ISA Dialog Manager

PROGRAMMIERTECHNIKEN

A.06.03.b

Dieses Handbuch vermittelt grundlegende Techniken für die Entwicklung von Benutzerschnittstellen mit dem ISA Dialog Manager. Zu den Themen gehören Modularisierung, Verwendung von Modellen, objektorientierte Programmierung und das Datenmodell.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Deutschland

Darstellungskonventionen

DM wird in diesem Handbuch synonym zu "Dialog Manager" verwendet.

Die Bezeichnung UNIX schließt generell alle unterstützten UNIX-Derivate ein - außer in den explizit angegebenen Fällen.

Dort wo für geläufige englische Fachbegriffe keine gängigen deutschen Übersetzungen existieren, wird zur Vermeidung von Unklarheiten der englische Begriff verwendet.

< >	muss durch einen entsprechenden Wert ersetzt werden
color	Schlüsselwort ("keyword")
.bgc	Attribut
{ }	optional (0 oder einmal)
[ ]	optional (0 oder n-mal)
<A> | <B>	entweder <A> oder <B>

Beschreibungsmodus

Alle Schlüsselwörter sind fett und unterstrichen, z.B.

variable **integer** **function**

Indizierung von Attributen

Syntax für indizierte Attribute:

[I]

[I,J] bzw. [row,column]

Identifikatoren

Identifikatoren müssen mit einem Großbuchstaben oder einem "Unterstrich" ('_') beginnen. Die weiteren Zeichen können Groß-, Kleinbuchstaben, Zahlen oder Unterstriche sein.

Der Bindestrich ('-') ist für die Benennung von Identifikatoren als Zeichen ***nicht*** zugelassen!

Die maximale Länge eines Identifikators beträgt *31* Zeichen.

Beschreibung der zugelassenen Identifikatoren in Backus-Naur-Form

<Identifikator> ::= <erstes Zeichen>{<Zeichen>}

<erstes Zeichen> ::= _ | <Großbuchstabe>

<Zeichen> ::= _ | <Kleinbuchstabe> | <Großbuchstabe> | <Ziffer>

<Ziffer> ::= 1 | 2 | 3 | … 9 | 0

<Kleinbuchstabe> ::= a | b | c | … x | y | z

<Großbuchstabe> ::= A | B | C | … X | Y | Z

Inhalt

1 Namenskonventionen

Wie auch in anderen Programmiersprachen üblich, kann die Verwendung von Namenskonventionen bei der Erstellung eines Dialogs die Lesbarkeit wesentlich erhöhen. Durch eine eindeutige Namensgebung werden die Mitentwickler in die Lage versetzt, mit wenig Aufwand fremde Dialogteile zu lesen, ohne dass sie nach allen verwendeten Elementen im Dialogskript suchen müssen. Aus diesem Grund sollte, bevor ein größeres Projekt gestartet wird, eine solche Namensgebung definiert werden.

Die im Folgenden vorgestellte Namensgebung kann nur als ein Grundgerüst oder als ein Vorschlag angesehen werden; sie muss in einzelnen Projekten an die dortigen Gegebenheiten angepasst werden.

1.1 Ziel der Namensgebung

Mit Hilfe der Namensgebung soll erreicht werden, dass dem Leser in möglichst wenigen Zeichen möglichst viel Information vermittelt wird. Dem Namen sollte man ansehen, um welche Art von Objekt es sich handelt, in welchem Modul das entsprechende Objekt definiert ist und welche Funktion es hat. Zusätzlich sollte man bemüht sein, die Anzahl der Namen so gering wie möglich zu halten und nur den Objekten einen Namen zu geben, die in den Regeln angesprochen werden und keinen Namen von ihrem Modell erben können. Es macht in der Regel keinen Sinn, einen vom Modell geerbten Namen durch einen neuen zu überschreiben. Das erhöht nur die Anzahl der Namen und trägt auf keinen Fall zur besseren Lesbarkeit des Dialogs bei.

