KAPITEL 9

Dialoge - Entwurfsmuster

Die Verwendung von Dialogboxen in Anwenderprogrammen ist inzwischen zu einer Selbstverständlichkeit geworden und es gibt eine ganze Reihe von kommerziell erfolgreichen Programmpaketen, mit denen sich in mehr oder minder einfacher Weise solche graphischen Benutzeroberflächen erstellen lassen. Im vierten Kapitel haben Sie bereits selbst einen Benutzerdialog für das Modul *TutKonvert1* konstruiert und Sie haben gesehen, daß die Fähigkeiten des BlackBox Systems Ihnen die Arbeit erheblich erleichtern. In diesem Kapitel lernen Sie einige weitere Möglichkeiten kennen, Dialoge besonderen Bedingungen anzupassen und benutzerfreundlich zu gestalten.

9.1. Interaktoren

Wie Sie wahrscheinlich erinnern, liegt eins der Probleme bei den Dialogboxen des Moduls *TutKonvert1* darin, daß nach einer Änderung der Daten in den Dialogen keine Aktualisierung der Anzeige erfolgt. Das Programm Konvert2.odc, das erste Programm dieses Kapitels, soll Ihnen zeigen, wie Sie diese Aktualisierung erreichen können. Die Hilfsmittel stellt Ihnen BlackBox in Form des Moduls *Dialog* zur Verfügung. *Dialog* ist eine Zusammenfassung vieler bei der Erstellung von Benutzerdialogen nützlicher BlackBox Systemdienste, von denen hier nur ein kleiner Teil benötigt wird

```
DEFINITION Dialog;

  CONST
    changed = 3;

  TYPE
    Par = RECORD
      disabled: BOOLEAN
    END;

    GuardProc = PROCEDURE (VAR par: Par);
    NotifierProc = PROCEDURE (op, from, to: INTEGER);

  PROCEDURE Update (IN interactor: ANYREC)

END Dialog.
```

Auszug der Schnittstelle des Moduls *Dialog*

Falls Sie sich die vollständige Schnittstellenbeschreibung des Moduls *Dialog* ansehen, sollten Sie nicht über die Menge der wahrscheinlich erst einmal unverständlichen Bezeichner erschrecken, bei Bedarf können Sie deren Bedeutung der Dokumentation des Moduls entnehmen.

Wenn Sie das Modul *TutKonvert2* mit dem Modul *TutKonvert1* des vierten Kapitels vergleichen, wird Ihnen als erstes wahrscheinlich die Ähnlichkeit beider Module auffallen. Ein Unterschied besteht in dem zusätzlichen Import des Moduls *Dialog*. Darüber hinaus hat sich scheinbar nur die Datenstruktur geändert, die im Modul *TutKonvert1* für die Erstellung des Dialogs einzeln exportierten Variablen sind in *TutKonvert2* zu der Verbundvariablen *dbox* zusammengefasst worden, die mit allen ihren Feldern exportiert wird.

Bei genauerem Hinsehen wird Ihnen auch die veränderte Struktur der Prozedur *Umrechnen* auffallen, diese enthält keinerlei explizite Ausgabeanweisungen mehr, in *TutKonvert2* ist das Modul *Out* anders als im Modul *TutKonvert1* nur noch für die Anzeige des Inhalts der Prozedur *Anleitung* im Log-Fenster, nicht mehr für die Anzeige der Ergebnisse von *Umrechnen* zuständig. Weiter fehlt der bereits für die zweite Version der Dialogbox des Moduls *TutKonvert1* überflüssige *ELSE*-Teil der CASE-Abfrage.

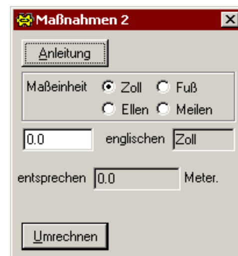

Abbildung 9
Dialogbox zu *TutKonvert 2*

Abbildung 9 zeigt eine mögliche Dialogbox für das Modul *TutKonvert2*. Der Abbildung können Sie entnehmen, daß der Dialog tatsächlich nicht nur die Eingabe, sondern auch die Ausgabe der Daten durchführt. Für den Transport der Daten in die entsprechenden Felder der Dialogbox und deren Anzeige ist das Modul *Dialog* zuständig, das für die Aktualisierung der Dialogbox nach einer Änderung der Daten sorgt.

In der Realisierung dieses Vorgangs geht das BlackBox System weit über die Möglichkeiten der meisten anderen, auch der auf die Erstellung von Benutzereingabemasken spezialisierten Programmiersysteme hinaus. In vielen Fällen müssen Sie nur eine einzige Anweisung ähnlich der am Ende der Prozedur *Umrechnen* zu findenden Zeile *Dialog.Update(dbox)* in Ihr Programm einfügen. Die Prozedur *Dialog.Update* veranlaßt die Ausführung aller Schritte, die bei der Aktualisierung des zu der Verbundvariablen *dbox* gehörenden Dialogs nötig sind, Sie müssen sich um die Einzelheiten nicht kümmern.

BlackBox entnimmt die notwendigen Informationen mit Hilfe des Moduls *Dialog* der Verbundvariablen *dbox*, die man deshalb als Vermittler zwischen dem Anwendermodul und dem System ansehen kann, derartige Verbundvariablen werden Interaktoren genannt. In der oben stehenden Schnittstellenbeschreibung des Moduls *Dialog* finden Sie die Signatur *PROCEDURE Update (IN interactor: ANYREC)*. Im Sinne der Vererbung, die Sie im vorigen Kapitel im Zusammenhang mit Verbundtypen kennengelernt haben, kann *ANYREC*, der Typ des Parameters *interactor*, als der - nicht explizit angeführte - Basistyp aller *RECORD*-Typen angesehen werden. Auf Grund dieses Typs stellt Ihnen BlackBox bei der Gestaltung von Dialogen keine Hindernisse in den Weg, ein formaler VAR- oder IN- Parameter des Typs *ANYREC* ist mit allen wie auch immer gearteten aktuellen Parametern eines Verbundtyps kompatibel, also auch mit *TutKonvert2.dbox*.

Eine detaillierte Darstellung der verschiedenen hinter dem simplen Aufruf *Dialog.Update(dbox)* stehenden Vorgänge geht verständlicherweise über eine Einführung in die Grundlagen des Programmierens hinaus, aber ich werde Ihnen einige der prinzipiellen, Entwurfsmuster (design patterns) genannten Strukturen in den Programmen dieses Kapitels vorstellen.

Das Modul *TutKonvert2* enthält sonst keine Überraschungen, jedoch gibt es in der Dialogbox der Abbildung 9 einige Ihnen bisher unbekannte Details, zu denen ich eine Erklärung geben möchte (Sie können mit dem Kommando *StdCmds.OpenAuxDialog('Tut/Rsrc/Konvert2', 'Maßnahmen2')*, das Sie am Ende des Programms Konvert2.odc finden, die der Abbildung 9 entsprechende Dialogbox öffnen, die zugehörige Datei *Konvert2.odc* steht im Verzeichnis *Tut/Rsrc*; unabhängig davon sollten Sie jedoch versuchen, die Dialogbox selbst zu erstellen).

Sie sehen als eine Neuigkeit um die vier Auswahlschaltflächen für die Maßeinheit einen Rahmen, dieser läßt sich mit *Controls* ➞ *Insert Group Box* in einen Dialog einfügen, wobei die Rahmengröße automatisch einer zuvor markierten Gruppe von Kontrollelementen angepaßt wird. Falls Sie für den Rahmen in das *Label*-Feld des Inspektors einen Bezeichner eingeben, erscheint dieser innerhalb der oberen Rahmenlinie auf der linken Seite, anderenfalls wird die Rahmenlinie durchgehend gezeichnet.

Als zweites möchte ich Sie auf die Möglichkeit aufmerksam machen, Dialoge nicht nur mit der Maus zu bedienen, sondern wahlweise auch mit der Tastatur. Hierfür finden Sie in BlackBox verschiedene Hilfsmittel, mit denen sich die bei Dialogboxen üblichen Tastatureingabebefehle realisieren lassen. Eines dieser Hilfsmittel besteht in einem von BlackBox automatisch um das jeweils aktive Element des Dialogs gezeichneten gepunkteten Rahmen. Dies ist insbesondere bei Schaltflächen wie *Anleitung* oder *Umrechnen* nützlich, da eine aktive Schaltfläche statt mit einem Mausklick auch mit der Eingabetaste betätigt werden kann.

