Was ist der Borland C++Builder ?

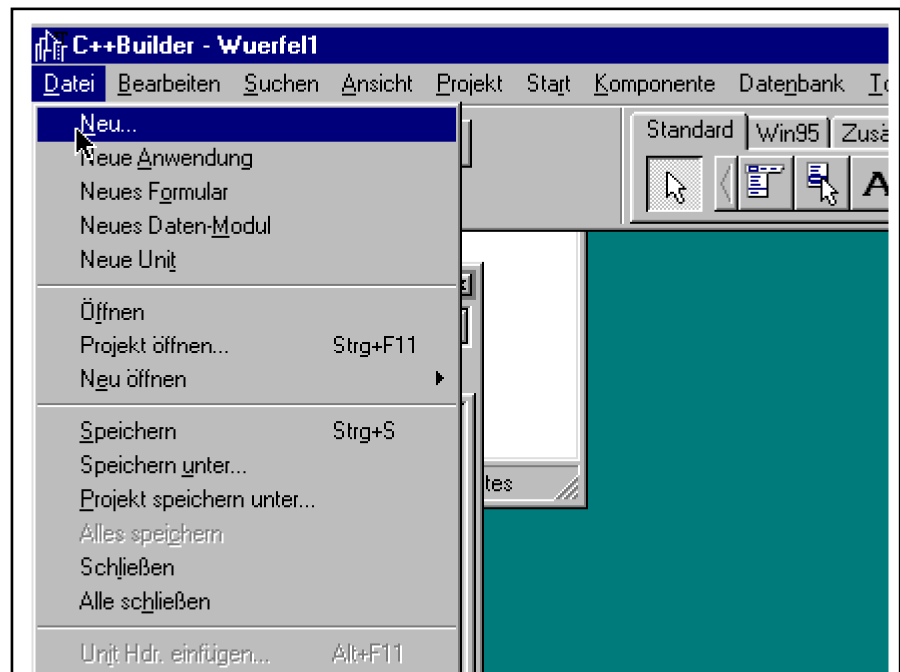
Dieses Teil ist ein sogenanntes *RDT* (***R*** apid ***D*** evolepment ***T*** ool), mit dem sich schnell, und unkompliziert Anwendungen für ein *GUI* (***G*** raphic ***U*** ser ***I*** nterface) wie Windows 95 erstellen lassen, ohne dass der Programmierer sich um Einzelheiten des *API* (***A*** pplica-tion ***P*** rogamming ***I*** nterface) kümmern muss.

D. h. , der Builder bringt eine vorgefertigte Anwendung mit, die sich "schon zu benehmen weiss", also ein Fenster besitzt, das auf Mausklicks reagiert, aufgefrischt wird, wenn es nö-tig ist, usw. usf.

Die einfachste Anwendung, ein Formular

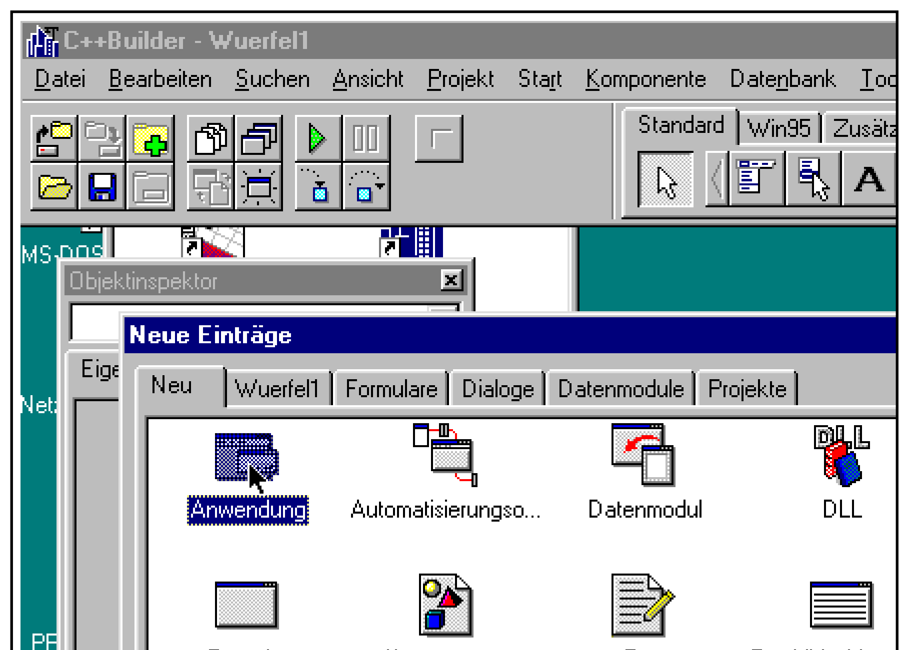
Im folgenden ist nun in Bild-schirmschnappschüssen die Entstehung einer Simpel-An-wendung, realisiert als For-mular, dokumentiert.

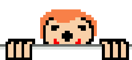
Wir wählen den Menuepunkt `Datei\Neu...` (die drei Punkte hinter dem Menue-punkt zeigen an, dass ein Di-alog folgen wird)



Wir wünschen uns eine An-wendung; was das ist, krie-gen wir gleich.

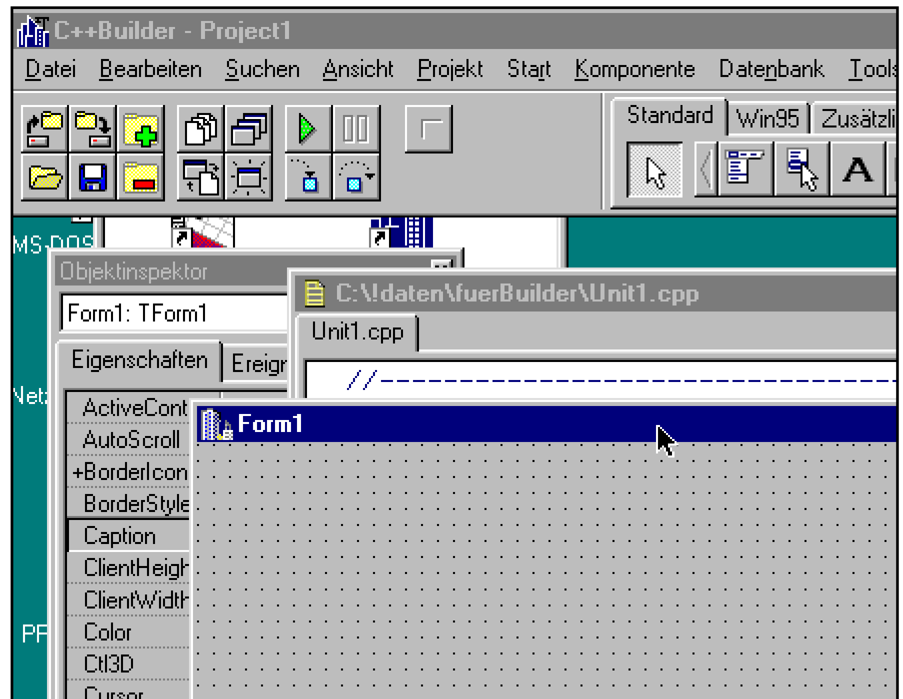
Hier hätten wir auch anderes auswählen können,ein neues Formular, eine DLL oder was immer.





Sobald wir unseren Wunsch, eine neue Anwendung zu erstellen, geäussert haben, bekommen wir sie.

