wie Sie aus der Fehlermeldung und anhand der Abbildung erkennen können, haben wir eine der eigenen Views aus der FilesystemExplorer-Anwendung in Kapitel 8, *Datei-I/O*, auf Seite 143 absichtlich zerstört und wiederhergestellt, um die Fehlermeldung und die Abbildung zu erzeugen.

können sie auch in der Console.app-Anwendung (zu finden in /Applications/Utilities) betrachtet werden, wo sie neben den Protokollmeldungen aller anderen Anwendungen und Systemprozesse erscheinen. Glücklicherweise bietet Console.app eine Filter in der oberen rechten Ecke an, so dass Sie die Ausgabe auf die aktuelle Anwendung einschränken können (indem Sie deren Namen in die Filterbox eingeben, da vor allen Protokollmeldungen der Prozessname steht).

Es kann gut sein, dass Sie als erfahrener Entwickler mit dem Schreiben von Text an stdout über die Funktion `printf()` von C, `print()` von Python oder `System.out.println()` von Java vertraut sind. Und es kann gut sein, dass Sie das auch bei Ihren iPhone-Apps machen wollen. Das Debuggen von sich seltsam verhaltendem Code durch die Ausgabe einer Reihe von Meldungen des Typs „Ich mache gerade dies oder das.“ mag zwar die Puristen vergrätzen, ist aber eine Standardtechnik vieler Entwickler.

In Objective-C können Sie die Funktion³ `NSLog()` nutzen, um Meldungen an stdout zu schicken. `NSLog()` erwartet eine `NSString`-Referenz, gefolgt von null oder mehr zusätzlichen Variablen. In ihrer einfachsten Form können Sie einen einzelnen String protokollieren:

```
NSLog(@"Hallo iPhone");
```

Damit wird der String `Hallo iPhone` an stdout geschickt, wo Sie ihn im Konsolenfenster sehen können.

Bei der fortgeschritteneren und praktischeren Version von `NSLog()` wird der String als Format-String genutzt. In diesem Fall nutzen Sie spezielle Zeichenfolgen, mit denen angegeben wird, wo die Werte von Variablen in den String einzufügen sind, bevor er an stdout geschickt wird. Wenn Sie zum Beispiel in einer Methode sind, die eine `NSString`-Variable namens `userName` besitzt, können Sie deren Wert mit der folgenden Anweisung protokollieren:

```
NSLog(@"Benutzername ist %@", userName);
```

Die Zeichenfolge `%@` wird genutzt, um eine String-Repräsentation eines beliebigen Objective-C-Werts auszugeben. Technisch gesehen, wird damit der Wert der `NSObject`-Methode `descriptionWithLocale:` ausgelesen (oder `description:`, wenn die Methode nicht definiert ist), die normalerweise von Cocoa-Klassen implementiert wird, um eine String-Darstellung anzubieten, die beim Debuggen sehr nützlich sein kann. Natürlich gibt ein `NSString` nur seinen eigenen Wert zurück, den wir hier nutzen, um den `userName`-String in den Format-String einzutragen.

3 Dies ist eine prozedurale *Funktion* und keine objektorientierte *Methode*.

Eine vollständige Liste der ersetzbaren Zeichenfolgen finden Sie im Abschnitt „String Format Specifiers“ von Apples *String Programming Guide for Cocoa*, den Sie wiederum erhalten, indem Sie nach der Dokumentation zur Foundation-Funktion `NSLogv( )` suchen (die Low-Level-Funktion, die von `NSLog( )` aufgerufen wird). Einige der gebräuchlichsten Format-Strings sehen sie hier:

Folge	Bedeutung
<code>%@</code>	String-Repräsentation eines Objekts, Ergebnis des Aufrufs von <code>descriptionWithLocale:</code> oder, falls nicht vorhanden, von <code>description:</code> .
<code>%d</code> , <code>%D</code> , <code>%i</code>	Vorzeichenbehafteter 32-Bit-Integer-Wert (<code>long</code>).
<code>%u</code> , <code>%U</code>	Vorzeichenloser 32-Bit Integer-Wert.
<code>%x</code>	Vorzeichenloser 32-Bit-Integer-Wert als Hexadezimalwert.
<code>%f</code>	64-Bit-Gleitkommawert (<code>float</code>).
<code>%s</code>	Ein String, der durch das Interpretieren des Werts als Adresse eines null-terminierten Strings aus 8-Bit-Zeichen ermittelt wird (ein klassischer „C-String“ in Systemkodierung).
<code>%S</code>	Ein String, der durch das Interpretieren des Werts als Adresse eines null-terminierten Strings aus 16-Bit-Unicode-Zeichen interpretiert wird.

Lassen Sie uns anschauen, wie man diese Funktion nutzen kann, um eine weitere absichtlich zerstörte Anwendung zu analysieren. Der `PathologicalPrimeCounter` ist eine App, die es dem Benutzer erlaubt, einen maximalen Wert anzugeben, bis zu dem alle Primzahlen gefunden werden. Eine funktionierende Version dieser Anwendung sehen Sie in Abbildung 27.7. Dabei handelt es sich um eine Anwendung mit einer einzelnen View mit einem `UITextField`, das mit dem View Controller als `countToField` verbunden ist, einer `UITextView`, verbunden als `primesView`, und einem `UIButton`, der `handleGoTapped:` aufruft, wenn er angetippt wird.

Um zu sehen, wie uns `NSLog( )` hier helfen kann, wollen wir es nutzen, um einen logischen Fehler zu debuggen. Die Beispielanwendung ruft `countPrimesToSelectedValue` auf, wenn der Button angetippt wird oder der Anwender nach der Eingabe eines maximalen Werts `Return` antippt. Hier eine erste, *falsche* Implementierung dieser Methode:



Abbildung 27.7: Primzahlenanwendung

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
- (void) countPrimesToSelectedValue {
    int maxPrime = [countToField.text intValue];
    if (maxPrime < 1)
        return;
    // Prüfe für jede ganze Zahl bis maxPrime jeden Teiler bis
    // maxPrime/2+1. Es gibt effizientere Algorithmen,
    // aber dieser ist leicht verständlich.
    for (int i=2; i<maxPrime; i++) {
        int maxDivisor = i/2 + 1;
        BOOL isPrime = YES;
        for (int j=1; j<maxDivisor; j++) {
            if (i%j== 0) {
                isPrime = NO;
                break;
            }
        }
        if (isPrime) {
            NSLog ("%d ist prim" , i);
        }
    }
}
```

In dieser Version haben wir uns noch nicht darum gekümmert, die Text-View mit Leben zu füllen, sondern wir geben die Primzahlen einfach an der Standardausgabe aus. Jede entdeckte Primzahl wird mit folgender Anweisung ausgegeben:

```
NSLog(@"%d ist prim" , i);
```

In diesem Fall gibt es eine Substitutionsfolge %d, deren Wert vom Argument i bereitgestellt wird.

Öffnen Sie also das Konsolenfenster mit --, klicken Sie auf BUILD AND RUN, geben Sie eine maximale Zahl ein, tippen Sie auf Los und ...

Nichts passiert.

Was ist hier geschehen? Läuft die Schleife nicht? Werden Primzahlen fälschlicherweise zurückgewiesen? Lassen Sie uns die zweite Hypothese prüfen, indem wir eine Log-Anweisung einbauen (direkt vor `isPrime = NO`), um alle potenziellen Primzahlen i anzuzeigen, die zurückgewiesen wurden, weil sie sich durch eine kleinere Zahl j teilen lassen. Wir werden einen Format-String verwenden, der beide Werte ausgibt:

```
NSLog(@"%d ist nicht prim, sondern teilbar durch %d" ,i,j);
```

Wie Sie sehen, reserviert der Format-String Platz für zwei vorzeichen-behaftete Integer-Werte, die dann in den folgenden beiden Argumenten bereitgestellt werden.

Führen Sie das Programm aus, wird der Fehler durch die Ausgabe offensichtlich.⁴

```
PathologicalPrimeCounter[13308:20b] 2 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 3 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 4 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 5 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 6 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 7 ist nicht prim,
    sondern teilbar durch 1
```

⁴ In der echten Ausgabe findet sich am Anfang jeder Zeile noch ein Zeitstempel. Wir haben ihn hier entfernt, damit die Ausgabezeilen nicht umbrechen.


```

PathologicalPrimeCounter[13308:20b] 8 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 9 ist nicht prim,
    sondern teilbar durch 1
PathologicalPrimeCounter[13308:20b] 10 ist nicht prim,
    sondern teilbar durch 1

```

Bei jeder potenziellen Primzahl lassen wir die Schleife mit den möglichen Divisoren j bei 1 beginnen. Aber da jede Ganzzahl durch 1 teilbar ist, schließen wir fälschlicherweise, dass jeder Wert von i keine Primzahl ist. Wir müssen j also einfach bei 2 beginnen lassen.

Verwenden des Debuggers

Das NSLog()-basierte Debuggen mag für viele von uns Gewohnheit sein, aber es hat seine Grenzen. Im vorigen Fall hat es funktioniert, weil wir ungefähr wussten, was passiert, und damit abschätzen konnten, wo wir unsere NSLog()-Anweisung platzieren. Aber was machen Sie, wenn Sie überhaupt nicht wissen, was Ihr Code da tut?

Lassen Sie uns dieses Beispiel ein bisschen schwieriger machen. Statt die innere Schleife so zu korrigieren, dass j anfänglich den Wert 2 hat, wollen wir die Dinge noch schlimmer machen und ihn in 0 umändern. Starten Sie das Programm, geben Sie einen Wert für die maximale Zahl ein, tippen Sie auf Los ... und Sie erhalten nicht einmal eine Log-Meldung im Konsolenfenster. Stattdessen erhalten Sie eine Meldung vom gdb, dem GNU-Debugger, die in etwa so aussieht:

```

Loading program into debugger...
GNU gdb 6.3.50-20050815 (Apple version gdb-962) (Sat Jul 26 08:14:40 UTC 2008)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.

```

Was ist passiert? Nun, Ihre Anwendung ist abgestürzt, und der Debugger hat die Kontrolle übernommen (solange Sie das Ganze über Xcode gestartet haben – sind Sie direkt über den Simulator oder das iPhone gegangen, wird die App schwarz und kehrt zum Home-Bildschirm zurück). Es bleibt nur das Konsolenfenster, in das Sie Befehle für den gdb eingeben können. Sind Sie mit dem gdb vertraut, dürfen Sie hier gern tätig werden, aber die meisten von uns werden mit dem Debugger-GUI von Xcode produktiver sein.

In Xcode öffnen Sie den Debugger über das Run-Menü oder über das Tastenkürzel **DBY**. In Abbildung 27.8 sehen Sie das Debugger-Fenster. Oben links findet sich der aktuelle Call Stack für den aktiven Thread. Scrollen Sie herunter, sehen Sie, dass der Call Stack bis zur Funktion `main()` reicht, die Xcode für Sie in `main.m` erzeugt hat. Methoden- und Funktionsaufrufe, für die der Quellcode verfügbar ist – also im Allgemeinen Ihr Code und nicht die diversen Systembibliotheken –, werden in Schwarz angezeigt, während der Rest in Grau zu sehen ist. Klicken Sie auf eine der schwarzen Textzeilen, um den Quellcode im unteren Bereich anzuzeigen (klicken Sie auf eine graue Zeile, sehen Sie den Assembler-Code). Im Quellcodebereich zeigt der rote Pfeil auf der linken Seite an, wo die Ausführung der Anwendung gestoppt wurde.

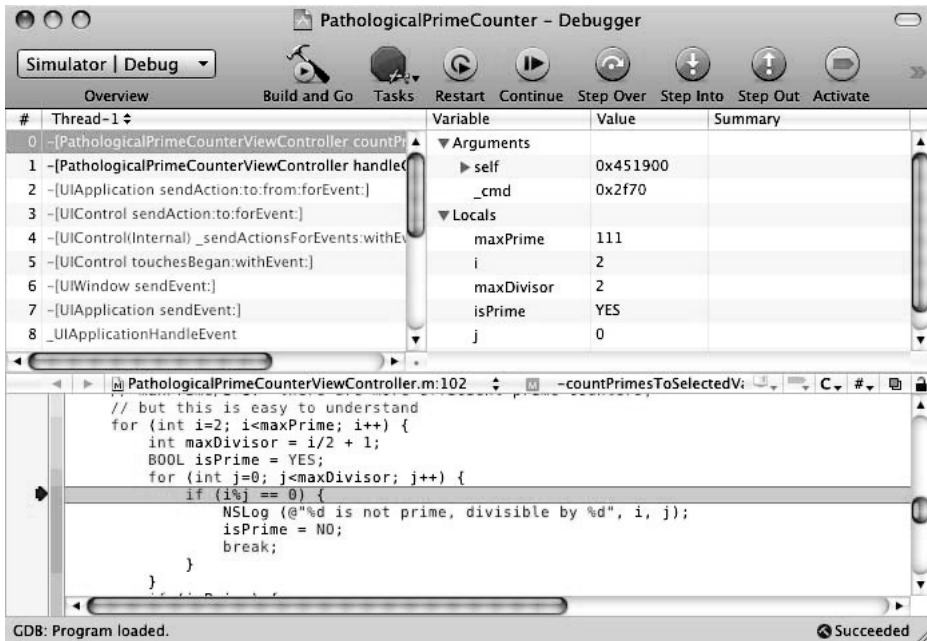


Abbildung 27.8: Der Debugger von Xcode

Oben rechts zeigt der Debugger eine Tabelle mit Variablen und ihren aktuellen Werten an. Schauen Sie sich die kritische Codezeile an:

```
if (i%j == 0)
```

Dann werfen Sie einen Blick auf die Variablen (Sie können den Wert der Variablen auch anzeigen, indem Sie mit der Maus über den Quellcode gehen, siehe Abbildung 27.9). Es gibt nur zwei Variablen, die in dieser

Zeile von Bedeutung sind: *i* hat den Wert 1, *j* den Wert 0. Was stimmt also an `1%0` nicht? Das Problem ist, dass es sich um eine Division durch Null handelt, was immer ein Garant für einen Absturz ist.

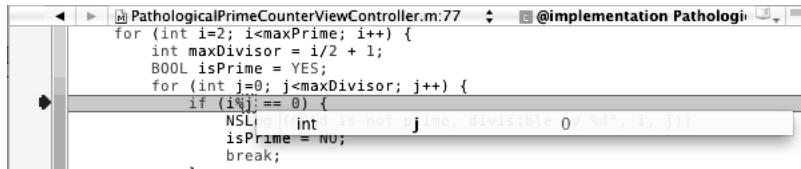


Abbildung 27.9: Variablenwerte im Quellcode anzeigen

Sie können sich das sogar bestätigen lassen, indem Sie die Anwendung weiterlaufen lassen, denn der Debugger hat den Status der Anwendung direkt vor dem Absturz gesichert. Klicken Sie auf den Button CONTINUE der Debugger-Toolbar, sehen Sie die Meldung am unteren Ende des Debuggers: Program received signal: „EXC_ARITHMETIC“.

Sie können mit dem Debugger noch viel mehr machen als nur zu beobachten, wie Ihre Anwendung abstürzt. Durchlaufen Sie doch einmal Ihren Code Schritt für Schritt und setzen Sie dabei die Werte von Variablen zurück: Stoppen Sie die Anwendung, wenn sie noch läuft, und wechseln Sie zu `PathologicalPrimeCounterViewController.m`. Klicken Sie in den grauen Bereich links neben der Anweisung `BOOL isPrime = YES;`, um einen Breakpoint zu erzeugen (siehe Abbildung 27.10).

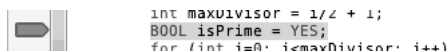


Abbildung 27.10: Einen Breakpoint hinzufügen

Lassen Sie die Anwendung per BUILD AND RUN erneut laufen. Mit dem gesetzten Breakpoint wird die Anwendung stoppen, sobald die Ausführung diesen Breakpoint erreicht. Damit haben Sie die Chance, den Status der Anwendung direkt vor dem Absturz zu begutachten. Jetzt zeigt die Liste der Variablen, dass *i* den Wert 2 hat, *isPrime* den Wert YES und *j* nicht im Gültigkeitsbereich liegt.

An dieser Stelle könnten Sie auf den Button CONTINUE klicken, um direkt in den Absturz zu laufen, oder die drei Schritt-Buttons nutzen, um sich an den Absturz heranzuarbeiten. Jeder dieser Buttons ermöglicht es Ihnen, eine bestimmte Menge von Schritten weiterzugehen. STEP INTO führt die nächste Anweisung aus. Handelt es sich dabei um einen Methoden- oder Funktionsaufruf oder eine andere komplexe Anweisung, wird die Ausführung in dieser Anweisung gestoppt (wobei

Call Stack und Quellcodeanzeige bei Bedarf aktualisiert werden). Bei einer Folge mehrerer Aufrufe sehen Sie so, wer wen aufruft, aber es kann auch aufwendig sein, so viele Ebenen hinabzusteigen. STEP OUT ist das entsprechende Gegenstück dazu: Sind Sie in einem Methodenaufruf, wird die Ausführung fortgeführt, bis die Methode zurückspringt. Und wenn Sie sich nicht für die Innereien der aufzurufenden Methode interessieren, können Sie STEP OVER nutzen, um den Methodenaufruf durchzuführen und direkt danach zu stoppen.

In diesem Fall klicken Sie auf STEP INTO, um weiterzuspringen. Damit kommen Sie in die for-Anweisung. `j` ist jetzt im Gültigkeitsbereich, aber bis zum nächsten Schritt noch nicht initialisiert, also hat es einen zufälligen Wert. Mit einem weiteren STEP INTO wird dieser Wert auf 0 gesetzt.

Jetzt stehen Sie kurz vor dem Absturz. Ein weiterer Schritt, und Sie sind in der Modulo-Anweisung, die durch null teilen und zum Absturz führen wird. Aber wissen Sie was? Umgehen Sie den Absturz. In der Variablentabelle klicken Sie doppelt auf den Wert von `j`. Er ist editierbar. Setzen Sie ihn auf 2 und klicken Sie erneut auf STEP INTO. Durch das Ändern des Werts zur Laufzeit können Sie ohne einen Absturz weitermachen!

Laufen Sie das nächste Mal durch die Schleife, wird das natürlich wieder zu einem Absturz führen, aber Sie sehen jetzt, was Sie im Code ändern müssen, damit er funktioniert. Sie haben gezeigt, dass ein Vermeiden von 0 als Wert für `j` den Absturz umgeht. Die Korrektur ist also nur ein BUILD AND RUN entfernt.⁵

Es ist sehr hilfreich, Ihren Code an Breakpoints zu stoppen, die Werte der Variablen zu begutachten und diese sogar ändern zu können. Damit lässt sich herausfinden, welche Werte für Ihre Anwendung korrekt sind.

⁵ Beim Entwickeln von Anwendungen für Mac OS X können Sie manchmal den Fix-Befehl von Xcode nutzen. Damit wird in eine laufende Anwendung Code eingefügt, aber das scheint nicht mit dem iPhone-Simulator zu funktionieren (zumindest nicht, als wir es versucht haben).

27.6 Über-freigegebene „Zombie“-Objekten finden

Im vorigen Abschnitt haben Sie gesehen, wie Sie mit dem Debugger einen Division-durch-Null-Fehler finden und beheben können. Die meisten Abstürze sind aber nicht so offensichtlich. Der weitaus häufigste Grund für einen Absturz ist ein Fehler des Typs **EXC_BAD_ACCESS**, der immer von einem Zeigerfehler herrührt, zum Beispiel durch eine vergessene Initialisierung eines Zeigers, der Verwendung eines referenzierten numerischen Werts (statt des Zeigers) als Adresse und so weiter.

Ein ganz einfaches Beispiel ist, einem Zeiger nichts zuzuweisen:

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
NSString *nullReferenceString;  
NSLog (nullReferenceString);
```

Da `nullReferenceString` nie alloziert und zugewiesen wird, ist die Referenz nicht nutzbar, und der erste Versuch, darauf zuzugreifen, führt zu einem Absturz mit dem Fehler `EXC_BAD_ACCESS`. Glücklicherweise lässt sich das mit dem Debugger sehr leicht herausfinden (siehe Abbildung 27.11).