Weiter sehen Sie bei den beiden Schaltflächen *Anleitung* und *Umrechnen* jeweils einen Buchstaben unterstrichen. Damit wird signalisiert, daß sich die Schaltfläche durch Drücken der entsprechenden Taste aktivieren läßt. Den Unterstrich und die entsprechende Funktionalität können Sie in BlackBox dadurch er-

zeugen, daß Sie bei dem jeweiligen Kontrollelement im *Label*-Feld des Inspektors ein kaufmännisches &-Zeichen vor den zu unterstreichenden Buchstaben setzen, wobei nicht nur Anfangsbuchstaben, sondern auch jedes beliebige andere Zeichen innerhalb des Bezeichners auf diese Weise aktivierbar gemacht werden können.

9.2. Notifiers

Die Möglichkeit, die Dialogbox des Moduls *TutKonvert2* mit Hilfe der Anweisung *Dialog.Update(dbox)* aktualisieren zu lassen, ist gegenüber der ersten Version im Modul *TutKonvert1* sicher eine erhebliche Verbesserung, möglicherweise wird Ihnen aber aufgefallen sein, daß die Anzeige noch nicht optimal reagiert. Die Wahl einer neuen Maßeinheit für die umzurechnende Länge führt keineswegs zu einer Änderung des Eintrags in dem entsprechenden Anzeigefeld des Dialogs, vielmehr wird dies Feld erst aktualisiert, nachdem der Benutzer die Schaltfläche *Umrechnen* aktiviert hat. Ein Blick auf den Quelltext des Moduls macht dies Verhalten verständlich; da die Anweisung *Dialog.Update* lediglich in der Prozedur *Umrechnen* aufgerufen wird, kann die Aktualisierung des Dialogs nur bei der Ausführung von *Umrechnen* erfolgen.

Mit dem Programm Konvert3.odc lernen Sie eine Möglichkeit kennen, die Anzeige unmittelbar nach der Wahl einer Maßeinheit vom BlackBox System ändern zu lassen. Der Quelltext des Programms unterscheidet sich vom Modul *TutKonvert2* lediglich durch die Existenz der zusätzlichen Prozedur *UnitNotifier*. Diese Prozedur ruft im Anschluss an die in der CASE-Abfrage durchgeführte Änderung des Wertes von *TutKonvert3.dbox.Sorte* die Anweisung *Dialog.Update(dbox)* auf, folglich wird der aktualisierte Wert des Feldes *dbox.Sorte* unmittelbar nach der Änderung in der Dialogbox angezeigt.

Auf welche Weise die dazu nötigen Schritte im Hintergrund von BlackBox ausgeführt werden, können Sie erfahren, wenn Sie sich mit dem Inspektor in den Dialogboxen Tut/Rsrc/Konvert2.odc und Tut/Rsrc/Konvert3.odc die Eigenschaften der Auswahlfelder ansehen, die den einzelnen Maßeinheiten zugeordnet sind. Im Gegensatz zu Konvert2.odc enthalten die mit *Notifier* beschrifteten Felder bei Konvert3.odc den Eintrag *TutKonvert3.UnitNotifier*. In der oben abgedruckten Schnittstelle des Moduls *Dialog* finden Sie den Prozedurtyp *NotifierProc = PROCEDURE (op, from, to: INTEGER)*. Vergleichen Sie die Signatur von *TutKonvert3.UnitNotifier* mit dem Typ *Dialog.NotifierProc*, wird Ihnen die Übereinstimmung der Parameterlisten auffallen (die Bedeutung der Parameter ist im Augenblick unwichtig, mit ihrer Hilfe lassen sich für spezielle Zwecke einzelne Anweisungen innerhalb der Prozedur präziser steuern, eine dieser Möglichkeiten werden sie im Programm Konvert5.odc kennenlernen). Im Programm ProcTyp.odc (Programm 6.1) haben Sie gesehen, daß jeder Variablen eines Prozedurtyps eine beliebige, in der Signatur übereinstimmende Prozedur zugewiesen werden kann, die Prozedur *TutKonvert3.UnitNotifier* kann also

einer innerhalb des Moduls *Dialog* (oder auch anderer BlackBox Systemmodule) deklarierten Variablen des Typs *Dialog.NotifierProc* zugewiesen werden. Die Zuweisung erfolgt dadurch, daß im Inspektor-Dialog der zu "benachrichtigenden" Variablen die zuzuweisende, exportierte Prozedur (*TutKonvert3.UnitNotifier*) in das *Notifier*-Feld eingetragen wird, BlackBox sorgt dann bei jeder Änderung des zugehörigen Dialogfelds automatisch für die Ausführung der zugeordneten *Notifier*-Prozedur und damit für die Aktualisierung der Dialogbox.

Mit den vorstehenden Informationen haben Sie die Wesentlichkeiten des Moduls *TutKonvert3* kennengelernt, aber es gibt wieder einmal einige Details, die mit der zu dem Modul gehörenden Dialogbox (Datei *Tut/Rsrc/Konvert3.odc*) zusammenhängen. Als bisher unbeachtete Neuigkeit finden Sie bei den Schaltflächen *Anleitung* und *Umrechnen* im Inspektor links unterhalb des mit *Notifier* gekennzeichneten Listenfeldes zwei Kontrollfelder (check boxes), die mit *Default* bzw. *Cancel* beschriftet sind. Wenn das *Default* Feld einer Schaltfläche aktiviert ist, wirkt die Eingabetaste der Tastatur wie ein Mausklick auf die entsprechende Schaltfläche, sofern das aktive, an dem Punktrahmen kenntliche Element des Dialogs nicht selbst eine Schaltfläche ist; wenn Sie also nach Auswahl einer umzurechnenden Maßeinheit eine Zahl in das Eingabefeld des Dialogs geschrieben haben, können Sie die Umrechnung statt durch einen Mausklick auf die Schaltfläche auch durch die Eingabetaste auslösen.

Ist statt des *Default*-Feldes im Inspektor das *Cancel*-Feld aktiviert, können Sie die Funktion der entsprechenden Schaltfläche statt durch einen Mausklick mit der Escape-Taste aktivieren, eine Möglichkeit, die insbesondere bei Schaltflächen zur Beendigung eines Dialogs nützlich ist. Ein solches Feld ist *Schließen* im Dialog *Maßnahmen3*; die zugehörige BlackBox Anweisung können Sie mit Hilfe des Inspektors finden, sie lautet *StdCmds.CloseDialog*. Modul *StdCmds* ist ein BlackBox Systemmodul, das ähnlich wie Modul *Dialog* eine Reihe nützlicher Routinen zusammenfasst, eine dieser Routinen, *StdCmds.OpenAuxDialog*, das Kommando, mit dem sich Dialogboxen öffnen lassen, haben Sie bereits kennengelernt.

Eine andere, üblicherweise vorhandene Möglichkeit, innerhalb einer Dialogbox mit Tastaturbefehlen zu arbeiten, besteht darin, durch Betätigen der Tabulatortaste von einem Eingabefeld zum nächsten zu wechseln; innerhalb einer zusammengehörenden Gruppe von Eingabefeldern, beispielsweise der vier Auswahlfelder der Gruppe *Maßeinheit* in der Dialogbox *Maßnahmen 3*, läßt sich außerdem zwischen den einzelnen Feldern mit den Pfeiltasten des Nummernblocks wechseln. Damit die Wechsel nicht entsprechend der Reihenfolge der Erstellung der einzelnen Kontrollelemente in einer für den Benutzer unverständlichen Weise erfolgen, gibt es im Menü *Layout* die Optionen *Sort Views*, *Set First/Back* und *Set Last/Front*. Mit *Sort Views* wird die Reihenfolge der Kontrollelemente so sortiert, daß sie einem intuitiven Benutzerverständnis entspricht, bei Betätigung der Tabulatortaste wechselt die Markierung jeweils zum nächsten Element, wobei das nächste Element das nächste rechts vom aktuell aktiven bzw. das direkt unterhalb am weitesten links stehende ist, diese Anordnung wird Z-Ordnung genannt.