Wir sehen folgende Fenster:

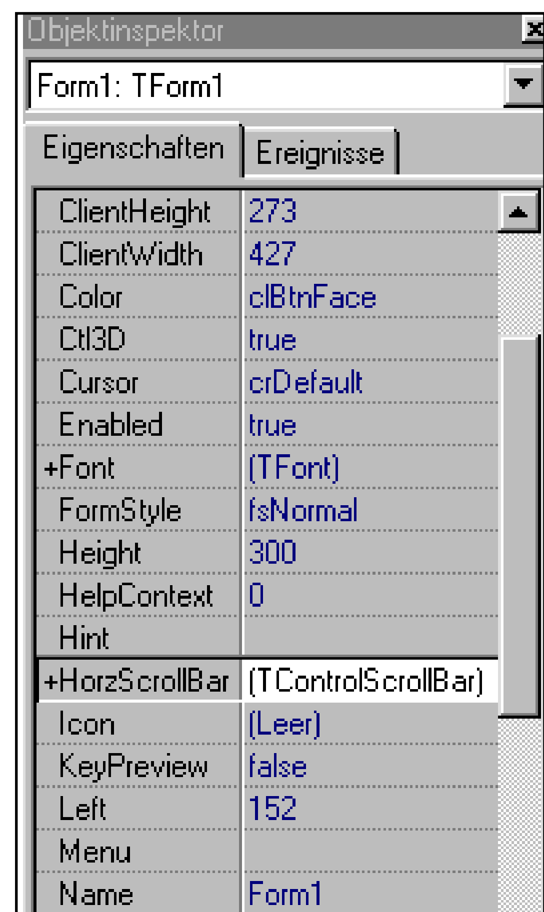


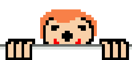
- Ein Fenster mit dem Titel `Form1`.
  Auf diesem Fenster befinden sich Rasterpunkte wie in einem Zeichenprogramm. Hier können wir unsere Bedienelemente (Komponenten wie Schaltflächen, Textfelder etc plazieren, die wir aus einer Werkzeugleiste mit der Maus in dieses Fenster ziehen.

- Ein Fenster mit einer Pfadangabe, hier `C:\!daten\fuerBuilder\Unit1.cpp`. Das ist der Quelltext der Formular-Unit, der von der *IDE* (***I*** ntegrated ***D*** evelopment ***E*** nvironment, das ist:Integrierte Entwicklungs-Umgebung) generiert wird.

- Das unterste Fenster ist der sogenannte Objektinspektor, hier rechts in voller Größe.
  Unser Formular ist ein Objekt vom Datentyp `TForm1` und hat den Bezeichner `Form1`.
  /*
  Im oberen Feld steht Form1: TForm1. Das ist eine Erbe von Pascal, in dem Objekte in der Reihenfolge `<Bezeichner> : <Typenbezeichner>` deklariert werden. Wir lesen das als **`class`** `TForm1 Form1`.
  */
  Die Bezeichner werden von der IDE automatisch vergeben. Wie das im Einzelnen funktioniert, gucken wir uns gleich genauer an.
  Mit ein bisschen Fantasie sollte klar sein, dass das Ändern der Eigenschaften, z.B. `left` in 252, dafür sorgt, dass unser Formularfenster nach rechts rutscht.

  Wenn wir die Eigenschaft Name ändern, hat das Einfluss auf den generierten Quelltext. Dazu später Genaueres





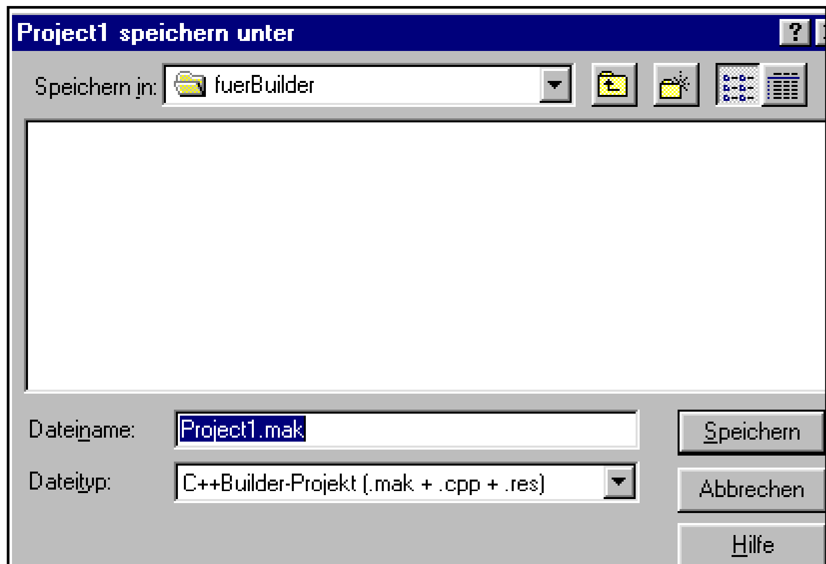
Wir sichern unser Projekt

Wir wählen uns als erstes den Menuepunkt `Datei\Projekt speichern unter...`
Jetzt gibt es etwas Anlass zur Verwirrung. Als erstes werden wir aufgefordert, die Datei `Unit1.cpp` zu speichern, also die Quelltextdatei, in der die Methoden der Klasse `class TForm1`, also unserer Formularklasse implementiert sind.

/*
Die Quelltexte der einzelnen Module eines Projekts werden von der IDE, wenn sie von ihr generiert werden, immer nach dem Muster `Unitx .cpp` benannt. Das vorangestellte "`Unit`" ist auch ein Erbe der PASCAL-Vorgeschichte.
*/

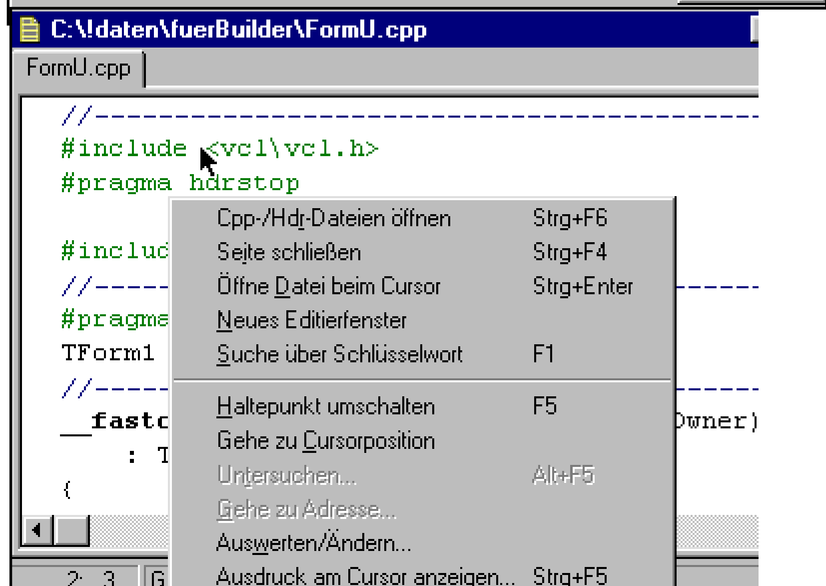
Ich finde eine aussagekräftige Bezeichnerwahl besser und wünsche mir als Dateiname `FormU.cpp`, das *`U`* ist noch eine Erinnererung an *`Unit`* , das *`Form`* steht hier für Formular.

Die IDE schlägt jetzt für den Namen des Projekt `Project1.mak` vor.

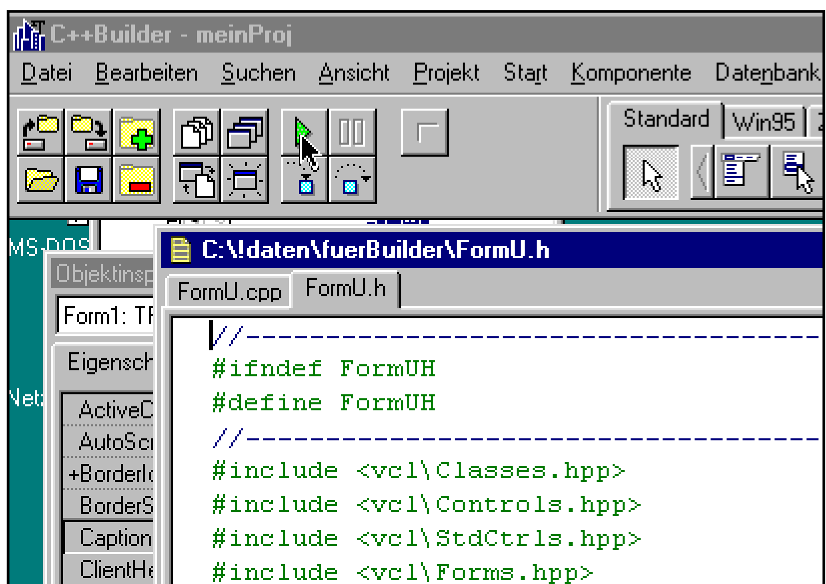


Das gefällt mir nicht und ich wünsche mir `meinProj.mak`.

Die Dateiendung `.mak` ist bei ordnungsgemäßer Installation des Builders mit ihm verknüpft. Ein Doppelklick darauf startet den Builder und öffnet das Projekt.
Zu den Dateitypen kommen wir gleich.