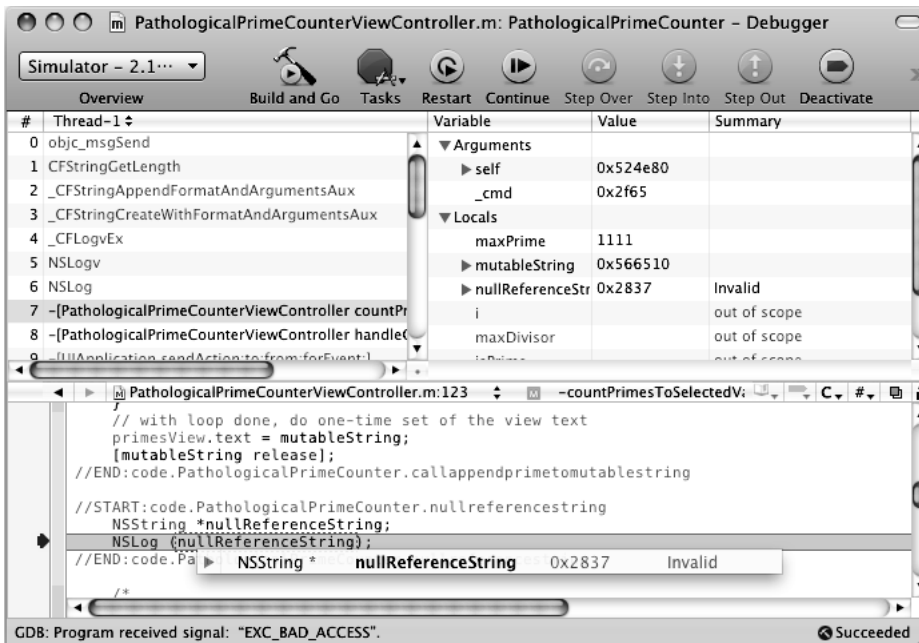


Abbildung 27.11: Ein Absturz durch einen EXC_BAD_ACCESS, verursacht durch einen kaputten Zeiger

Wie Sie sehen können, hat der Debugger beim problematischen Aufruf von `NSLog()` gestoppt. Geht man mit der Maus über den `nullReference-String`, sieht man, dass dieser „invalid“ ist, was auch in der Variablenliste im oberen rechten Bereich angezeigt wird.

Das lässt sich sehr leicht debuggen, aber schauen Sie sich diesen Versuch am Ende der Methode `countPrimesToSelectedValue` an, um eine Meldung zu protokollieren:

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
NSString *headerString = [NSString stringWithFormat:
    @"Primzahlen gezählt bis %@", countToField.text];
NSLog (headerString);
[headerString release];
```

Das sieht auf den ersten Blick ausgesprochen harmlos aus, führt aber beim Laufenlassen zu einem Absturz. Schlimmer noch: Eine Analyse des Absturzes mit dem Debugger zeigt kein offensichtliches Problem. Keine Zeile des Anwendungscodes scheint am Stack des abgestürzten Threads beteiligt zu sein, da der Absturz entsteht, wenn `NSPopAutoreleasePool` versucht, eine Message an ein Objekt zu senden ... leider wissen wir nicht, welches Objekt oder welche Methode hier nun schuld ist.

Wir werden zunächst die Ursache aufdecken und Ihnen dann ein Tool vorstellen, mit dem Sie diese Art von Absturz verfolgen können. Das Problem ist eine *Über-Freigabe* (Over Release) eines Objekts. Die Klassenmethode `stringWithFormat:` erzeugt einen Autorelease-`NSString`, daher ist das darauffolgende `[headerString release]` eigentlich nicht korrekt. Was passiert? Unser `release` verringert den Referenzzähler auf null und gibt das Objekt frei, aber der Autorelease-Pool versucht später, das Objekt erneut freizugeben, und schickt eine weitere `release`-Nachricht an das Objekt, das schon weg und dessen Referenz nicht mehr nutzbar ist. Also müssen wir entweder unser `release` weglassen oder `stringWithFormat:` durch das nicht per Autorelease freigebende Paar aus `alloc` und `initWithFormat:` ersetzen.

Das Problem ist: Selbst wenn Sie vermuten, dass es irgendwo eine Über-Freigabe gibt, kann es sehr schwierig sein, herauszufinden, *welches* Objekt davon betroffen ist.

Das ist ein Job für *Zombies*, genauer gesagt für `NSZombies`. In Cocoa sind *Zombies* Objekte, die zwar schon tot sind, aber immer noch herumirren und für Ärger sorgen. Glücklicherweise gibt es in Xcode Tools, um *Zombies* zu finden und zu jagen. Wir können eine Build-Einstellung setzen, die Objekte nicht freigibt, wenn ihr Referenzzähler auf 0 fällt, sondern sie

in Objekte vom Typ `NSZombie` umwandelt. Der Zweck dieser Klasse ist, jeden Aufruf einer Instanz von ihr zu protokollieren, da dies bedeutet, dass der Code versucht, eine Methode eines toten Objekts aufzurufen.

Klappen Sie in Ihrem Projektfenster unter „Groups & Files“ das Element „Executables“ auf und markieren Sie die Anwendung `PathologicalPrimeCounter`. Öffnen Sie deren Inspector mit GET INFO (DI oder Info-Button in der Toolbar) und wählen Sie den Tab ARGUMENTS. In der unteren Hälfte „Variables to be set in the environment“ nutzen Sie den Button +, um zwei Variablen `NSZombieEnabled` und `MallocStackLogging` hinzuzufügen. Setzen Sie beide auf den Wert YES, wie in Abbildung 27.12 gezeigt.

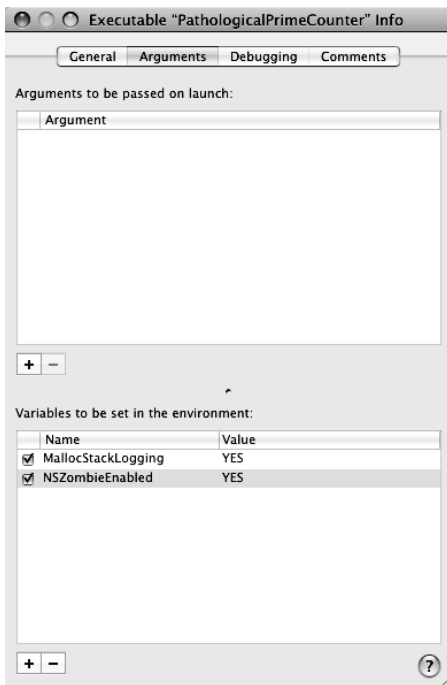


Abbildung 27.12: `NSZombieEnabled` in den Umgebungsvariablen der ausführbaren Datei aktivieren

Öffnen Sie wieder die Konsole (DR) und starten Sie die Anwendung erneut. Sie werden ein paar Meldungen erhalten, in denen es darum geht, dass „malloc stack logs“ geschrieben werden – normalerweise in Ihr `/tmp`-Verzeichnis:⁶

⇒ `launchd(2018) malloc: stack logs being written into /tmp/stack-logs.2018.launchd.odBbpt`

⁶ Wir haben die Ausgabe umbrochen, damit sie ins Buch passt.

Lassen Sie jetzt ein paar Primzahlen ermitteln, um den Absturz auszulösen. Dieses Mal erhalten Sie direkt vor dem Absturz eine Nachricht vom Zombie:

```
⇒ 2009-03-10 16:56:01.128 PathologicalPrimeCounter[2018:20b]
-[CFString release]: message sent to deallocated instance 0x591f40
```

Jetzt haben wir etwas, mit dem wir weitermachen können. Wir kennen die Nachricht, die an das über-freigegebene Objekt gesendet wurde (ein weiteres release), die Klasse des über-freigegebenen Objekts (CFString, das Core Foundation-Äquivalent zu NSString) und sogar die Adresse des Zombies. Damit kennen wir das Ende der Geschichte, aber nicht den Anfang. Wie kommen wir zurück an die Stelle, an der alles begann? Hier kommt der malloc-Trace ins Spiel. Mit der Adresse des Zombies können wir herausfinden, wo er allokiert wurde.

Geben Sie im Konsolenfenster nach dem (gdb) Folgendes ein:⁷

```
shell malloc_history pid address
```

wobei pid die Prozess-ID ist, die in Klammern in der Log-Meldung von malloc: angegeben ist, und address die Adresse des Zombies, wie sie in der Meldung „deallocated instance“ steht. Mit der eben gezeigten Log-Meldung sieht Ihre Eingabe an der Konsole so aus:

```
⇐ (gdb) shell malloc_history 2018 0x591f40
```

```
⇒
```

```
Call [2] [arg=32]: thread_a024c720 |0x1 | start | main | UIApplicationMain |
-[UIApplication _run] | GSEventRun | GSEventRunModal | CFRunLoopRunInMode |
CFRunLoopRunSpecific | PurpleEventTimerCallBack | SendEvent |
 UIApplicationHandleEvent | -[UIApplication sendEvent:] |
-[UIWindow sendEvent:] |
-[UIControl touchesBegan:withEvent:] |
-[UIControl(Internal) _sendActionsForEvents:withEvent:] |
-[UIControl sendAction:to:forEvent:] |
-[UIApplication sendAction:to:from:forEvent:] |
-[PathologicalPrimeCounterViewController handleGoTapped:] |
-[PathologicalPrimeCounterViewController countPrimesToSelectedValue] |
+[NSString stringWithFormat:] |
-[NSPlaceholderString initWithFormat:locale:arguments:] |
 _CFStringCreateWithFormatAndArgumentsAux | CFStringCreateCopy |
 _CFStringCreateImmutableFunnel3 | _CFRuntimeCreateInstance |
 malloc_zone_malloc
```

⁷ Die Anweisung shell sendet den Rest des Befehls an die Darwin-Shell, Sie könnten also den Befehl malloc_history auch direkt im Terminal oder in xterm an der Befehlszeile eingeben.

Sie erhalten hier eventuell mehrere `Call[]`, von denen manche nicht einmal von Ihrer Anwendung kommen (weil sie von anderen Programmen protokolliert wurden). Aber wenn Sie nach Ihren eigenen Klassen suchen, können Sie den Call Stack finden, der das Erstellen des Objekts repräsentiert, das schließlich zu einem Zombie wurde. In der vorigen Ausgabe können Sie sehen, dass dieses Objekt über die `NSString`-Klassenmethode `stringWithFormat:` erzeugt wurde, die wiederum innerhalb von `countPrimesToSelectedValue` aufgerufen wurde. Damit sollten Sie dazu in der Lage sein, sich den Quellcode anzuschauen, herauszufinden, welches Objekt den Ärger macht, und auch zu erkennen, wie Sie es geschafft haben, es über-freizugeben.

Haben Sie das Problem gelöst, ist es wichtig, die Umgebungsvariablen `NSZombieEnabled` und `MallocStackLogging` zu entfernen (oder sie auf `NO` zu setzen), bevor Sie Ihre App herausgeben. Denn `NSZombieEnabled` wandelt Objekte, die eigentlich freigegeben werden sollten, stattdessen in `NSZombie`-Objekte um, was bedeutet, dass kein einziges Objekt jemals freigegeben wird, wenn Sie `NSZombieEnabled` gesetzt haben.

Mit diesen Tools zum Aufspüren und Erlegen von Zombies sollten Sie dazu in der Lage sein, Abstürze durch Über-Freigaben zu beheben.⁸ Verbinden Sie sie mit den weiter oben beschriebenen Debugger-Techniken, können Sie jegliche Zeigerprobleme analysieren und verstehen, die zu `EXC_BAD_ACCESS`-Abstürzen führen.

Haben wir eigentlich erwähnt, dass alles, was der Debugger so tut – Breakpoints setzen, schrittweise durch den Code gehen sowie Variablen betrachten und verändern –, auch funktioniert, wenn Ihre Anwendung auf dem iPhone selbst läuft und nicht nur im Simulator? Das Gleiche gilt ebenso für die Anzeige Ihrer Protokollmeldungen. Xcode stellt Ihnen ausgesprochen wichtige Tools für die Entwicklung von iPhone-Anwendungen bereit, und je mehr Sie sie nutzen, desto unentbehrlicher werden Sie sie finden.

⁸ Und mit den in Abschnitt 28.3, Abschnitt 28.3, *Performanceanalyse mit dem Clang Static Analyzer*, auf Seite 557 vorgestellten Techniken sollten Sie Fehler durch Über-Freigaben ganz vermeiden können.

Kapitel 28

Performance-Tuning

Als wir die Primzahlenanwendung in Abschnitt 27.5, *Debuggen*, auf Seite 530 vorgestellt haben, schrieben wir, dass wir die gefundenen Primzahlen in eine UITextView stecken würden. Nachdem die Logik zum Finden der Zahlen jetzt läuft, wollen wir uns darum kümmern. Die einfachste Variante ist, den Text der View jedes Mal zu erweitern, wenn wir eine neue Primzahl finden. Der Code könnte so aussehen:

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
-(void) appendPrimeToPrimeArea: (int) aPrime {
    NSString *primeString = [[NSString alloc] initWithFormat: @"%d ", aPrime];
    primesView.text = [primesView.text stringByAppendingString: primeString];
}
```

Immer wenn diese Methode mit einer neuen Primzahl aufgerufen wird, erzeugt sie einen NSString mit der Zahl und einem folgenden Leerzeichen, holt den aktuellen String der UITextView und ersetzt ihn durch einen neuen Wert, der aus einer Verkettung des alten Werts mit dem neuen String besteht. Rufen Sie diese Methode im **if**-Block auf, der die gefundenen Primzahlen meldet (wenn Sie schon mal da sind, kommentieren Sie auch gleich die nicht mehr benötigte Anweisung `NSLog()` aus):