Als besonders wichtig stellen sich bei der Verwendung von Objekten immer wieder die Klasse und die Art (Modell oder Instanz) heraus. Aus diesem Grund sollte diese Information in den ersten Buchstaben enthalten sein. Da die Instanz im Normalfall die am häufigsten verwendete Objektart ist, sollte diese nicht durch einen besonderen Buchstaben gekennzeichnet sein, sondern nur die Modelle. Anschließend sollte am Kennzeichen, das anzeigt, ob es sich um ein Modell oder eine Instanz handelt, die Klasse des Objektes angegeben werden. Falls Module verwendet werden, sollte dann das Modul - abgetrennt durch einen Underline - angegeben werden, in welchem das entsprechende Objekt definiert ist. Wieder durch einen Underline zur besseren Lesbarkeit abgetrennt, sollte dann der eigentliche Name des Objektes folgen. Aus diesem Namen sollte hervorgehen, welche Aufgabe das entsprechende Objekt hat.

Damit ergibt sich folgendes Schema für die Namensvergabe:

Definition

```
{M}<Abkürzung Objektklasse>{_<ModulName>}_NameDesObjektes
```

Der Modulname sollte nur verwendet werden, wenn auch wirklich mit Modulen gearbeitet wird. Sonst entfällt dieser Teil des Namens. Bei der Namensvergabe muss darauf geachtet werden, dass die Länge eines Namens 31 Zeichen nicht überschreiten darf.

1.2 Abkürzungen für die einzelnen Objektklassen

In diesem Kapitel werden die Abkürzungen für die einzelnen Objektklassen definiert.

Objekttyp	Abkürzung für die Namenskonvention
Bild	Im
Canvas	Cn
Checkbox, Tristate-Pushbutton	Cb
Editierbarer Text	Et
Fenster	Wn
Groupbox	Gb
Listbox	Lb
Menübox	Mb
Menuitem	Mi
Menuseparator	Ms
Messagebox	Mx
Notebook	Nb
Notepage	Np
Poptext	Pt
Pushbutton	Pb
Radiobutton	Rb
Rechteck	Re
Scrollbar	Sc
Statusbar	Sb
Statischer Text	St
Tablefield	Tf

Objekttyp	Abkürzung für die Namenskonvention
Application	Ap
Record	Rc
Timer	Ti
Accelerator	Ac
Cursor	Cu
Farbe	Cl
Format	Fm
Muster	Ti
Textressource	Tx
Zeichensatz	Fn
Funktion	Fc
globale Variable	V
Regel	Rl
Benutzerdefiniertes Attribut	A

Bei den benutzerdefinierten Attributen und den globalen Variablen kann anschließend noch eine Bezeichnung für den Datentyp folgen, so dass man dem Namen direkt den zugehörigen Datentyp ansehen kann.

1.3 Abkürzungen für Datentypen

Die Datentypen können wie folgt abgekürzt werden:

Datentyp	Abkürzung für die Namenskonvention
anyvalue	Av
attribute	At

Datentyp	Abkürzung für die Namenskonvention
boolean	Bo
class	Cl
datatype	Dt
enum	En
event	Ev
index	Ix
integer	In
method	Mt
object	Ob
pointer	Pt
string	St

1.4 Beispiele für die Namensvergabe

In diesem Kapitel werden Beispiele aufgeführt, wie aus den beschriebenen Konventionen Namen für Objekte gebildet werden können.

Objektbeschreibung	Objektname
Modell eines Tablefields	MTf_Uebersicht
Modell eines OK-Pushbuttons	MPb_OK
Globale Objekt-Variable	VOb_Aktuellesfenster
Objekt-Attribut	AOb_InitFenster
Tristate-Pushbutton	Cb_Schalter

2 Einsatz von Modellen

In diesem Kapitel wird beschrieben, wie die Vererbung von Attributen intern funktioniert. Dadurch erhalten Sie Gestaltungshinweise, wie Sie mit Hilfe des Dialog Managers einen Baukasten aufbauen können, aus dem Sie immer wieder Objekte verwenden können.