Der Builder legt (wie sich das gehört) standardmäßig zu (fast) jeder *xyz* `.cpp`-Datei eine *xyz* `.h`-Datei an. Ein Klick mit der rechten Maustaste in `FormU.cpp` eröffnet ein Kontext-Menue, aus dem wir `CPP-/Hdr.-Datei öffnen` wählen. Das sind die kleinen Annehmlichkeiten.

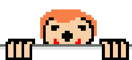


Wir sehen nun die zugehörige Datei `FormU.h`, die standardmäßig mit dem bewährten

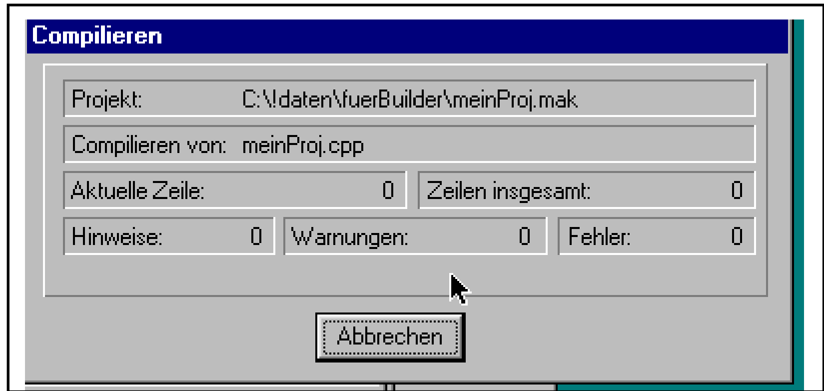
```
#ifndef xyz H
#define xyz H
    //.....
```

`#endif`- Mechanismus ausgestattet ist.

Wir starten unser Projekt, indem wir den grünen Pfeil betätigen.

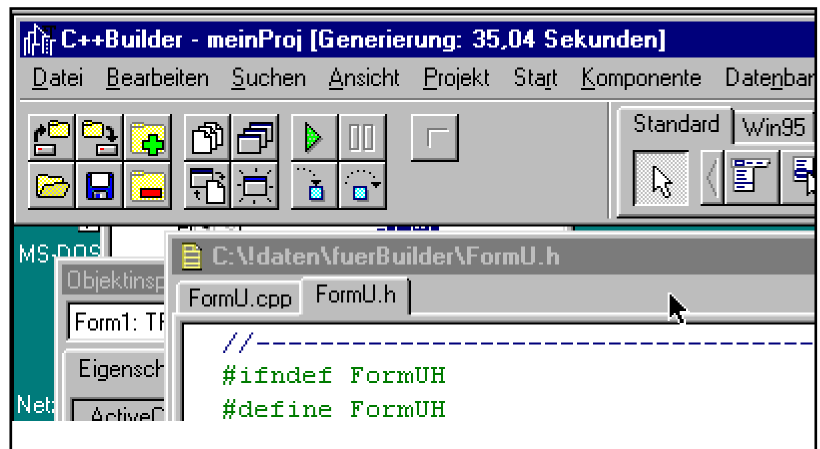


Es kommt ein Statusfenster und beim ersten Lauf eine Abschlussmeldung.

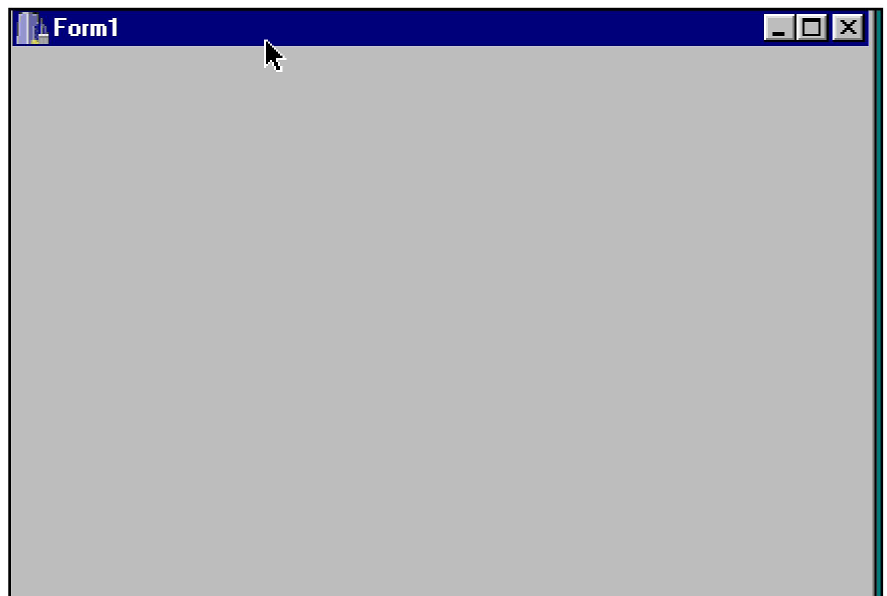


Nicht immer dauert es 35,04 Sekunden, das hängt von vielen Faktoren ab, nicht nur von den MHz der CPU.

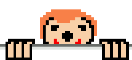
Beim ersten Lauf werden einige MB an temporären Dateien erzeugt, und wer sein Projekt auf einem Netzwerklaufwerk anlegt, sollte nicht soviel über die Armut des Bundeslandes Bremen (wg. lahmer Rechner) lamentieren sondern sich an die eigene Nase fassen.



Unser Projekt, genauer gesagt das ausführbare Programm `meinProj.exe` wird unter der Kontrolle der IDE gestartet und präsentiert sich auf dem Monitor.



Die Basisfunktionalität eines Windows-Programms ist vorhanden.Das Fenster lässt sich an der Titelleiste anfassen und verschieben, geht in den Hintergrund, wenn -sagen wir- ein Explorer-Fenster in den Vordergrund kommt, läßt sich minimieren und maximieren, in der Größe ändern und schließen.



Die Dateitypen des Builders

Rechts nun alle Dateien in demDirectory, in dem das Projekt angelegt wurde.

Die "eigentliche" Projektatei hat den Namen `meinProj.mak`.

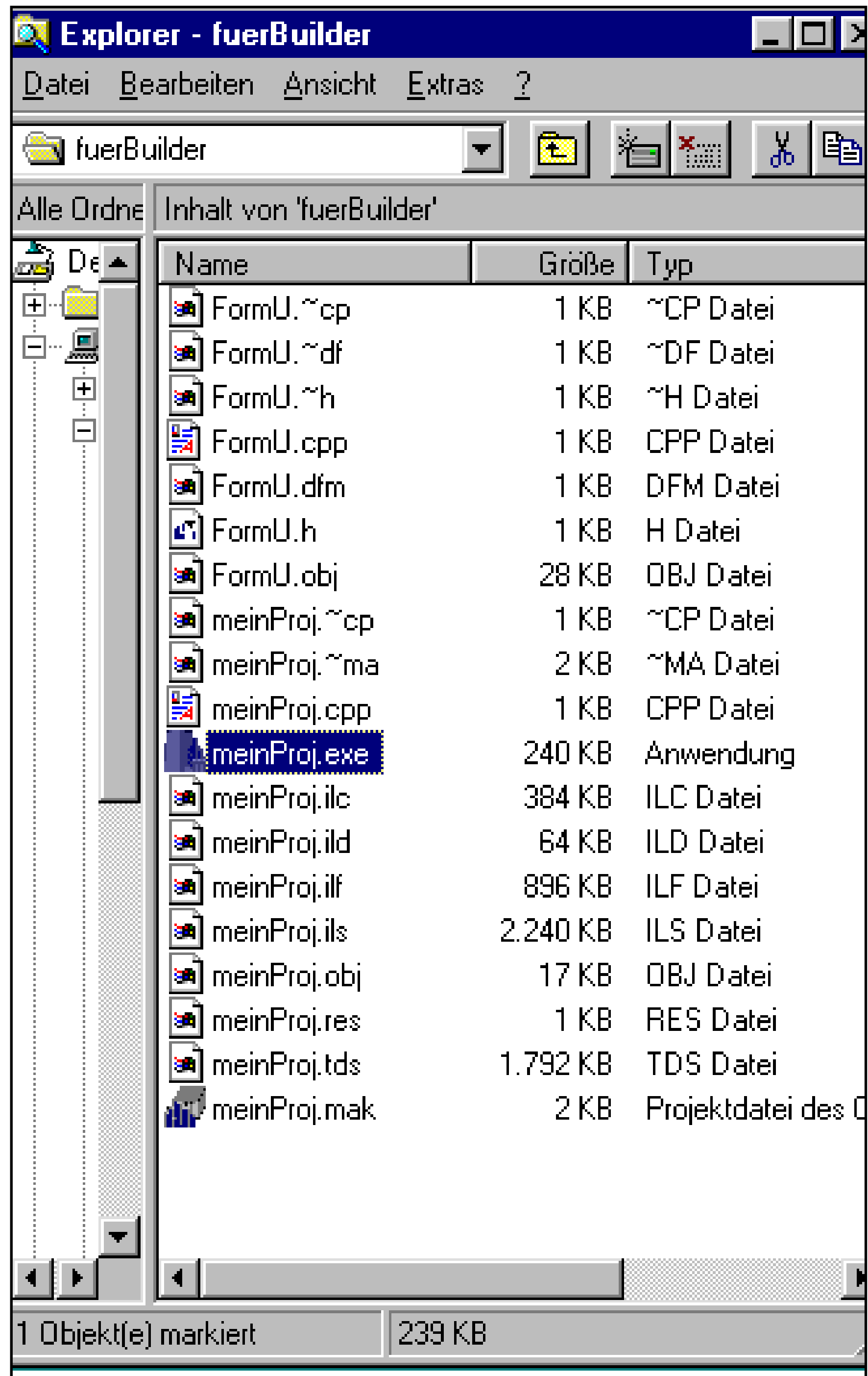
Die IDE erzeugt eine Menge weitere Dateien nach dem Strickmuster `meinProj.*`.