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
if (isPrime) {
    // NSLog(@"%d ist prim", i);
    [self appendPrimeToPrimeArea: i];
}
```

Führen Sie diese App aus, sieht die Ausgabe mehr oder weniger so aus wie Abbildung 27.7. Wollen Sie Primzahlen bis etwa 100 haben, geht das sofort. Bei 1.000 dauert es ein bisschen. Bei Primzahlen bis 10.000 müssen Sie einige Sekunden warten, bis sie angezeigt werden. Bei 100.000 ... nun ja ... sieht es so aus, als hätte sich unsere Anwendung aufgehängt.

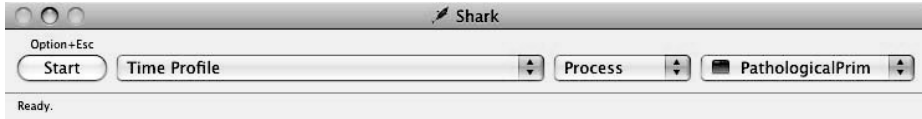


Abbildung 28.1: Einen Prozess auswählen für den mit Shark ein Profile erstellt werden soll

Dabei haben wir noch gar nichts wirklich Aufwendiges gemacht. Klar, wir haben eine verschachtelte Schleife, die wir mehr als 100.000 Mal durchlaufen, aber die Hälfte davon wird schon nach der ersten Runde durch die innere Schleife abgebrochen, weil alle geraden Zahlen durch 2 teilbar sind – den ersten Wert für *j*. Der kluge Programmierer weiß, dass etwas faul ist. Die Frage ist nur, was?

Manche Programmierer gehen gleich in die Vollen und beginnen, diverse Optimierungen auf das Problem loszulassen – Hacks mit Schleifen und Nebeneffekte, durch die die Zahl der Anweisungen reduziert werden sollen. Der kluge Programmierer fragt sich dagegen: „Wo liegt das Problem, und was kann ich da tun?“

28.1 Performanceanalyse mit Shark

Xcode enthält eine Reihe von Tools zur Performanceanalyse. Eines davon gibt es schon seit ein paar Jahren: *Shark*, benannt aufgrund des unablässigen Jagdinstinkts eines Hais, der zum Finden von Performanceproblemen notwendig ist. Shark finden Sie in `/Developer/Applications/Performance Tools`, und auch wenn es ursprünglich dazu gedacht war, mit Mac-Anwendungen genutzt zu werden, stellt es sich auch beim Analysieren von Apps auf dem iPhone als ausgesprochen hilfreich heraus.

Um Shark zu verwenden, starten Sie die Anwendung im Simulator mit einem klassischen BUILD AND RUN. Läuft sie, wechseln Sie zu Shark, das Sie mit seinem Hauptfenster begrüßt (siehe Abbildung 28.1). In diesem Fenster bietet ein Pop-up die Wahl des Profiling-Typs (wir werden den Standardtyp „Time Profile“ nutzen) und eines Zieltyps (dies ist entweder

ein bestimmter Prozess oder das gesamte System). Wählen Sie für das Ziel „Process“ und suchen Sie dann im letzten Pop-up den PathologicalPrimeCounter. Der Name kann allerdings abgeschnitten sein.

Mit dem gewählten Ziel klicken Sie auf START, um mit dem Profiling zu beginnen. Sie können jetzt zum iPhone-Simulator zurückwechseln und die Anwendung nutzen, die dabei von Shark daraufhin überwacht wird, womit sie ihre Zeit verbringt. Geben Sie für die maximale Primzahl 10000 ein und tippen Sie auf LOS. Ist der Textbereich schließlich gefüllt, wechseln Sie zurück zu Shark und klicken auf STOP, um das Profiling zu beenden. Nun analysiert Shark die gesammelten Daten und erzeugt ein Zeitprofilfenster (siehe Abbildung 28.2).

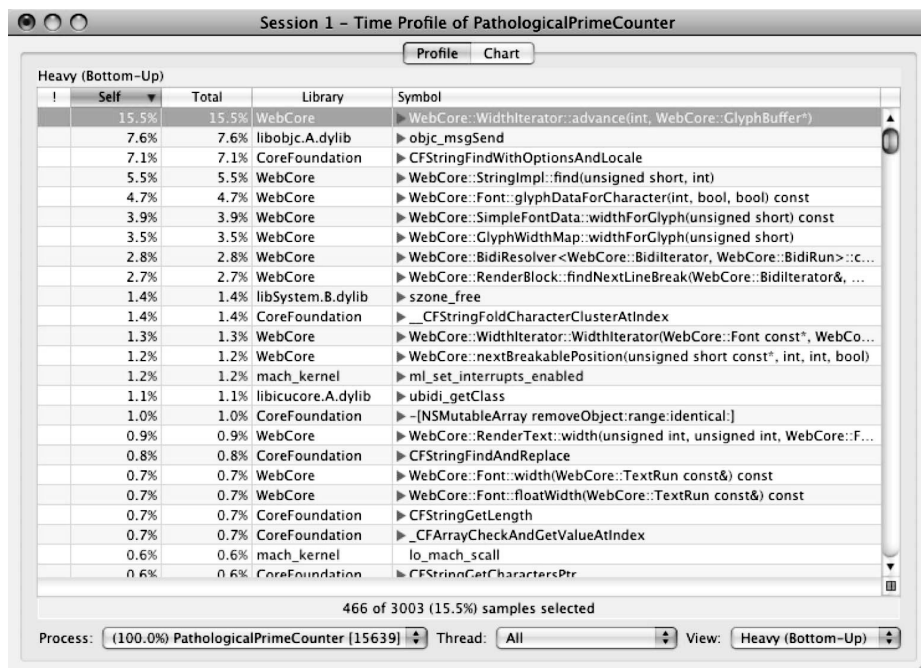


Abbildung 28.2: Zeitprofilergebnisse von Shark in der „Heavy“ View

Das Zeitprofilfenster besitzt zwei Anzeigemodi, die durch ein Pop-up in der unteren rechten Ecke kontrolliert werden. Im *Heavy (Bottom-Up)* Mode, den Sie in Abbildung 28.2 sehen, wird angezeigt, welche Methoden- und Funktionsaufrufe jeweils wie viel Zeit in der Anwendung verbrauchen. Die Tabelle lässt sich nach ihren diversen Spalten sortieren: *Self* und *Total* (was wir gleich erläutern), *Library* und *Symbol*. Schon bei einem ersten Blick auf die Tabelle sollte etwas auffallen: Die Anwendung verbringt fast die gesamte Zeit in der WebCore-Bibliothek. Aufrufe

von WebCore-Methoden sind für sieben der zehn am häufigsten aufgerufenen Methoden verantwortlich und allein diese Methoden verbrauchen nahezu die Hälfte der Zeit, die unsere Anwendung benötigt.

Jetzt schauen Sie sich genauer an, um welche Methoden es sich dabei handelt: `WidthIterator::advance()`, `Font::glyphDataForCharacter()`, `SimpleFontData::widthForGlyph()`... selbst ohne wirklich zu wissen, was die Methoden genau machen, könnte man durchaus die Hypothese aufstellen, dass unsere App sehr viel Zeit auf das grafische Layout verwendet, vermutlich aufgrund des Texts, den wir in die `UITextView` stecken.

Das können wir überprüfen, indem wir die Ansicht von *Heavy (Bottom-Up)* nach *Tree (Top-Down)* wechseln. Nun sehen wir nicht, wie viel Zeit pro Methode verbraucht wird, sondern wir erfahren eher, wer welche Methoden aufruft. Die Tabelle beginnt mit den Zeilen `start()`, `main()` und `UIApplicationMain()`, was die Ausgangsbasis für den Start einer iPhone-Anwendung ist. Weiter unten finden Sie die Main Run Loop der App (`CFRunLoopRunInMode()`), die das Tap-Event (`[UIApplication sendAction:to:from:forEvent:]`) schickt. Noch weiter unten finden Sie schließlich die Methoden aus unserer Anwendung `PathologicalPrimeCounter` (siehe Abbildung 28.3).

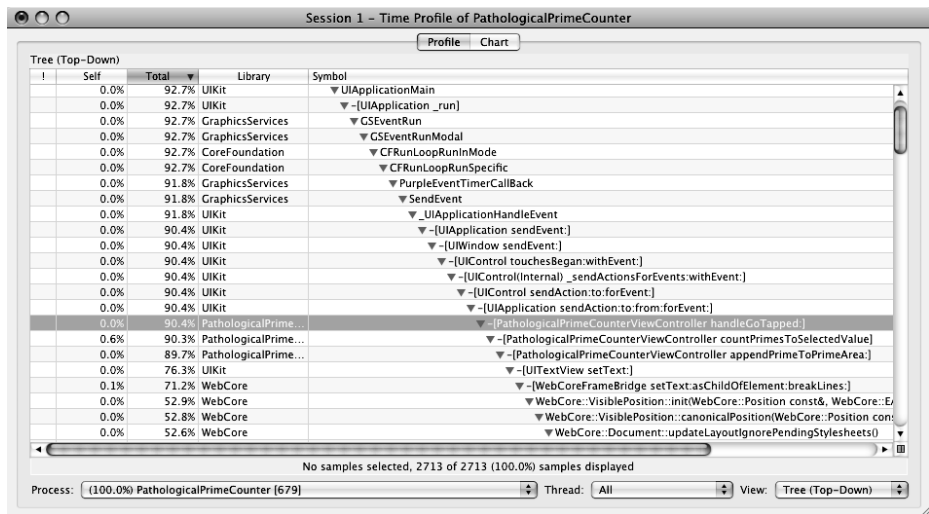


Abbildung 28.3: Zeitprofilergebnisse von Shark in einer Baumansicht

Hier kommen die Spalten *Self* und *Total* ins Spiel. Die Spalte *Self* gibt den Prozentsatz der von der Anwendung verbrauchten Zeit an, den diese Methode selbst benötigt, während in *Total* auch die Zeiten enthalten sind, die von durch diese Methode aufgerufenen Methoden benötigt

werden. Wie Sie sehen können, verbraucht von den drei `PathologicalPrimeCounter`-Methoden auf dem Stack unsere Behandlung des `TapEvents` insgesamt 90,6% der Anwendungszeit. Allerdings wird so gut wie überhaupt keine Zeit in `handleGoTapped`: selbst verbraucht und nur 0,6% in den Schleifen von `countPrimesToSelectedValue`. Die wirklich meiste Zeit, genauer 89,7%, wird mit den GUI-Methoden verbraucht, die von `appendPrimeToPrimeArea`: aufgerufen werden.

Führt man das Profiling auf diese Art und Weise durch, wissen Sie, wo Sie nach Möglichkeiten für Performanceoptimierungen suchen sollten – und wo nicht. Es ist egal, ob unser Algorithmus zum Finden von Primzahlen effizient ist – er trägt weniger als 1% zur Laufzeit bei, daher würden alle Schleifenoptimierungen der Welt keinen echten Unterschied ausmachen. Das Problem liegt an der teuren „Übertragung“ des Inhalts der `UITextView`. Denken wir darüber genauer nach, sollte die Tatsache, dass diese Layoutoperationen so teuer zu sein scheinen, uns zu einem alternativen Ansatz führen. Statt den Inhalt der View jedes Mal neu zu setzen, wenn wir eine Primzahl finden, könnten wir ihn doch auch nur einmal am Ende der Schleife setzen.

Lassen Sie uns das versuchen. Statt `appendPrimeToPrimeArea`: jedes Mal aufzurufen, wenn wir eine Primzahl finden, fügen wir die Zahl stattdessen an einen `NSMutableString` an und nutzen diesen zum Schluss, um den View-Inhalt erst dann zu setzen, wenn wir alle Zahlen gefunden haben. Schreiben wir also eine einfache Methode, die einen `String`-Formatierer nutzt, um eine Primzahl und ein zusätzliches Leerzeichen an einen veränderbaren `String` anzuhängen:

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
-(void) appendPrime: (int) aPrime
    toMutableString: (NSMutableString*) mutableString {
    NSString *primeString =
        [[NSString alloc] initWithFormat: @"%d ", aPrime];
    [mutableString appendString: primeString]; }
```

Und wir schreiben `countPrimesToSelectedValue` so um, dass diese Hilfsfunktion aufgerufen wird. Hier folgen die geänderten Schleifen und der Aufruf am Ende, mit dem der Text der View gesetzt wird.

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
NSMutableString *mutableString =
    [[NSMutableString alloc] initWithCapacity: maxPrime 5];
for (int i=2; i<maxPrime; i++) {
    int maxDivisor = i/2 + 1;
    BOOL isPrime = YES;
    for (int j=2; j<maxDivisor; j++) {
```

```

        if (i%j == 0) {
            // NSLog(@"%d ist nicht prim, teilbar durch %d", i, j);
            isPrime = NO;
            break;
        }
    }
    if (isPrime) {
        // NSLog(@"%d ist prim", i);
        [self appendPrime:i toMutableString: mutableString];
    }
}
// nach der Schleife einmal den View-Text setzen
primesView.text = mutableString;
[mutableString release];

```

Probieren Sie es aus und vergleichen Sie die Ergebnisse. Die Originalversion im Simulator auf einem Mac Pro mit acht Kernen – wir wollen so einen ineffizienten Code gar nicht erst auf die iPhone-Hardware loslassen – braucht für das Finden von Primzahlen bis 10.000 insgesamt 2,5 Sekunden bis zur Anzeige, bis 50.000 dauert es 66,8 Sekunden. Wechseln wir zur Version, die einen NSMutableString nutzt und den Text erst am Ende setzt, kommen wir auf Zeiten von 0,02 Sekunden beziehungsweise 0,25 Sekunden. Mit einer ziemlich einfachen Änderung sind wir von über einer Minute auf eine Viertelsekunde Laufzeit gekommen, was ziemlich genau dem entspricht, was zu erwarten war, nachdem die wiederholten Aufrufe entsorgt wurden.

Und gerade darin liegt die Mächtigkeit des Profiling: Sie können genau herausfinden, womit Ihre Anwendung ihre Zeit verbringt, um kritische Punkte zu finden, für die Sie entweder effizienteren Code schreiben oder einen alternativen Weg finden, der das Problem gleich ganz umgeht.

Anwendungen mit Shark auf dem iPhone profilieren

Auch wenn es nicht gleich offensichtlich ist: Shark kann auch an eine iPhone OS-Anwendung gehängt werden, die auf dem iPhone selbst läuft.

Sie müssen natürlich Ihre Umgebung so konfiguriert haben, dass der Code Ihrer Anwendung für die Ausführung auf einem mit Ihrem Mac verbundenen iPhone OS-Gerät signiert ist. Dann starten Sie Ihre Anwendung (entweder über BUILD AND RUN von Xcode oder ganz normal über das Antippen auf dem Home-Bildschirm Ihres Geräts) und starten auch Shark, wenn es nicht schon läuft. In Shark wählen Sie den Menüeintrag SAMPLING NETWORK/IPHONE PROLING (BDN). Damit wird ein neuer Bereich am unteren Ende des Shark-Fensters angezeigt (siehe Abbildung 28.4).

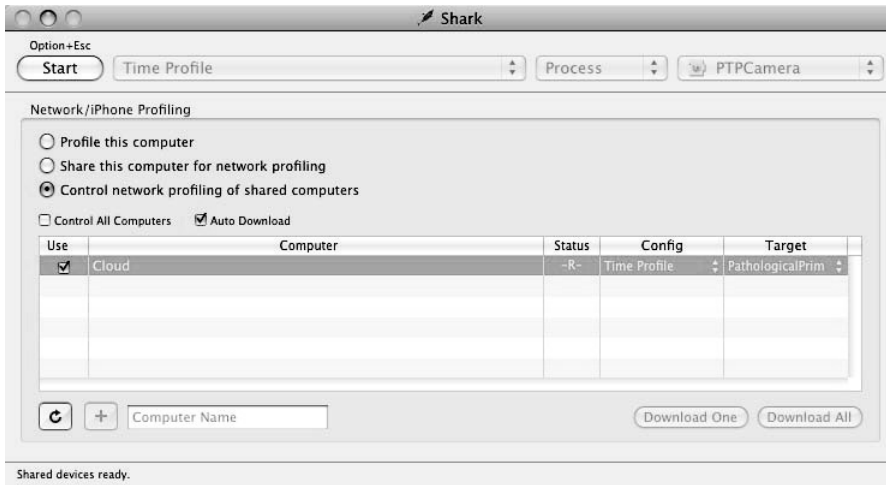


Abbildung 28.4: Profiling-Prozesse auf dem Gerät mit Shark