Dagegen können Sie mit *Set First/Back* und *Set Last/Front* gezielt die Z-Ordnung verändern. Dies ist nützlich, wenn sich Kontrollelemente überlappen bzw. überdecken. Wenn beispielsweise eine *Group Box* nach der Erstellung eines Dialogs über einem Eingabefeld liegt, kann der Benutzer keine Eingaben in das hinter dem Rahmen liegende Feld machen, weil der Rahmen zwar durchsichtig aber undurchlässig ist; in diesem Fall können Sie den Rahmen markieren und mit *Set First/Back* gezielt hinter dem Eingabefeld plazieren.

9.3. Textfenster

Dialogboxen sind eine wichtige, aber zweifellos nicht die einzige Möglichkeit, die Kommunikation zwischen einem Programm und dem Benutzer zu gestalten. Eine weiteres, insbesondere bei größerem Umfang der auszugebenden Daten wichtiges Verfahren besteht darin, die Resultate einer Programmaktion in einem Textfenster darzustellen. BlackBox als Programmentwicklungsumgebung enthält für die Ein- und Ausgabe von Daten das Log-Fenster, das für diese Zwecke sehr nützlich ist, in den meisten professionellen Anwenderprogrammen sind derartige Log-Fenster jedoch nicht üblich.

Das Programm Konvert4.odc macht Sie mit der Erstellung von Fenstern für die Ausgabe von Texten bekannt. Als erstes wird Ihnen wahrscheinlich wieder einmal die Ähnlichkeit mit den vorangegangenen Programmen auffallen, das Modul enthält in unveränderter Form die Deklaration der Variablen *dbox* sowie die Prozeduren *Umrechnen* und *UnitNotifier*. Auch die zugehörige Dialogbox, die Sie mit dem am Ende des Moduls stehenden Kommando *StdCmds.OpenAuxDialog('Tut/Rsrc/Konvert4', 'Maßnahmen 4')* öffnen können, erscheint im Vergleich mit dem Programm Konvert3.odc unverändert, beide Dialoge enthalten exakt die gleichen Felder und Schaltflächen mit der entsprechenden Funktionalität. Den einzigen Unterschied zwischen beiden Programmen erleben Sie, wenn Sie auf die Schaltfläche *Anleitung* klicken. Während bei der Dialogbox *Maßnahmen 3* die entsprechende Information im Log-Fenster dargestellt wird, erfolgt die Ausgabe des Textes bei *Maßnahmen 4* in einem separaten Fenster.

Die Erstellung eines neuen Fensters und das Anzeigen eines Textes in diesem Fenster sind keine triviale Programmierarbeit, wie Sie bereits an der Zahl der vom Modul *TutKonvert4* importierten Module sehen können, um so mehr werden Sie sich freuen, daß diese Arbeit von den BlackBox Programmierern in einer Weise geleistet worden ist, die die restlichen Schritte für Gelegenheitsprogrammierer wie Sie und mich beinahe zum Kinderspiel macht.

Die Prozedur *TutKonvert4.Anleitung* deklariert als eine ihrer drei Variablen *m: TextModels.Model*. Dem nachstehenden Auszug der erweiterten Schnittstelle des Moduls *TextModels* können Sie entnehmen, daß *TextModels.Model* eine Klasse im Sinn von Kapitel 8 darstellt (den Zusatz *POINTER TO* in der Typ-

deklaration bitte ich vorläufig zu ignorieren, er wird im nächsten Kapitel erklärt werden). Außerdem finden Sie zwei weitere Klassen, *TextModels.Directory* und *TextModels.Writer*

```
DEFINITION TextModels;

  IMPORT Containers;

  TYPE
    Directory = POINTER TO ABSTRACT RECORD
      (d: Directory) New (): Model, NEW, ABSTRACT
    END;

    Model = POINTER TO ABSTRACT RECORD (Containers.Model(Models.Model(Stores.Store)))
      (m: Model) Length (): INTEGER, NEW, ABSTRACT
    END;

    Writer = POINTER TO ABSTRACT RECORD
      (wr: Writer) Base (): Model, NEW, ABSTRACT
    END;

  VAR
    dir-: Directory

END TextModels.
```

Auszug der erweiterten Schnittstelle des Moduls*TextModels*

Jede der drei Klassen exportiert eine Methode als typgebundene Prozedur. Von diesen ist im Augenblick nur die Prozedur *New* in der Klasse *TextModels.Directory* wichtig. *New* erscheint in der Prozedur *TutKonvert4.Anleitung* in der ersten Anweisung *m := TextModels.dir.New()*. Es wird ein neues (leeres) Textmodell erzeugt und der Variablen *m* zugewiesen, diese ist danach in der Lage, jede Art von (Text-) Zeichen aufzunehmen. Da die Methode *New* nur von einem Objekt (einer Variablen) der Klasse *TextModels.Directory* aufgerufen werden kann, stellt das Modul *TextModels* mit *dir: Directory* ein passendes Objekt zur Verfügung. (Die dritte exportierte Klasse *TextModels.Writer* bitte ich im Moment zu ignorieren, sie wird im Abschnitt 9.4. erläutert werden.)

Die ebenfalls in *TutKonvert4.Anleitung* deklarierte Variable *f: TextMappers.Formatter* ist ein sogenanntes Formatierungsobjekt. Wie Sie leicht aus den weiteren Anweisungen der Prozedur ersehen können, dient ein Formatierungsobjekt dazu, Daten in ein (Text-) Modell zu schreiben. Bevor dies jedoch möglich ist, muß das Objekt mit dem Modell verbunden werden.

```
DEFINITION TextMappers;

    IMPORT TextModels;

    CONST
        real = 5;

    TYPE
        Formatter = RECORD
            rider-: TextModels.Writer;
            (VAR f: Formatter) ConnectTo (text: TextModels.Model), NEW;
            (VAR f: Formatter) SetPos (pos: INTEGER), NEW;
            (VAR f: Formatter) WriteLn, NEW;
            (VAR f: Formatter) WriteRealForm (x: REAL; precision, minW, expW: INTEGER;
                                              fillCh: CHAR), NEW;
            (VAR f: Formatter) WriteString (x: ARRAY OF CHAR), NEW
        END;

        Scanner = RECORD
            type: INTEGER;
            real: REAL;
            (VAR s: Scanner) ConnectTo (text: TextModels.Model), NEW;
            (VAR s: Scanner) Pos (): INTEGER, NEW;
            (VAR s: Scanner) Scan, NEW;
            (VAR s: Scanner) SetPos (pos: INTEGER), NEW
        END

END TextMappers.
```

Auszug der Schnittstelle des Moduls*TextMappers*

Der Schnittstellenauszug des Moduls *TextMappers* zeigt, daß die Klasse *TextMappers.Formatter* eine Methode *(VAR f: Formatter) ConnectTo (text: TextModels.Model), NEW* enthält, diese Methode wird in der Zeile *f.ConnectTo(m)* der Prozedur *TutKonvert4.Anleitung* benutzt, um die gewünschte Verbindung herzustellen. Außer *ConnectTo* benutzt die Prozedur *TutKonvert4.Anleitung* zwei weitere Methoden der Klasse *TextMappers.Formatter*, die Ihnen vermutlich unmittelbar verständlich sind. Die Deklarationen *(VAR f: Formatter) WriteLn, NEW* und *(VAR f: Formatter) WriteString (x: ARRAY OF CHAR), NEW* machen Sie mit den "erwachsenen" Verwandten der Prozeduren *Out.Ln* und *Out.String* bekannt (die übrigen Deklarationen der Schnittstelle können Sie im Augenblick ebenfalls ignorieren, sie werden im Zusammenhang mit dem Modul *TutScanForm* im Abschnitt 9.5 erläutert).

Die Ausgabeanweisungen von *TutKonvert4.Anleitung* sind nahezu selbsterklärend, das Formatierobjekt *f* schreibt eine Reihe von Textstücken und Zeilenumbrüchen in das Textmodell. Wichtig ist, sich

klarzumachen, daß die Zeichenketten und Zeichen damit zwar in das Textmodell, aber im Gegensatz zu den Ausgabeanweisungen des Moduls *Out*, die unmittelbar im Log-Fenster erscheinen, keineswegs auf den Bildschirm geschrieben werden.