Unverzichtbar sind die Dateien

- `meinProj.mak`
  Projektdatei
- `meinProj.res`
  Ressourcen (Bitmaps, Sound)
- `meinProj.cpp`
  Eintrittspunkt, Funktion `WinMain()`
- `meinProj.exe`
  die ausführbare Anwendung, lässt sich neu erstellen.

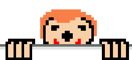
Die restlichen `meinProj.*` sind nicht nötig, wnn man sein Projekt auf Diskette nach Hause nehmen will.

Ausser den `meinProj.*` gibt es in diesem Fall nur noch die `FormU.*`.



Die Bedeutung von `FormU.cpp` und `FormU.h` ist klar, in `FormU.dfm` sind Position und Eigenschaften des Formulars und seiner Bedienelemente gespeichert.

Die `*.~*` sind Backup-Dateien , die `*.obj` sind ebenfalls verzichtbar.



Die ersten Erweiterungen

Bis jetzt haben wir nur ein leeres Formular. Das ist immerhin schon etwas.
Um ein Fenster zu programmieren, das sich "vernünftig benimmt", sind "zu Fuß", d.h. bei Benutzung der nackten Window-API, etwa 500 Zeilen C++-Quellcode nötig, wobei dieses Zeug sich auch noch ziemlich hässlich liest.

Was macht die IDE hinter unserem Rücken ?

Rechts noch einmal der Objektinspektor und ein Teil des Formulars.
Beim Anlegen des Formulars wurde automatisch `Form1` für den Namen des Formulars (Name als Bezeichner innerhalb des C++Quelltextes) vergeben.

Und nicht nur das; es wurde auch gleich eine neuer Datentyp, nämlich **class** `TForm1`**: public** `TForm` deklariert.

Da taucht nun etwas Neues auf, nämich *Vererbung* . **class** `TForm1`**: public** `TForm` ist nämlich eine neue Klasse, die all das "kann und weiss", was **class** `TForm` schon kann.

Und **class** `TForm` weiss sich eben als Fenster innerhalb eines W95-Programms zu benehmen.

Der von der IDE vorgeschlagene Name `Form1` gefällt mir nicht, ich finde `TestForm` besser. Also, rin in den Objektinspektor und die Eigenschaft `Name` geändert

Caption	Form1
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollBar)
Icon	(Leer)
KeyPreview	false
Left	219
Menu	
Name	Form1

rmU.cpp | FormU.h

```
//------------------------
#ifndef FormUH
#define FormUH
//------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//------------------------
class TForm1 : public TForm
{
__published:    // IDE-verw
private:    // Benutzer-Dek
public:    // Benutzer-Dek
        __fastcall TForm1(TComp
};
```

Objektinspektor

TestForm: TTestForm

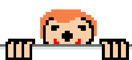
Eigenschaften | Ereignisse

Caption	TestForm
ClientHeight	273
ClientWidth	427
Color	clBtnFace
Ctl3D	true
Cursor	crDefault
Enabled	true
+Font	(TFont)
FormStyle	fsNormal
Height	300
HelpContext	0
Hint	
+HorzScrollBar	(TControlScrollBar)
Icon	(Leer)
KeyPreview	false
Left	219
Menu	
Name	TestForm

C:\!daten\fuerBuilder\FormU.h

FormU.cpp | FormU.h

```
//------------------------
#ifndef FormUH
#define FormUH
//------------------------
#include <vcl\Classes.hpp>
#include <vcl\Controls.hpp
#include <vcl\StdCtrls.hpp
#include <vcl\Forms.hpp>
//------------------------
class TTestForm : public T
{
__published:    // IDE-ver
private:    // Benutzer-De
public:    // Benutzer-De
        __fastcall TTestForm(T
};
//------------------------
extern TTestForm *TestForm
```



Huups, jetzt hat sich im Objektinspektor und in Quelltext von `FormU.h` einiges geändert.

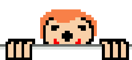
- Auf einmal ist auch die Eigenschaft `Caption` des Formulars verändert, sie lautet jetzt `TestForm`, genau so wie der Name des Formulars.
- Die Deklaration von **`class`** `TForm1`**`: public`** `TForm` gibt es nicht mehr in `FormU.h`, jetzt heisst diese Klasse **`class`** `TTestForm`**`: public`** `TForm`., und auch **`extern`** `TForm1 *Form1` ist zu **`extern`** `TTestForm *TestForm` mutiert.

Anscheinend funktioniert das so:

- Wird im Qbjektinspektor der Namen des Formulars geändert, so ändert sich auch die Eigenschaft `Caption`. (Das ist der Text in der Titelleiste des Formulars. `Caption` ist immer so etwas wie ein Etikett, das der Anwender des Programms sieht, nicht zu verwechseln mit den Bezeichnern innerhalb eines C++Quelltext!!)
- Darüberhinaus wird gemäß dem Formularnamen auch der Typenbezeichner der Formularklasse angepasst, und zwar nach dem Strickmuster vorangestelltes großes `T` und dann der Formularname. (Hier soll das *`T`* in bester Borland-Tradition wohl daran erinnern, dass eine Klasse auch immer ein Daten*T*yp ist). Die Elternklasse `TForm`, also die Tatsache, dass **`class`** `TTestform` von **`class`** `TForm` abgeleitet ist, ändert sich natürlich dabei nicht.
  Wir werden die Anpassung von Bezeichnern, seien es Objekt-, Methoden- oder Typbezeichner, durch die IDE bei Änderungen im Objektinspektor auch an anderen Beispielen sehen.
- Zu dem **`extern`** `TTestForm * TestForm` ist auch noch einiges zu sagen.
  Hier wird ein Zeiger auf ein Objekt vom Datentyp `TTestForm` *deklariert* . Das reservierte Wort **`extern`** sagt allen Modulen im Projekt (vorausgesetzt natürlich, sie inkludieren `FormU.h`), dass eine solche Zeigervariable in einem Modul des Projekts *definiert* ist.
  Wir werden gleich sehen wo.

