Ulf Neubert

dBASE lebt!

Band 3

Klassen und Objekte

dBASE Programmierung unter Windows

Für dBASE Plus und seine Vorgänger
dBASE SE, dBASE 2000, Visual dBASE

Copyright © 2006 Ulf Neubert

Inhaltsverzeichnis

1.7 Schwerpunkt dieser Ausgabe

Die dritte Ausgabe dieser Reihe beschäftigt sich mit *Klassen und Objekten*, sowie mit weiteren Punkten zum Thema *Objektorientierte Programmierung*.

Ich beginne mit einer grundsätzlichen Einführung des Themas und zeige an sinnbildlichen Beispielen aus dem „richtigen Leben", was unter Klassen und daraus abgeleiteten Objekten zu verstehen ist. Sie brauchen dazu keinerlei Vorwissen über *OOP* (das Kürzel für *Objekt-Orientierte Programmierung*), sollten aber etwas Erfahrung und Grundkenntnisse über dBWin besitzen.

ⓘ Band 3 baut auf dem in Band 1 *Einführung* und Band 2 *Grundlagen* erworbenen Wissen auf. Natürlich wird nicht vorausgesetzt dass Sie die ersten Bände besitzen, es genügt dass Sie die dort vermittelten Grundlagen kennen, wie auch immer Sie diese erworben haben. Ich werde aber bei Bedarf auf entsprechende Stellen der früheren Bände verweisen, falls Sie das eine oder andere Detail zu einzelnen Befehlen oder Techniken doch nachlesen wollen.

Nach einer Einführung werden wir gemeinsam eigene Klassen entwickeln, die alle von den dBWin-Basisklassen abgeleitet werden. Dabei verwende ich am Anfang die visuellen Klassen, das sind die Dinge die Sie in Ihren Formularen verwenden, also Texte, Eingabefelder, Buttons, Rechtecke, Linien usw.

Danach werden aus diesen Klassen eine Reihe nützlicher Dialoge erstellt, die Sie später in all Ihren Programmen flexibel und unabhängig nutzen können.

Nachdem Sie sich so eine solide Basis für die Programmierung mit Objekten geschaffen haben stelle ich wichtige nicht-visuellen Klassen von dBWin vor, die nicht in Formularen sondern direkt im Programmcode verwendet werden.

Nach einem Kapitel über fortgeschrittene Techniken der OOP-Entwicklung programmieren wir gemeinsam einige weitere nützliche Klassen, mit denen Sie Ihre eigenen Programme bereichern können. Sie werden Ihnen auch noch viele weitere wichtige Details der *Objektorientierten Programmierung* zeigen.

ⓘ Bei so einem Buch ist es immer eine Gratwanderung, weder einen Teil der LeserInnen zu überfordern, noch andere zu langweilen. Im Zweifelsfall werde ich bei neuen Informationen immer einer genauen Beschreibung den Vorzug geben und auch die Hintergründe erläutern. Bei bereits in früheren Bänden ausführlich erklärten Themen werde ich dagegen dorthin verweisen.

So, und nun sitzen Sie bitte bequem, wir fangen an ...

3. Visuelle Basisklassen

Unter *visuellen Basisklassen* sind alle Klassen zusammengefasst, die Sie in Formularen verwenden und die für den Anwender später auch sichtbar sind. Es gibt mehrere Möglichkeiten, die vielen Basisklassen von dBWin in Ihren Programmen zu benutzen. Einmal natürlich im *Formular-Designer*, mit dem Sie Formulare, Eingabemasken, Dialoge, Meldungsfenster und was Ihnen sonst noch alles einfällt mit ein paar Mausklicks zusammenbasteln können.

Sie legen einfach über das *Regiezentrum* oder über das Menü *Datei - Neu ...* ein neues Formular an und lassen Ihrer Phantasie freien Lauf. Dabei werden Sie die Basisklasse *form* für das Formular, aber auch diverse andere Klassen für Eingabefelder, Buttons, Linien, Texte etc. benutzen. Dass Sie dabei mit Klassen und Objekten arbeiten wird Ihnen nicht bewusst, denn der Designer versteckt diese Details so gut es geht und erstellt selbständig den Quellcode.

Der Designer legt eine Formulardatei *.wfm* an, schreibt den nötigen Code für Ihr Formular dort hinein und Sie kommen damit in sehr kurzer Zeit sehr weit. Nur: wirklich verstanden haben Sie Klassen, Objekte und *Objektorientierte Programmierung* damit vermutlich noch nicht. Sobald die Anforderungen spezieller werden oder ein per Designer gebautes Formular nicht tut was es soll haben Sie ein Problem und kommen nicht weiter. Das ändert sich jetzt!

Alles was die diversen Helferlein von dBWin können, das können Sie auch indem Sie den Code dazu selbst schreiben. Und glauben Sie mir, Sie können das mit ein klein wenig Übung schon bald sehr viel besser als jeder Designer.

Wenn Sie Ihre Produkte von der Masse der 0815-Programme abheben wollen werden Sie um selbst geschriebenen Code sowieso nicht herumkommen. Und genau das sollte als Profi ja auch ihr Ziel sein. Ok?! Prima, dann schreiben Sie bitte mal die folgenden einfachen Zeilen in eine Programmdatei beliebigen Namens, die Sie danach mit dem Befehl *do <programmdatei>* starten.

Listing *form1.prg* (alle Beispiele auf meiner Homepage zum Download!)

```
1.   public oForm
2.   oForm = new form()
3.   oForm.text = "Hallo Welt"
4.   oForm.open()
5.   wait "Bitte eine Taste ..."
6.   oForm.close()
7.   oForm.release()
```

3.2.5 Eigene Klasse für Dialoge

Als Ergänzung zum Formular brauchen wir noch das Gegenstück, eine Klasse für Dialoge. Sie erinnern sich, das Formular ist ein MDI-Fenster, der Dialog nicht. Ein wichtiger Unterschied für Ihre Programmierung unter Windows.