Dieser Bereich stellt die Remote Debugging-Optionen von Shark bereit, die durch die Radiobuttons im oberen Bereich kontrolliert werden. Wählen Sie „Profile this computer“, wird der Bereich wieder verschwinden, und man kehrt zum lokalen Profiling-Modus zurück. Wir wollen aber die letzte Option nutzen: „Control network proling of shared computers“. Markieren Sie die Checkbox neben dem Namen Ihres Geräts (in diesem Beispiel „Cloud“) und nutzen Sie die Pop-ups, um die Art der Erfassung (bisher haben wir das Standardzeitprofil genutzt) und den Zielprozess zu bestimmen. Läuft Ihre App, sollten Sie ihren gekürzten Namen, zum Beispiel `PathologicalPrim`, im Target-Pop-up sehen. Ab hier funktioniert Shark wie bisher: Lassen Sie Ihre Anwendung durch ihre kritischen Punkte laufen, stoppen Sie die Erfassung in Shark und begutachten Sie die Profilergebnisse.

28.2 Performanceanalyse mit Instruments

Während Shark zwar mächtig ist und sehr hilfreich beim Finden offensichtlich kritischer Stellen, ist sein Vorgehen doch entscheidend eingeschränkt: Es behandelt alle Aktivitäten innerhalb des Erfassungszeitraums mehr oder weniger gleich. Es kann keine kurzen Zeitabschnitte erfassen – zum Beispiel das Ergebnis eines externen Events –, die vielleicht für die Aktivitäten verantwortlich sind, die Sie interessieren. Im vorigen Beispiel geschah ein Großteil der Aktivitäten nach dem Antippen des Los-Buttons, aber Shark nutzte alles innerhalb des Erfassungszeitraums, egal ob es vor oder nach dem Beginn der Primzahlenberechnung geschah.

Hier kommt die Instruments-Anwendung ins Spiel. Instruments merkt sich die Aktivitäten in Ihrem Programm zusammen mit einem Zeitstempel und zeigt sie auf einer Timeline an, so dass Sie erkennen können, wo Ihre Anwendung CPU, Speicher und andere Ressourcen als Reaktion auf eine Aktivität benötigt.

Lassen Sie uns kurz anschauen, wie Instruments das gleiche Problem mit dem `PathologicalPrimeCounter` angeht. Wechseln Sie zur ineffizienten Version von `countPrimesToSelectedValue` zurück. Haben Sie den Beispielcode heruntergeladen, sind die verschiedenen Versionen dieser Methode auskommentiert, so dass Sie nur die eine wieder aktivieren müssen, die sagt, sie würde `appendPrimeToPrimeArea`: aufrufen. Um mit Instruments zu laufen, bauen Sie die App erneut, öffnen das RUN-Menü und wählen **START WITH PERFORMANCE TOOL**. Dann wählen Sie aus dem Untermenü **CPU SAMPLER** aus. Die Instruments-Anwendung wird gestartet und öffnet ihr Hauptfenster. Wechseln Sie zum iPhone-Simulator und nutzen Sie die Anwendung wie zuvor, indem Sie einen schönen hohen maximalen Wert eingeben (wir haben wieder 50.000 verwendet). Wenn Sie fertig sind, wechseln Sie zurück zu Instruments und klicken auf den **STOP**-Button, um die Datenerfassung zu beenden. Lassen Sie uns nun das Instruments-Fenster anschauen (siehe Abbildung 28.5).

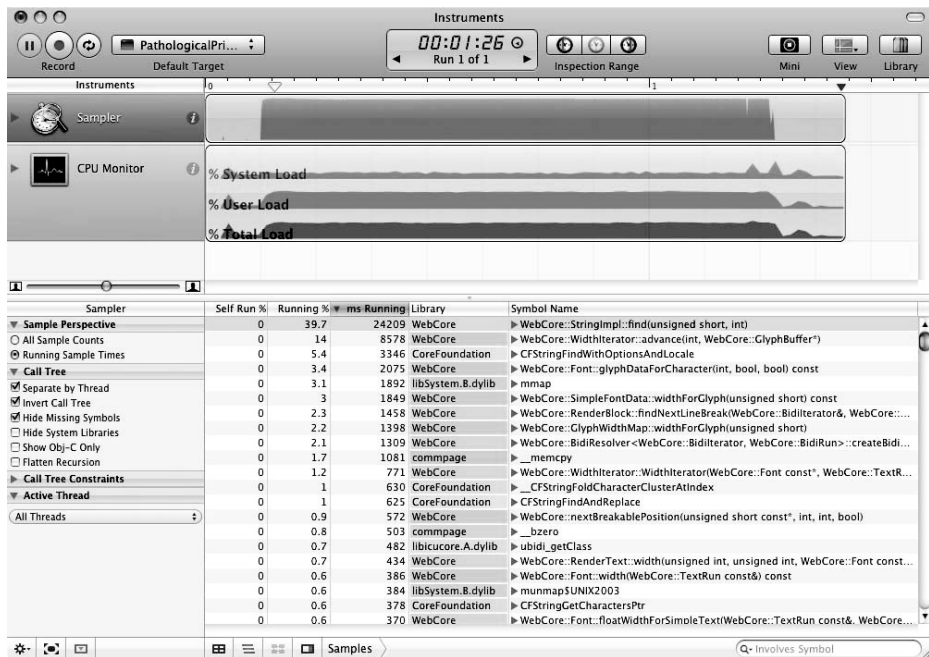


Abbildung 28.5: Instruments-Fenster mit der Sampler-Timeline

Der oberste Bereich des Instruments-Fenster zeigt eine Timeline, die Sie vielleicht schon aus Audio- oder Videoanwendungen wie GarageBand oder Final Cut Pro kennen. Diese Timeline repräsentiert die Zeit, die Ihre Anwendung gelaufen ist – vom Anfang der Aufzeichnung an (also als Instruments gestartet wurde, da es bei einem Aufruf aus Xcode direkt mit dem Aufzeichnen beginnt) bis zum Anklicken von STOP. Diese Timeline zeigt in der Abbildung zwei „Tracks“: einen für das Sampler-Instrument und einen für den CPU-Monitor. Über der Timeline gibt es ein kleines weißes Dreieck. Das ist die „Abspielposition“, die Sie vor- und zurückschieben können, um sich bestimmte Momente bei der Ausführung der Anwendung zur weiteren Analyse herauszupicken. Verschieben Sie das Dreieck, wird entlang der verschiedenen Tracks der Status angezeigt, zum Beispiel die CPU-Nutzung (in Prozent) im Sampler-Track.



Joe fragt...

Wenn Instruments doch schon die CPU-Auslastung und noch mehr erfasst – warum soll ich mich dann mit Shark herumschlagen?

Instruments hat gegenüber Shark einige große Vorteile, denn es fasst eine Reihe von Tools an einer Stelle zusammen (CPU-Auslastung, Speicherleckererkennung und so weiter) und besitzt ein Timeline-Konzept, mit dem Sie diese Messwerte über die Zeit verfolgen können. Apple hat signalisiert, dass Instruments das Tool der Zukunft sein wird.

Aber um schnell zu schauen, was denn eigentlich passiert, kann die Einfachheit von Shark manchmal den umfassenden Ansatz von Instruments ausstechen. Es ist ziemlich simpel, in Shark einen Überblick über die Auslastung zu erhalten und sich mit den beiden schlichten Views die kritischen Stellen genauer anzuschauen. In Instruments gibt es einen engen Zusammenhang zwischen der Timeline und den beiden Detail-Views.

Wenn Sie sich nur in ein Tool einarbeiten wollen, nutzen Sie Instruments. Aber wir mögen Shark immer noch.

Wie Sie im Sampler-Track an der lilafarbenen Balkengrafik sehen können, gibt es nach 14 Sekunden einen außerordentlichen Aktivitätsausbruch. Sie vermuten sicherlich, dass dies der Moment ist, in dem der Anwender auf den Los-Button getippt hat und die Anwendung damit beschäftigt war, Primzahlen zu berechnen und eine Menge unnötiger Layoutarbeit in der UITextView zu erledigen. Haben Sie den Sampler-Track markiert, erhalten

Sie im unteren Bereich des Fensters, der *Detailansicht*, zudem einen Überblick über die Funktions- und Methodenaufrufe. Setzen Sie einige der Optionen auf der linken Seite, können Sie Systemaufrufe ausblenden, die für Ihre Analyse nicht relevant sind (zum Beispiel das Message Dispatching von Objective-C) und sich darauf konzentrieren, wo Ihre Anwendung die Zeit vertrödelt. Obwohl sich Layout und Präsentation von Shark unterscheiden, sehen Sie erneut, dass die Anwendung den größten Teil ihrer Zeit mit den WebCore-Layoutmethoden verbringt.

Das Instruments-Fenster besitzt außerdem auf der rechten Seite ein „Extended Detail Pane“, auch wenn es standardmäßig ausgeblendet ist. Um es anzuzeigen, wählen Sie im VIEW-Menü den Eintrag EXTENDED DETAIL, nutzen das Tastenkürzel $\text{⌘} - \text{E}$ oder klicken auf den Button mit dem halb gefüllten Rechteck am unteren Rand des Instruments-Fensters. Sind beide Listen sichtbar, können Sie ein Element im Detailfenster anklicken und weitere Informationen dazu im „Extended Detail Pane“-Fenster erhalten. Ist zum Beispiel der Sampler-Track markiert und Sie klicken auf einen der Methodennamen in der Mitte, zeigt das „Extended Detail Pane“-Fenster den „Heaviest Stack Trace“, der aus diesem Methodenaufruf resultiert. In Abbildung 28.6 sehen Sie einen Stack Trace im „Extended Detail Pane“-Fenster mit Call Stack für den WebCore-Aufruf, der für mehr als 39% der Runtime-Aktivitäten zuständig ist.

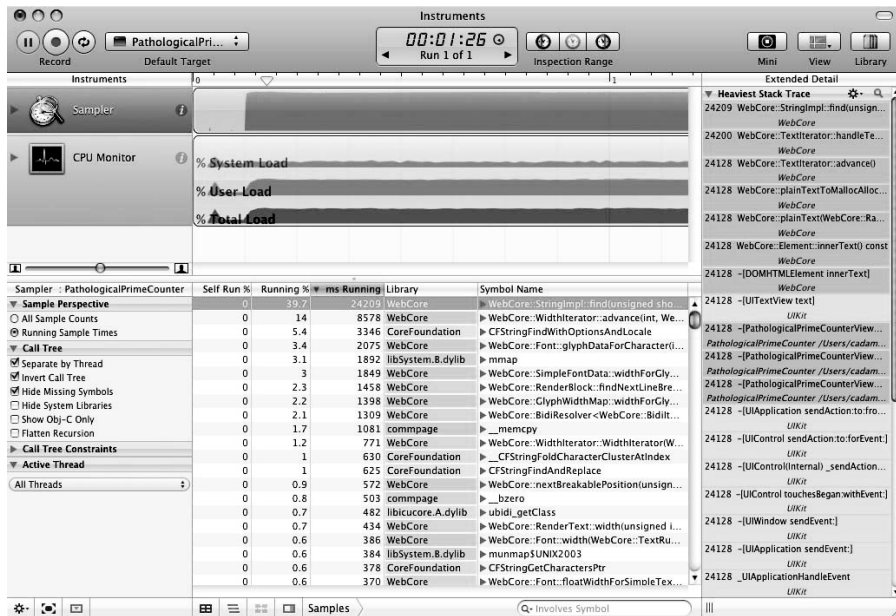


Abbildung 28.6: Instruments-Fenster mit dem „Extended Detail Pane“-Fenster

Wenn Sie Instruments von Xcode aus starten, geben Sie dabei ein Instrument an, das Sie nutzen wollen, wie zum Beispiel den CPU-Sampler, mit dem wir weiter oben begonnen haben. Aber sobald Instruments gestartet ist, können Sie noch weitere Instrumente als Tracks hinzufügen und damit genauer untersuchen, wie Ihre Anwendung die verschiedenen Systemressourcen nutzt. Die verfügbaren Instrumente finden Sie im Library-Fenster, das Sie im WINDOW-Menü oder über das Tastenkürzel - erreichen. Die Library zeigt alle Instrumente an, die Sie in eine Mac- oder Xcode-Instruments-Session ziehen können. Natürlich passen einige davon nicht auf das iPhone, wie zum Beispiel die Core Data- oder Java-Tools. Markieren Sie ein Tool, zeigt die Beschreibung am unteren Ende des Library-Fensters an, ob das Tool für den Mac, für das iPhone oder für beides nutzbar ist.

Wir haben zwar schon dafür gesorgt, dass keine unnötige CPU-Zeit mehr verbraten wird, aber wir wollen sichergehen, dass wir nicht auch noch Speicher verbrauchen. Markieren Sie das ObjectAlloc-Instrument und ziehen Sie es in die Liste der Instrumente auf der linken Seite des Fensters. Mit einem Klick auf RECORD führen Sie die Anwendung erneut aus. Nachdem wir wissen, dass wir eine ganze Menge Primzahlen in sehr kurzer Zeit bestimmen können, wollen wir gleich einmal mit 50.000 loslegen. Wenn die Text-View gefüllt ist, wechseln Sie zurück zu Instruments und schauen sich die Timeline an. Sie können sehen, dass es parallel zum Aktivitätsausbruch im CPU-Monitor auch eine Delle im ObjectAlloc-Graphen gibt. Das bedeutet, dass wir bei unserer Primzahlenmethode eine Menge Objekte erzeugen. Eine genauere Analyse des Graphen zeigt zudem, dass die Anzahl der Objekte nicht wieder zurückgeht. Sie können das überprüfen, indem Sie sich die Tabelle mit Kategorien¹ in der Detailliste anschauen und diejenigen anklicken, die Sie interessieren. In Abbildung 28.7 haben wir CFString angeklickt (das Core Foundation-Äquivalent zu NSString). Dadurch wird im lilafarbenen Graphen der *All Allocations* eine rote Linie eingezeichnet. Sie können sogar die *All Allocations* über deren Checkbox abwählen und nur CFString selektiert lassen, der Graph zeigt dann, dass die Anzahl der Objekte steil nach oben schießt und nicht wieder sinkt.

1 Kategorien sind in diesem Zusammenhang als Cocoa-Klassen und opake Foundation-Typen zu verstehen, die keine „echten“ Klassen sind.

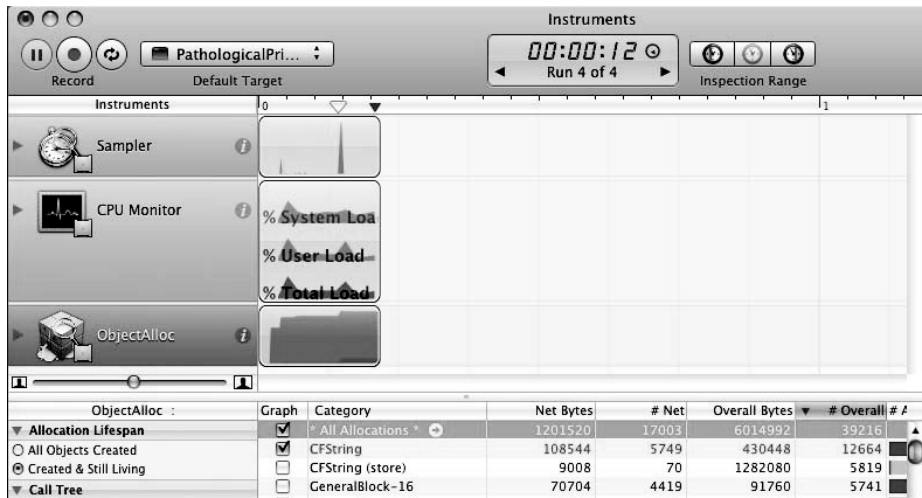


Abbildung 28.7: Ein mögliches Speicherleck mit dem ObjectAlloc-Instrument untersuchen

Mit so vielen Objekten, die erzeugt, aber nie freigegeben werden, ist es sehr wahrscheinlich, dass wir ein Speicherleck haben.

Um uns das genauer anzuschauen, öffnen Sie das „Extended Detail Pane“-Fenster und klicken auf den Pfeil, der neben CFString erscheint, wenn Sie mit der Maus darübergehen. Damit wird die Tabelle mit den Kategorien durch eine Liste aller Instanzen von CFString ersetzt mit der Angabe, wann und durch welchen Aufruf sie erzeugt wurden. Da unser Speicherleck in der Timeline eher später auftaucht, können wir nach der Zeit sortieren und in der Tabelle nach unten scrollen, um Instanzen zu finden, die auch erst später während der Lebensdauer der Anwendung entstanden sind. Wir finden hier Tausende von Instanzen, die durch `-[NSString initWithFormat:locale:arguments:]` entstanden sind. Klicken Sie eine an, erhalten wir den Call Stack im „Extended Detail Pane“-Fenster und direkt in der Mitte eine Reihe von Aufrufen durch den `PathologicalPrimeCounter`. Klicken Sie doppelt auf den letzten dieser Einträge, gelangen Sie zu der Codezeile, die den übrig gebliebenen String erzeugt hat. Diese befindet sich in `appendPrime:toMutableString:`. Lassen Sie uns die Methode nochmals anschauen:

Debugging/Primes/Classes/PathologicalPrimeCounterViewController.m