2.1 Objektarten

Die auf dem Bildschirm für den Benutzer sichtbaren Attribute können auf verschiedenen Objektdefinitionsebenen definiert werden.

Die verschiedenen Objekttypen sind:

- DM-interne Defaultwerte für einzelne Attribute
- Default
- Vorlage bzw. Modell (verschiedene Schichten, Anzahl ist beliebig)
- Instanz

Diese unterschiedlichen Arten von Objekten werden in den nachfolgenden Kapiteln genauer vorgestellt.

2.1.1 Defaultobjekte

Zu jeder im Dialog verwendeten Objektklasse sollte ein Defaultobjekt definiert werden. Durch die Verwendung dieser Defaultobjekte können einzelne Objektattribute einmalig global definiert werden. D.h. alle nachfolgend deklarierten Objekte derselben Objektklasse enthalten implizit die in der Objektvorlage definierten Attribute, sofern diese nicht lokal umdefiniert werden. Mit diesem Hilfsmittel sind also nur noch zusätzliche Attribute zu deklarieren oder solche, die vom Default abweichen.

Die Deklaration einer Objektvorlage beginnt mit dem Schlüsselwort **default**, anschließend steht die Klasse des Objektes, gefolgt von der in geschweiften Klammern stehenden Objektdefinition.

Definition

```
{ export | reexport } default <Objektklasse> { <Bezeichner> }
{
  <Attribute>
}
```

Innerhalb der Definition der Defaults können alle zu der Objektklasse gehörenden Attribute verwendet werden. Zusätzlich können selbstverständlich auch benutzerdefinierte Attribute dem Defaultobjekt hinzugefügt werden.

Beispiel

```
default window
```

```
{
    .sensitive true;
    .visible  true;
    .fgc Medium_Gray;
    .bgc White;
    .titlebar true;
    .closeable true;
    .sizeable true;
    object MeinZusaetzlichesAttribut := null;
}
default pushbutton
{
    .sensitive true;
    .visible true;
    .fgc Medium_Gray;
    .bgc White;
    .width 80;
    .height 30;
    .xauto 1;
    .yauto 1;
}
```

Bei der späteren Deklaration eines Pushbuttons muss dann nur noch die Position der linken oberen Ecke und die Beschriftung angegeben werden.

D.h. es genügt dann:

```
pushbutton OKAY
{
    .xleft 10;
    .ytop 10;
    .text "OKAY";
}
```

Im Gegensatz zu allen anderen Objekten können Defaultobjekte keine Kinder haben. Wenn dem Defaultobjekt, wie üblich, kein Name gegeben wird, wird es über den Namen seiner Klassen in Großbuchstaben angesprochen; mit PUSHBUTTON ist das Defaultobjekt der Pushbuttons gemeint, mit WINDOW das Defaultobjekt aller Fenster.

Die nachfolgende Tabelle zeigt alle Namen der Defaultobjekte.

Tabelle 1: *Namen der Defaultobjekte*

Objektklasse	Name des Defaultobjektes
Bild	IMAGE
Canvas	CANVAS

Objektklasse	Name des Defaultobjektes
Checkbox, Tristate-Button	CHECKBOX
editierbarer Text	EDITTEXT
Fenster	WINDOW
Groupbox	GROUPBOX
Poptext (Combobox)	POPTEXT
Menubox	MENUBOX
Menueintrag	MENUITEM
Menuseparator	MENUSEP
Messagebox	MESSAGEBOX
Notebook	NOTEBOOK
Notepage	NOTEPAGE
Pushbutton	PUSHBUTTON
Radiobutton	RADIOBUTTON
Rechteck	RECTANGLE
Scrollbar	SCROLLBAR
statischer Text	STATICTEXT
Statusbar	STATUSBAR
Tablefield	TABLEFIELD
Timer	TIMER

2.1.2 Modelle

Ein weiteres Hilfsmittel zur Objekterzeugung sind die Vorlagen. Mit ihnen kann ein benanntes Vorlagenobjekt definiert werden, das zur Definition des realen Objektes herangezogen wird.