Listing *myclass.prg* bzw. *myclass.cc* (Ausschnitt)

```
class MyDialogClass ( sTitel, fCenter ) of FORM
  this.text = sTitel
  this.top = 0
  this.left = 0
  this.mdi = .f.                                  && !
  this.escexit = .t     .                         && !
  this.maximize = .f.                             && !
  this.minimize = .f.                             && !
  this.sysmenu = .t.
  this.sizeable = .f.                             && !
  this.moveable = .t.
  this.autoSize = .f.
  this.showspeedtip = .t.
  this.autoCenter = fCenter
  this.metric = 0
  this.ScaleFontName = "Arial"
  this.ScaleFontSize = 10
endclass
```

Im Vergleich zur Formular-Klasse sind nur wenige Zeilen anders, die ich zur besseren Übersicht mit *&& !* markiert habe. Wichtig ist *mdi*, der Rest ergibt sich meist autom. daraus, bzw. hängt von der Einstellung für *mdi ab*. Ein Dialogfenster ist nicht in der Grösse änderbar und kann nicht zum Symbol minimiert werden. Das ist Standard unter Windows, also auch bei dBWin.

Ein Anwendungsbeispiel folgt später, erst brauchen wir noch weitere Klassen.

ⓘ Sie können die Quelldatei mit ihren eigenen Klassen wie jede „normale“ Quelldatei kompilieren. So können Sie prüfen, ob Syntaxfehler enthalten sind.

```
compile myclass.prg          && oder myclass.cc
```

Ebenso können Sie Quelldateien, die „nur“ Klassen enthalten, auch ausführen.

```
do myclass.prg               && oder myclass.cc
```

Aber Sie werden danach im Regelfall nichts weiter bemerken, denn es gibt ja darin meist keinen Code, der etwas für den Anwender sofort sichtbares tut. Die Ausführung von Klassendateien mit *do* ... ist daher selten wirklich nötig.

3.2.6 Eigene Klasse für Pushbuttons

Ein Pushbutton hat einige typische Eigenschaften. So z. B. einen fetten Text (*fontbold = .t.*), er ist in der Tab-Reihenfolge der Formularobjekte enthalten (*speedbar = .f.*) und hat eine gewisse Breite und Höhe (*width* und *height*).

Buttons sollten im Programm einheitlich sein, damit das optische Bild stimmt. Da Sie aber grosse und kleine Buttons benötigen habe ich gleich drei Klassen definiert. Von der Basisklasse *Pushbutton* wird erst *MyPushButton* abgeleitet, und daraus entstehen wiederum *MySmallPushButton* und *MyMiniPushButton*.

Listing *myclass.prg* bzw. *myclass.cc* (Ausschnitt)

```
1.   class MyPushButton ( oForm, sName ) of PUSHBUTTON
                                        ( oForm, sName )
2.     this.height = 1.5
3.     this.width = 14
4.     this.text = ""
5.     this.speedbar = .f.
6.     this.fontbold = .t.
7.   endclass
8.
9.   class MySmallPushButton ( oForm, sName ) of
                           MyPushButton ( oForm, sName )
10.    this.height = 1.50
11.    this.width = 7
12.  endclass
13.
14.  class MyMiniPushButton ( oForm, sName ) of
                           MyPushButton ( oForm, sName )
15.    this.height = 1
16.    this.width  = 4
17.  endclass
```

ⓘ Ich habe die Zeilen nummeriert, damit Sie längere Befehle, die nicht in eine Zeile im Buch passen, besser als solche erkennen. Diese Zeilennummern stimmen aber bei *myclass.prg* bzw. *myclass.cc* nicht mit der Download-Datei überein, denn Sie sehen im Buch immer nur einen Ausschnitt dieser Dateien.

Die 2. und 3. Klasse „erben" alle Eigenschaften von *MyPushButton*. Lediglich die zwei Eigenschaften *width* und *height* werden dort explizit anders definiert. Die von mir verwendeten Grössen sind natürlich nur Vorschläge und es steht Ihnen wie immer frei, diese nach Ihren individuellen Wünschen zu ändern.

Die beiden Parameter *oForm* und *sName* sind später die Objektreferenz auf das Formular, in dem der Button erscheinen soll, sowie sein interner Name. Über den Namen wird der Button später angesprochen, um bei Bedarf zur Laufzeit seine Details und seine Eigenschaften zu ändern (z. B. die Grafik).

4.1 Dialog-Klasse für Texteingaben

Einen simplen Dialog für die Eingabe eines Strings werden Sie immer wieder benötigen. Sei es für die Abfrage des Usernamens oder des Passworts beim Start des Programms, zur Abfrage eines Suchbegriffs oder wozu auch immer.

Für solch einen Dialog brauchen wir die folgenden Komponenten:

- **das Dialog-Fenster (kein MDI)**
- **einen Erklärungstext**
- **ein Eingabefeld für den Text**
- **Pushbuttons für Ok, Abbruch und (falls gewünscht) Hilfe**

Für diese Objekte haben wir im vorherigen Kapitel bereits eigene Klassen von den dBWin-Basisklassen abgeleitet. Damit ist die wichtigste Vorarbeit getan.

Ausserdem werde ich eine kleine Routine aufbauen, in welcher der komplette Dialog gesteuert wird. Die Erstellung des Dialogs im Speicher, die flexible Einstellung des Eingabefelds, der Aufruf des Dialogs und zuletzt auch noch die Auswertung der Eingabe. Alles wird in dieser einen Routine ausgeführt.