```
-(void) appendPrime: (int) aPrime
    toMutableString: (NSMutableString*) mutableString {
    NSString *primeString = [[NSString alloc]
initWithFormat: @"%d ", aPrime];
    [mutableString appendString: primeString]; }
```

Sehen Sie das Leck? Wir haben einen `NSString` erzeugt, ihn an den `NSMutableString` angehängt ... und ihn nie freigegeben! Nach dem Anhängen an den `NSMutableString` sind wir nicht mehr länger an ihm interessiert und können ihn direkt wieder freigeben. Fügen Sie also am Ende der Methode diese Zeile hinzu:

```
[primeString release];
```

Erzeugen Sie einen neuen Build, wechseln Sie zu Instruments und starten Sie mit einer Aufzeichnung. Richten Sie das *ObjectAlloc*-Instrument so ein, dass nur CFStrings dargestellt werden, sollten Sie das viel bessere Ergebnis erhalten, das Sie auch in Abbildung 28.8 sehen. Es gibt eine Spitze beim Objektzähler zu dem Zeitpunkt, an dem auch die CPU mit dem Finden von Primzahlen und dem Füllen der `UITextView` beschäftigt ist, aber dieser geht wieder zurück, da wir alle temporären String-Objekte aufgeräumt haben.

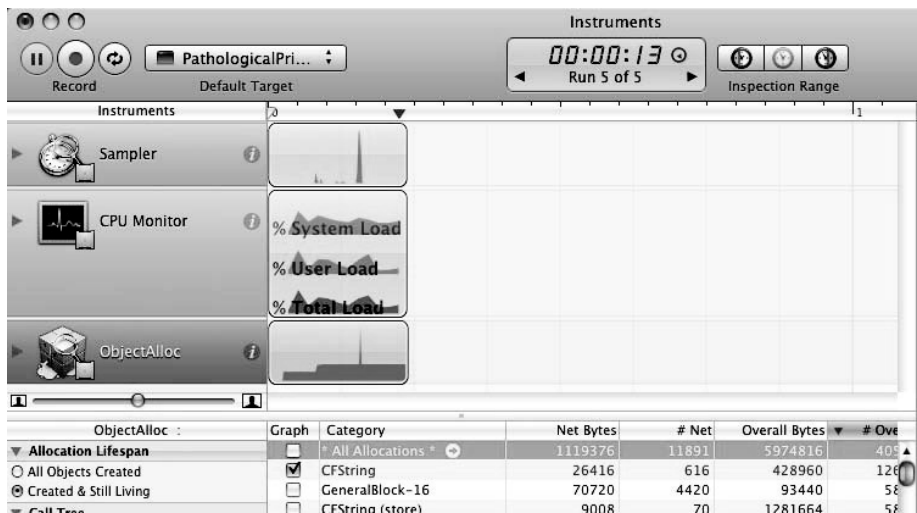


Abbildung 28.8: Ein behobenes Speicherleck mit dem ObjectAlloc-Instrument untersuchen

28.3 Performanceanalyse mit dem Clang Static Analyzer

Ein Probing Ihres Codes mit Shark oder Instruments ist ausgesprochen nützlich, wenn Sie Ihren Code unter realen Bedingungen testen wollen. Aber Sie finden damit nicht alles. Bugs, die sich in Seitenzweigen des Codes verstecken, die Sie normalerweise nicht testen, zeigen sich dort auch nicht, sondern erst, wenn die Anwender in diesen Bereich der Anwendung vordringen. Und wie wir im Abschnitt über Instruments

gesehen haben, erzeugt Ihre Anwendung Tausende von Objekten auf einmal, die dann über einen längeren Zeitraum nach und nach Objekte „heraströpfeln“ lassen.

Eine *statische Analyse* bietet hier ein ganz anderes Vorgehen, um Fehler zu finden. Anstatt den Code zur Laufzeit zu analysieren, modelliert sie den Code mit ihren eigenen Techniken und versucht so, Fehler zu finden. Gerade vor ein paar Monaten hat ein von Apple gesponsertes Open Source-Projekt den *LLVM/Clang Static Analyzer* veröffentlicht. Er arbeitet mit Xcode-Projekten, unter anderem auch mit iPhone-Anwendungen.

Clang entwickelt sich

Arbeiten Sie mit Xcode 3.2 oder neuer unter Snow Leopard, ist der Clang Static Analyzer schon mit installiert und in Xcode integriert. Entwickeln Sie unter Leopard, müssen Sie die Befehlszeilenversion des Analyzer selbst einrichten. Sie finden eine Anleitung dazu in einem PDF, das dem herunterladbaren Code zu diesem Buch beiliegt.

Auch wenn sich die Schnittstelle zum Static Analyzer noch ändert, sind die Tests, die er durchführt, und die Berichte, die er ausgibt, recht stabil.

Der Analyzer ist ein Ableger des *Clang*-Projekts, das ein Frontend für den LLVM-Compiler² für C, C++, Objective-C und Objective-C++ bereitstellt.

Wenn Sie den Analyzer auf Ihrem Mac installiert haben (siehe den Kasten mit einem Hinweis auf eine Installationsanleitung für die Befehlszeilenversion), wollen wir ihn mit echtem Code testen. Stellen Sie sicher, dass Ihr Projekt mit Standardoptionen und ohne Fehler gebaut werden kann. Wenn nicht, erhalten Sie eventuell eine verwirrende Fehlermeldung, die besagt, dass scan-build eine benötigte Version des gcc nicht finden kann. In der Praxis ist uns diese Meldung begegnet, wenn ein Projekt standardmäßig für ein Gerät und nicht für den Simulator gebaut wird und die aktuelle Umgebung keinen Code signieren kann. Das kann passieren, wenn Sie den Code eines anderen heruntergeladen haben, dessen Projekt für das Bauen für das Gerät eingerichtet ist,

² Bei der Low Level Virtual Machine (LLVM), deren Heimat <http://www.llvm.org/> ist, handelt es sich um einen mehrgleisigen Ansatz zum Verbessern der Anwendungsperformance. Dabei wird ein virtuelles Instruction Set genutzt, das intelligent an seine Host-Umgebung angepasst werden kann, indem es seinen nativen Bytecode zur Laufzeit und sogar nach der Installation optimiert.

oder wenn Sie noch nicht dem kostenpflichtigen iPhone Developer Programm beigetreten sind und noch kein Zertifikat zum Signieren von Apps erhalten haben. Sie können dieses Problem umgehen, indem Sie die Projekteinstellungen so ändern, dass standardmäßig für den Simulator gebaut wird. Dazu wählen Sie in Ihrem Xcode-Projekt das Projektsymbol ganz oben im *Groups & Files*-Baum und klicken dann in der Toolbar oder im FILE-Menü auf GET INFO. Wie in Abbildung 28.9 zu sehen, setzen Sie dann *Base SDK* auf „Simulator“ und nicht auf „Device“.

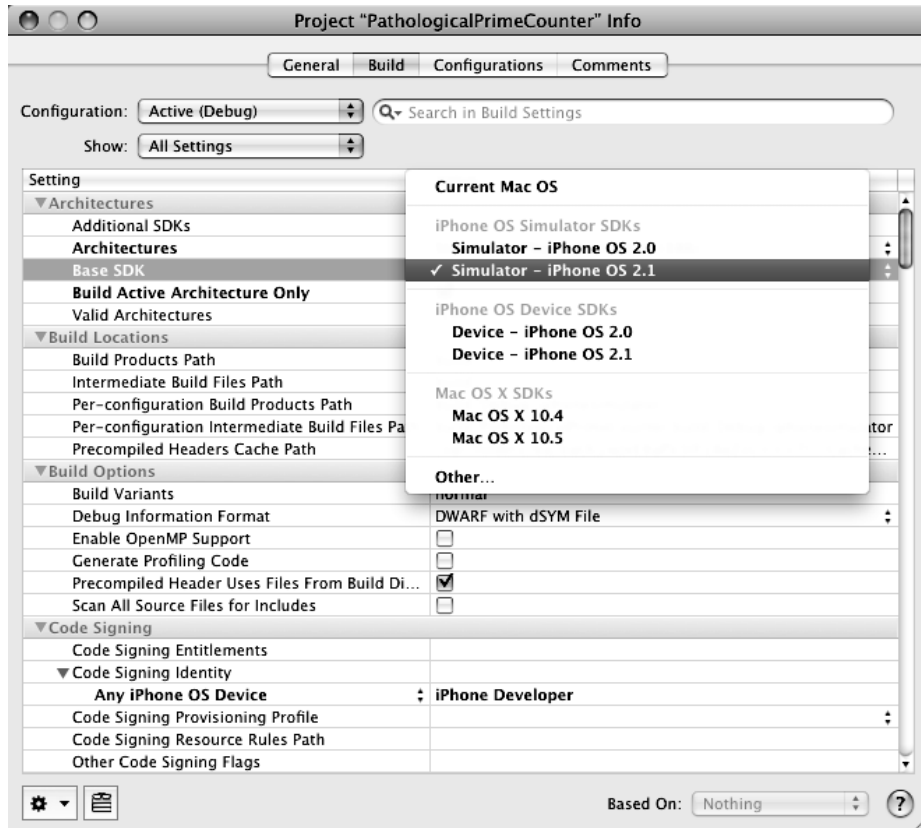


Abbildung 28.9: Setzen Sie Base SDK auf „Simulator“, um den Clang Static Analyzer nutzen zu können

Lassen Sie uns nun einen Scan für das Projektverzeichnis durchführen. Wenn Sie die Befehlszeilenversion nutzen, geben Sie `scan-build xcodebuild` ein. Können Sie einen Scan von Xcode aus starten, wählen Sie **BUILD & ANALYZE**.

Wenn alles geklappt hat,³ werden Sie darüber informiert, dass ein Fehler gefunden wurde.

In der Web-View wird jeder Fehler, den scan-build findet, im Reportbereich aufgeführt – zusammen mit Links, über die ein Bericht über den Fehler angezeigt werden kann. Klicken Sie auf den Link „View Report“, um eine Schritt-für-Schritt-Bewertung des Fehlers zu erhalten. Die Webseite, die erzeugt wird, sieht ungefähr so wie in Abbildung 28.10 aus. Wird der Static Analyzer aus Xcode heraus ausgeführt, erhalten Sie die Fehlerberichte auf nahezu dem gleichen Weg wie die Fehler und Warnungen. Trotz des Unterschieds in der Schnittstelle enthält der Fehlerbericht diese Informationen:

- „Method returns an object with a +1 retain count (owning reference).“
- „Object allocated on line 61 and stored into 'primeString' is no longer referenced after this point and has a retain count of +1 (object leaked).“

Hier hat die Analyse einen weiteren Fall gefunden, in dem wir eine Instanz von NSString allokkieren und nie wieder freigeben.

Was macht das Tool – abgesehen vom Finden von Speicherlecks – noch? Hier eine kurze Liste seiner nützlichsten Tests:

- Korrekte Verwendung von retain und release.
- Ungenutzte Instanzvariablen.
- Nicht initialisierte Variablen.
- Nicht erreichbare Codeabschnitte.
- Verweise auf Null-Zeiger.
- Division durch null.
- Tote Speicher (Zuweisen eines Werts an eine Variable, die nie genutzt wird).
- Methodensignaturen in Objective-C mit Typinkompatibilitäten.
- Fehlendes dealloc .

³ Wir sind sicher, dass Sie uns im (englischsprachigen) Forum zum Buch (<http://forums.pragprog.com/forums/83>) darüber informieren werden, wenn dem nicht so ist. Zudem hat einer der Leser darauf hingewiesen, dass ein sauberer Build als Teil der Ausführung des Static Analyzers helfen kann, während iterativ Fehler behoben werden.

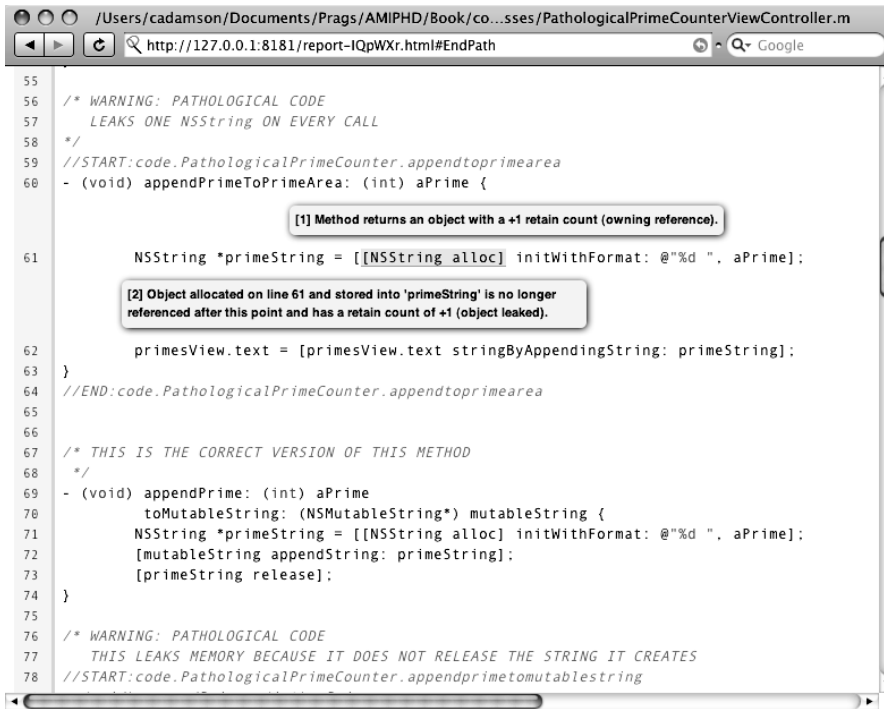


Abbildung 28.10: Bericht des Clang Static Analyzers über ein Speicherleck in Objective-C

Interessant daran ist, dass einige dieser Tests Dinge finden, die Sie durch Profiling nie finden können. Wir haben die nicht freigegebenen Objekte in Abschnitt 28.2, *Performanceanalyse mit Instruments*, auf Seite 551 vor allem deshalb gefunden, weil wir so viele davon auf einmal erzeugt haben. Da im Verlauf einer Anwendung viele Objekte zwischendurch erzeugt werden, kann es viel schwieriger sein, einen Fehler zu finden, bei dem nur nach und nach Objekte übrig bleiben. Aber solch ein Fehler, der den Referenzzähler beeinflusst, lässt sich von scan-build sehr gut finden. Versuchen Sie, in eine Variable zu schreiben, die schon freigegeben ist, erzeugen Sie damit potenziell einen Fehler, der zum Absturz führen kann. Findet man diesen Fehler frühzeitig durch eine statische Analyse, ersparen Sie sich, Ihren Testern und Ihren Anwendern so manchen Ärger. Es sei auch nochmals darauf hingewiesen, dass Sie durch ein Profiling keine Fehler in Ablaufpfaden finden, die Sie nicht testen. Der Static Analyzer hingegen schaut sich jeden möglichen Pfad im Code an und findet damit auch diese Fehler.

Der Clang Static Analyzer ist noch sehr frisch und auch unvollständig, aber schon jetzt kann er eine sehr nützliche Erweiterung Ihres Testbaukastens sein.

Was kommt als Nächstes?

In diesem und im vorigen Kapitel haben wir uns eine Reihe von Tools angeschaut, die zu Xcode gehören und die dabei helfen, dass Ihre Anwendung korrekt und effizient läuft. Wir haben uns im vorigen Kapitel mit Kompilierfehlern und deren Dokumentation sowie mit fortschrittlicheren Debugging-Techniken zur Diagnose und zum Lösen von kritischeren Problemen befasst. In diesem Kapitel ging es um Performance – die Analyse der CPU-Aktivität mit Shark und Instruments, wobei wir bei Letzterem auch noch die Objekt-Allokation verfolgen konnten, um Speicherlecks zu finden. Zudem haben wir uns mit dem Clang Static Analysis Tool das Neueste vom Neuesten angeschaut, um Speicherlecks und andere Fehler direkt im Code zu finden.

Es gibt ein paar Instrumente, die wir in diesem Kapitel nicht angesprochen haben, die aber für manche iPhone-Anwendungsentwickler von Nutzen sein können. Es gibt Instrumente, um die Grafikperformance und die CPU-Auslastung von Core Animation- und Open GL ES-Aufrufen zu messen, aber auch ein „System Usage“-Instrument, um I/O-Aufrufe zu messen (zum Beispiel das Öffnen, Schließen, Lesen und Schreiben von Dateien).

Zudem gibt es eine Reihe von Mac-spezifischen Instrumenten, zum Beispiel einen ausgesprochen interessanten UI Recorder, der es Ihnen ermöglicht, Interaktionen mit Ihrem GUI aufzuzeichnen und dann wieder „abzuspielen“, um Performanceprobleme im Zusammenhang mit dem Gebrauch der Anwendung zu finden. Auch wenn diese Tools noch nicht für das iPhone SDK zur Verfügung stehen, kann man aufgrund der Verfügbarkeit für den Mac hoffen, dass sie in der Zukunft vielleicht portiert werden.

Auch wenn es sich nicht um ein Performancetool im eigentlichen Sinne handelt, gibt es im iPhone SDK 3.0 eine Unterstützung für *Unit-Tests*, die es Ihnen erlauben, Code zu schreiben, der die Logik und das Laufzeitverhalten Ihres Codes prüft, und damit einige Ihrer Tests zu automatisieren. Dazu wählen Sie die Targets des Projekts und fügen neue Unit-Test-Bundles hinzu. Dann erzeugen Sie neue Klassen in Ihrem Projekt, die als Target statt des Application Target das Unit-Test-Bundle-Target haben. Der Prozess ist ein bisschen aufwendig; sollten Sie in Ihrem Entwicklungsprozess Unit-Tests nutzen, ist es sinnvoll, das Kapitel „Unit Testing Applications“ des *iPhone Development Guide* [App08d] zu lesen.

Kapitel 29

Davor und danach

Okay, kommen wir zum Ende. Es war ein langer Weg seit dem ersten Absatz, in dem wir Sie dazu ermuntert haben, etwas Großartiges zu schaffen. Wir sind sicher, dass Ihr Kopf nach dem Lesen dieses Buchs vor Ideen nur so summt. In diesem Kapitel werden wir einen Schritt zurücktreten und uns über Dinge um die Technologie herum unterhalten, die auch zu einer erfolgreichen Anwendung dazugehören.

Beginnen wir dieses Kapitel damit, auch das Entwickeln einer App richtig zu beginnen. Bevor Sie Code schreiben können, müssen Sie entscheiden, was Sie überhaupt haben wollen. Das kann der aufwendigste Teil der App-Erstellung sein. Es ist schwer, sich darüber klar zu werden, was Sie nun genau bauen wollen, wenn Ihnen viele Ideen im Kopf herumschwirren. Aber es ist auch einer der wichtigsten Aspekte für eine erfolgreiche App. Wenn Sie hier keinen Plan haben, landen Sie eventuell bei einer Kraut-und-Rüben-Anwendung. Solche Anwendungen können beim Erstellen sehr interessant und unterhaltsam sein, aber sie lassen sich auch häufig nur schlecht verkaufen. Wir werden einige Strategien vorstellen, die Sie nutzen können, um herauszufinden, was Sie genau bauen wollen.

Nachdem die eigentliche Funktionalität Ihrer Anwendung läuft, ist es wichtig, Ihre App zu durchforsten und nach Stellen zu suchen, die Sie noch abrunden können. Die Anwender wissen so etwas zu würdigen; sie werden eher davon erzählen und sie damit weiterempfehlen. Das ist einer der Bereiche, der sehr häufig ignoriert wird, der aber zum Erfolg Ihrer App entscheidend beiträgt.

Nachdem Sie Ihre Anwendung gebaut haben, ist es wichtig, sie zu testen. Möglicherweise schrecken Sie davor zurück, Ihre App in die Hände von Testern zu geben. Der nächste Abschnitt wird beschreiben, was Sie tun müssen, um einen erfolgreichen Betatest per Ad-hoc-Distribution durchzuführen. Wir werden auch ein paar Dinge erwähnen, mit der Sie Ihre Anwendung für Ihre Tester verbessern können, so dass deren erster Eindruck so positiv wie möglich ist.

Ist Ihre Anwendung nun abgerundet und getestet, ist sie bereit für den nächsten Schritt – Werbung. Im letzten Abschnitt dieses Kapitels stellen wir ein paar Ideen vor, die Sie nutzen können, um Ihre App bekannt zu machen. Es gibt viele verschiedene Möglichkeiten, die aber davon abhängen, welche Zielgruppe Ihre App ansprechen soll. Wir werden eine Reihe allgemeiner Ideen aufzeigen, die Ihnen eine erhöhte Aufmerksamkeit bescheren können.

Aber beginnen wir am Anfang.

29.1 Richtig beginnen

Wenn wir eine tolle Idee haben, sind wir oft so begeistert, dass wir gleich Xcode anwerfen und mit dem Coden beginnen. Lange, anstrengende Nächte des Refaktorisierens haben uns aber gezeigt, dass das der falsche Ansatz ist. Auch wenn wir schnell Fortschritte machen, sobald wir unsere Ideen mit Xcode und dem Interface Builder zum Leben erwecken, geht der Aufwand für Änderungen doch dramatisch in die Höhe, wenn eine Idee in Code umgesetzt ist.

Bevor Sie Xcode starten, sollten mindestens zwei Dinge vor Ihnen liegen: eine Anwendungsdefinition und ein Prototyp auf Papier. Anstatt einen Prototyp direkt in Code umzusetzen, um Ihre eigene Idee möglicherweise besser zu verstehen, sollten Sie dafür erst mal Ihren Kopf und ein Blatt Papier nehmen. Schreiben Sie auf, was Ihre Anwendung tun soll und welche die geplante Zielgruppe ist. Nutzen Sie ganz real Papier und Bleistift.