Die Bildschirmdarstellung des gesamten Textes erfolgt nach dessen Fertigstellung mit Hilfe des Moduls *Views*. Für die Darstellung benötigen Sie nur eine einzige Anweisung, die Sie am Ende von *TutKonvert4.Anleitung* finden, *Views.OpenView(v)*.

```
DEFINITION Views;

    IMPORT Stores;

    TYPE
        View = POINTER TO ABSTRACT RECORD (Stores.Store) END;
        Title = ARRAY 64 OF CHAR;

    PROCEDURE OpenView (view: View);
    PROCEDURE OpenAux (view: View; title: Title)

END Views.
```

Auszug der Schnittstelle des Moduls *Views*

Dem Schnittstellenauszug des Moduls *Views* können Sie entnehmen, daß die von *Views* exportierte Prozedur *OpenView* als Parameter ein Objekt der Klasse *Views.View* benötigt, das von *TutKonvert4.Anleitung* mit der dritten Variablen *v: TextViews.View* zur Verfügung gestellt wird.

Im Kapitel 8 habe ich Ihnen eine Klasse als erweiterbaren Verbundtyp vorgestellt. Der Schnittstellenauszug des Moduls *TextViews* zeigt, daß *TextViews.View* eine Unterklasse von *Containers.View* und damit von *Views.View* ist (s. auch Abb. 13 im Abschnitt 9.5). Auf Grund der Vererbung "kennt" die Klasse *Views.View* die Klasse *TextViews.View*, in *Views.OpenView* ist der formale Parameter *view: View* mit dem aktuellen Parameter *v* der Unterklasse *TextViews.View* kompatibel. Der Erfolg der objektorientierten Programmierung beruht auf dieser als Polymorphie bekannten Flexibilität von Klassen, da es mit ihrer Hilfe möglich ist, bestehende Programme an Hand ihrer Schnittstellen zu jeder Zeit ohne Kenntnis der Programmquelltexte und ohne Neukompilation der bestehenden Programme zu erweitern und mit zusätzlichen Fähigkeiten zu versehen.

```
DEFINITION TextViews;

    IMPORT TextModels, Containers;

    TYPE
        Directory = POINTER TO ABSTRACT RECORD
            (d: Directory) New (text: TextModels.Model): View, NEW, ABSTRACT
        END;

        View = POINTER TO ABSTRACT RECORD (Containers.View(Views.View(Stores.Store)))
            (v: View) ThisModel (): TextModels.Model, EXTENSIBLE;
            (v: Containers.View) ThisController (): Containers.Controller, NEW, EXTENSIBLE
        END;

    VAR
        dir-: Directory

END TextViews.
```

Auszug der erweiterten Schnittstelle des Moduls *TextViews*

Die Schnittstelle des Moduls *TextViews* ist der des Moduls *TextModels* sehr ähnlich; Sie sehen, daß eine Textsicht wie ein Textmodell erst mittels der *New*-Methode der Klasse *TextViews.Directory* erzeugt werden muß; ebenso wie Modul *TextModels* stellt Modul *TextViews* dafür das passende *dir*-Objekt zur Verfügung. Bevor also in *TutKonvert4.Anleitung* mit der Anweisung *Views.OpenView(v)* das Textfenster geöffnet werden kann, muß die Variable *v* mit einer zuvor erzeugten Sicht verbunden werden, beides zusammen geschieht mit der Anweisung *v := TextViews.dir.New(m)*. Beachten Sie bitte, daß *TextViews.dir.New* anders als *TextModels.dir.New* einen Parameter vom Typ *Views.View* oder einer Unterklasse erwartet. Dies liegt daran, daß eine Sicht nur in Verbindung mit einem Modell existiert, während ein Modell durchaus ohne Sicht vorkommen kann, auf diesen Unterschied und das damit zusammenhängende MVC Entwurfsmuster werde ich im Abschnitt 9.5 genauer eingehen.

9.4. Guards

Das nächste Programm dieses Kapitels, Konvert5.odc, stellt Ihnen eine weitere mit Dialogboxen in Verbindung stehende Neuigkeit vor, die sogenannten guards. Darunter sind in BlackBox Prozeduren zu verstehen, die Felder eines Dialogs oder einen Menüeintrag überwachen. Guards sorgen auf eine angenehme Weise dafür, daß dem Benutzer eines Programms signalisiert wird, ob ein Menüeintrag, eine Schaltfläche oder ein Eingabefeld in einem Dialog aktiv oder passiv ist. Dies geschieht dadurch, daß bei einem

Auswahlmenü oder einem Dialog diejenigen Elemente gekennzeichnet werden, die im aktuellen Zustand des Systems deaktiviert sind. Die Art der Kennzeichnung ist abhängig von der jeweiligen Systemumgebung, oft werden deaktivierte Elemente "unsichtbar" gemacht, d. h. in einer Farbe dargestellt, die der Hintergrundfarbe ähnelt, sodaß nicht wählbare Elemente zwar noch erkennbar, aber unauffälliger als die übrigen sind. Versucht der Benutzer trotzdem, ein derartiges Element zu aktivieren, erfolgt keine Reaktion des Systems, die passiven Elemente sind gesperrt.

In älteren Programmen war es dagegen üblich, den Benutzer jede der angebotenen Aktionen erst ausführen zu lassen und im Fall einer unzulässigen Aktion anschließend eine Fehlermeldung auf den Bildschirm zu bringen. Diese Verfahrensweise hat zwei wesentliche Nachteile, zum einen wird der Benutzer im Unklaren darüber gelassen, welche Aktionen das System in der jeweiligen Situation sinnvoll durchführen kann, zum anderen wird die Programmerstellung erheblich kompliziert, der Programmierer muß die unterschiedlichsten "Fehl"bedienungen des Programms durch den Benutzer vorhersehen und für jeden Einzelfall Abfangmechanismen einbauen, beides erweist sich immer wieder als äußerst schwierig. Eine sinnlose oder gefährliche Aktion gar nicht erst zuzulassen ist dagegen sehr viel einfacher, der Benutzer wird zu jeder Zeit über die Möglichkeiten des Programms informiert, und auf der Seite der Programmerstellung trägt die Existenz einer sicheren Standardschutzmaßnahme, die sich bequem anwenden läßt, wesentlich zur Programmstabilität bei.

Aufbau und Inhalt des Programms Konvert5.odc, das Sie mit beiden Möglichkeiten der Fehlerbehandlung bekannt macht, stimmen weitgehend mit dem Programm Konvert4.odc überein. Bei den Importen finden Sie zusätzlich das Modul *StdCmds*, das Sie bereits mehrfach zum Öffnen von Dialogboxen benutzt haben, es wird zu diesem Zweck ebenfalls in der Prozedur *UnitNotifier* verwendet. Die CASE-Abfrage der Prozedur ist gegenüber dem Modul *TutKonvert4* um zwei Fälle erweitert worden. Öffnen Sie mit dem Kommando *"StdCmds.OpenAuxDialog('Tut/Rsrc/Konvert5', 'Maßnahmen 5')"* die zum Modul gehörende Dialogbox (Datei Tut/Rsrc/Konvert5.odc), finden Sie die beiden Fälle als die Auswahlfelder *Unzen* und *Quart* wieder.

Während die ersten vier Auswahlfelder für die Maßeinheit genauso reagieren, wie Sie es aus dem Programm Konvert4.odc kennen, öffnet sich bei einem Klick auf das Auswahlfeld *Unzen* entsprechend der oben geschilderten älteren Programmlogik ein zweiter Dialog mit einer Fehlermeldung (s. Abb. 10). Die notwendigen Programmschritte finden Sie in der CASE-Abfrage der Prozedur *TutKonvert5.UnitNotifier* bei dem zugehörigen Fall *unze*. Nach dem Setzen eines Wertes für das Feld *dbox.Einheit* wird die Fehlermeldung in den dafür deklarierten Verbund *Fehlerbox* geschrieben und dessen Inhalt wird in einer eigenen Dialogbox angezeigt. Die anschließend aufgerufene Prozedur *Umrechnen* schreibt den symbolischen Wert *inf* in das Feld *dbox.Ergebnis* und die Dialogbox wird von der Prozedur *Dialog.Update* aktualisiert.