Mit Hilfe dieser Prozedur können Sie dann den Dialog zur Texteingabe in all Ihren Programmen mit nur einer einzigen Befehlszeile beliebig oft benutzen. Dafür ist natürlich gründliche Vorarbeit nötig. Aber die machen Sie einmal, während Sie die spätere Zeitersparnis daraus immer wieder aufs neue haben!

Um das Ganze von Anfang an modular und flexibel zu gestalten werde ich die Codes auf mehrere Dateien aufteilen. Eine Quelldatei nur für die Klassen der Dialoge und eine weitere für die Routinen, innerhalb derer die Klassen dann verwendet werden. Die Dateien sind völlig unabhängig von einem speziellen Programm, so dass Sie sie später in all Ihre Programme einbinden können.

ⓘ Dieser modulare Aufbau und die strikte Trennung von Klassen einerseits und Routinen welche die Klassen verwenden andererseits ist für den Anfang etwas mehr Arbeit. Für dBASE-Entwickler der „alten Schule“, die noch aus alten DOS-Zeiten daran gewöhnt sind den Code einfach in einer Datei ohne jede Struktur „runterzuklopfen“ ist das evtl. eine gewisse Umstellung. Und es fordert anfangs auch eine gewisse Disziplin, sich konsequent daran zu halten.

Aber es erleichtert die spätere Anwendung und Erweiterung der Klassen und der dazugehörenden Routinen enorm. Sie werden die Vorteile des modularen Aufbaus sehr schnell erkennen und schon bald nicht mehr missen wollen. Die langfristige Zeitersparnis ist ein Vielfaches des anfänglichen Mehraufwands.

4.3 Dialog-Klasse für Eingaben von ... bis ...

Als nächsten Dialog zeige ich Ihnen eine Erweiterung des Spinbox-Dialogs, um nicht nur einen sondern zwei Werte eingeben zu können. Das ist nützlich für alle Fälle, in denen Zahlen- oder Datumsbereiche *von ... bis ...* nötig sind.

Hier ist die Zutatenliste um solch einen Dialog zu backen:

- **das Dialog-Fenster (kein MDI)**
- **einen Erklärungstext**
- **zwei Beschriftungstexte**
- **zwei Spinbox-Eingabefelder**
- **Pushbuttons für Ok, Abbruch und (falls gewünscht) Hilfe**

Wie wollen Sie vorgehen? Wieder eine neue Dialog-Klasse anlegen mit allen nötigen Objekten, so wie bisher schon? Jein! Natürlich wäre es möglich, jetzt eine weitere Dialogklasse analog zu den bisherigen anzulegen, aber das wird auf die Dauer doch etwas langweilig. Wie wäre es also, wenn wir stattdessen den Dialog des letzten Kapitels „vererben" und den neuen Dialog als eine Art Ableitung daraus erstellen. Dann müssten Sie im neuen Dialog nur noch die fehlenden Objekte (die zweite Spinbox und den zweiten Text) ergänzen.

Hört sich kompliziert an? Ist es aber nicht. Und ausserdem lernen Sie so, wie einfach es ist eine eigene Klasse als Basis einer anderen Klasse zu verwenden. Es wird jetzt also etwas komplexer und Sie müssen etwas weiter und auch mal „um die Ecke" denken. So macht **O**bjekt-**O**rientierte **P**rogrammierung Spass! Übrigens, diese „Vererbung" von Klassen und Objekten ist völlig steuerfrei ...

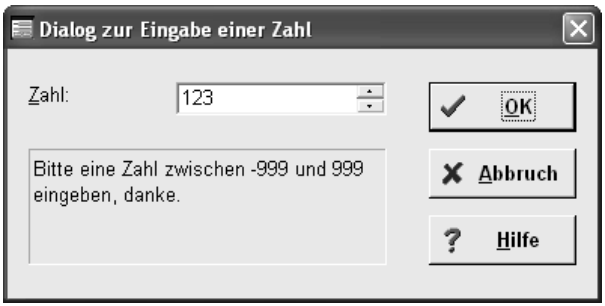
4.3.1 Eingabe von ... bis ..., Klassen-Definition

Diese Klasse sieht auf den ersten Blick schon etwas böser aus. Obwohl in ihr weniger Objekte definiert werden als in der vorherigen mit nur einer Spinbox. Auch sehen Sie die Ihnen bislang evtl. noch unbekannte Anweisung *super::*

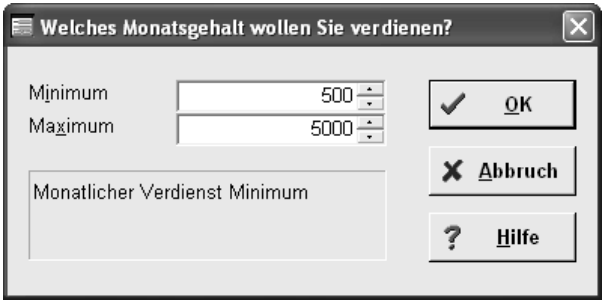
Listing *dlgclass.prg* (Ausschnitt)

```
1.   class My2SpinboxDialogClass (sTitel,fCenter,fEscExit)
         of MySpinboxDialogClass (sTitel,fCenter,fEscExit)
2.
3.     this.Rect1.height = this.Rect1.height - 0.75
4.     this.Rect1.top    = this.Rect1.top    + 0.75
5.     this.InfoText.height = this.InfoText.height - 0.75
6.     this.InfoText.top    = this.InfoText.top    + 0.75
7.
8.     define MyTextclass Text2 of this property;
9.       top 2, left 2, alignment 0, width 14, height 1
10.    define MySpinboxClass Spin2 of this property;
11.      top 2, left 16, width 20
```

Hier noch die beiden Dialoge mit einer und zwei Spinboxen zum Vergleich.