Die Anwendungsdefinition (Application Definition Statement, ADS) ist vermutlich eines der wichtigsten Elemente Ihrer Anwendung. Dieser Text, vielleicht lediglich ein Satz, hilft Ihnen, zu verstehen, worum es bei Ihrer App überhaupt geht. Damit können Sie Entscheidungen zu Features und UI-Elementen treffen, durch die Ihre App fokussierter und sauberer wird.

Es ist nicht schwer, ein ADS zu schreiben, aber Sie sind dadurch gezwungen, auszudrücken, worum es bei Ihrer Anwendung geht. Es gibt in jedem guten ADS drei Teile:

- das Unterscheidungsmerkmal
- die Lösung
- die Zielgruppe

Das Unterscheidungsmerkmal gibt an, was Ihre Anwendung so besonders macht. Ist sie besonders leicht zu bedienen? Dann kann das Ihr Unterscheidungsmerkmal sein. Statt einer besonders leichten Bedienbarkeit kann Ihre Anwendung vielleicht auch grafisch besonders attraktiv sein. Was auch immer Ihre Anwendung als Alleinstellungsmerkmal hat – dies ist das Unterscheidungsmerkmal.

Die Lösung definiert, welches Problem Ihre Anwendung lösen wird. Wollen Sie Informationen bereitstellen oder organisieren, oder wollen Sie es Ihren Anwendern ermöglichen, Fotos freizugeben? Die Lösung, die Sie geben wollen, definiert das Problem, das Ihre App für Ihre Benutzer lösen wird. Aus diesem Teil der Definition erhalten Sie eine Liste mit Features, aus der Sie all die löschen, die nicht auf Ihre Zielgruppe passen.

Die Zielgruppe definiert, für wen die App geschrieben wird. Bauen Sie ein Tool für Profifotografen oder eine unterhaltsame Anwendung für Kinder? Durch den Fokus auf Ihre Zielgruppe können Sie Ihre Feature-Liste auf die Elemente reduzieren, die für Ihre Zielgruppe wichtig sind. Wenn Sie eine Unterhaltungs-App für Kinder bauen wollen, wird sich Ihre Feature-Liste und die Benutzerschnittstelle deutlich von der einer App unterscheiden, die für Vertreter gedacht ist.

Für manche klingt das vielleicht verrückt. „Ich kann eine ganze Woche damit zubringen, ein ADS zu schreiben und mein UI zu zeichnen, ohne dass ich eine Zeile Code fertig habe!“ Sie haben vielleicht keinen Code, aber Sie werden ein viel besseres Gefühl dafür haben, wie sich Ihre Idee in eine echte Anwendung umsetzen lässt.

Viel zu oft haben wir vor Xcode gesessen und losprogrammiert, nur um Hunderte von Codezeilen später zu bemerken, dass das Konzept, das wir im Kopf hatten, nicht funktioniert. Dann sind wir gezwungen, die Anwendung umzuschreiben oder an dem Konzept zu drehen – beides keine schönen Aufgaben. Die Kosten eines Umschreibens können sehr hoch werden, also ändern wir das Konzept und damit schließlich auch das „Gefühl“ der Anwendung, die wir eigentlich machen wollten.

Nutzen Sie Papier und Bleistift, um Ihre Anwendung aufzuzeichnen, ergeben sich immer ein paar Ideen:

- Verwenden Sie wirklich Papier und Bleistift, nicht OmniGraffle. Auch wenn es verführerisch ist, ein UI per Software zu entwerfen, führt das oft dazu, dass man nur die Zeichnung aufhübscht, statt die eigentliche Idee mit Leben zu füllen. Es geht darum, Ihre Ideen zu dokumentieren, und nicht um die Erkenntnis, wie schwierig es ist, ein pixelgenaues Bild zu erhalten.
- Zeichnen Sie Ihre Ideen auf Karteikarten oder auf Papier mit der Größe des Bildschirms. Damit können Sie das echte UI simulieren, indem Sie den nächsten Bildschirm darüberlegen, wenn Sie einen Button „antippen“. So können Sie Ihren Anwender die App recht wirklichkeitsnah zeigen, ohne eine Zeile Code schreiben zu müssen.
- Wenn Sie feststellen, dass Teile Ihres UI nicht so funktionieren, wie Sie es sich vorgestellt haben, schmeißen Sie diese Zeichnungen weg und erstellen schnell neue. Auf Papierzeichnungen verzichtet man eher, und das ist es, was Sie an diesem Punkt wollen – schnelle Ideen, die weggeworfen werden können, wenn sie nicht sinnvoll umzusetzen sind.
- Kümmern Sie sich vor allem um die Dinge, die entscheidend sind. Es ist nicht immer einfach, zu erkennen, was „entscheidend“ ist, aber überlegen Sie sich, was Ihre Anwendung einmalig macht, was auf jedem Bildschirm zu sehen sein wird, wie es zu präsentieren ist – diese ganzen Dinge. Verbringen Sie nicht so viel Zeit mit Features, die Sie einfach nur wiederverwenden, wie eine Navigationsleiste oder eine Auswahlliste. Mit anderen Worten – es ist wichtig, zu zeigen, wie Ihre Anwendung aussehen wird, nicht, eine bestimmte Auswahlliste exakt darzustellen.
- Zeigen Sie Ihre Skizzen jemand anderem und bitten Sie ihn um eine Bewertung. Wenn Sie ein Team haben, stellen Sie sicher, dass es bei der Bewertung dabei ist (und natürlich auch die Unterlagen erhält). Haben Sie niemanden, der eine Bewertung abgeben könnte, versuchen Sie, jemanden zu finden, der ein „repräsentativer“ Anwender Ihrer App sein wird. Je mehr Feedback Sie an diesem Punkt bekommen, desto leichter wird es fallen, die perfekte App zu bauen.

- Da sich bisher alles nur auf Papier befindet, ist es kein großer Aufwand, beide Layouts (horizontal und vertikal) auszuprobieren und das zu verwerfen, das sich nicht so gut nutzen lässt. Anwender lieben es zwar, ein anderes Layout zu erhalten, wenn sie das iPhone drehen, aber wenn eine der Richtungen nicht funktioniert, ist es besser, das vorher auf Papier festzustellen als nach vielen Stunden des Codeschreibens.

Nachdem Sie nun eine ganz gute Vorstellung davon haben, wie Ihre Anwendung aussehen wird, ist es an der Zeit, den üblichen Rahmen einmal zu verlassen. Das Zeichnen eines UI auf dem Papier zeigt uns, wie unsere Anwendung aussehen wird und sogar wie sie funktioniert. Aber es gibt auf dem iPhone eine ganze Menge ziemlich coole Technologie, und da Sie jetzt wissen, was Ihre App tun wird und wie sie dabei aussieht, ist es Zeit, über den „Wow-Faktor“ nachzudenken. Was können Sie Ihrer Anwendung hinzufügen, das die Benutzer dermaßen begeistert, dass sie ihren Freunden davon erzählen? Gibt es eine Möglichkeit, eine „Schüttelgeste“ sinnvoll zu nutzen? Können Sie die „Always On“-Internetverbindung verwenden, um Ihrer App einen sozialen Aspekt hinzuzufügen?

Diese Art von bemerkenswerten Features sind das i-Tüpfelchen Ihrer App – und ein weiterer Grund dafür, Papierprototypen statt Code zu erzeugen. Denn dort ist es einfach, eine Kontextsensitivität hinzuzufügen. Wenn sich zeigt, dass sie für Ihre App nicht sinnvoll nutzbar ist, können Sie sie auf dem Papier auch wieder genauso einfach fallen lassen.

Denken Sie beim Sammeln von Features mit „Wow-Faktor“ daran, dass man sich leicht mitreißen lässt. Und wenn Sie sich beim Schreiben von Code haben mitreißen lassen, ist die emotionale Nähe deutlich größer. Papier lässt sich einfacher zusammenknüllen und wegwerfen, daher kann man dort Features mit „Wow-Faktor“ leichter wieder loswerden, als müsste man sie aus dem Code löschen.

Wie immer muss bei jedem Feature, das Sie Ihrer Anwendung hinzufügen wollen, der Benutzer im Mittelpunkt stehen. Wenn das tolle Feature oder die coole Geste die App nur „voller“ werden lässt, ist es besser, es zu lassen. Selbst wenn es cool aussieht – am Ende lenkt es vom Herz Ihrer Anwendung ab.

Denken Sie einmal an eine Schüttelgeste. Es ist ziemlich cool, dass Sie ein iPhone schütteln können und es passiert etwas. Aber ist das Schütteln für Ihre Anwender sinnvoll? Stellen Sie sich genau solche Fragen. Werden meine Benutzer wissen, dass sie das Gerät schütteln können, um diese Funktion oder dieses Feature aufzurufen? Wenn nicht – gibt es eine andere Möglichkeit, an die Funktion zu kommen? Die Antworten auf diese Fragen sind sehr wichtig und können leicht untergehen, wenn man sich von den „Wow-Faktor“-Features mitreißen lässt.

Eine andere Möglichkeit, coole Features zu finden, ist, Ihren Prototyps Benutzern zu präsentieren, um deren Feedback zu erhalten. Oft haben Ihre potenziellen Anwender ein paar tolle Ideen für coole Features, auf die Sie nie gekommen wären.

Es ist doppelt schwer, wenn man diese Features gleich in Code umsetzt. Wer denkt mitten im Codieren schon daran, ob ein „Wow-Faktor“ wirklich nützlich ist? Wir hassen es, uns zu wiederholen, aber das ist ein weiterer Grund dafür, Ihre Ideen zuerst auf Papier umzusetzen. Es ist viel einfacher, schlechte Ideen wieder fallen zu lassen, wenn Sie nur ein bisschen Grafit und Papier dafür aufgewendet haben.

Es sind häufig diese scheinbar kleinen Dinge – die Entscheidung darüber, was man nun doch nicht nutzt, welche Details man ergänzt, welche coole Geste man einsetzt –, die wichtig sind. Damit sorgen Sie dafür, dass die Leute die Anwendung ihren Freunden zeigen wollen. Aber wenn eines dieser Features oder der Gesten überflüssig ist oder sogar stört, kann es den gegenteiligen Effekt haben. Denn schließlich wollen Sie mit einer App ja das Leben von jemandem vereinfachen und es nicht noch komplizierter gestalten. Lassen Sie immer den Anwender Mittelpunkt Ihrer Entscheidungen sein, werden Sie auch eher die richtigen Wege beschreiten.

Bevor Sie nun immer noch engagiert mit dem Programmieren beginnen, sollten Sie sich überlegen, ob Sie Ihre Anwendung internationalisieren beziehungsweise lokalisieren wollen. Berücksichtigt man das von Anfang an, hat man später weniger Probleme, die App in andere Sprachen zu übersetzen oder andere Währungs- und Zeitformate zu nutzen. Apple bietet unter <http://developer.apple.com/iphone/library/documentation/MacOSX/Conceptual/BPInternational/BPInternational.html> eine Reihe von Artikeln zu diesem Thema an. Und unter <http://adeem.me/blog/2009/05/09/tutorial-iphone-localization-in-xib-nib-files/> finden Sie den ersten Teil eines Tutorials zum Lokalisieren Ihrer .xib-Dateien.

Nachdem wir nun über den Anfang des Prozesses gesprochen haben, wollen wir direkt zum anderen Ende springen und uns über das Abrunden Ihrer Anwendung unterhalten, bevor wir sie in die Freiheit entlassen und sie zum Testen schicken.

29.2 Abrunden

Sobald die Anwendung läuft, ist es meist verführerisch, sie sofort zum Testen herauszugeben, um all Ihre harte Arbeit auch echten Anwendern präsentieren und deren Reaktionen einsammeln zu können. Auch wenn ein frühes Feedback und Tests wichtig sind, denken wir doch, dass es besser ist, nach all den schweren, aufwendigen Arbeiten einen Schritt zurückzutreten und nach Bereichen zu suchen, die noch etwas Politur vertragen können.

Abrunden kann vieles bedeuten, aber vornehmlich ist hier damit gemeint, Ihrer Anwendung die Ecken und Kanten zu nehmen und die Bedienbarkeit zu vereinfachen. Dabei sind folgende Fragen wichtig:

- Wie geht meine App mit Unterbrechungen um, zum Beispiel durch eingehende Anrufe?
- Sind die wichtigen Features/Funktionen leicht zu erreichen und einfach zu nutzen?
- Wie ist die gefühlte Performance? Startet die Anwendung schnell genug?
- Gibt es Bereiche in der Anwendung, die mehr Strom brauchen, als sie sollten/könnten?

Auch wenn es auf keine dieser Fragen eine universelle Antwort gibt, ist es wichtig, dass Sie sie sich stellen, bevor Sie Ihre Anwendung als fertig bezeichnen. Die Anwender legen nämlich durchaus Wert darauf, dass man auch solchen Fragen Aufmerksamkeit schenkt, auch wenn sie Ihnen keine E-Mail schreiben und sich überschwänglich bei Ihnen bedanken, nur weil eingehende Anrufe kein Chaos anrichten.

Einen anderen Bereich, den Sie beim Abrunden Ihrer Anwendung berücksichtigen sollten, ist das Startbild. Auf dem iPhone selbst kann es ein bisschen dauern, bis Ihre Anwendung ganz geladen ist, insbesondere wenn dabei noch auf das Netzwerk zugegriffen wird oder ein anderer aufwendiger Prozess läuft, bevor das Programm eingabebereit ist.

Eine Möglichkeit, die gefühlte Performance zu verbessern, ist das Hinzufügen einer Datei `Default.png` zu Ihrer Anwendung. Wird Ihre App gestartet, wird dieses Bild geladen und dem Anwender direkt angezeigt. Viele Anwendungen nutzen daher einen typischen Screenshot davon, wie die App nach dem Starten aussehen könnte, als `Default.png`. Damit hat der Anwender den Eindruck, dass die App schon bereit ist, bevor sie es wirklich ist. Zusätzlich zur Datei `Default.png` können Sie eine Datei für jedes URL-Schema definieren, das Ihre Anwendung laden kann. So haben wir zum Beispiel in Kapitel 26, *Anwendungsintegration*, auf Seite 511 eine Anwendung gebaut, die über eine URL mit dem integrierten Schema geöffnet wurde. Hier könnten wir ein Bild `Default-integrated.png` hinzufügen, so dass bei jedem Aufruf über eine URL dieses Bild genutzt würde anstelle von `Default.png`.

Ein ordentlicher Umgang mit Unterbrechungen ist genauso wichtig. Versuchen Sie, herauszufinden, was Ihre Benutzer von Ihrer Anwendung erwarten. Gibt es nur sehr wenig Kontext, ist es nicht so wichtig, nach einem Neustart der App dort weitermachen zu können, wo der Anwender aufgehört hat. Aber sobald der Kontext umfangreicher ist, wird es immer wichtiger, den vorherigen Status Ihrer Anwendung bei jedem Start herstellen zu können. In unseren Anwendungen haben wir SQLite genutzt, um den Kontext abzuspeichern, so dass wir beim Starten der Anwendung an der Stelle weitermachen können, wo wir zuvor aufgehört haben. (Mehr Informationen über SQLite finden Sie in Kapitel 10, *Die Datenbank SQLite*, auf Seite 205.) Jede Anwendung ist anders, aber machen Sie sich zumindest Gedanken darüber, wie Ihre Anwendung aus Sicht der Benutzer funktionieren sollte – und orientieren Sie sich dann auch daran.

Eine letzte Verbesserung, auf die Sie achten sollten, bevor Sie die Beta-version Ihrer App an die Tester mailen, sind die Symbole. Sie brauchen zwei – eines für das iTunes-Symbol, das genutzt wird, um Ihre Anwendung dort darzustellen, und eines für Ihre Anwendung auf dem Home-Bildschirm des iPhone.

Für Ihr Anwendungssymbol müssen Sie ein paar Dinge beachten. Es sollte 57 x 57 Pixel groß und im PNG-Format vorliegen. Fügen Sie es dem *Resources*-Abschnitt in Xcode hinzu. Zudem müssen Sie `Info.plist` anpassen, um die Datei festzulegen.

Der Code dazu sieht wie folgt aus:

```
<key>CFBundleIconFile</key>
<string>icon.png</string>
```

Ersetzen Sie einfach den Wert `icon.png` durch den Namen Ihres Symbols, wird das iPhone dieses Bild statt des generischen weißen Blobs einsetzen.

Für iTunes müssen Sie der *Resources*-Gruppe in Xcode ein Bild mit dem Namen `iTunesArtwork` (beachten Sie die fehlende Erweiterung) hinzufügen. Die Datei wird von iTunes genutzt, um Ihre Anwendung im Programmbereich anzuzeigen. Dabei sollte es sich um eine PNG- oder JPEG-Datei mit der Größe 512 x 512 handeln und möglichst das Gleiche zeigen wie das Symbol für Ihre Anwendung auf dem Gerät.

Nachdem Ihre Anwendung nun abgerundet ist, sollten Sie sie einer Gruppe realer Anwender schicken, damit diese einen Betatest durchführen können.

29.3 Weitere Features

Was man noch in seine Überlegungen einbeziehen sollte, ist die Verwendung einiger der webbasierten Features, die Apple mit dem iPhone OS 3.0 eingeführt hat. Dazu gehören vor allem der *In-Application Purchase* und die *Push Notifications*.

Beim In-Application Purchase können Ihre Nutzer zusätzliche Add-ons für Ihre App erwerben, nachdem sie schon die Anwendung an sich gekauft haben. Ist es in Ihrer App möglich, zusätzliche Features oder Komponenten anzubieten, die in dieses Szenario passen würden? Wenn ja, könnte es sinnvoll sein, sie über das In-App Purchase-Feature zu integrieren. Sie müssen nicht nur die Elemente in Ihre Anwendung einfügen können, Sie brauchen zudem einen Webserver im Internet, der mit Ihrer Anwendung und mit den Apple-Servern interagieren kann. Die Entwicklerdokumentation enthält weitere Details zu diesem Feature.

Ein weiteres cooles Feature des 3.0-Release ist die Push Notification. Ist es in Ihrer Anwendung sinnvoll, Ihre Benutzer über neue Inhalte zu informieren? Wenn ja, sollten Sie darüber nachdenken, dazu den Push Notification Service zu verwenden. Auch bei diesem Feature benötigen Sie einen Server im Internet, der mit den Push Notification-Servern von Apple kommunizieren kann.

29.4 Betatesten

Das Testen ist einer der wichtigsten Abschnitte im Lebenszyklus Ihrer Anwendung. Wenn Ihre App an die echten Anwender gelangt, ist der entscheidende Punkt gekommen. Es ist außerordentlich wichtig, hier nicht zu hastig zu sein. Lassen Sie Ihre Anwender die App ausgiebig nutzen und geben Sie ihnen die Möglichkeit zum Feedback.

Hören Sie Ihren Testern zu. Wir wissen, dass sie manchmal komische Sachen sagen, aber warum sollten Sie sonst Leute dazu bringen, Ihre Anwendung zu testen, wenn Sie ihnen dann nicht zuhören? Das Problem ist, dass das Feedback der Tester durchaus auch negativ sein kann. Es ist oft schwer, negative Rückmeldungen anzunehmen, aber gerade bei ihnen handelt es sich auch häufig um die hilfreichsten. Es ist viel einfacher, auf negative Meldungen in dieser frühen Phase zu reagieren, als sich dann im App Store mit den negativen Bewertungen herumärgern zu müssen. Damit kommen wir zu einem weiteren wichtigen Punkt.

Wählen Sie Ihre Testgruppe sehr sorgfältig aus. Sie brauchen Personen, die sich für das Testen tatsächlich Zeit nehmen und auch ein ehrliches Feedback geben. Erhalten Sie nur positives Feedback, ist es wichtig, nachzuhaken, um herauszufinden, was die Leute nicht gemocht haben. Drängen Sie darauf, zu erfahren, was nicht ging und was sie nicht wollten. Auch wenn es schön ist, positives Feedback zu erhalten, hilft dies selten, die Anwendung zu verbessern.

Die Details des Prozesses, Ihre Anwendung den Betatestern zukommen zu lassen, entwickeln sich fortlaufend weiter, aber grundlegend ist, dass Sie ein „Ad Hoc Distribution Profile“ für die Geräte der Tester brauchen. Dazu benötigen Sie die Geräte-IDs von jedem Gerät, das im Test verwendet wird. Sie können das Gerät mit einem Computer verbinden, auf dem Xcode installiert ist, das Organize-Fenster öffnen, das Gerät auswählen und die ID aus der Übersichtsseite kopieren. Haben Sie keinen Zugriff auf das Gerät, müssen die Anwender ihr iPhone an den Rechner anschließen, iTunes starten und das Gerät markieren. In der Zusammenfassungs-Tab von iTunes klicken sie dann auf das Feld für die Seriennummer, das damit zum Feld für die Identifizierung (UDID) wird. Danach können sie auf KOPIEREN klicken (oder - tippen), um die Nummer in die Zwischenablage zu kopieren. Diese ID ist es, die Sie benötigen.

Haben Sie die IDs der Testgeräte, können Sie sie im iPhone Developer Portal eintragen und ein Ad Hoc Distribution Profile für jedes Gerät erhalten. Halten Sie sich exakt an die Instruktionen von Apple, da jede Abweichung dazu führen kann, dass sich Ihre App nicht auf die Geräte verteilen lassen kann.

Nachdem sich Ihre Anwendung nun in den Händen echter Benutzer befindet, müssen Sie einen erstklassigen Support leisten. Wenn Fehler gemeldet werden, sollten Sie sicherstellen, dass die Tester erfahren, dass Sie die Meldung erhalten haben und dass Sie sich um das Problem kümmern werden. Die Leute werden durch positives Feedback motiviert, und es ist wichtig, dass Ihre Tester motiviert bleiben.

Noch ein Hinweis zum Schluss – Sie sollten zwar so viele Fehler wie möglich in der Betaphase beheben, aber aufpassen, dass sich keine zusätzlichen Funktionen als Fehlermeldungen einschleichen. Konzentrieren Sie sich auf das „Hauptthema“, das Ihre Anwendung anspricht. Ihre Tester werden viele tolle Ideen liefern. Aber wenn es kein Fehler ist, legen Sie es für eine spätere Implementierung beiseite. Zu viele Anwendungen sind durch ein Überladen mit Features ins Straucheln geraten. Es ist immer besser, eine Software ausgeliefert zu bekommen, als fortlaufend nur an der „besten“ Software zu arbeiten.

29.5 Die App in den Store bringen

Es gab viele Berichte über den Zulassungsprozess von Apple für Anwendungen, und einige davon können durchaus entmutigend sein. Lassen Sie sich aber nicht davon abhalten. Wie so oft im Leben schaffen es die schlechten Nachrichten auf die Titelseiten, während die guten Nachrichten übersehen werden. Es ist sehr wahrscheinlich, dass Ihre Anwendung zugelassen wird, aber wir wollen hier ein paar Tipps dazu geben, wie Sie gleich beim ersten Mal Erfolg haben können.

Zuallererst stellen Sie sicher, dass Sie die rechtlichen Rahmenbedingungen geklärt haben, bevor Sie eine App freigeben wollen. Das bedeutet, dass Sie ein Bankkonto für Ihre geschäftlichen Aktivitäten haben, eine Steuernummer vorweisen können und all die anderen lokalen Anforderungen erfüllen, die eine echte Firma erfüllen muss. Sorgen Sie dafür, dass Sie die Verträge mit Apple unterschrieben haben und dass Apple das auch weiß.

Der Prozess kostet Zeit, und es dauert auch, zu prüfen, ob Sie derjenige sind, der Sie zu sein vorgeben. Erwarten Sie also nicht, dass Sie direkt nach dem Einreichen eine App auch ausliefern können. Denken Sie ebenfalls daran, dass es Tausende von Entwicklern gibt und jeden Tag neue hinzukommen. Apple hat eine ganze Reihe von Mitarbeitern, die sich um diese Arbeit kümmern, aber trotzdem dauert es seine Zeit, jeden neuen Entwickler aufzunehmen.

Wenn die Dokumente sagen, dass Sie keine privaten APIs nutzen sollen, dann ist das auch so gemeint. Bindet Ihre Anwendung private Frameworks ein, wird sie abgewiesen. Apple geht darauf in seinen Dokumenten recht umfassend ein und spricht an vielen Stellen auch Warnungen aus, aber viele Apps werden trotzdem abgewiesen, weil die Leute diese Beschränkungen ignorieren. Es mag verlockend sein, Text to Speech zu nutzen, aber wenn Sie das tun, erhalten Sie eine Ablehnung.

Wird Ihre App abgelehnt und die Gründe erscheinen Ihnen merkwürdig, atmen Sie erst einmal tief durch, lassen Sie alles einen oder zwei Tage liegen und schauen Sie sich die Sache dann mit etwas Abstand an. Die meisten Ablehnungen sind korrekt, und das Einzige, was Sie dann tun können, ist ein Ändern der App.

Wenn Sie nach einigem Nachdenken immer noch der Meinung sind, dass die Ablehnung nicht richtig ist, versuchen Sie, die App erneut einzureichen. Die Reviewer sind auch nur Menschen, und andere Personen sehen Ihre Anwendung vielleicht mit anderen Augen. Denken Sie auch daran, dass die Reviewer täglich Hunderte von Anwendungen begutachten müssen. Da geschehen natürlich auch Fehler, also nehmen Sie das Ganze besser nicht persönlich. Reichen Sie die App erneut ein und hoffen Sie, dass ein neuer Reviewer eine andere Sichtweise hat. Prüfen Sie zusammen mit Kollegen, ob Sie eine Beschränkung übersehen haben oder ob Apple wirklich einen Fehler gemacht hat. Wird Ihre App erneut zurückgewiesen, lesen Sie sich nochmals den vorigen Absatz durch.

29.6 Für Ihre Anwendung werben

Marketing und Werbung für Ihre Anwendung kann ein Vollzeitjob sein. Sind Sie ein unabhängiger Entwickler, ist es leicht, sich nur dafür aufzureiben. Sind Sie nicht unabhängig, gibt es vielleicht Leute in Ihrer Firma, die sich darum kümmern können.

Haben Sie keine Marketingabteilung in der Hinterhand, lesen Sie weiter. Müssen Sie all die Rollen selbst einnehmen und suchen Sie nach der effektivsten Möglichkeit, Ihre Anwendung bekannt zu machen, sind Sie hier genau richtig.

Es kann hart sein, sich bekannt zu machen, aber das eigentliche Geheimnis ist, ein Netzwerk mit Leuten aufzubauen, die Ihrer Arbeit vertrauen, so dass sie Ihnen helfen können. Lernen Sie Leute über soziale Netzwerke wie Twitter, Facebook oder LinkedIn kennen. Besuchen Sie lokale Entwicklertreffen. Ist Ihr Produkt so weit, versorgen Sie Ihr Netzwerk mit den Fakten. Spammen Sie die Leute nicht zu, aber lassen Sie sie freundlich wissen, dass Ihre App jetzt zur Verfügung steht.

Eine weitere, häufig unterschätzte Gruppe sind Ihre Tester. Hatte jemand die Gelegenheit, Ihre Anwendung zu nutzen, und mochte er sie, sollten Sie ihn um einen Review auf iTunes bitten, aber auch darum, von Ihrer Anwendung zu erzählen.

Pressemeldungen können ebenfalls effektiv sein. Eine vollständige Liste von Sites, die iPhone-Anwendungen beschreiben, ließe sich aufgrund der dynamischen Natur des Markts nicht sinnvoll erstellen. Aber googeln Sie einmal eine halbe Stunde nach Sites, finden Sie häufig eine Reihe von Plätzen, an denen Sie Ihre Anwendung vorstellen könnten.

Achten Sie bitte darauf, Ihre Pressemitteilung an eine Site anzupassen. Schreiben Sie eine ganz allgemeine Pressemitteilung, erhalten Sie auch eine allgemeine Antwort (oft die des Spamfilters). Haben Sie ein Spiel und schreiben Sie eine Site an, die sich auf Spiele spezialisiert hat, sollten Sie sich auf die Spieleigenschaften Ihrer App konzentrieren. Schreiben Sie an eine Site, die Reviews zu allen möglichen Apps veröffentlicht, schreiben Sie über den Spaß oder die Herausforderung Ihres Spiels. Gebrauchen Sie Ihren gesunden Menschenverstand und schreiben Sie eine Pressemitteilung speziell für eine Person oder Site. Man wird Ihnen dankbar sein, und die Chancen für eine Veröffentlichung steigen deutlich.

Schließlich noch ein Wort zum Support. Schickt Ihnen jemand eine E-Mail mit Fragen zu Ihrer Anwendung oder mit der Bitte um ein neues Feature, reagieren Sie schnell und freundlich. Wir wissen, wie seltsam es klingt, erzählen zu müssen, man solle doch freundlich zu seinen Kunden sein, aber manchmal können Benutzeranfragen etwas sehr seltsam sein. Bedanken Sie sich immer und seien Sie nett. Denn eine freundliche Antwort auf eine Support-E-Mail kann den Unterschied zwischen einem verärgerten Kunden, der schlecht über Ihre App redet, und einem Kunden, der sie seinen Freunden empfiehlt, ausmachen.

Dieses Kapitel ist bei Weitem nicht vollständig, wenn es darum geht, was Sie tun können und sollten, um eine erfolgreiche Anwendung zu präsentieren. Das, was wir hier beschrieben haben, hat bei uns als unabhängige iPhone-Anwendungsentwickler funktioniert (oder auch nicht). Wir hoffen, dass die Informationen aus diesem Kapitel Ihnen dabei helfen, Ihre Ziele in Bezug auf Ihre Anwendung zu erreichen. Haben Sie andere Ideen, teilen Sie sie mit uns im (englischsprachigen) Forum zum Buch.

Viel Glück!

Literaturverzeichnis

- [Apa09] Apache Software Foundation. *Authentication, Authorization and Access Control*. <http://httpd.apache.org/docs/2.0/howto/auth.html>, 2009.
- [App06a] Apple, Inc. *Cocoa Fundamentals*. http://developer.apple.com/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html#//apple_ref/doc/uid/TP40002974-CH6-SW1, 2006.
- [App06b] Apple, Inc. *Toll Free Bridging*. <http://developer.apple.com/documentation/Cocoa/Conceptual/CarbonCocoaDoc/Articles/InterchangeableDataTypes.html>, 2006.
- [App07a] Apple, Inc. *Carbon-Cocoa Integration Guide*. <http://developer.apple.com/documentation/Cocoa/Conceptual/CarbonCocoaDoc/index.html>, 2007.
- [App07b] Apple, Inc. *Core Audio Overview*. <http://developer.apple.com/documentation/MusicAudio/Conceptual/CoreAudioOverview/>, 2007.
- [App07c] Apple, Inc. *Introduction to Quartz 2D Programming Guide*. http://developer.apple.com/documentation/GraphicsImaging/Conceptual/drawingwithquartz2d/dq_intro/chapter_1_section_1.html, 2007.

- [App08a] Apple, Inc. *Archives and Serializations Programming Guide for Cocoa*. <http://developer.apple.com/DOCUMENTATION/Cocoa/Conceptual/Archiving>, 2008.
- [App08b] Apple, Inc. *Event-Driven XML Programming Guide for Cocoa*. <http://developer.apple.com/documentation/Cocoa/Conceptual/XMLParsing/>, 2008.
- [App08c] Apple, Inc. *iPhone Developer Program*. <http://developer.apple.com/iphone/program/>, 2008.
- [App08d] Apple, Inc. *iPhone Development Guide*. http://developer.apple.com/iphone/library/documentation/Xcode/Conceptual/iphone_development/, 2008.
- [App08e] Apple, Inc. *Key-Value Coding Programming Guide*. <http://developer.apple.com/documentation/Cocoa/Conceptual/KeyValueCoding>, 2008.
- [App08f] Apple, Inc. *Obtaining your iPhone Development Certificate*. <http://developer.apple.com/iphone/manage/certificates/team/howto.action>, 2008.
- [App08g] Apple, Inc. *Predicate Programming Guide*. <http://developer.apple.com/documentation/Cocoa/Conceptual/Predicates/predicates.html>, 2008.
- [App09a] Apple, Inc. *HTTP Live Streaming Overview*. <http://developer.apple.com/iphone/library/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/>, 2009.
- [App09b] Apple, Inc. *Introduction to Core Data Programming Guide*. <http://developer.apple.com/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html>, 2009.
- [App09c] Apple, Inc. *iPhone Application Programming Guide*. <http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide>, 2009.
- [App09d] Apple, Inc. *Technical Note TN2188: Exporting Movies for iPod, Apple TV and iPhone*. <http://developer.apple.com/technotes/tn2007/tn2188.html>, 2009.

- [App09e] Apple, Inc. *The Objective C Programming Language*.
<http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC>, 2009.
- [Dud08] Bill Dudley. *Core Animation for OS X: Creating Dynamic Compelling User Interfaces*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2008.
- [GL06] David Gelphman and Bunny Laden. *Programming with Quartz, 2D and PDF Graphics in Mac OS X*. Morgan Kaufman, San Francisco, 2006.
- [Hip09] Inc. Hipp, Wyrick & Company. *Sql features that sqlite does not implement*.
<http://www.sqlite.org/omitted.html>, 2009.
- [KP88] Al Kelley and Ira Pohl. *A Book on C*. Addison-Wesley Professional, New York, 1988.
- [KR98] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall PTR, Englewood Cliffs, NJ, second edition, 1998.
- [Pan09] R. Pantos, Ed. *Http live streaming* (internet-draft).
<http://www.ietf.org/internet-drafts/draft-pantos-http-live-streaming-01.txt>, 2009.
- [PT08] Maija Palmer and Paul Taylor. *Google homes in on revenues from phones*. Financial Times, February 13, 2008.
- [Ras07] Bertis Rasco. *Where's the wiimote? using kalman filtering to extract accelerometer data*. Gamasutra, June 20, 2007.
- [RSWLH07] Jr. Richard S. Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL SuperBible*. Addison Wesley Longman, Reading, MA, fourth edition, 2007.
- [SPHB08] Thomas Schlömer, Benjamin Poppinga, Niels Henze, and Susanne Boll. *Gesture recognition with a wii controller*. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, Seiten 11–14, New York, 2008. ACM Press.

- [Ste09] Daniel H Steinberg. *Cocoa Programming: A Quick-Start Guide for Developers*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.
- [Sto08] Brad Stone. *iphone users love that mobile web*. The New York Times, June 19, 2008.
- [Uni04] Inc. Unicode. *Unicode technical standard 35: Locale data markup language (ldml)*. Unicode Technical Reports, 2004.
- [Zar09] Marcus Zarra. *Core Data: Apple's API for Persisting Data under Mac OS X*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

Index

Symbole

#include 522
#pragma mark 287
%@ Formatangabe 39, 531–532
*-Zeichen 52, 522
+-Zeichen 34
_ (Unterstrich) 297
µ-law 364

A

A Book On C: Programming in C
(Kelley Pohl) 480
AAC 364
Abrunden von Anwendungen 569, 571
Abspielposition 553
Abspielstatus 332, 335
Abstand 488
Abtasten 360
Accelerometer 453, 470
 Ausrichtung des Geräts 455
 Filtern von Daten 463, 470
 Rohdaten 456, 462
 Schüttelerkennung 455–456,
 464, 466
 Simulator 453–454
 Überblick 453
accessoryType 120
Actions 33, 261
 hinzufügen 168
 implementieren 41
AddressGeocoder 503, 507

Adressbuch
 Benutzerschnittstelle 472
 Hinzufügen von Kontakten 477
 Kontakte hinzufügen 480
 People Picker Delegation 473–474
 People Picker konfigurieren 475
 Person Controller 475, 477
ALAC 364
A-law 364
Alignment 35
allgemeine
 Medienobjekteigenschaften 333
Allocator 390
Animation 439, 452
 anpassen 443
 Delegation 444
 Layer 447, 449
 OpenGL 449, 452
 Überblick 439–440
 UIView 440, 446
Animationsblöcke 440–441
Anwendungen 563
 abrunden 569, 571
 Absturz 535
 Betatesten 572–573
 bewerben 574–575
 Flippable Views 181, 184
 Fotos 427
 Integration 511, 517
 Starten anderer Anwendungen
 511, 513
 Überblick 511
 Vorbereitung 513, 517
 navigationsbasiert 117, 121

- Start und Planung 564, 569
 - webbasierte Features 571
 - Zulassung 573–574
- Anwendungen starten, Integration 511
- Anwendungs-Anatomie 43
- Anwendungs-Delegate 44
- Anwendungsdelegation 41
- Anwendungsintegration 44
- Anwendungs-Speicherverwaltung 47
- Anzeigestile 83
- Apache, Websites für Bonjour 294
- App Store 4, 573–574
- Application Definition Statement (ADS) 564
- Application ID 154
- Application Music Player 332
- Arrays 88, 136
 - PreferenceSpecifiers 194
- Aspect Fit 430–431
- asynchrones Einlesen von Dateien 164
- Attributes Inspector 35
 - Tab-Controller 134
- Audio Conversion Services 399
- Audio File Services 398
- Audio File Stream Services 399
- Audio Queue Services 355, 398
- Audio Session Services 397–398
- Audio- /Video-Formate 326
- Audiodateiformate 358
- Audioformate 326
- Audiokategorie 382
- Audio-Sessions 382, 385
- Audio-Unit Graph 400
- Audio-Units 399, 401
- Aufzeichnungsinformationen
 - anzeigen 372
- Ausrichtung, Accelerometer 455
- Aussteuerung 372, 375
- Aussteuerung anzeigen 377
- Aussteuerungsanzeige 372
- Auswirkungen 4
- Autorelease Pool 41
- AVAudioPlayer 380–381
- AVAudioRecorder 356, 359
 - Aufzeichnungsinformationen 372
 - Aussteuerung anzeigen 377
 - Delegate-Callbacks 370
 - Methoden 370
 - nutzen 368, 372
 - Tab-basierte Anwendung 356
- AVAudioSession 382, 385

- AVFoundation 356
 - Abspielen von Audio-Daten 377, 382
 - Aufzeichnungsinformationen
 - anzeigen 372
 - Aussteuerung anzeigen 377
 - AVAudioRecorder nutzen 368, 372
 - AVAudioRecorder, Delegate-
 - Callbacks 370
 - AVAudioRecorder-Methoden 370
 - Beispielanwendung 359
 - Beispielanwendung aufsetzen 356
 - Eigenschaften 362
 - Einstellungen 362
 - kodierte Formate 364, 367
 - Schlüssel/Wert-Paare 363
 - unkomprimierte Audioformate
 - 359, 364

B

- BalanceBall-Beispiel 456
- Balkengrafik zeichnen 419
- Befehlsobjekte 404
- benutzerdefinierte
 - Objekteigenschaften 333
- Benutzeroberfläche
 - Bilder 429
- Benutzerschnittstelle
 - Adressbuch 471–472
 - AudioRecorderPlayer 362
 - Browser 261
 - iPod 330, 332
 - Planung 567
 - Tab-Controller 131, 133
 - Videos abspielen 321–322
- Betatesten 572–573
- Bevölkerungsdaten, sortiert 136
- Bewerben der Anwendung 574
- Bézierkurven 418
- Bibliothek
 - iPod 341, 351
 - libXML2 503
- Bilder 427, 437