Abbildung 10
Fehlermeldung des Feldes *Unzen* im Dialog *Konvert 5*

Dagegen verkörpert der zweite zusätzlich in der CASE-Abfrage stehende Fall *quart* die modernere der beiden geschilderten Programmlogiken. Klicken Sie in der Dialogbox auf das zugehörige Auswahlfeld *Quart*, gibt es als Reaktion lediglich die oben beschriebene Änderung einiger Elemente des Dialogs, diese sind in den passiven Zustand übergegangen und zeigen keine Inhalte mehr an. Außerdem ist es nicht mehr möglich, in das Eingabefeld zu klicken, um eine umzurechnende Zahl einzugeben, das Feld ist gesperrt. Bei *Quart* handelt es sich um ein Dialogelement, mit dem Teile des Dialogs und damit Teile des Programms aktiviert und passiviert werden können.

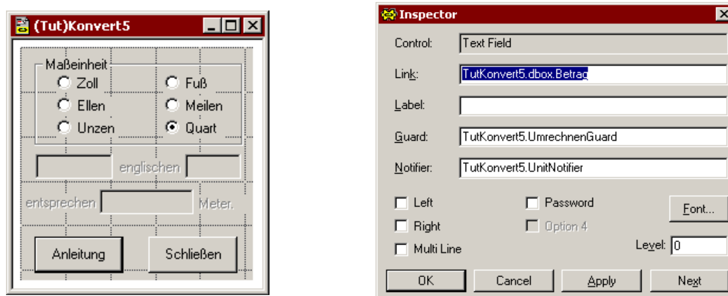

Abbildung 11
Guard* Prozedur des Feldes *TutKonvert5.Betrag* im Dialog *Konvert 5

Obwohl die Dialogbox sich beim Anklicken der "Maßeinheit" *Quart* anders verhält als bei den ersten vier Maßeinheiten, unterscheiden sich die zugehörigen Teile der CASE-Abfrage innerhalb des Programms nicht, der Fall *quart* hat in der Prozedur *TutKonvert5.UnitNotifier* keinerlei Sonderrolle, diese ergibt sich

vielmehr aus der zusätzlichen Prozedur *TutKonvert5.UmrechnenGuard*. Wenn Sie sich die Eigenschaften der deaktivierten Elemente im Dialog Tut/Rsrc/Konvert5.odc mit dem Inspektor anzeigen lassen, sehen Sie, daß bei jedem dieser Elemente als *Guard* die Prozedur *TutKonvert5.UmrechnenGuard* eingetragen ist (s. Abb. 11).

Die Schnittstelle des Moduls *Dialog* (s. Kapitel 9.1) enthält den Typ *GuardProc = PROCEDURE (VAR par: Par)*, mit dem die Prozedur *UmrechnenGuard* kompatibel ist. Wie Sie ebenfalls der Schnittstelle von *Dialog* entnehmen können, enthält der Verbundtyp *Par* (neben anderen, nicht aufgeführten Feldern) ein Feld *disabled: BOOLEAN*. Eine Guard-Prozedur kann über dies Feld das Modul *Dialog* benutzen, um Elemente einer Dialogbox oder auch einen Menüeintrag zu deaktivieren. Die einzige Anweisung der Prozedur *TutKonvert5.UmrechnenGuard* lautet *par.disabled := ~(dbox.Sorte IN {zoll, fuss, elle, meile, unze})*. Sie setzt den Wert des Feldes *par.disabled* auf *FALSE*, sofern das Feld *dbox.Sorte* einen der Werte *zoll* bis *unze* enthält, anderenfalls enthält *par.disabled* den Wert *TRUE*. Das Modul *Dialog* sorgt dafür, daß alle Elemente der Dialogbox deaktiviert werden, in deren *Guard*-Feld die Prozedur *TutKonvert5.UmrechnenGuard* eingetragen ist, sobald das Feld *dbox.Sorte* einen anderen Wert als 1 bis 5 annimmt, beispielsweise also 6, den Wert der Konstanten *quart*.

Auf zwei Einzelheiten der Prozedur *TutKonvert5.UnitNotifier* möchte ich Sie aufmerksam machen. Die gesamte CASE-Abfrage steht innerhalb der Abfrage *IF op = Dialog.changed THEN*. Die drei Parameter *op, from, to* einer *Notifier* -Prozedur lassen sich dazu verwenden, die Änderungen der zugehörigen Dialogbox gezielt feinzusteuern. In der Prozedur *TutKonvert5.UnitNotifier* wird der Parameter *op* in der *IF*-Abfrage daraufhin getestet, ob sich eines der Felder der Dialogbox geändert hat, nur in diesem Fall wird die CASE-Abfrage durchgeführt (für die übrigen möglichen Parameterwerte und die damit verbundenen Optionen studieren Sie bitte die vollständige Dokumentation des Moduls *Dialog*).

Außerdem sollten Sie beachten, daß die Prozedur *TutKonvert5.UnitNotifier* einen gegenüber der Prozedur *TutKonvert4.UnitNotifier* veränderten Aufbau hat. An ihrem Ende wird die Prozedur *TutKonvert5.Umrechnen* statt der Prozedur *Dialog.Update* aufgerufen, *Dialog.Update* selbst wird lediglich einmal in *TutKonvert5.Umrechnen* aktiviert. Im Zusammenhang damit gibt es gegenüber der Dialogbox in *TutKonvert4* einige weitere Änderungen. Zum einen finden Sie, wie bereits erwähnt, bei dem Eingabefeld *Betrag* sowie den Anzeigefeldern *Einheit* und *Ergebnis* als *Guard* jeweils die Prozedur *TutKonvert5.UmrechnenGuard*. Außerdem sehen Sie bei dem Eingabefeld *Betrag* als *Notifier* die Prozedur *TutKonvert5.UnitNotifier* eingetragen, während beim Modul *TutKonvert4* die analoge *Notifier*-Prozedur *TutKonvert4.UnitNotifier* dem Feld *Einheit* zugeordnet wurde. In der Dialogbox *TutKonvert4* ergibt die Aktivierung eines der Auswahlfelder, denen die *Notifier*-Prozedur ebenfalls zugeordnet ist, also lediglich eine Änderung des Anzeigefeldes *Einheit*, das Feld *Ergebnis* bleibt unverändert. Ebenso führt eine Änderung des Wertes im Eingabefeld *Betrag* nicht zu einer Änderung der übrigen Dialogfelder, diese erfolgt erst nach Aktivierung der Schaltfläche *Umrechnen*.

Dagegen führt jede Änderung in der Dialogbox von *TutKonvert5* zu einer sofortigen Aktualisierung aller Felder. Dies liegt einmal daran, daß die *Notifier*-Prozedur dem Feld *Betrag* zugeordnet ist, eine Änderung wird also von ihr unmittelbar verarbeitet. Zum anderen liegt es daran, daß die *Notifier*-Prozedur anders als im Modul *TutKonvert4* nicht direkt die Prozedur *Dialog.Update* aktiviert, dies geschieht in *TutKonvert5* vielmehr auf dem Umweg über die Prozedur *Umrechnen*, die vor der Aktivierung von *Dialog.Update* ihrerseits die Felder der Dialogbox aktualisiert. Im Zusammenspiel dieser Änderung mit der geänderten Zuordnung der *Notifier*-Prozedur erfolgt daher bei jeder Aktion des Benutzers eine automatische Neuanzeige der Dialogbox. Als Konsequenz ist die in *TutKonvert4* gesondert vorhandene Schaltfläche *Umrechnen* in *TutKonvert5* verschwunden und die Prozedur *Umrechnen* ist faktisch Bestandteil der *Notifier*-Prozedur geworden. Als selbständige Prozedur wird sie nicht mehr benötigt, beide Prozeduren könnten in *TutKonvert5* auch zu einer einzigen (*Notifier*-)Prozedur zusammengefasst werden.