Dialog mit einer Spinbox, Inhalt linksbündig



Dialog mit zwei Spinboxen, Inhalt rechtsbündig

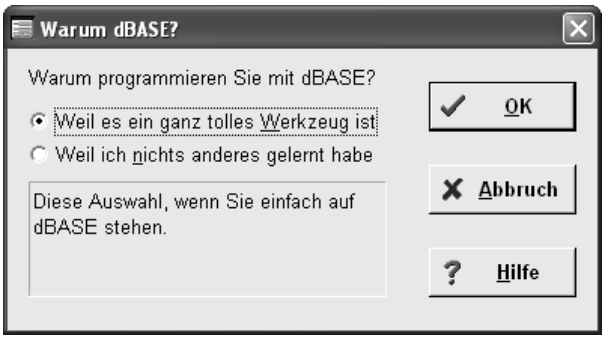
Der Unterschied von links- und rechtsbündiger Darstellung wird durch die Eigenschaft *function* der Spinbox erreicht. Im ersten Dialog wird hier beim Aufruf *„ “* als Parameter übergeben. Es wird keine spezielle Formatierung gewählt, und damit ist eben die Standard-Formatierung (linksbündig) aktiv.

Im zweiten Fall wird per Option *„J“* eine rechtsbündige Darstellung erzielt. Diese wird im vorletzten Parameter von *DoMy2SpinboxDialog* übergeben. Damit lassen sich Details beider Spinboxen flexibel steuern, indem je nach Bedarf die beiden Eigenschaften *picture* und/oder *function* geändert werden.

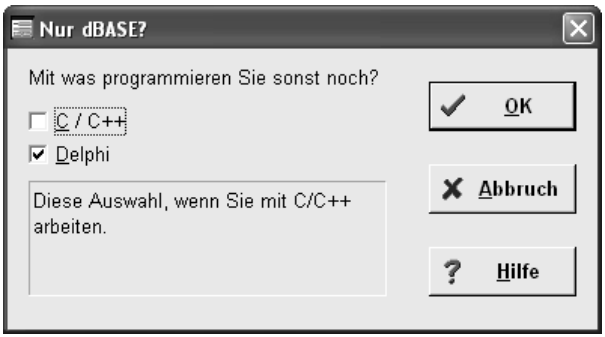
ⓘ Wenn Sie in der dBWin Onlinehilfe nach *picture* oder *function* suchen finden Sie alle zulässigen Einstellungen für diese nützlichen Eigenschaften.

Wenn Sie genau hinsehen bemerken Sie auch, dass das Rechteck mit dem darin eingebetteten Infotext im zweiten Dialog etwas kleiner ist und um wenige Pixel nach unten geschoben wurde. Die Buttons sind dagegen in beiden gleich, da sie vom ersten Dialog an den zweiten „vererbt“ werden.

Hier noch die Auswahl-Dialoge mit je zwei Radiobuttons und Checkboxen.



Auswahl-Dialog mit zwei Radiobuttons



Auswahl-Dialog mit zwei Checkboxen

Sie werden Abfragen und Auswahlen nach diesem Schema sicher häufiger in Ihren Programmen benötigen. Mit Hilfe der Klassen und Objekte können Sie sich das schnell und einfach als fertige Dialoge programmieren und dabei Ihre individuellen Wünsche, Vorstellungen und Ansprüche völlig frei umsetzen.

Sie haben jetzt das Wissen und genügend Übung mit Klassen und Objekten, um bei Bedarf weitere Dialoge aller Art und für jeden Zweck zu entwickeln.

ⓘ Vielleicht fiel Ihnen auf, dass ich die in mehreren Dialogen verwendeten Komponenten immer einheitlich benenne. So z. B. *InfoText* für das Textobjekt, das dem Anwender einen zusätzlichen Informationstext in den Dialogen zeigt. Das hat den Vorteil, dass gleichartige Objekte in den verschiedenen Dialogen immer gleich heissen und erleichtert die Pflege dieser Klassen ganz erheblich. Nach einer gewissen Zeit und nach mehreren programmierten Klassen wissen Sie beim Anblick eines Elements im Quellcode sofort für was es gedacht ist.

4.6 Dialog-Klasse zur Datenbank-Anzeige

In diesem Kapitel zeige ich einen Dialog, der eine beliebige dBASE-Tabelle **.dbf* öffnet und die Datensätze zeigt. Auch eine flexible Suche über mehrere Indexfelder wird möglich sein, ebenso das (optionale) bearbeiten der Daten.

Für diesen Dialog werde ich verwenden:

- **das Dialog-Fenster (kein MDI)**
- **ein Browse-Objekt zur Anzeige der Daten**
- **ein Textobjekt**
- **ein Eingabefeld**
- **eine Checkbox**
- **drei Radiobuttons**
- **Pushbuttons für Ok, Abbruch und Hilfe**

Die späteren Vorbereitungen, um diesen Dialog im Programm zu benutzen, werden etwas umfangreicher ausfallen als bei den bisher gezeigten Dialogen. Daher zeige ich sowohl die gewohnte Anwendung über Prozeduren wie bei den vorherigen Dialogen, als auch die Steuerung mit einer eigenen Klasse.

Ich verwende die bereits in Band 1 *Einführung* erstellte Tabelle *adressen.dbf*, um den praktischen Einsatz des Dialogs zu zeigen. Sie können das Beispiel aber leicht so ändern, dass auch beliebige andere Dateien angezeigt werden.

Zuerst aber wie immer die Klasse des Dialogs mit den Erklärungen dazu.