Aufgabe:
Was ändert sich, wenn im Objektinspektor die Eigenschaft `Name` des Formulars zu `Wollebautz` geändert wird)

Die Eigenschaft `Caption` lautet jetzt `Wollebautz`
`class` `TTestForm`**`: public`** `TForm.` wird zu
`class` `TWollebautz`**`: public`** `TForm.`
`extern` `TTestForm *TestForm` wird zu
`extern` `TWollebautz *Wollebautz`



Das ersten Bedienelemente

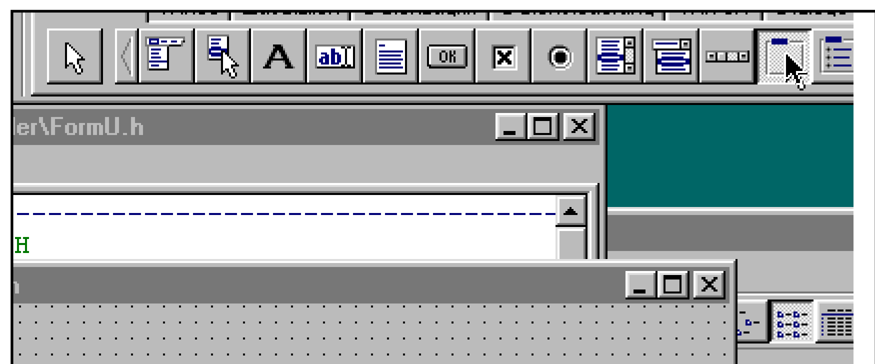
Bis jetzt haben wir nur ein leeres Formular. Das ist zwar immerhin etwas, aber zu einer "richtigen" Anwendung gehört mehr.

Wir wollen ein Formular haben, das

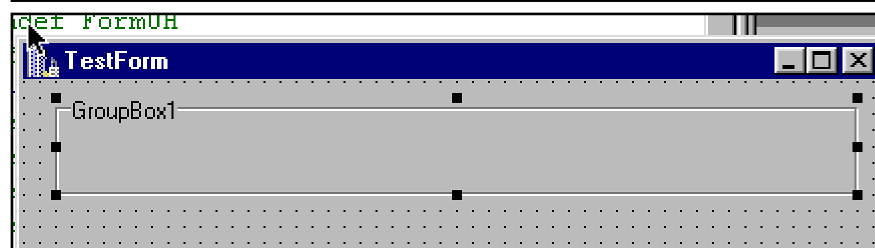
- ein Textfeld (Datentyp `TEdit`) enthält, in das der Benutzer einen Text eingeben kann.
- eine Schalfläche (Datentyp `TButton`) enthält, das beim Anklicken eine Aktion auslöst. Hier soll die Aktion ein modaler Dialog sein, der den in das Textfeld eingegebenen Text ausgibt.

Der schöneren Gestaltung wegen wird das Textfeld in eine *GroupBox* (Datentyp `TGroup-Box`) gepackt.

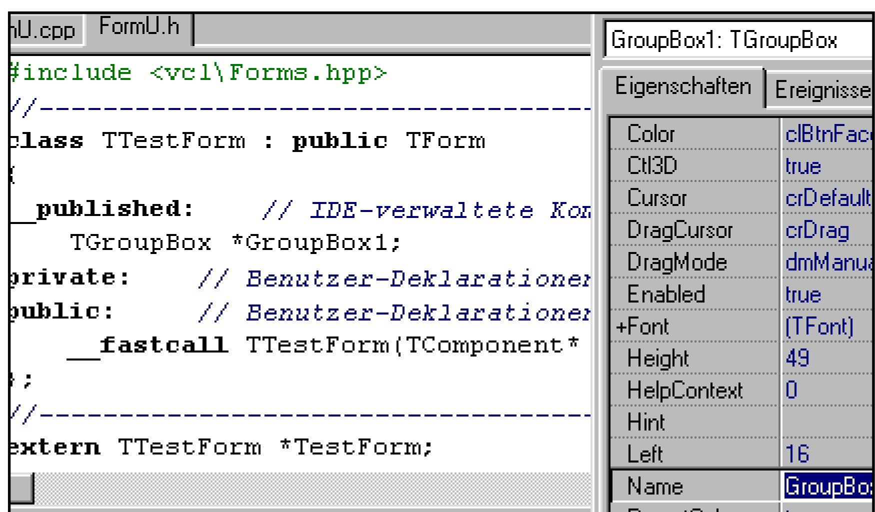
Und nun ans Werk



Mit dem Mauszeigerwird Werkzeug zur Erzeugung einer Groupbox angewählt.



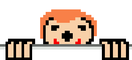
Im Formularfenster wird diese Box aufgezogen.
Oben links erscheint (Und das ist hier genau so wie beim Formular.) als Beschriftung (Eigenschaft `Caption`) der von der IDE automatisch vergebene Name (hier `GroupBox1`).



In FormU.h hat sich jetzt etwas getan. Nach dem *Zugriffsprivileg* **`__published:`** gibt es nun ein Datenfeld `TGroupBox *GroupBox1`.
Der Name gefällt mir nicht. Ich ändere ihn im Objektinspektor zu `EingGrpBox`.

Was wird sich ändern ?

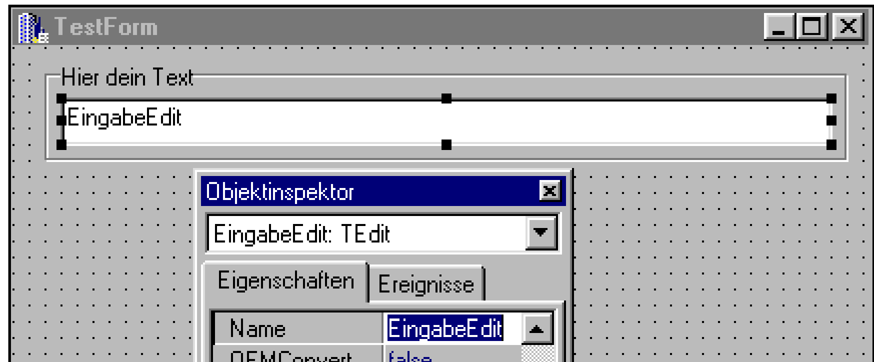
Die Eigenschaft `Caption` und somit auch der Text oben links im Formular
Hinter **`__published:`** steht jetzt `TGroupBox *EingGrpBox`.



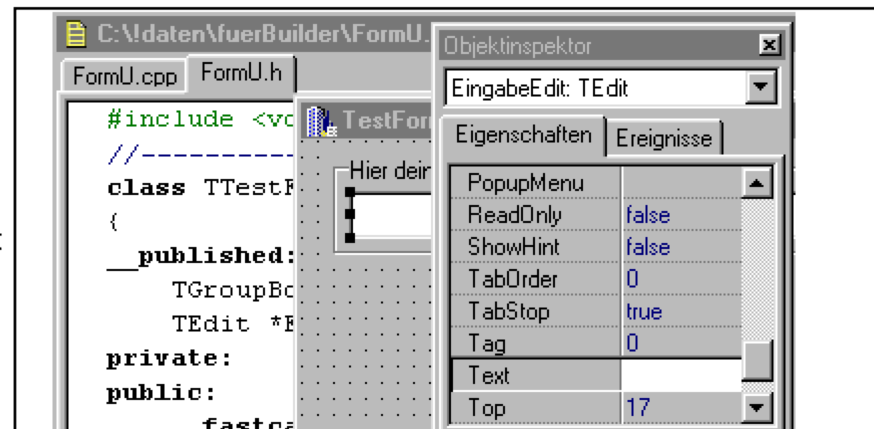
Die nächsten Schritte

In dieser Groupbox wird nun mit dem Edit-Werkzeug ein Textfeld aufgezogen.
Der automatisch vergebene Name (`Edit1`) dieses Textfelds wird abgeändert.

Die Beschriftung (Eigenschaft `Caption`) der GroupBox wird ebenfalls abgeändert.

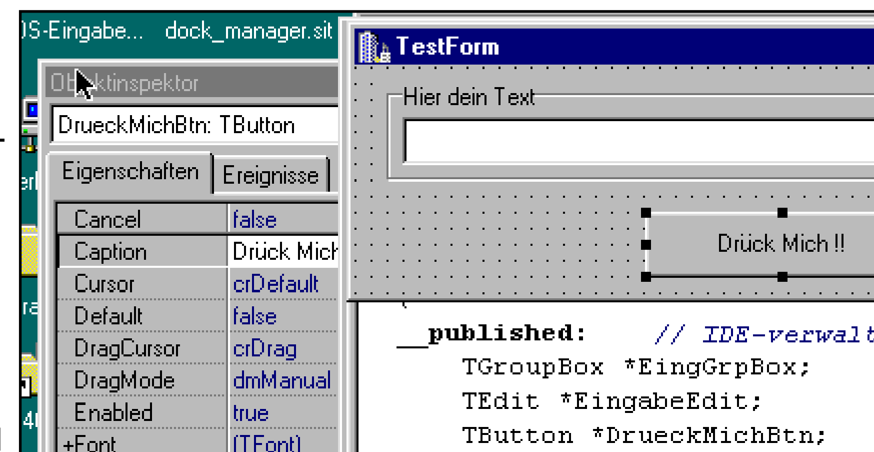


Nach der Umbenennung von `Edit1` in `EingabeEdit` taucht dieser Name nun als voreingestellter Inhalt des Textfeldes auf.
Das ist nun -im Gegensatz zum Formular und der Group-box- nicht die Eigenschaft `Caption`, sondern hier die Eigenschaft `Text`.