Eine weitere Neuigkeit des Moduls steht im Zusammenhang mit der Anzeige des Umrechnungsergebnisses in der Dialogbox. Geben Sie in das *Betrag*-Feld der Dialogbox *TutKonvert4* beispielsweise die Zahl *1.23456789* ein und wählen als umzurechnende Einheit *Meilen*, erhalten Sie im Feld *Ergebnis* eine unleserliche Anhäufung von Ziffern. Dagegen liefert die gleiche Eingabe in der Dialogbox *TutKonvert5* eine lesbare Anzeige mit vier Dezimalziffern. Dies unterschiedliche Verhalten werden Sie verstehen, wenn Sie in den beiden Dialogboxen mit Hilfe des Inspektors die Eigenschaften der jeweiligen *Ergebnis*-Felder vergleichen.

Im Dialog *TutKonvert4* finden Sie im Feld *Level* in der unteren rechten Ecke den Standardeintrag "0", bei dem die Länge des Ergebnisses unbestimmt ist. Bei der Dialogbox *TutKonvert5* steht als Wert in diesem Feld die Zahl "- 4". Im Zusammenhang mit der zweiten Dialogbox des Moduls *TutKonvert1* haben Sie erfahren, daß Sie einer Auswahlschaltfläche wie *Meilen* über das Inspektorfeld *Level* einen festen Ganzzahlwert zuweisen können. Handelt es sich statt der Auswahlschaltfläche um ein Textfeld, das mit einer Ganzzahlvariablen verknüpft ist, ergibt ein positiver Wert im Feld *Level* eine Anzeige des Ganzzahlwertes als Pseudodezimalzahl, der *Level*-Wert "2" beispielsweise führt dazu, daß die ganze Zahl "453" als "4.53" angezeigt wird. Handelt es sich beim Typ der assoziierten Variablen jedoch um reelle Zahlen, ergibt ein positiver *Level*-Wert eine Anzeige der Zahl in wissenschaftlicher Notation, die Dezimalzahl "0.9144567" wird bei dem *Level*-Wert "4" als "9.145E-1" mit insgesamt vier gültigen Ziffern, also drei Dezimalen in der Mantisse, angezeigt. Dagegen ergibt ein negativer *Level*-Wert eine Fixierung der Nachkommastellen auf den Betrag des *Level*-Werts, bei dem *Level*-Wert "-4" wird die Dezimalzahl "234.9144567" als "234.9145" mit vier Dezimalen angezeigt.

Eine erwähnenswerte Einzelheit des Moduls *TutKonvert5* betrifft die Prozedur *Anleitung*. Vergleichen Sie die Variablendeklarationen mit denen von *TutKonvert4.Anleitung*, finden Sie von den drei dort deklarierten Variablen nur eine in *TutKonvert5.Anleitung* wieder, *f: TextMappers.Formatter*, die beiden anderen erscheinen lediglich zur Verdeutlichung der folgenden Änderungen in Kommentarklammern. Der Vergleich der Prozeduren *TutKonvert4.Anleitung* und *TutKonvert5.Anleitung* zeigt als erste Änderung in der Zeile *f.ConnectTo(TextModels.dir.New())*, daß ein Formatierobjekt *f* sich ohne Zwischenschaltung eines Objekts vom Typ *TextModels.Model* selbst mit einem anonym erzeugten neuen Textmodell verbinden kann.

Die zweite Änderung betrifft die Darstellung des Textes auf dem Bildschirm. Der Schnittstelle des Typs *TextMappers.Formatter* (s. Abschnitt 9.3) können Sie entnehmen, daß ein *Formatter*-Objekt seinerseits ein Objekt *rider: TextModels.Writer* enthält. Ein *rider*-Objekt ähnelt einem *Formatter*-Objekt, beide dienen dazu, sich durch eine Datenstruktur bewegen zu können, ohne daß der Klient wissen muß, wie diese im Einzelnen aufgebaut ist. Sowohl *rider* als auch *formatter* sind Vertreter eines allgemein als Iterator bezeichneten Entwurfsmusters, das dazu dient, eine Datenstruktur, deren Darstellung und die Aktionen des Benutzers zu entkoppeln (näheres dazu finden Sie im folgenden Abschnitt).

Der Typ von *rider* - die Klasse *TextModels.Writer* - exportiert die Methode *(wr: Writer) Base(): Model, NEW, ABSTRACT*, die das zugrunde liegende Modell zurückgibt. Anders als in *TutKonvert4.Anleitung*, wo mit den Zeilen *v := TextViews.dir.New(m)* und *Views.OpenView(v)* zu einem explizit deklarierten Textmodell zuerst eine Sicht erzeugt und anschließend geöffnet wird, kann dieser Vorgang unter Verwendung eines *rider*-Objekts in der Anweisung *Views.OpenAux(TextViews.dir.New(f.rider.Base()), "Anleitung")* zusammengefasst und anonymisiert werden, die explizite Deklaration eines *View*-Objekts ist nicht mehr erforderlich. Die Prozedur *Views.OpenAux* ist eine erweiterte Form von *Views.OpenView*, mit der ein Fenster geöffnet und mit einem Titel versehen werden kann (s. die Schnittstelle des Moduls *Views* im Abschnitt 9.3).

9.5. Das Model - View - Controller Muster

Zwei der drei in der Prozedur *TutKonvert4.Anleitung* deklarierten Variablen sind *m: TextModels.Model* und *v: TextViews.View*. Deren Klassen finden Sie zusammen mit einer dritten, *TextControllers.Controller*, im Programm ScanForm.odc wieder. Die drei Klassen repräsentieren - auch namentlich - eines der wichtigsten Entwurfsmuster in der modernen Programmiertechnik, das Model-View-Controller (MVC) Muster. Darunter versteht man die Aufgliederung der Programmlogik in drei weitgehend unabhängige, aber zusammenspielende Teile. Der erste Teil ist das Modell, das eine Abstraktion der zu verarbeitenden Daten darstellt, der zweite ist die Sicht auf die Daten, ihre Darstellung auf dem Bildschirm oder einem anderen Ausgabemedium. Zwischen beiden Teilen und dem Benutzer vermittelt als dritter Beteiligter der Kontrollteil.

Aufteilungen komplexer Vorgänge in dieser oder einer ähnlichen Art finden sich in vielen Situationen wieder. Nehmen Sie beispielsweise den Ihnen wahrscheinlich vertrauten Vorgang der Wiedergabe einer

Musikaufzeichnung. Der erste Teil, das Datenmodell (model) sind die auf einem Datenträger befindlichen Musikstücke. Der zweite Teil, die Präsentation der Daten in Form von Klängen (view), wird vom Endverstärker und den Lautsprechern der Musikanlage ausgeführt; den dritten Teil, den Kontrollteil (controller), bildet die Abspieleinheit zusammen mit den Regelmöglichkeiten von Vorverstärker, Equalizer und anderen Geräten zur Klangbeeinflussung.

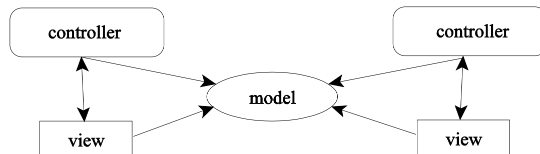

Abbildung 12
Das Model - View - Controller Muster

Abbildung 12 zeigt das Grundkonzept des MVC Musters. Eine mögliche Realisierung dieses Musters ist ein Text im Computer. Der Text besteht aus einem Textmodell, das eine Folge von Zeichen (*ARRAY OF CHAR*) darstellt. Damit ist nicht nur der reine Text gemeint, ein Text enthält neben den darstellbaren Zeichen auch Formatierungsmerkmale, mit denen Wortabstände, Zeilenumbrüche, Schriftart, Schriftgrad, Schriftschnitt und andere Attribute festgelegt werden. Ein moderner Texteditor wie zum Beispiel der BlackBox Editor kann außerdem Video- oder Klangschnipsel enthalten sowie Graphikboxen, die ihrerseits beliebige weitere Elemente enthalten könnten, wobei der Schachtelungstiefe solcher Elemente prinzipiell keine Grenzen gesetzt sind, Texte im engeren Sinn sind also nur ein Teil der in einem Computer zu verarbeitenden und darzustellenden Daten. Im MVC Muster müssen alle Daten ein Modell in dem oben dargestellten Sinn haben, ein solches Modell wird im BlackBox System durch die Klasse *Models.Model* zur Verfügung gestellt.