4.6.1 Datenbank-Anzeige, Klassen-Definition

Listing *dlgclass.prg* (Ausschnitt)

```
1.   class MyDbfListDialogClass ( sTitel, fCenter,
          fEscExit ) of MyDialogClass ( sTitel, fCenter )
2.     form.width = 100
3.     form.height = 15
4.     form.sIdx1 = ""
5.     form.sIdx2 = ""
6.     form.sIdx3 = ""
7.     form.sHelp = ""
8.     form.iRecNo = 0
9.
10.    form.onOpen = {;class::MyOnOpenFunc()}
11.
12.    define BROWSE DataList of this property;
13.      append .f., delete .f., modify .f., fields "",;
14.      alias "", left 1.5, top 0.5, cuatab .f.,;
15.      width form.width-3, height form.height -5,;
16.      StatusMessage "Liste der gefundenen Daten",;
17.      OnLeftDblClick class::DoDoubleClick
```

4.6.2 Datenbank-Anzeige, Aufruf-Routine

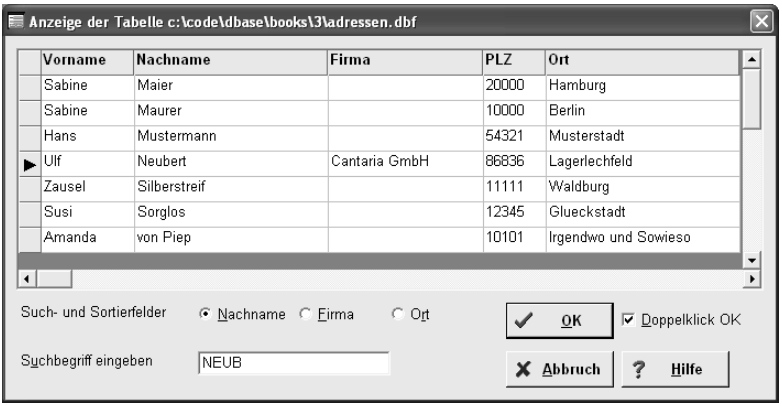
Die vielen Parameter machen *DoMyDbfListDialog* etwas unhandlich, aber im Einsatz ist die Routine so simpel wie alle bisher gezeigten Dialog-Routinen.

Listing *dlgfuncs.prg* (Ausschnitt)

```
1.   Procedure DoMyDbfListDialog ( sTitel, sRT1, sRT2,
                    sRT3, sI1, sI2, sI3, sFields, sAlias,
                    fEdit, fDblClk, sHelp, fEscExit )
2.   local oDlg, fReturn, iSelection
3.
4.     oDlg =NEW MyDbfListDialogClass(sTitel,.t.,fEscExit)
5.     oDlg.InitDialog (sRT1, sRT2, sRT3, fDblClk, sHelp)
6.     oDlg.InitData   (sI1,sI2,sI3,sFields,sAlias,fEdit)
7.
8.     fReturn = oDlg.ReadModal()
9.
10.    iSelection = -1
11.    if fReturn
12.      iSelection = oDlg.GetRecNo()
13.    endif
14.
15.    oDlg.Release()
16.
17.  return iSelection
```

Die Parameter sind: der Titel des Dialogs, die drei Radiobutton-Texte und die drei Indexnamen („“ falls weniger als drei), die Liste der zu zeigenden Felder, der Alias der Datenbank, Flags für „Daten ändern erlaubt“ und die Checkbox-Vorgabe, ein Hilfstext und die Angabe ob der Dialog mit **[Esc]** abbrechbar ist.

Die Routine gibt bei **[OK]** die Nummer des letzten Datensatzes zurück, oder falls der Dialog vom Anwender abgebrochen wurde wird *-1* zurückgegeben.



Der Dialog zur flexiblen Datenbank-Anzeige in Aktion

5. Nicht-visuelle Basisklassen

Bis auf die Basisklasse *Object* waren bislang alle benutzten Klassen *visuelle* Komponenten für Formulare. Es gibt bei dBWin aber auch noch diverse nicht sichtbare Klassen, die direkt im Programmcode Verwendung finden. Das sind die *nicht-visuellen* Klassen, und jetzt erfahren Sie was es damit auf sich hat.

Vorab ein Tip, der nicht nur auf die in diesem Kapitel besprochenen Klassen, sondern auf alle Klassen anwendbar ist. Aber gerade bei den hier erwähnten Klassen, die evtl. noch völlig neu für Sie sind, ist es sehr hilfreich, sich mit

```
inspect ( <objektvariable> )
```

ihre Bestandteile im Detail anzusehen. Sie können dann im Objekt-Inspektor alle Eigenschaften, Methoden und Ereignisse der Klasse sehen. Alles was Sie dazu tun müssen, ist ein Objekt aus der zu untersuchenden Klasse zu erstellen und dieses Objekt an den Befehl *inspect* zu übergeben. Das geht auch schnell und bequem im *Befehlsfenster*, hierzu ein Beispiel mit der Basisklasse *Timer*:

```
oObj = new timer()
inspect ( oObj )
```

ⓘ Eine Erklärung des *Objekt-Inspektors* finden Sie in Band 2 ab Seite 33.

Bitte beachten Sie auch die Onlinehilfe und lesen Sie die Informationen über die Klassen und deren Bestandteile darin nach. Suchen Sie dort einfach nach „Klasse ...“ oder „Class ...“, je nach dem welche Sprach-Version Sie nutzen.

Es kommt auch mal vor, dass Klassen bei Updates erweitert werden und neue Eigenschaften oder Methoden hinzukommen. Davon erfahren Sie aber nichts, wenn Sie sich nicht die Mühe machen die Online-Dokumentation zu lesen. Natürlich nicht am Stück, sondern immer dann, wenn Sie sich neu mit einer Klasse, einem Befehl oder einer sonstigen Funktion von dBWin beschäftigen.

Falls Sie noch mit einer älteren dBWin-Version arbeiten können Details einer Klasse fehlen. Und wenn Sie Pech haben gibt es die ganze Klasse noch nicht. Aber das werden Sie schnell merken, denn dann gibt´s eine Fehlermeldung.

```
oObj = new gibtsnicht()
```

Die Klasse *gibtsnicht* gibt es nicht, und dBWin meckert auch prompt los ...