Beim Start des Programms soll das Textfeld leer sein, und so ändere ich im Objekt-inspektor die Eigenschaft Text zum Leerstring ab.

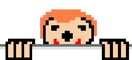
Diese Eigenschaft `Text` ist, genau so wie `Caption`, vom Datentyp `AnsiString`, das ist eine Borlandspezifische komfortable Stringklasse. Dazu gleich mehr.



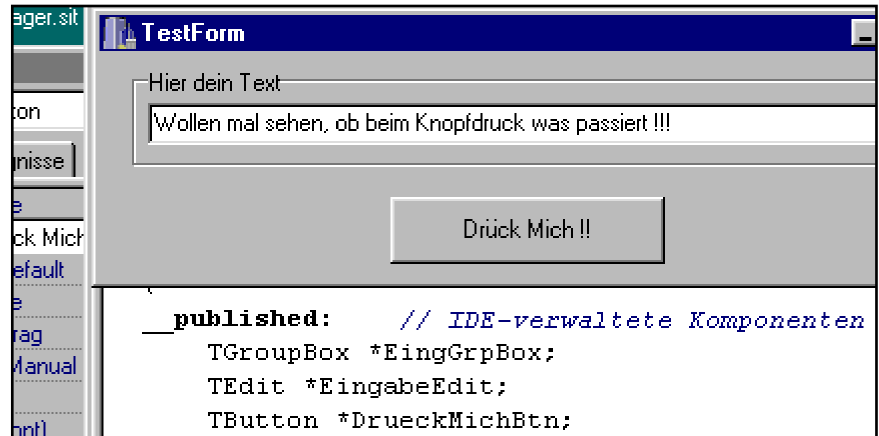
Nun fehlt nur noch Schaltfläche (Datentyp `TButton`), deren `Name` und `Caption` ebenfalls abgeändert werden.

Dazu wird aus der Werkzeugleiste das Button-Werkzeug gewählt und der Knopf wird im Formular aufgezogen.

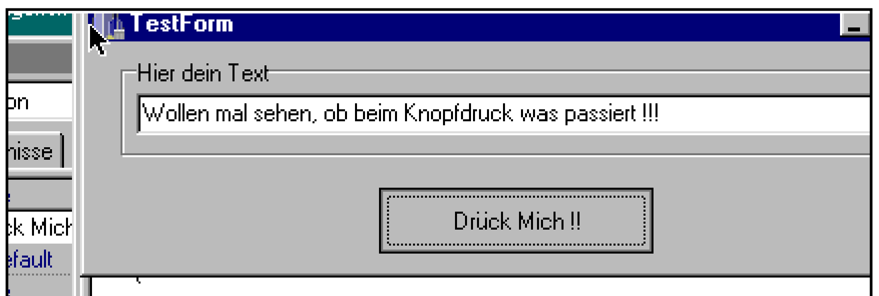
Die Größe des Formulars wird ebenfalls angepasst.



Wir haben nun eine Anwendung, erkennbar daran, dass im Formular keine Rasterpunkte sichtbar sind, und dass nun Text in das Textfeldeingegeben werden kann.

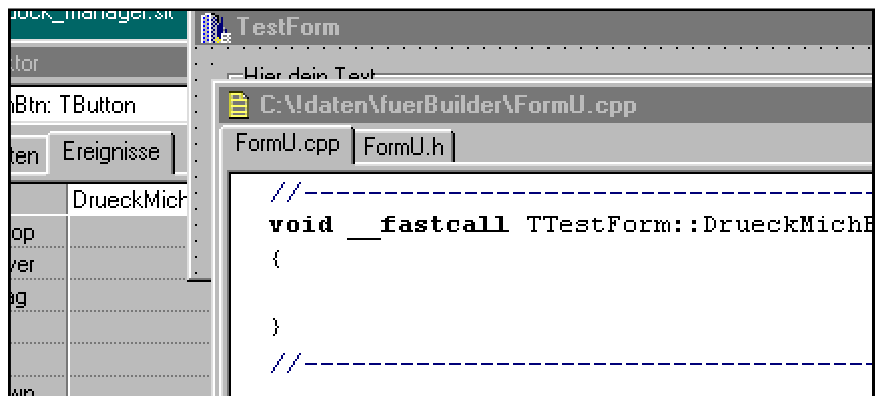


Sogar die Schaltfläche funktioniert , aber nur insoweit, was in die Klasse eingebaute Funktionalität der Klasse `TButton` angeht. Das beschränkt sich darauf, dass der Knopf beim Drücken sein Aussehen wechselt, so wie ein Knopf im richtigen Leben.



Wir hauchen ihm jetzt weiteres Leben ein.

Dazu beenden wir unsere `*.exe` durch `alt-F4` oder wie auch immer und wechseln zur IDE.



Ein Doppelklick auf die Schaltfläche produziert einige Action.

Der Texteditor bringt die Datei `FormU.cpp` in den Vordergrund, es ist automatisch eine leere Methode **`void __fastcall`** `TTestForm::DrueckMichBtnClick(TObject *Sender)` generiert worden. die natürlich in `FormU.h` deklariert ist.

Im Objektinspektor wird das Reiterfeld Ereignisse aktiviert und bei dem Ereignis `OnClick` taucht genau die eben generierte Methode `DrueckMichBtnClick()` auf

Aufgabe: Was ändert sich, wenn der Name des Formulars von `TestForm` zu `MeinFormular` und der Name der Schaltfläche von `DrueckMichBtn` zu `MeinButton` geändert wird ?

Caption des Formulars wird zu `MeinFormular`
`class` `TTestForm: public TForm` wird zu **`class`** `TMeinFormular: public TForm`
in `FormU.h` wird **`extern`** `TTestForm *TestForm` zu **`extern`** `TMeinFormular *MeinFormular`
Der Konstruktor bekommt den neuen Klassenbezeichner `TMeinFormular`,
ausserdem ändert sich der Klassenbezeichner vor dem Scopeoperator in der Datei `FormU.cpp`
Die Ereignismethode `TTestForm::DrueckMichBtnClick()` wird zu `TMeinFormular::MeinButtonClick()`.

Bevor wir dem Knopf Leben einhauchen, sollen zuerst die Dateien `FormU.cpp` und `FormU.h` im Einzelnen durchgesprochen werden

Zuerst `FormU.h`

Sie ist "eingerahmt" durch den üblichen
`#ifndef xyz`
`#define xyz`
`#endif`
-Mechanismus.

Dann gibt es einige `#include` auf die Builder-Header-Dateien.

Dann folgt der Klassenprototyp, benannt nach der Eigenschaft `Name` des Formulars, und zwar nach dem Strickmuster

```
class Txyz:
   public TForm
{
};
```

```
//---------------------------------------
#ifndef FormUH
#define FormUH
//---------------------------------------
#include <vcl\Controls.hpp>
#include <vcl\StdCtrls.hpp>
#include <vcl\Forms.hpp>
//---------------------------------------
class TTestForm : public TForm
{
__published:
// IDE-verwaltete Komponenten
    TGroupBox *EingGrpBox;
    TEdit *EingabeEdit;
    TButton *DrueckMichBtn;
    void __fastcall DrueckMichBtnClick
          (TObject *Sender);
private:
// Benutzer-Deklarationen
public:
    // Benutzer-Deklarationen
    __fastcall TTestForm(TComponent* Owner);
};
//---------------------------------------
extern TTestForm *TestForm;
//---------------------------------------
#endif
```

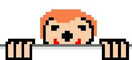
Das Borland-eigene reservierte Zugriffsprivileg **`__published`** sagt erstens aus, dass diese Zeigervariablen für alle Benutzer dieser Klasse **`public`** sind und zweitens, dass sie von der IDE verwaltet werden, d.h. Namensänderungen im Objektinspektor wirken sich hier aus. Dazu gehört auch die Ereignisbehandlungsmethode `DrueckMichBtnClick()`.