Das Modell kümmert sich in keiner Weise um die Präsentation der Daten, dafür ist der Sichtteil (view) des MVC Musters zuständig, im BlackBox System die Klasse *Views.View*. Eine Sicht weiß ihrerseits nichts Inhaltliches über die Daten, sie ist ausschließlich dazu da, die Daten in angemessener Weise zu präsentieren. Zu jedem Modell kann es eine oder mehrere Sichten geben, jede Sicht hat ihren eigenen Kontrollteil, der Änderungen der Sicht z. B. durch Eingabe von Zeichen mit der Tastatur oder Mausklicke registriert und an das Modell weiterleitet. Im BlackBox System sind Kontrollteile Instanzen der Klasse *Controllers.Controller* oder einer ihrer Unterklassen.

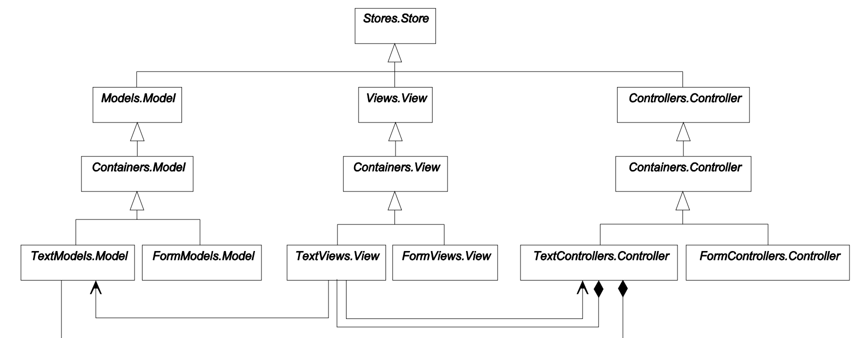

Abbildung 13
UML Diagramm der Klassenhierarchien des MVC Musters im BlackBox framework

Der Zusammenhang der verschiedenen Klassen läßt sich am einfachsten in graphischer Weise veranschaulichen (s. Abb. 13). Die Darstellung folgt den Regeln der *Unified Modeling Language* (UML), mit der sich die Beziehungen von Klassen, Objekten und Methoden in einer programmiersprachenunabhängigen Weise aufzeigen lassen. Für die Erläuterung des MVC Musters in BlackBox werden aus dem umfangreichen Katalog der UML Graphiken nur vier benötigt [12]. Ein Rechteck repräsentiert eine Klasse, der Klassenname wird im Rechteck vermerkt. Ein Dreieck stellt eine Vererbungsbeziehung dar (class inheritance). Eine Linie mit Pfeil (manchmal auch ohne) stellt eine allgemeine Beziehung zwischen Klassen dar (class association), diese Art Beziehung wird auch als "Benutzt"- Beziehung bezeichnet, eine Klasse benutzt Eigenschaften einer zweiten (am Pfeilende stehenden) Klasse. Dem Auszug der erweiterten Schnittstelle des Moduls *TextViews* können Sie entnehmen, daß die Klasse *TextViews.View* derartige Referenzen zur Klasse *TextModels.Model* als Methode *ThisModel* und zur Klasse *TextControllers.Controller* als Methode *ThisController* enthält.

Das vierte graphische Symbol ist eine Linie mit einer Raute am Anfang und repräsentiert einen etwas komplexeren Sachverhalt. Das UML Diagramm verdeutlicht die früher bereits erwähnte Tatsache, daß Texte lediglich einen Teil der in einem Computer verarbeitbaren Daten ausmachen. Da das MVC Muster sich nicht nur auf Texte, sondern ebensogut auf andere Daten anwenden läßt, liegt zwischen der Ebene mit den Klassen *Models.Model*, *Views.View*, *Controllers.Controller* und der Ebene mit den entsprechenden Textklassen eine weitere mit den Klassen *Containers.Model*, *Containers.View* und *Containers.Controller*, in denen fast alle für die Modellierung, Darstellung und Kontrolle beliebiger Daten benötigten Eigen

[12] Eine Zusammenstellung der relevanten UML-Diagramme finden Sie im Anhang D.

schaften realisiert sind, die Textebene enthält lediglich darüber hinausgehende textspezifische Ergänzungen. Diese Klassenhierarchie findet sich im erweiterten Schnittstellenauszug des Moduls *TextControllers* wieder

```
DEFINITION TextControllers;

    IMPORT Containers, TextModels, TextViews;

    TYPE
        Controller = POINTER TO ABSTRACT RECORD
                                    (Containers.Controller(Controllers.Controller(Stores.Store)))
            text-: TextModels.Model;
            view-: TextViews.View;
            (c: Controller) GetSelection (OUT beg, end: INTEGER), NEW, ABSTRACT;
            (c: Containers.Controller) HasSelection (): BOOLEAN, NEW, EXTENSIBLE
        END;

    PROCEDURE Focus (): Controller

END TextControllers.
```

Auszug der erweiterten Schnittstelle des Moduls*TextControllers*

In dieser Schnittstelle ist die Klasse *TextControllers.Controller* als Unterklasse von *Containers.Controller* aufgeführt, das Modul *TextControllers* muß deshalb das Modul *Containers* importieren. Die Klasse *TextControllers.Controller* enthält nun nicht nur Methoden, sondern zwei weitere Felder, *text: TextModels.Model* und *view: TextViews.View*, die Importliste des Moduls *TextControllers* muß also auch die beiden exportierenden Module *TextModels* und *TextViews* enthalten. Die Beziehung der Klasse *TextControllers.Controller* zu den Klassen *TextModels.Model* und *TextViews.View* ist ebenso wie die Benutzt-Beziehung keine Vererbungsbeziehung, beide Klassen sind in Form der Felder *text: TextModels.Model* und *view: TextViews.View* in der Klasse *TextControllers.Controller* enthalten. Diese Art der Klassenbeziehung wird als Klassenkomposition (class composition) oder auch Klassenaggregation (class aggregation) bezeichnet und durch eine Linie mit Raute auf der Seite der enthaltenden Klasse repräsentiert.

Wie die Schnittstellenauszüge und die Abbildung 13 zeigen, sind alle Klassen des MVC Musters Unterklassen der Klasse *Stores.Store*, die im Sinn der Polymorphie alle Objekte ihrer Unterklassen kennt und sie verarbeiten kann. Die Klasse *Stores.Store* stellt im BlackBox System die gemeinsame Basis für alle auf externen Medien speicherbaren Daten dar, sie sorgt dafür, daß beliebig komplexe, ineinandergeschachtelte Dokumente so in ein externes Medium geschrieben werden, daß ihre Datenstruktur beim erneuten Einlesen vollständig rekonstruiert werden kann.

Abbildung 13 zeigt auch, daß sich Unterklassen von *Stores.Store* nicht generell kennen. Weder kennt die Klasse *TextModels.Model* die Klasse *TextControllers.Controller* noch die Klasse *TextViews.View*. Ein Modell kann deshalb seine Veränderungen den zu ihm gehörenden Sichten nur über anonyme Botschaften (message broadcasts) mitteilen, jede Sicht weiß selbst, ob und wie sie auf die jeweilige Botschaft reagieren