Dieses Hilfsmittel ist vor allem dann nützlich, wenn eine größere Anzahl gleichartiger Objekte erzeugt werden soll (z.B. mehrere OK-Knöpfe oder mehrere editierbare Texte, in denen der Benutzer jeweils eine Artikelnummer eingeben kann). Das Objekt, das sich auf eine Vorlage bezieht, erbt von dieser alle Attribute, die es **nicht lokal** neu definiert.

Definition

```
{ export | reexport } model <Objektklasse> <Bezeichner>
{
  <Attribute>
}
```

Innerhalb der Definition der Vorlagen können alle zu der Objektklasse gehörenden Attribute verwendet werden. Solche Vorlagen können dann als Kinder von anderen Objekten oder Vorlagen dienen. Zusätzlich können einer solchen Vorlage auch Kinder gegeben werden, die dann automatisch an alle davon abgeleiteten Instanzen vererbt werden. Dadurch können also komplexe hierarchische Modelle aufgebaut werden, die im aktuellen Dialog immer wieder verwendet werden sollen.

Die Definition solcher hierarchischer Vorlagen sieht dann wie folgt aus:

```
{ export | reexport } model <Objektklasse> <Bezeichner>
{
  { export | reexport } { child } <Objektklasse> { <Bezeichner> }

{
    <Attribute>
  }
}
```

Um den Einsatz von Vorlagen möglichst effizient zu gestalten, sollte man sich die Mühe machen, komplexe Modelle zu erstellen, wenn im Dialog gewisse Strukturen immer wieder auftauchen. Dadurch werden der Wartungsaufwand und der Aufwand bei Änderungen wesentlich verringert.

Wenn nun eine Vorlage für die Definition eines Objektes genutzt werden soll, muss das Objekt wie folgt definiert werden:

```
<Modellbezeichner> { <Bezeichner> }
{
  <Attribute>
}
```

Beispiel: Eingabefeld für Artikelnummer

In einer Anwendung sollen in verschiedenen Fenstern Eingabefelder definiert werden, in denen eine Artikelnummer eingetragen werden kann. Im Normalfall ist für eine Anwendung das Schema, wie eine Artikelnummer aussieht, überall gleich. Aus diesem Grund sollte man hierfür also eine Vorlage schaffen und diese dann in den entsprechenden Fenstern verwenden. Wenn sich dann später das Schema für die Artikelnummer ändert, muss nur eine Stelle im Dialog, nämlich die Vorlage, verändert werden.

```
model edittext MEtArtikelNummer
{
  .posraster true;
  .sizeraster true;
  .format "AA.NN.NN/NN";
```

```
  .width 12;
}
```

Diese Vorlage kann dann in verschiedenen Fenstern verwendet und dabei in ihren Attributen verändert werden.

```
window WnFenster1
{
    child MEtArkitelNummer
    {
        .ytop 2;
        .xleft 2;
    }
}

window WnFenster2
{
    child MEtArtikelNummer EtErsterArtikel
    {
        .ytop 1;
        .xleft 10;
    }
    child MEtArtikelNummer EtZweiterArtikel
    {
        .ytop 2;
        .xleft 10;
    }
}
```

Im ersten Fenster WnFenster1 wird die ArtikelNummer verwendet, ohne dass ihr dabei ein neuer Name gegeben wird. Die von der Vorlage abgeleitete Instanz kann daher über WnFenster1.MEtArtikelNummer angesprochen werden. Im zweiten Fenster WnFenster2 wird die Vorlage gleich an zwei Stellen benutzt. Hier sollten den Objekten neue Namen gegeben werden, damit diese eindeutig angesprochen werden können.