Die letzte Methodendefinition ist der Konstruktor.
Dazu gleich mehr

Alle von der IDE generierten Klassenmethoden tragen den Modifizierer **`__fastcall`**.

Das ist eine Borland-Spezialität und bewirkt, dass in Abweichung von der gängigen C++Praxis die Aufrufparameter nicht auf dem Stack, sondern in Prozessorregistern übergeben werden, was die Programme natürlich schneller ablaufen läßt.

Schließlich folgt noch die *Deklaration* **`extern`** `TTestForm *TestForm`, um allen Benutzern dieser Formularklasse, vorausgesetzt sie inkludieren `FormU.h`, mitzuteilen, dass in einem Modul diese Zeigervariable *definiert* ist. Wir sehen gleich, wo.



Zu FormU.cpp:

Hier wird ein Builder-spezifischer Header inkludiert.

#pragma hdrstop wird wohl bewirken, dass auch ohne das #ifndef-#define- #endif-Ritual das mehrfache Inkludieren verhindert wird.

/*
mit den #pragma können Einstellungen der IDE gesteuert werden, das ist allerdings herstellerspezifisch. Auf der Lieblingsplattform des Autors lautet das vergleichbare Teil #pragma once.
*/

```
//-------------------------------------------
#include <vcl\vcl.h>
#pragma hdrstop

#include "FormU.h"
//-------------------------------------------
#pragma resource "*.dfm"
TTestForm *TestForm;
//-------------------------------------------
---------------------------------
__fastcall TTestForm::TTestForm
    (TComponent* Owner): TForm(Owner)
{
}
//-------------------------------------------
void __fastcall TTestForm::
    DrueckMichBtnClick (TObject *Sender)
{
}
//-------------------------------------------
```

Dann wird FormU.h inkludiert.

Als nächstes wird TTestForm *TestForm *definiert* . Das bedeutet, die Zeigervariable auf das Formular befindet sich in diesem Modul, also nach der Kompilierung in FormU.obj.

Der Konstruktor ist noch (fast) leer.

Was Aussieht wie das Initialisieren eines Datenfeldes (nach dem Doppelpunkt und vor dem Methodenrumpf) ist hier der Aufruf des Konstruktors der Elternklasse TForm.

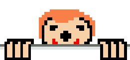
In der Implementation der Klasse **class** TApplication, deren Quelltext uns nicht ohne weiteres zugänglich ist, wird wohl eine Anweisung lauten
TestForm = **new** TTestform(**this**);
wobei der Parameter TObject *Sender der Zeiger auf das Objekt Application ist, dem Konstruktor der Elternklasse durchgereicht

Somit "weiß " unser Formular, wer sein "Herr und Meister" ist.

/* Das sind erstmal Vermutungen, die Dokumentation erweist sich dabei als recht zugknöpft. */

Die Methode DrueckMichBtnClick(TObject *Sender) ist bis jetzt noch leer. Hier gibt es als Parameter ein Zeiger auf das Objekt, das diese Methode aufgerufen hat, also ein Zeiger auf den Knopf. Das ist praktisch, wenn man die gleiche Ereignismethode auf viele knöpfe legen will, so kann innerhalb der Methode entschieden werden, wer das Ereignis ausgelöst hat.

Wir haben aber nur eine Knopf, der dieses Ereignis auslöst, und so werden wir diesen Parameter nicht verwerten.



Was soll nun unser Knopf auslösen ?

Wir wollten ein Formular haben, das

- ein Textfeld (Datentyp `TEdit`) enthält, in das der Benutzer einen Text eingeben kann.
- eine Schalfläche (Datentyp `TButton`) enthält, das beim Anklicken eine Aktion auslöst. Hier soll die Aktion ein modaler Dialog sein, der den in das Textfeld eingegebenen Text ausgibt.

Textfeld und Schaltfläche haben wir schon, nun müssen wir "nur" noch `DrueckMichBtnClick(TObject *Sender)` ausprogrammieren.

Nur wenn im Textfeld etwas steht, soll etwas passieren. Das "riech"t nach einem

```
if (xyz )
{
}
```

Wir müssen unser Textfeld daraufhin untersuchen, ob es einen Leerstring enthält. Hier ist nun der Einsatz der kontextsensitiven Hilfe angesagt. Im ObjektInspektor wird die Eigenschaft `Text` aktiviert und mit `<strg><F1>` die aktive Hilfe aufgerufen.

Es wird ein Hilfefenster zu `TControl::Text` aufgebaut.

Objektinspektor
EingabeEdit: TEdit
Eigenschaften | Ereignisse

OEMConvert	false
ParentColor	false
ParentCtl3D	true
ParentFont	true
ParentShowHint	true
PasswordChar	#0
PopupMenu	
ReadOnly	false
ShowHint	false
TabOrder	0
TabStop	true
Tag	0

Borland C++Builder Hilfe
Datei Bearbeiten Lesezeichen Optionen ?
Inhalt Index Zurück Drucken <<
Bücher

TControl::Text
TControl Siehe auch Beispiel

Die Eigenschaft Text enthält einen String, de
mit dem Steuerelement verbunden ist.

```
__property System::AnsiString
  Text;
```

Beschreibung
Mit dieser Eigenschaft können Sie
- den Text eines Steuerelements abrufen.

Hier lernen wir, dass Text vom Datentyp `AnsiString` ist.

Die (in Farbe grün) unterstrichenen Begriffe sind sogenannte Links im Hilfesystem. Ein Klick darauf öffnet den nächsten Hilfetext.

Wir können uns dann entlang der Links weiterklicken oder aber wie in einem WebBrowser mit den Knöpfen navigieren.

Objektinspektor
EingabeEdit: TEdit
Eigenschaften | Ereignisse

OEMConvert	false
ParentColor	false
ParentCtl3D	true
ParentFont	true
ParentShowHint	true
PasswordChar	#0
PopupMenu	
ReadOnly	false
ShowHint	false
TabOrder	0
TabStop	true

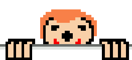
Borland C++Builder Hilfe
Datei Bearbeiten Lesezeichen Optionen ?
Inhalt Index Zurück Drucken <<
Bücher

AnsiString
Methoden Beispiel

C++Builder implementiert den Typ AnsiString
als Klasse.

Header
vcl/dstring.h

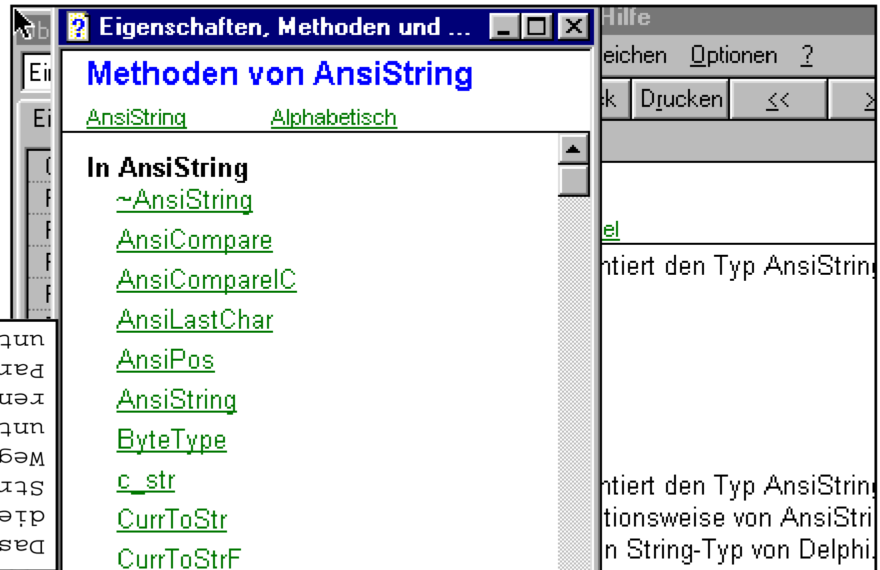
Beschreibung
C++Builder implementiert den Typ AnsiString



Spannend sind nun die Methoden der Klasse. Wir lassen servieren.

Anscheinend gibt es (mindestens) einen Konstruktor.

<u>Aufgabe</u>
Woran erkennt man das ?