 - angepasste Anzeige 430
 - angepasste Ausgabe 432
 - ausgeben 428
 - Formate 428
 - Image Picker 432, 436
 - Programmstart 569
 - Überblick 427
 - Video 437
 - zeichnen 430

- Bogen 418
- Bonjour
 - Apache aktivieren 294
 - Netzwerkdienste 293, 295
 - Philosophie 294
 - Service Discovery 296, 301
- Build and Run 19
- Build-Fehler 520, 522
- Bundesland-Beispiel (Tabs) 132
 - Anpassen durch View-Controller 135, 140
 - Konfiguration 141
 - Tab Bar Template 133, 135
- Bunny, Laden 424
- Button 55, 59
- Button Action 55, 59
- Buttons
 - Callback-Methode 169
 - Fertig 253
 - Text 35

C

- cabinetController 121
- CAEAGLLayer 449
- .caf-Dateien 358
- CALayer 446
- Callback-Methode 169
- Carbon-Cocoa Integration Guide (Apple) 474
- Cell Identifier 93
- center 442
- CFRelease 390
- CGContextDrawImage() 432
- CGRect 92
- Checkmark 120
- Clang Static Analyzer 562
- Classes 31
- CLLocationManager 482
- Cocoa Touch 26, 30
 - Core Foundation-Referenzen 389
 - delegieren 44, 47
 - Grundlagen, Zusammenfassung 52–53
 - Klassen 29–30
 - Reference Counting System 47
 - Schemata 513
 - Service Resolution 298
 - Unterstützung 513
 - URL Loading System 265
- Collection-Klassen 29
- Compiler-Fehler 520, 522
- Completion Process 394
- Connections Inspector 42, 66, 76
- Contact Mapper
 - AddressGeocoder 503, 507
 - Anmerkungen 497, 507
 - Anmerkungen auswählen 508–509
 - Beschreibung 493–494
 - eine Karte anzeigen 494, 497
 - eine Karte zentrieren 500
- ContactAnnotation 498
- Containerformate 326
- contentView 105
- Core Animation 439, 452
 - anpassen 443
 - Delegation 444
 - Layer 447, 449
 - OpenGL ES 449, 452
 - Überblick 439–440
 - UIView 440, 446
- Core Animation for OS X: Creating Dynamic Compelling User Interfaces (Dudney) 444, 446, 449
- Core Audio 387, 401
 - Audio Conversion Services 399
 - Audio File Services 398
 - Audio File Stream Services 399
 - Audio Queue Services 398
 - Audio Session Services 397–398
 - Audio-Units 399, 401
 - Einstellungen 362
 - OpenAL 401
 - prozedurale C-APIs 388, 390
 - Systemklänge 391, 397
 - Überblick 387
 - Vier-Zeichen-Codes 369
- Core Data 206, 226, 258
 - API für persistente Daten unter Mac OS X (Apple) 244, 246
 - Core Data-Konferenzprojekt
 - Core Data-Stack 232, 234
 - erstellen 232
 - Klassen modellieren 235
 - Modellklassen 242
 - Navigation 252, 258
 - Tabellen-Views 243, 247
 - Tracks ändern 248, 252
 - Tracks einlesen 247
 - Dokumentation 246
 - im Vergleich zu SQLite 206
 - Persistent Object Stores 231

Core Data-Konferenzprojekt 229

- Auswahl eines Vortrags 228
- Beschreibung 226, 229
- Core Data-Stack 232, 234
 - erstellen 229, 232
 - Modellklassen 242
 - Navigation 252
 - Predicate Programming Guide (Apple, Inc.) 258
 - Tabellen-Views 243, 247
 - Tracks ändern 248, 252
 - Tracks bearbeiten 227
 - Tracks einlesen 247
 - Tracks hinzufügen und löschen 227
- Core Data-Projekt 226
- Core Data-Stack
 - Aufbau 232, 234
 - Klassen modellieren 235
 - Komponenten 232
 - Modellklassen 242
 - Objekte 230
- Core Foundation
 - Cocoa 389
 - Regeln 390
 - Speicherverwaltung 390
 - String-Objekte 479
- Core Graphics 426
 - Balkengrafik 419
 - Grafikkontext 423–424
 - neu zeichnen 425
 - Pfade 418, 422
 - Vektorgrafik 416–417
 - Zeichenmodell 416
- Core Location 481, 492
 - Abstand 488
 - Genauigkeit 487
 - Kompass 491
 - Komponenten 485
 - Koordinaten 482
 - Positions-Updates 486
 - Service-Parameter 484
- create table 207
- CreateDirectoryViewController 145
- CreateFileViewController 145, 174
- createNewDirectory 170–172, 174
- createNewFile 170
- customizableViewControllers (Eigenschaft) 140

D

- Datei-/Ressourcenliste 15
- Dateisystem 143, 179
 - Alternativen 143
 - asynchrones Einlesen von Dateien 164, 168
 - Code refaktorisieren 148
 - Dateiattribute 154, 160
 - Daten einlesen 163
 - Eigenschaftslisten 178–179
 - FilesystemExplorer (Anwendung), erstellen 147, 154
 - Inhalte lesen 160
 - Navigation 144
 - Schreiben in Dateien 175, 178
 - Übersicht 143
 - Verzeichnisse 144, 146–147, 150, 154
 - Verzeichnisse erstellen und löschen 168, 172, 174
- Datenbanken 205, 223
 - auf das iPhone bringen 214
 - auf dem Desktop anlegen 213
 - Beispielanwendung 211
 - Core Data vs. SQLite 206
 - Erstellen 206
 - erstellen 209
 - Funktionsaufrufe 215
 - SQLite, Überblick 205
 - übertragen aufs iPhone 212
 - Werte einfügen 215, 218
 - Werte lesen 218, 222
- Dead Reckoning 310
- dealloc 63
- Debuggen 543
 - Compiler-Fehler 520–522
 - Debugger 530, 535, 538
 - Dokumentation nutzen 525, 527
 - Importing-/Linking-Fehler 522, 524
 - Interface Builder 528–530
 - Objekte 539, 543
 - remote mit Shark 551
- Deklarationen 52
- Delegate-Protokoll 53, 165
- Delegation 47
 - Adressbuch 473–474
 - Animation 444
 - Bilder 433
 - Location Manager 482
- deleteObject 249
- Detail Disclosure Button 120

- Detail Pane 554
- Dezibel 374
- Dictionary
 - Daten 126–127
 - Einstellungen 195
- DirectoryViewController 145, 153
- Disclosure Indicator 120
- displayName 307
- Documents-Verzeichnis 146
- Dokumentation 12, 525
 - Kürzel 527
- done 189
- doppeltes Tippen 408
- Dot 407
- drawRect 425
- Dudney, Bill 444, 446, 449
- durchschnittliche Leistung 372
- DVDCase-App 117, 121

E

- editButtonItem 95
- Editor-Bereich 15
- Eigenschaften 51, 53
 - Layer 440
 - Punktoperator 39
 - Variablen 49
- Eigenschaften (Properties) 39
- Eigenschaftslisten 178–179
- einfache View 90
- Einkaufsliste
 - erstellen 209
 - Werte lesen 222
- Einkaufszettel
 - Anwendungsdetails 211
 - Datenbank
 - Werte einfügen 215
 - erstellen 206
 - Werte einfügen 218
 - Werte lesen 218
- Einstellungen 203
 - ändern und aktualisieren 187, 190
 - Anwendung 203
 - Flippable View 181
 - flippable View 184
 - Flips vs. Bundles 196
 - laden 200, 203
 - regelmäßiges Aktualisieren (Uhr) 191–192
 - Systemanwendung 192, 199
 - verwalten 184, 187
- Einstellungsanwendung 192, 203
- einzelnes Tippen 408
- Elemente einer Tabelle 84–85
- E-Mail 289, 292
- E-Mails versenden 289, 292
- Entitäten 232, 236, 238
- Entwicklerzertifikate 22–23
- Entwicklung 64
- Entwurfsmuster 55
- Event-Driven XML Programming
 - Guide for Cocoa (Apple) 288
- Event-gesteuerter Parser 281
- Event-Modell 404–405
- EXC_BAD_ACCESS 539
- explizite Animation 440
- Extended Audio File Services 399
- Extended Detail, Instruments-App 554

F

- Fast Fourier Transform 469
- Felder 91, 94
- Fertig-Button 253
- Fetched Results Controller (FRC) 243, 247
- File's Owner 38, 107, 119, 187, 261, 529
- FileContentsViewController 145
- FileOverviewViewController 145
- FillingView 417
- Filter 463
- Filterprädikate 347
- Flippable View 181, 184
 - ändern und aktualisieren 187, 190
 - Einstellungen 184, 187
- FlippableClock 184
- FlipsideViewController 189
- Formatspezifizierer (%@) 39
- Format-Strings 531–532
- Fotos 427
- Fourier-Transformation 469
- Fragmentierung 3–4
- Frameworks 32, 522
- freigeben 48

G

Game Kit

- Datenmodus 310
- Empfangen von Daten 314
- GKSession-Kommunikation 315
- Kommunikation 311
- Netzwerkspiel 303
- Peer Picker einrichten 304–305
- Peer Picker-Delegate 306, 308
- Peer-to-Peer-Chat 316–317
- Spiellogik im Netz 308, 310
- Statusänderung 312
- Überblick 301–302
- Unterstützung 302
- gebührenfreies Wechseln 389
- Gelphman, David 424
- GKSession 307–308, 311, 315
- GPS-Empfänger 485
- Grafikkontext 423–424
- Grouped-Stil 84
- gruppierte Tabellen 84, 92
- GUI-Layout 34

H

- Hallo iPhone-Projekt 30
 - Anatomie 41
 - Gruß zurück 32
- HalloUserController 45
- Header-Dateien 27, 52, 62, 522
 - ConferenceAppDelegate 230
 - Instanzvariablen 283
 - Session-Klasse 241
 - SQLite-API 212
 - Track-Klasse 241
- Heads-Up-Display (HUD) 37
- Hochpass-Filter 463, 465
- HTTP Live Streaming 323
- HTTP-Authentifizierung 274, 277
- Human Interface Guidelines (HIG) 16

I

- IBAction 33, 37
- IBOutlet 33, 37, 51
- Identifizier 23
- im Code navigieren 21
- IMA4 364
- implementieren 70
- Implementierungsdatei 27

- implizite Animation 440
- Importing-/Linking-Fehler 522, 524
- indexPath 89
- init-Methode 62
- initWithSessionID 307
- insertTrack 248
- Installation, Developer Tools 13
- Instanzvariablen 51–53, 71, 88, 125, 283
 - Daten in Dateien schreiben 175
 - Flippable Views 186
- Instruments 12
- Instruments-Anwendung 551, 557
- integrated:// 516
- Integration 511, 517
 - Anwendungen starten 511, 513
 - Überblick 511
 - Vorbereitung 513, 517
- Interaktive Anwendung 32
- Interface Builder (IB) 12, 16, 19, 38
 - Bilder 429
 - Browser erstellen 261
 - Dateitypen 19
 - debuggen 528–530
 - GUI-Layout 34
 - Im Code navigieren 21
- Internet 259, 292
 - Daten lesen 265
 - Einen Browser erstellen 260, 265
 - E-Mails versenden 289, 292
 - HTTP-Authentifizierung 274, 277
 - Netzwerkdaten lesen 270
 - Überblick 259
 - XML parsen 289
- Internet Low-Bitrate Codec 364
- Introduction to Quartz 2D
 - Programming Guide (Apple) 424
- iPhone
 - Anwendungen ausführen 22
 - Audioaufnahmemöglichkeiten 384
 - Audioformate 358
 - Bildformate 428
 - Dateisystem 144
 - Grundlagen, Zusammenfassung 52–53
 - Interface Builder 19
 - Internetnutzung 259
 - Medienanwendung 330
 - Multi-Touch 403
 - SDK-Dokumentation 525, 527
 - Software-Stack 26
 - Tastatur 46

- Terminologie in diesem Buch 12
- Tools 12
- Versionen 184, 302, 384, 436, 456
- iPhone Application Programming Guide (Apple) 327
- iPhone Dev 12
 - iPhone Developer Program 12
- iPhone Development Book (Apple) 562
- iPhone Development Guide 562
- iPod 329, 354
 - abspielen 339
 - Abspielstatus 332, 335
 - Anwendungsname 330
 - Benutzerschnittstelle 330, 332
 - Bibliothek 341, 351
 - Medienbibliothekssuche 345
 - Medien-Queues 342
 - Player 329
 - Player-Benachrichtigungen 337, 339
 - Steuerung 339, 341
 - Zeit 352
 - Zeiteigenschaften 335, 337

K

- Key-Value Coding (Apple) 242, 246
- Klasse Movie 61
- Klassen 29
- kodierte Audioformate 364, 367
- Kompass 491
- Kreise 418
- Kurven 418

L

- Label 17
- Layer 439, 447, 449
 - CAEAGLLayer 449
 - CALayer 446
 - OpenGL 449, 452
- layoutSubviews 451
- Library 16, 147
- libsqlite3.dylib 212
- libXML2-Bibliothek 503
- Linear PCM 359–360, 362, 364
- Linien 418
- Linking-Fehler 522, 524
- Löschen 94, 97
- Lossy 364
- Lösung 565

M

- Magnetometer 491
- Mail, Navigation 115–116
- main() 43
- make-table-script 208
- malloc 542
- Managed Object Context (MOC) 227, 232
- Managed Object Model (MOM) 232–235, 242
- Map Kit 493, 509
 - Contact Mapper
 - AddressGeocoder 503, 507
 - Anmerkungen 497, 507
 - Anmerkungen auswählen 508–509
 - Eine Karte anzeigen 494, 497
 - Eine Karte zentrieren 500
 - Überblick 493–494
 - Überblick 493
- Map View 495
- Marker 125
- Media Player-Framework 341
- Medienformate 328
- Methoden, Kartenanmerkungen 497
- Methodenaufrufe 27
- Methodendeklaration, Syntax 34
- Methodendeklarationen 52
- Methodensignaturen 46
- MKAnnotation 497, 507
- modalTransitionStyle 184
- Modell erstellen 61, 64
- MovieEditorViewController 83
- Movie-Klasse 61, 64–65
- MPMoviePlayerController 319, 324
 - Benutzerschnittstelle 321–322
 - Nachrichten 324–325
 - unterstützte Medienformate 328
 - Video auswählen 322, 324
- Multi-Touch 403
 - Berührungen verfolgen 406, 408
 - Event-Modell 405
 - Gesten 410, 414
 - tippen 408–409
- Multi-Touch-Tastatur 76
- Multivalue-Objekte 479

N

Nachrichten 27
 Name/Wert-Paare 195
 Navigation 85, 87, 115, 129
 Anwendungen 117, 121
 Controller 116–117
 Mail 115–116
 Navigationsleiste, Anpassen
 122, 128
 Tracks 252, 258
 View-Controller 122
 View-Controller entfernen 128
 Navigation Controller 86, 117
 Navigation Item 86
 navigationsbasiertes Anwendungs-
 Template 230
 Navigations-Controller 100, 116
 Navigationsleiste 86
 Navigations-Objekt 102
 Navigations-Objekte 86
 navigieren 96, 104
 Netzwerkschlüssel 309
 Netzwerkspiele 303
 neu zeichnen 425
 New Project 13
 nextResponder 404
 .nib-Dateien
 Besitzer 36
 hinzufügen 74
 Inhalt 42
 Tab-Controller 139
 Nicht-FTP-Protokoll 268
 Nicht-HTTP-Protokoll 268
 Notification Center 337
 NSDateFormatter 186
 NSFileManager 152
 NSLog() 531–532
 NSLog() 540
 NSNetServiceBrowser 297
 NSOutputStream 175
 NSSStream 175
 NSString 178
 NSTimer 191
 NSZombies 540

O

ObjectAlloc, Instrument 555
 objectForKey 138

Objective-C 25, 27
 Eigenschaften 49, 51
 Methodenaufrufe 27
 Methodendeklaration, Syntax 34
 Notwendigkeit 28
 Parameter 28
 Programmieren 28
 vs. prozedurales C 388
 Objective-C-Protokoll 44
 Objects 52
 Objekte 230
 als Zeiger 522
 Autorelease 48
 Befehlsobjekte 404
 Core Data-Stack 232
 debuggen 539, 543
 delegieren 44, 47
 Multivalue-Objekte 479
 opaque Typen 388–389
 Over-Release 540
 Speicher 53
 verarbeitende Objekte 404
 Observer 337
 opaque Typen 388–389
 opaque 450
 OpenAL 401
 OpenGL ES 449, 452
 OpenGL SuperBible
 (Wright, Lipchak & Haemel) 450–451
 openURL 512
 Outlets 261
 Action 33, 64–65

P

P2PTapWar 303
 Parameter 28, 389
 Parsen von XML 289
 pause 370
 Peer Picker einrichten 304–305
 Peer Picker-Delegate 306, 308
 Peer-to-Peer-Chat 316–317
 Peer-to-Peer-Netzwerke 293, 317
 Bonjour 293, 295
 Bonjour Services 296, 301
 Game Kit, GKSession 311, 315
 Game Kit, Netzwerkspiel
 einrichten 303
 Game Kit, Peer Picker einrichten
 304–305

- Game Kit, Peer Picker-Delegate 306, 308
- Game Kit, Peer-to-Peer-Chat 316–317
- Game Kit, Spiellogik 308, 310
- Game Kit-Überblick 301–302
- Überblick 293
- Performance 562
 - Clang Static Analyzer 558, 562
 - Instruments vs. Shark 553
 - Instruments-Anwendung 551, 557
 - Profiling 549–550
 - Shark 546, 551
- Persistent Object Store (POS) 231, 233
- Persistent Store Coordinator (PSC) 232
- Pfade 418, 422
- Picker 187–188, 210, 379, 434, 499
- Pin-Farbe 501
- Plain-Stil 84
- Platzhaltertext 78
- Podcast-Objekteigenschaften 333
- Position 481, 492
 - Abstand 488
 - Delegate 482
 - Genauigkeit 487
 - iPhone-Komponenten 485
 - Kompass 491
 - Koordinaten 482, 486
 - Updates 486
- Preferences 147
- PreferenceSpecifiers 194
- Primitive 30
- Primzahlenanwendung 532
- Products 32
- Profiling 549–550, 557
- Programming with Quartz (Gelphman & Bunny) 424
- Properties Inspector 360
- Protokolle 44, 53
- prozedurale C-APIs 388
- Pulse Code Modulation (PCM) 359, 362
- Punktoperator 39
- Push Notifications 571

Q

- Quelldateien 31
- Queue 342, 351

R

- record 370
- removeItemAtPath:error: 173
- Research Assistant 527
- Resolution 298
- Resource Specifier 515
- Responder-Kette 404
- Ressourcen 31
- Root.plist 194
- RootViewController 86, 103, 148
- Rows 83
- Run Loop 165

S

- Sandbox-Sicherheitssystem 144
- save 249, 254
- Scan-build 559–561
- Schall messen 374
- Schlüssel/Wert-Paare 111, 195, 363
- Schlüsselbildanimationen 444
- Schüsselpfad 246
- Schreiben in Dateien 175, 178
- Schüttelerkennung, Accelerometer 455–456
- Scrubbing 331, 339
- selectedTimeZone 190
- Service Record 298
- Session 256
- Session (Entität) 238
- sessionMode 308
- setMessage 514
- Settings.bundle 194
- sharedApplication 512
- Shark 546, 553
- Signaturen 46
- Simulator 454
- Softwarearchitektur des iPhone 29
- Software-Stack 26
- Sort Descriptor 136
- Sortierdeskriptoren 111
- Sortieren 110
- Sortieren der Tabelle 113
- Speicher 555–556
 - Core Foundation 390
 - Eigenschaften 49
 - freigeben 40, 53
 - Verwaltung 47–48, 390
- Speicherverwaltung 41, 50
- Spiellogik 308, 310

- Spitzenleistung 372
- sqlite3 206–207
- sqlite3_open() 215
- SQLite-C-API 212
- SQLite-Datenbank 205, 223
 - auf das iPhone bringen 214
 - auf dem Desktop anlegen 213
 - aufs iPhone übertragen 212
 - Beispielanwendung 211
 - erstellen 206, 209
 - Funktionsaufrufe 215
 - Homepage 205
 - Überblick 205
 - Werte einfügen 215, 218
 - Werte lesen 218, 222
- Startbild 569
- statische Analyse 558
- Stauben-/Strecken-Geste 413
- Steinberg, Daniel 26
- stop 370
- Streaming-Unterstützung 323
- Streams 161, 163, 167
- String Programming Guide for Cocoa (Apple) 532
- String-Repräsentation 39
- Strings
 - Formatangaben 39, 531
 - Format-Strings 532
- Struts und Springs (Konfiguration) 20
- Suchergebnisse 348
- Symbole 570
- Symbolliste 21
- synthesize-Anweisungen 62
- synthetisieren 50
- Systemanwendung für
 - Einstellungen 192
- Systemklänge 391, 397
- SystemSoundsDemo 391, 397

T

- Tab-Controller 131, 142
 - Anwendung 131, 133
 - erstellen 133, 135
 - mehrere 140, 142
 - View-Controller 135, 140
- Tabellen bearbeiten 94–97
- Tabellen editieren 99
- Tabellendaten modellieren 87, 90

- Tabellendaten sortieren 110
- Tabellenfelder 90
- Tabellen-View 84–85
- Tabellen-View Data Source 84
- Tabellen-View Delegate 85
- Tabellen-Views 83, 112
 - Medien-Queue 348
 - Tab-Controller 137
- Tags 108
- Tastatur 46, 71, 76
- Technical Note TN2188: Exporting Movies for iPod, Apple TV, and iPhone (Apple) 328
- Templates 13, 133
 - MediaPlayer 320
 - Navigations-basiert 230
 - Utility Application 184, 343
 - View-based 260
- temporäres Verzeichnis (tmp) 147
- testen 572–573
- Textfelder benennen 35
- textFieldDidEndEditing 73, 512
- textFieldShouldReturn 512
- The Objective-C 2.0 Programming Language 26
- Thread-Handling 50
- Tidy 504
- Tiefpass-Filter 463, 465
- timeLabel 187
- Timeline, Instruments-Anwendung 553
- Tippen 409
- Toolbar 15
- Touch Fighter II 468
- TouchXML 503–504, 506
- Track 227, 236
- Track einlesen 247
- TrackEditingViewController 253
- Tracks ändern 248, 252
- Tracks einlesen 247
- Track-Tabellen-Views 243, 247
- transform 447
- TrivialCompilerErrors 520
- twentyFourHourSwitch 187
- Twitter 277
 - Daten laden 279
 - Daten parsen 281

U

- Über-freigegebene Objekte 540
- Uhr regelmäßig aktualisieren 191–192
- Uhranwendung 132
- UIAccelerometerDelegate-Protokoll 458
- UIBARBUTTONITEM 103
- UIButton 508
- UIImage 428
- UIImage-Objekt 430, 432
- UIImagePickerController 432, 436
- UIImageView 428
- UIKit 403
 - Animations-Layer 447
- UIKit-Framework 30
- UIPickerView 378
- UIResponder 455–456
- UISearchDisplayController 346
- UITabBarDelegate 142
- UITABLEVIEW 90
- UITableView 84–85, 87, 91, 93
- UITableViewCell 105
- UIView 440, 446
- UIWebView 260, 265
- unkomprimierte Audioformate 359, 364
- Unterscheidungsmerkmal 565
- Unterstrich 297
- updateClockView 187
- URL Loading System 265, 268
- URL, Anwendung starten 513
- User Interface
 - rotieren, Anpassung 19
- uses24Hour 190
- Utility Application Projekt-Template 343
- Utility Application Template 183–184

V

- Variablen 49
 - Eigenschaften 51
 - Outlets 33
- Vektorgrafik 416–417
- Veränderbarkeit 50
- verarbeitende Objekte 404
- verlustbehaftet 364
- Verzeichnisse 146, 150, 154
 - erstellen und löschen 168, 172, 174

- Video 319, 328, 437
 - MPMoviePlayerController 319, 324
 - Benutzerschnittstelle 321–322
 - Nachrichten 324–325
 - unterstützte Medienformate 328
 - Video auswählen 322, 324
 - Überblick 319
- Vier-Zeichen-Codes (4cc) 369
- View-based Application Template 13–14, 260
- View-Controller (VC) 45, 55, 68
 - Accelerometer 459
 - Eigenschaften und Anpassung 122, 128
 - entfernen 128
 - mit Views verbinden 157
 - MovieEditorViewController 83
 - Navigation 122
 - per Code erzeugen 156
 - Tabs 132, 135, 140
 - verbinden 128
- viewWillAppear 69
- viewWillDisappear 167
- Voice Chat 302, 316–317
- Vorwärts-Deklaration 65, 79

W

- Währungs-Formatter 74
- Webbrowser 260, 265
- Webservices 289
- Werben für Anwendungen 575
- Where's the Wiimote? Using Kalman
 - Filtering to Extract Accelerometer Data (Rasco) 470
- Wow-Faktor 567

X

- Xcode 13, 505
 - App Bundle 16
 - Application Definition
 - Statement 565
 - Autocompletion 525
 - Build Results-Fenster 521
 - Debugger-Fenster 535
 - Debugger-Konsole 529
 - Dokumentation 12
 - Editor-Bereich 15
 - Mediaplayer hinzufügen 320

- refaktorisieren 148
- Research Assistant 527
- Shark 546, 551
- Syntax-Highlighting 520, 525
- Templates 13, 133
- Toolbar 15
- Utility Application Template
 - 183–184
- Wichtigkeit 32
- .xib-Dateitypen 528
- XML parsen 286, 289

Z

- Zeichnen 426
 - Balkengrafik 419
 - Grafikkontext 423–424
 - Modell 416
 - neu zeichnen 425
 - Pfade 418, 422
 - Vektoren 416–417
- zeichnen
 - Vor dem Programmieren planen
 - 567
- Zeiger 539
- Zeitprofilfenster, Shark 548
- Zeitverläufe 443
- Zero Configuration Networking, The Definitive Guide (O'Reilly) 301
- Zertifikate für Entwickler 22–23
- Ziel/Action-Paradigma 60
- Zielgruppe 565
- Zombie-Objekte 539, 543
- zoomen 413
- zPosition 447