Beispiel: Fenster mit Abbrechen und OK-Pushbutton

In einem Dialog sollen mehrere Fenster angelegt werden, in denen Daten eingegeben werden können. Diese Fenster sollten daher mit zwei Pushbuttons ausgestattet werden. Der erste Pushbutton "OK" übernimmt die Veränderungen des Benutzers in das Programm, der "Abbrechen" Pushbutton verwirft alle Aktionen des Benutzers. Damit das entsprechende System komfortabel aufgebaut werden kann, wird zunächst ein Modell eines Pushbuttons "MPbButton" angelegt, bei dem nur die Fixierung des Pushbuttons an der unteren Kante des Vaters definiert wird. Davon werden dann die eigentlichen Pushbuttons "MPbOK" und "MPbAbbrechen" ebenfalls als Vorlage abgeleitet. Anschließend wird ein Modell eines Fensters definiert, das Ableitungen der Pushbuttons "MPbOK" und "MPbAbbrechen" beinhaltet. Abschließend wird dann von diesem Fenster eine Instanz gebildet. Die Instanz hat dann folgendes Aussehen:

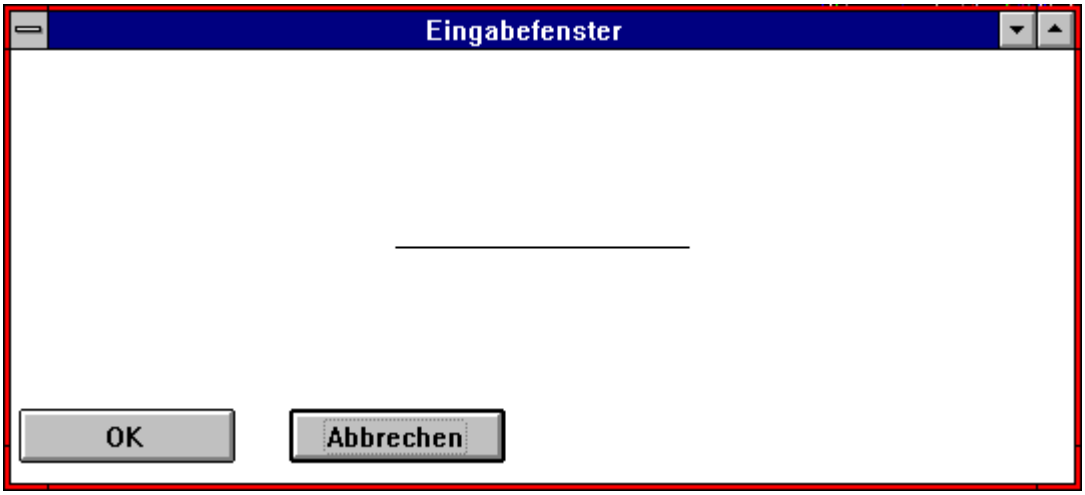

Abbildung 1: *Fenster mit zwei Pushbuttons*

Das zugehörige Dialogskript sieht wie folgt aus:

```
model pushbutton MPbButton
{
  .yauto -1;
}
model MPbButton MPbAbbrechen
{
  .xleft 15;
  .text "Abbrechen";
}
model pushbutton MPbOK
{
  .yauto -1;
  .text "OK";
}
model window MWnEingabeFenster
{
  .title "Eingabefenster";
  child MPbAbbrechen
  {
  }
  child MPbOK
  {
  }
}
MWnEingabeFenster WnAddresse
{
}
```

In dem Fenster "WnAdresse" können selbstverständlich beliebig viele weitere Kinder definiert werden.