5.1 Die Basisklasse *Timer*

Der *Timer* ist hilfreich, wenn Sie regelmässige Vorgänge in Ihrem Programm automatisieren wollen. Beispielsweise eine Zeitanzeige oder andere Aktionen, die mehrfach und meist immer in denselben Zeitabständen ausgeführt werden.

Listing *xtimer1.prg*

```
1.   oTimer = new timer()
2.   oTimer.interval = 5
3.   oTimer.onTimer = {; ? "5 Sekunden umsonst gewartet"}
4.   oTimer.enabled = .t.
5.   wait ( "Warte auf irgendeine Taste ..." )
6.   ? "na endlich!"
```

Die Anwendung ist denkbar einfach: mit der Eigenschaft *interval* (mit einem *l*, hier ist das engl. *interval* und nicht das dt. *Intervall* gemeint) wird der Intervall in Sekunden eingestellt, das Ereignis *onTimer* gibt an was passieren soll, wenn der Intervall abgelaufen ist. *enabled* aktiviert bzw. deaktiviert den Timer.

Aber: absolut verlässlich ist es leider nicht. Zeitkritische Operationen wie das kopieren von Daten, die sequentielle Suche in grossen Tabellen oder andere Aktionen, auch von anderen Programmen oder sonstigen im Hintergrund ablaufenden Prozessen, funken hier schon mal „gern“ dazwischen. Sogar der dBWin-Befehl *sleep* lässt den Timer im wahrsten Sinne des Wortes schlafen.

Listing *xtimer2.prg*

```
1.   oTimer = new timer()
2.   oTimer.interval = 1
3.   oTimer.onTimer = {; ? time() }
4.   oTimer.enabled = .t.
5.   wait ( "Warte auf irgendeine Taste ..." )
6.   ? "schlafe 10 Sekunden ohne Timer ..."
7.   sleep 10
8.   wait ( "Warte auf irgendeine Taste ..." )
9.   ? "fertig!"
```

Solange Sie nicht haargenaue und extrem zeitkritische Aktionen ausführen wollen reicht der *Timer* im Regelfall aber völlig aus. Für eine regelmässige Aktualisierung von Statusanzeigen o. ä. ist es meist kein Problem, wenn die zeitgesteuerte Aktion für einen kurzen Moment mal nicht ausgeführt wird.

Dieser *Timer* ist nicht dazu geeignet, absolut zuverlässige und auf die Millisekunde exakte Ereignisse zu starten. Er wird nur ausgeführt, wenn Ihr dBWin-Programm gerade nichts „besseres“ zu tun hat. Selbst Windows oder ein x-beliebiges Fremdprogramm kann die Timer-Funktion vorübergehend stören, wenn anderweitige zeitkritische Aktionen ausgeführt werden müssen.

6. Fortgeschrittene Techniken

Nachdem Sie nun bereits eine sehr fundierte Praxis mit Klassen und Objekten haben möchte ich in diesem Kapitel einige bereits erwähnte Aspekte vertiefen, sowie neue Anwendungsvarianten und fortgeschrittene Techniken vorstellen.

Das geht nicht immer ganz ohne Theorie ab, aber ich versuche es dennoch so lebendig wie möglich zu gestalten. Die Beispiele die ich dafür verwende sind entweder später von Ihnen universell nutzbar oder möglichst knapp gehalten, damit das zu vermittelnde Wissen präzise und schnell bei Ihnen ankommt.

Auch hier gilt fast immer: man *kann* es so machen, man muss es aber nicht. Wenn Ihnen ein anderer Weg einfällt um zur selben Lösung zu kommen, gehen Sie Ihren Weg. Nichts ist daran falsch, wenn das Ergebnis richtig ist.

Einige der besprochenen Punkte kann man sowohl mit Klassen und Objekten, als auch konventionell (dBWin-Befehle mit „normalen“ Prozeduren) lösen. Und auch hier ist es wieder allein Ihre Entscheidung, wie Sie es machen.

6.1 Vererbung von Eigenschaften

Sie hatten bereits häufig mit Eigenschaften von Klassen und Objekten zu tun. Und „vererbt“ haben Sie im Laufe dieses Buchs schon so oft, dass Ihnen ein Notar liebend gern eine Gebührenrechnung dafür ausstellen würde. Dennoch, eine weitere Vertiefung dieser wichtigen Angelegenheit schadet nicht.

Wenn Sie aus einer bestehenden Klasse eine neue Klasse ableiten, so erbt die neue Klasse sämtliche Eigenschaften der alten. Und auch die Inhalte dieser geerbten Eigenschaften sind gleich, zumindest solange Sie das nicht ändern. Ein einfaches Beispiel: ich hatte im Kapitel zur Entwicklung eigener Klassen aus der Basisklasse *Text* eine eigene Klasse namens *MyTextClass* abgeleitet. Diese bekam autom. alle Eigenschaften und alle Inhalte der Basisklasse *Text*, so wie es die Vorgaben (die sog. *Defaults*) von dBWin vorgeben. Lediglich einige speziell von mir ausdrücklich geänderte Eigenschaften (z. B. *height*) bekamen einen Wert, der ggf. von der dBWin-Vorgabe abweichen kann.

Wenn ich nun aus meiner Klasse *MyTextClass* nochmals eine weitere Klasse ableite, bekommt die neue Klasse autom. die Eigenschaften von *MyTextClass*. Es spielt keine Rolle, ob das die Vorgaben von dBWin waren oder ob es von mir explizit geänderte (oder auch neu hinzugefügte) Eigenschaften sind.