muß. Dagegen kennt die Klasse *TextViews.View* sowohl die Klasse *TextModels.Model* als auch die Klasse *TextControllers.Controller*, das zu einer Sicht gehörende Modell läßt sich ebenso wie das der Sicht zugeordnete Kontrollobjekt über die Methoden *ThisModel* und *ThisController* ermitteln. Die Klasse *TextControllers.Controller* kennt ihrerseits die beiden Klasse*n TextModels.Model* und *TextViews.View*, ihre Kenntnis dieser Klassen ist allerdings von anderer Art, ein Kontrollobjekt enthält das zu ihm gehörende Modell und die mit ihm verbundene Sicht als aggregierte Objekte.

```
MODULE TutScanForm;                         (*   Das MVC Muster   *)

 IMPORT
   TextModels, TextControllers, TextMappers, TextViews, Views;
(*  ------------------------------------------  Summieren  *)
 PROCEDURE Summieren*;
   VAR➜ ←
     m1: TextModels.Model;➜ ← ➜ ←
     c: TextControllers.Controller;➜ ←
     s: TextMappers.Scanner;
     f0, f1: TextMappers.Formatter;➜ ←
     beg, end: INTEGER;
     Summe: REAL;
 BEGIN
   Summe := 0.0;
   c := TextControllers.Focus();➜ ←
   IF (c # NIL) & c.HasSelection() THEN➜ ← ➜ ←
     s.ConnectTo(c.text);➜ ←
     c.GetSelection(beg, end);➜ ←
     s.SetPos(beg);
     s.Scan;
     WHILE (s.type = TextMappers.real) & (s.Pos() <= end + 1) DO
       Summe := Summe + s.real;
       s.Scan;
     END;➜ ← ➜ ←
     f0.ConnectTo(c.text);
     f0.SetPos(c.text.Length());➜ ←
     f0.WriteRealForm(Summe, 5, 7, 0, 8FX);➜ ←
     m1 := TextModels.dir.New();➜ ←
     f1.ConnectTo(m1);➜ ←
     f1.WriteString("Die Summe der markierten Zahlen ist: ");
     f1.WriteRealForm(Summe, 5, 7, 0, 8FX);➜ ← ➜ ←
     Views.OpenView(TextViews.dir.New(m1));➜ ←
   END;
 END Summieren;

END TutScanForm.
```

Modul *TutScanForm* mit verborgenen Textfaltungen

Bevor die Einzelheiten des Programms ScanForm.odc erläutert werden, möchte ich Sie mit einer in diesem Programm verwendeten nützlichen Fähigkeit des BlackBox Systems bekannt machen, den Textfaltungen. Im Menü *Tools* finden Sie die Einträge *Create Fold, Expand All, Collapse All* und *Folds...*, mit denen Sie Textfaltungen erzeugen und bearbeiten können. Wenn Sie das Modul *TutScanForm* in den

BlackBox Editor laden, werden Sie innerhalb des Programmcodes entweder ausgefüllte schwarze oder leere weiße Pfeile bemerken. Diese Pfeile sind stets paarig, jedes Paar besteht aus einem nach rechts weisenden Pfeil, der den Anfang und einem nach links weisenden Pfeil, der das Ende einer Textfaltung markiert. Die Bezeichnung Faltung ist dabei eigentlich nicht ganz exakt, es handelt sich bei gefaltetem Text vielmehr um Alternativen; schwarze Pfeile signalisieren die eine, weiße die andere Variante. Ist eine der Varianten leer, erscheint die andere als verborgen - gefaltet. Umgekehrt kennzeichnet in diesem Fall die zweite Pfeilart Textpassagen, die gefaltet werden können. Klicken Sie auf einen Anfangs- oder Endpfeil, wird die Faltung in ihr Gegenteil verkehrt. Unabhängig vom aktuellen Zustand einzelner Faltungen lassen sich mit dem Menüeintrag *Expand All* alle Faltungen eines Textes gleichzeitig öffnen, ebenso schließt der Eintrag *Collapse All* alle Faltungen gleichzeitig (der Menüpunkt *Folds...* bietet weitere, mit Textfaltungen verbundene Möglichkeiten, s. dazu die Dokumentation zum BlackBox Systemmodul *StdFolds* im Hilfesystem).

Im Modul *TutScanForm* enthalten die Textfaltungen Erklärungen der einzelnen Programmschritte. Wenn Sie mit *Tools* → *Expand All* die Faltungen des Programms öffnen, finden Sie unter einem einzelnen oder mehreren zusammengehörenden Programmschritten diese Erklärungen als Kommentare, umgekehrt erhalten Sie mit *Tools* → *Collapse All* den reinen Quelltext des Programms, wie er vorstehend abgedruckt ist und wie ihn der Compiler sieht.

Zusammen mit den kommentierenden Zeilen im Programmtext des Moduls *TutScanForm* sollten die folgenden Erläuterungen ausreichen, um die Arbeitsweise des Programms zu verstehen. Die Prozedur *Summieren* liest im aktuellen Textfenster einen Block markierter Dezimalzahlen und gibt anschließend deren Summe einmal am Ende des aktuellen Fensters und ein zweites Mal in einem neuen Fenster aus.

Das Modul *TutScanForm* enthält zusätzlich zu dem Ihnen bereits aus dem Modul *TutKonvert4* bekannten Objekt *m1: TextModels.Model* ein Kontrollobjekt *c: TextControllers.Controller*, das entsprechend den vorangegangenen Darlegungen zwischen Benutzer, Textmodell und Textsicht vermittelt. Ähnlich wie in *TutKonvert5* gibt es keine explizit deklarierte Sicht, das Objekt *v1: TextViews.View* steht in Kommentarklammern und wird nicht direkt benötigt, es dient wie die analogen Objekte in *TutKonvert5* nur zur Verdeutlichung einiger nachfolgender Programmschritte.

Während Ihnen die Deklaration der Formatierungsobjekte *f0, f1: TextMappers.Formatter*, mit denen Texte in ein Modell geschrieben werden, bekannt ist, finden Sie in der anschließenden Zeile mit *s: TextMappers.Scanner* das dazu passende Gegenstück, ein Objekt, mit dem vorhandener Text gelesen werden kann (s. dazu auch den Schnittstellenauszug des Moduls *TextMappers* im Abschnitt 9.3). Objekte der Klasse *TextMappers.Scanner* sind wie Objekte der Klasse *TextMappers.Formatter* Vertreter des Iterator-Musters.

Im Anweisungsteil von *Summieren* finden Sie die Zeile *c := TextControllers.Focus()*, in der das Kontrollobjekt *c* mit dem Textmodell des aktiven Fensters verbunden wird. Die anschließende *IF*-Abfrage

enthält zwei Teile, *c # NIL* und *c.HasSelection()*. Die Funktion des zweiten Teils sollte mit der darunter stehenden Erläuterung klar sein, während Sie mit dem Component Pascal Schlüsselwort *NIL* in der Boole'schen Bedingung einer Neuigkeit begegnen, die Sie vollständig erst im Zusammenhang mit den im nächsten Kapitel vorgestellten Zeigern verstehen werden. Kurz gesagt wird in dieser Bedingung geprüft, ob der vorangegangene Versuch, das Kontrollobjekt *c* mit dem aktuell aktiven Fenster zu verbinden, erfolgreich war, anderenfalls enthält das Objekt *c* den speziellen Wert *NIL*. Dies kann verschiedene Ursachen haben, beispielsweise kann das aktive Fenster nicht ein Text- sondern ein Graphikfenster sein, oder es gibt kein geöffnetes Fenster auf dem Bildschirm.

In Verbindung mit der Schnittstelle des Moduls *TextMappers* dürften Ihnen die weiteren Programmschritte keine Verständnisschwierigkeiten bereiten, das Erschreckendste könnte die Parameterliste in dem Aufruf *f1.WriteRealForm(Summe, 5, 7, 0, 8FX)* sein. Der Vergleich mit den formalen Parametern der zur Klasse *TextMappers.Formatter* gehörenden Methode *(VAR f: Formatter) WriteRealForm (x: REAL; precision, minW, expW: INTEGER; fillCh: CHAR), NEW* hilft Ihnen vielleicht etwas weiter, für die Einzelheiten muß ich Sie bitten, die Dokumentation des Moduls *TextMappers* im Hilfesystem zu studieren, eine unvermeidliche Notwendigkeit, die bei Ihrer Arbeit in Zukunft immer häufiger auftauchen wird.

Als Letztes möchte ich auf die Anweisung *Views.OpenView(TextViews.dir.New(m1))* eingehen. Der Vergleich mit den beiden darüber stehenden Zeilen und mit den entsprechenden Anweisungen in den Prozeduren *Anleitung* der Module *TutKonvert4* und *TutKonvert5* zeigt noch einmal, daß die Deklaration eines separaten Objekts für die Darstellung des Ergebnisses unnötig ist, die beiden Einzelanweisungen für die Allozierung der Sicht und deren Darstellung lassen sich in einer Anweisung zusammenfassen.