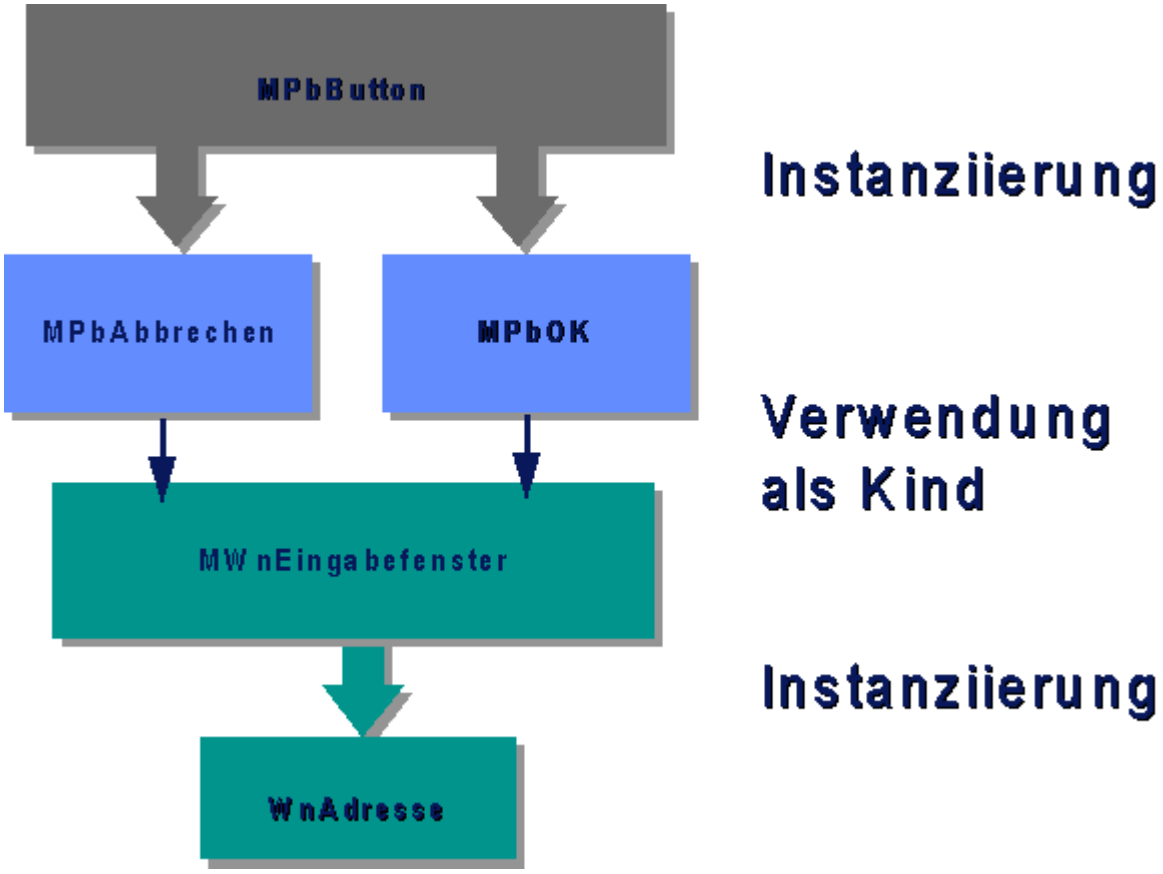

Abbildung 2: *Darstellung der Modellhierarchie*

2.1.2.1 Problematische Modellhierarchie

Die folgende Modellhierarchie ist zwar möglich, wird aber ausdrücklich nicht empfohlen, da sie dem Charakter einer Modellhierarchie eigentlich widerspricht und in größeren Dialogen zu Verständnisproblemen führen kann.