7. Eigene Klassen für Dies & Das

In diesem Kapitel habe ich noch einige spezielle Klassen parat, die Sie gern bei Bedarf für Ihre Formulare und Programme benutzen können. Und die Ihnen weitere interessante Aspekte der Klassen-Programmierung zeigen.

Für alle Klassen gibt es natürlich Beispielprogramme oder -Formulare, damit Sie auch gleich den Einsatz in der Praxis sehen. Der Schritt, diese Klassen in Ihren eigenen Programmen zu verwenden, ist damit nur noch ein Kinderspiel.

7.1 Eine Ampel als Klasse

Wie wäre es mit einer Ampel, mit den bekannten Farben rot, gelb und grün. Damit lassen sich viele Informationen darstellen (z. B. die Zahlungsmoral eines Kunden, der Lagerbestand eines Artikels, die Höhe des Bankkontos ...), oder Einstellungen vom Anwender vornehmen (z. B. bei Abfragen mit drei verschiedenen Antwortmöglichkeiten (ja=grün, nein=rot, vielleicht=gelb).

Das schöne an so einer Ampel ist, dass sie jeder Anwender kennt und sofort versteht. Sie ist völlig unabhängig von Zielgruppe, Branche oder Sprache.

Ich habe die Ampel als sog. *Custom Class* realisiert und in einer eigenen Datei *ampel.cc* gespeichert. Damit ist sie flexibel in Ihren Programmen verwendbar.

7.1.1 Ampel, Klassen-Definition

Listing *ampel.cc*

```
1.   class MyAmpelClass(oForm) of CONTAINER(oForm) custom
2.
3.     && --- Eigenschaften ---
4.     this.left = 1
5.     this.top  = 1
6.     this.width = 9.9
7.     this.height = 5.9
8.
9.
10.    && --- Objekte ---
11.    this.Rect1 = new RECTANGLE ( this )
12.    with (this.Rect1)
13.       left = 0.5
14.       top = 0.25
15.       width = 8.25
16.       height = 5.25
17.       text = ""
18.       pageno = 0
19.    endwith
```

Hier noch die Ampel, wie sie im beispielhaften Formular verwendet wird:

Die Ampel in ein Formular eingebunden

Und als einfaches Programm ohne Formular-Designer sieht es z. B. so aus:

Die Ampel direkt per Programm

Die hier gezeigten Formulare sind natürlich nur Beispiele. Ihnen fallen sicher viele andere Möglichkeiten ein, wie Sie so eine Ampel verwenden können.

7.2 Ein LED-Fortschrittsbalken als Klasse

So, gleich noch etwas für´s Auge. Sie kennen den üblichen Fortschrittsbalken, dBWin hat dafür sogar eine eigene Basisklasse namens *Progress* im Angebot.

Ein Windows-typischer Fortschrittsbalken

Wirklich schön ist er ja nicht (ok, das ist Geschmackssache), und da fast jedes Programm so einen Balken verwendet hat man sich längst daran satt gesehen.

Wie wäre es mit einer Anzeige in moderner „LED-Optik“, z. B. so:

LED-Fortschrittsbalken (in Farbe noch viel schöner)

Dazu brauchen Sie auch keinen Kurs über Computer-Grafik oder Web-Design belegen, das lässt sich, wie die Ampel, alles mit dBWin-Bordmitteln erreichen.

Die „Zutatenliste“ für solch eine Klasse besteht aus:

- **einem Container**
- **einem Rechteck**
- **50 Pushbuttons mit je drei verschiedenen Grafiken**
- **diverse Methoden zur Steuerung der Anzeige**

Der Container ist wichtig, damit die einzelnen Komponenten wie ein Objekt behandelt werden können, was die Verwendung im Designer sehr erleichtert. Das Rechteck hat nur optischen Zweck und soll einen Rahmen erzeugen.

Die 50 Pushbuttons sind das eigentliche Highlight, denn sie sorgen für das Spiel mit den Farben und die Anzeige eines Prozentwerts von 0% bis 100%.

7.3 Zeiteingabe als Klasse

Die Eingabe der Uhrzeit ist ein oft benötigter Punkt in vielen Programmen. Leider bietet dBWin dafür keine passende Eingabeklasse und die normalen Objekte wie *Entryfield* und *Spinbox* sind ohne Feintuning dafür nur bedingt brauchbar. Also basteln wir uns eben selbst eine eigene Zeiteingabe-Klasse.

Erneut wird die Klasse als *Container* realisiert, da die neue Klasse wieder aus mehreren einzelnen Grafik-Komponenten besteht. Ich brauche ein Rechteck, einen Text und zwei Spinboxen, wobei ich diesmal meine eigenen Klassen aus *myclass.cc* verwende. Erstens macht das in diesem Fall mehr Sinn, weil ich die Klasse für der Zeiteingabe diesmal sowieso in der Datei *myclass.cc* speichere. Und zweitens haben Sie dann auch ein Beispiel für diese Art der Entwicklung. Ausserdem verwende ich diesmal *define* statt *new* zur Erzeugung der Objekte.