Das Teil heisst genau so wie die Klasse, nämlich Ansi-String.
Wegen *overloading* können es unterschiedliche Konstruktoren sein, die sich in ihrer Parameterliste (der *Signatur*) unterscheiden.

Dieser Konstruktor gleich noch eine wesentliche Rolle spielen.

Nur wenn das Textfeld keinen Leerstring enthält, soll das Mitteilungsfenster gezeigt werden.

Zur Erinnerung ein Auszug aus `FormU.h`

```
//..
TEdit *EingabeEdit;
//..
```

Alle Komponenten eines Formulars liegen als Zeigervariablen auf eben solche Objekt vom Typ `TEdit` oder `TButton` vor.

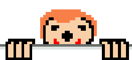
Um an die Eigenschaften oder Methoden heranzukommen, müssen wir mit Hilfe des **->** Operators *dereferenzieren* , also den "Karton aufmachen" auf den der Zeiger zeigt.

Das wäre hier `EingabeEdit->`. Nach dem **->** Operator können wir uns das Klassenelement wünschen, hier die Eigenschaft `Text`.
Nun ist diese Eigenschaft vom Datentyp `AnsiString`, und der kann eine ganze Menge.

So gibt es da eine Operatormethode **`bool operator`** `!=()`, die einen Stringvergleich durchführt. Wir müssen also nicht `strcmp()` aus `string.h` bemühen.

Deshalb lautet unser **`if`**-Konstrukt

```
if (EingabeEdit->Text != "")
{
   // Modalen Dialog zeigen
}
```



Was C++ "hinter unserem Rücken macht.

Dieses "unschuldige" `EingabeEdit->Text != ""` liefert einen Wahrheitswert, und zwar **`true`**, wenn in dem Eingebefeld ein Text steht.

Hier läuft eine ganze Menge ab. Wenn C++ ein menschlicher Interpreter wäre, würde er grübeln:
Hmm.. Hmm.. ein Vergleich (**`!=`**). Na was haben wir denn links und was rechts.
Guck mal da, links gibt es `AnsiString`, der hat
`bool operator != (const AnsiString &as)` im Programm. Dieser Operator erwartet aber rechts irgendwas in Richtung `AnsiString`.

Rechts steht aber **`char *`** als Datentyp. Das ist aber blöd, gucken wir mal in unserm Werkzeugkasten nach, ob wir da was machen können.

Ah ja, da seh' ich, `AnsiString` hat einen Konstruktor, wie gemacht dafür.

Den benutze ich, um aus dem ordinären C-Leerstring `""` einen *transienten* `AnsiString` (ein Schmierzettelobjekt) zu erzeugen, den ich dann dem Vergleichsoperator als zweiten Parameter übergeben kann.

Jede Builder-Anwendung hat eine globale Variable `TApplication *Application`, deren Methoden und Eigenschaften (soweit **`public`**) von allen Modulen genutzt werden können. Auch hier sei die aktive Hilfe empfohlen.

`TApplication` hat eine Methode

```
int __fastcall MessageBox(char *Text, char *Caption, unsigned short Flags).
```

Der Rückgabewert ist eine Nummer für den Knopf, der gedrückt wurde. Wie in C üblich, muss dieser Rückgabewert nicht verwendet werden, hier verzichten wir ebenfalls darauf.

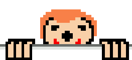
`char *Text` ist der c-String, der im Fenster erscheint, `char *Caption` der Titel in der Kopfzeile des Fensters und `unsigned Short Flags` ein Wert, der darüber entscheidet, wieviel Knöpfe mit welcher Beschriftung erscheinen. So sorgt die vordefinierte Konstante `MB_OKCANCEL`, dass es zwei Knöpfe gibt, einer zur Bestätigung und einer zum Abbrechen.

Hier soll es nur einen Knopf geben, und zwar einen, den der Benutzer quittieren muss, damit es weitergeht.

Dann muss als 3. Parameter `MB_OK` übergeben werden.

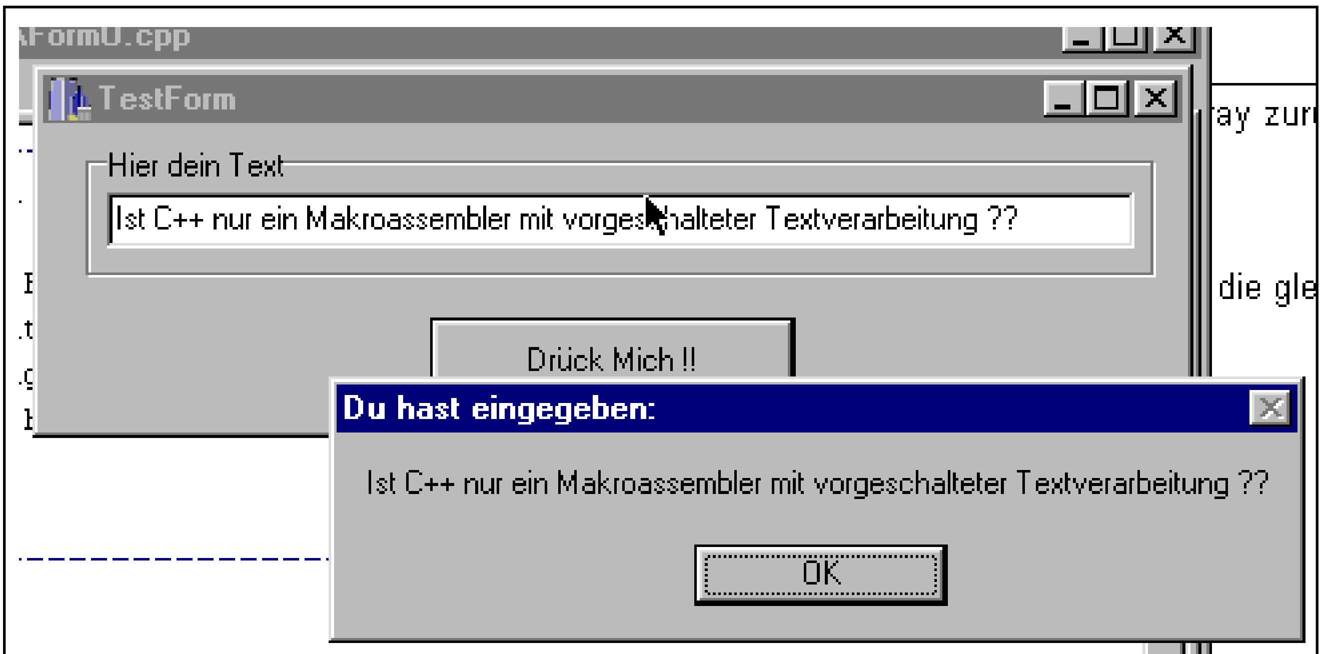
Der 2. Parameter ist auch einfach, das ist ein stinknormaler c-String, d.h. *StringLiteral* .

Nun müssen wir noch die Eigenschaft `Text` unseres Textfelds in einen c-String verwandeln. Die aktive Hilfe zeigt eine Methode `char *c_str()`, die das anscheinend kann.



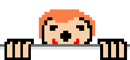
Nun "haben wir fertig"

```
void __fastcall TTestForm::DrueckMichBtnClick
  (TObject *Sender)
{
  if (EingabeEdit->Text != "")
   Application->MessageBox(
     EingabeEdit->Text.c_str(),
     "Du hast eingegeben:", MB_OK);
}
```



Wir merken uns:

- Ohne die aktive Hilfe geht nix.
- Es ist ein bisschen Fantasie nötig, um aus den Eigenschaften, die im Objektinspektor bzw. aus den Methoden, die in der Hilfe aufgeführt werden, auf den Verwendungszweck zu schließen.
  Aber das wird mit etwas Übung schnell besser.
- Alle Komponenten eines Formular liegen als `TXyz * abc` vor. Dabei ist *abc* der im Objektinspektor vergebene Komponentenname.
  Konsequenz daraus ist: Auf Eigenschaften bzw. Methoden `bcd` wird mit `abc->bcd` zugegriffen.
- Alle von Borland gewählten Bezeichner fangen (bis auf `TAnsiString::c_str()`) mit Grossbuchstaben an und es sind nach dem Muster `ErsterWortteilZweiterWortteil` Grossbuchstaben eingestreut.
  /* bei näherem Hinsehen ist das aber auch nicht ohne Witz */



Notizen