Folgende Definition bildet ein Instanzenfenster WnInstanz mit zwei von Et1 abgeleiteten Editierfeldern (edittext) und einem weiteren, unabhängigen Fenster, welches ebenfalls eine Instanz von Et1 enthält:

```
model window MWnModel
{
child edittext Et1
  {
    .xleft 16;
    .ytop 2;
  }
}
MWnModel WnInstanz
{
  child Et1 NeuerEt
  {
```

```
    .xleft 30;
  }
}
window AnderesFenster
{
  child Et1 Eingabe
  {
    .xleft 10;
    .ytop 5;
    .width 20;
  }
}
```

Die hier skizzierte Vererbung ist auch über Modulgrenzen hinweg möglich, allerdings müssen dafür die entsprechenden Objekte mit **export** oder **reexport** freigegeben werden.

In diesen Fällen ist die getrennte Definition eines Modells für Et1 besser und auch leichter verständlich.

2.1.3 Instanz

Instanzen sind die Objekte, die auf dem Bildschirm sichtbar gemacht werden können. Sie können von einem Modell abgeleitet sein und neben den vom Modell geerbten Kindern eine beliebige Anzahl weitere Kinder definiert haben. Bei der Instanz können alle Attributwerte überschrieben werden und auch die Attribute von geerbten Kindern verändert werden. Im Modell definierte Kindern müssen dabei übernommen werden, sie können nicht in der Instanz gelöscht werden.

2.2 Vererbung von Attributen

Bei einem Objekt werden Attributwerte immer vom zugrundeliegenden Modell geerbt. Diese Attributwerte können dann bei dem Objekt durch neue Werte überschrieben werden. Damit nimmt die Bedeutung von den Attributen von Werten im Default zu Werten im Modell zu Werten in der Instanz immer mehr zu. D.h. wenn z.B. eine Instanz eine bestimmte Farbe haben soll, wird die Farbe bei der entsprechenden Instanz definiert. Damit erhält die Instanz die entsprechende Farbe, unabhängig davon, was in den zugrundeliegenden Modellen definiert worden ist.

Oberhalb der im Dialog definierten Defaults gibt es noch Dialog Manager Defaultwerte, die Werte für einzelne Attribute darstellen, mit denen der Dialog Manager intern arbeiten kann. Auf diese Werte sollte man sich bei der Dialogprogrammierung auf keinen Fall verlassen, denn diese können immer wieder angepasst werden. Zusätzlich gibt es für bestimmte Attribute noch Defaultwerte im Fenstersystem, die dann zum Tragen kommen, wenn weder bei der Instanz, noch beim Modell oder dem Default ein Wert für das Attribut gesetzt worden ist. Dieses ist vor allem bei Layoutattributen wie der Hintergrundfarbe oder dem Zeichensatz sinnvoll, da hierdurch der Benutzer des Systems durch Systemeinstellungen die entsprechenden Werte verändern kann. Im nachfolgenden Schaubild wird die Hierarchie dieser Attributwerte dargestellt.