7.3.1 Zeiteingabe, Klassen-Definition

Listing *myclass.cc* (Ausschnitt)

```
1.   class MyTimeEditClass(oForm) of CONTAINER(oForm)
                                                    custom
2.
3.     && --- Eigenschaften ---
4.     this.left = 1
5.     this.top  = 1
6.     this.width = 15
7.     this.height = 1.5
8.     this.borderstyle = 3          && 3 = kein Rand
9.     this.f24Modus = .t.           && 24-Stunden Modus
10.
11.    && --- Objekte ---
12.    define MyRectClassNorm rect1 of this property;
13.    left 0.7, top 0.15, width 13.6, height 1.2, text ""
14.
15.    define MySpinboxClass SpinStunde of this property;
16.    left 1, top 0.25, width 6, height 1, value 0,;
17.    rangemin 0, rangemax iif (this.f24Modus, 23, 11),;
18.    rangerequired .t., picture "99", function "IL",;
19.    step 1, border .f., pageno 0,;
20.    OnLeftDblClick {;class::OnDblClickStunde()}
21.
22.    define MyTextClass TextDP of this property;
23.    left 7, top 0.2, width 1.0, height 1, text ":",;
24.    fontbold .t., alignment 4
25.
26.    define MySpinboxClass SpinMinute of this property;
27.    left 8, top 0.25, width 6, height 1, value 0,;
28.    rangemin -1, rangemax 60,;
29.    rangerequired .t., picture "99", function "IL",;
30.    step 1, border .f., pageno 0,;
31.    OnLeftDblClick {;class::OnDblClickMinute()},;
32.    onChange {;class::OnChangeMinute(this.value)}
```

7.4 Datenstrukturen als Klasse

Klassen sind natürlich auch für die Abbildung von Daten jeglicher Art prima geeignet. Die Klasse ist der Datensatz, die Eigenschaften sind die Datenfelder. Dazu ein kleines Beispiel, das diesen Gedanken praktisch umsetzt und Ihnen zusätzlich noch zeigt, wie sich Klassen auch verschachteln lassen. Ausserdem spielen wir etwas mit dynamischen Arrays, das kann ja auch nicht schaden ...

Nehmen wir doch mal eine Musik-CD als Vorlage. Diese hat eine Reihe von individuellen Eigenschaften (Interpret, Genre, Erscheinungsjahr, Label etc.). Und die einzelnen Songs lassen sich gut als kleine Unterklasse definieren, mit ebenfalls speziellen Eigenschaften (Titel, Spielzeit, Position auf der CD etc.).

Listing *musik1.prg*

```
1.   public oCD1, oCD2, oCD3
2.
3.   oCD1 = CDInit ( "Pink Floyd", "Atom Heart Mother",
                     "Alternative Rock", 1970 )
4.   oCD2 = CDInit ( "Therion", "Vovin",
                     "Symphonic Metal", 1998 )
5.   oCD3 = CDInit ( "Vivaldi", "4 Jahreszeiten",
                     "Klassik", 1725 )
6.
7.   ? "Meine drei liebsten CDs"
8.   CDShow ( oCD1 )
9.   CDShow ( oCD2 )
10.  CDShow ( oCD3 )
11.
12.  Procedure CDInit ( sBand, sName, sArt, iVon )
13.  local oObj
14.    oObj = new MyAlbum()
15.    with oObj
16.      sInterpret = sBand
17.      sTitel = sName
18.      sGenre = sArt
19.      iJahr = iVon
20.    endwith
21.  return oObj
22.
23.  Procedure CDShow ( oObj )
24.    ? oObj.sInterpret + ", " + oObj.sTitel + " - " +
         oObj.sGenre + " von " + ltrim(str(oObj.iJahr))
25.  return
26.
27.  class MyAlbum of OBJECT
28.    this.sTitel = ""
29.    this.sGenre = ""
30.    this.sInterpret = ""
31.    this.iJahr = 0
32.    this.iSpielzeit = 0
33.  endclass
```

7.5 Allgemeine Routinen als Klasse

Im Laufe der Zeit wird sich bei Ihnen sicher eine kaum noch überschaubare Zahl von Routinen aller Art ansammeln. Diese häufig von einem bestimmten Programm unabhängigen Prozeduren und Funktionen können Sie in eigenen Dateien sammeln, um bei Bedarf die ganze Datei mit *set procedure to* in ein neues Projekt einbinden (Infos dazu siehe Band 2 Seite 165). Natürlich bietet sich auch eine Klasse als „Sammelbecken" für solche flexiblen Routinen an.

7.5.1 Routinen für Datums-Berechnungen

Die Methoden sind Berechnungen, die weder in der Basisklasse *Date* noch in anderen Grundfunktionen von dBWin enthalten sind. Sie haben sicher genug eigene Ideen um diese Klasse bei Bedarf selbst zu erweitern. Ein Beispiel für die praktische Anwendung in einem beliebigen Programm folgt im Anschluss.

Natürlich, statt einer Klasse könnten Sie alle Routinen auch als „normale" Prozeduren oder Funktionen schreiben, so wie Sie das bisher getan haben. Aber es geht eben auch als Klasse, und genau darum geht es uns hier ja ...

ⓘ Ich verwende Variablennamen *iD*, *iM* und *iY* für Tag, Monat und Jahr (*iY* weil ich dabei immer von den engl. Worten *day*, *month* und *year* ausgehe). Zwar sind diese Namen nicht sehr aussagekräftig und damit auch nicht ideal, aber Sie wissen hier um was es geht und bei einheitlicher Verwendung kann man solche Namen schon mal durchgehen lassen. Ausserdem passen so alle Codezeilen in eine Druckzeile im Buch, das erleichtert Ihnen die Übersicht.

Listing *datetool.prg*

```
1.   *-------------------------------------------------
2.   * Eine Klasse allgemeiner Zeit- und Datums-Routinen
3.   *-------------------------------------------------
4.   Class MyDateTools of OBJECT
5.
6.     Procedure MakeADate ( iD, iM, iY )
7.     local sHlp
8.       sHlp  = ltrim (str ( iD, 2, 0, "0" ) ) + "."
9.       sHlp += ltrim (str ( iM, 2, 0, "0" ) ) + "."
10.      sHlp += ltrim (str ( iY, 4, 0, "0" ) )
11.    return ctod ( sHlp )
12.
13.
14.    Procedure AddYear ( dDate, iAdd )
15.    local iD, iM, iY
16.      iD = day   ( dDate )
17.      iM = month ( dDate )
18.      iY = year  ( dDate ) + iAdd
19.    return class::MakeADate ( iD, iM, iY )
20.
```