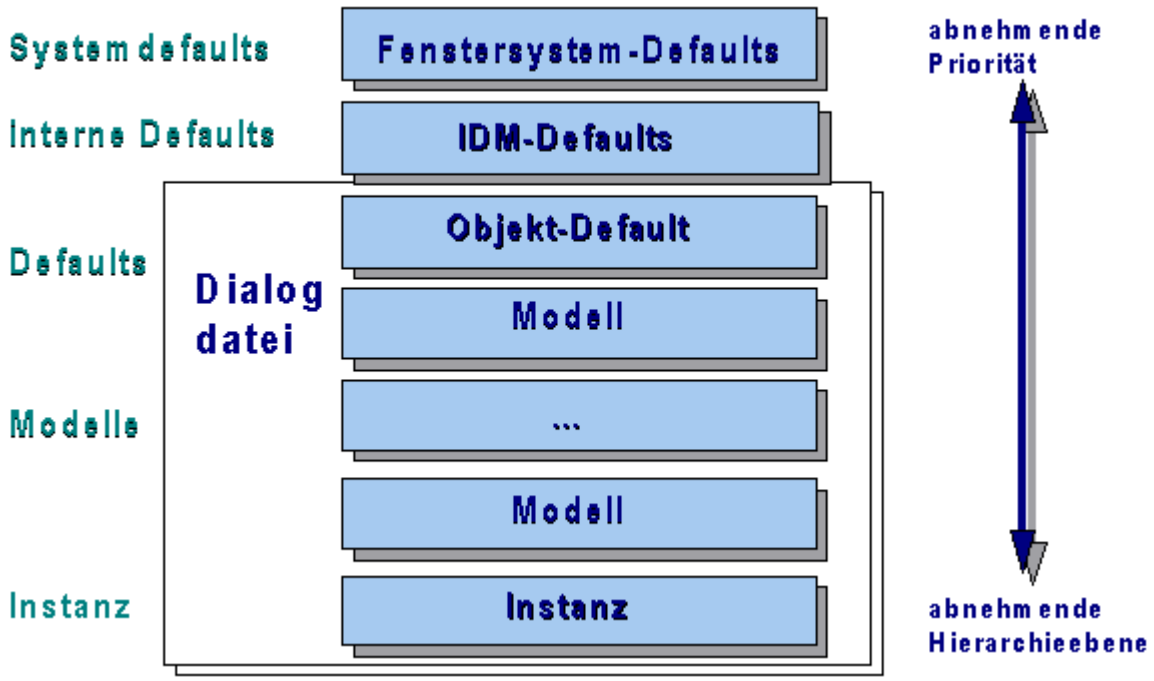

Abbildung 3: *Vererbungspriorität*

Wenn nun eine Instanz auf dem Bildschirm sichtbar gemacht wird, wird ein Objekt gebildet, das die Attribute der Instanz und der zugrundeliegenden Modelle beinhaltet. Dazu wird die Hierarchie immer so weit durchsucht, wie es unbedingt notwendig ist. Sobald für ein Attribut ein Wert definiert ist, wird die Suche für dieses Attribut abgebrochen und der gefundene Wert übernommen. In der nachfolgenden Tabelle sollen die Spalten die einzelnen Attribute darstellen, in den Zeilen werden die Werte der verwendeten Defaults, Modelle und Instanzen angezeigt. In der letzten Zeile werden die Attributwerte des angezeigten Objektes dargestellt.

Tabelle 2: *Berechnung der Attributwerte*

	Attribut 1	Attribut 2	Attribut 3	Attribut 4	Attribut 5
Fenstersystem	FS	FS			FS
interner Default			ID	ID	ID
Modell 1	M1				
Modell 2			M2		
Modell 3					M3
Instanz				I	
angezeigter Wert	M1	FS	M2	I	M3

2.3 Besonderheiten bei der Vererbung von Attributen

Neben der bereits geschilderten Vererbung von Attributen von der Vorlage auf die Instanz gibt es für die Attribute *.visible* und *.sensitive* noch eine spezielle Art der Vererbung vom Vater auf die Kinder. Wenn eines dieser Attribute beim Vater mit dem Wert *false* belegt ist, sind auch automatisch die Werte bei den Kindern mit diesem Wert belegt. Das kommt daher, dass z.B. ein Pushbutton nur dann sichtbar sein kann, wenn das Fenster, in dem er definiert ist, auch sichtbar ist. Ist es nicht sichtbar, ist auch der Pushbutton automatisch nicht sichtbar. Dasselbe gilt für das Verhalten der Selektierbarkeit.

Um also zu bestimmen, ob der Pushbutton sichtbar ist, müsste man den Pushbutton und alle seine Väter fragen, ob sie sichtbar sind. Nur wenn alle sichtbar sind, ist der Pushbutton auch wirklich sichtbar. Um dies zu vereinfachen, kann man das Objekt nach dem Attribut *.real_visible* bzw. *.real_sensitive* fragen. Mit einer Abfrage erkennt dann das Programm, ob das Objekt wirklich sichtbar bzw. wirklich selektierbar ist.

Die nachfolgende Tabelle zeigt das Verhalten des Attributes *.visible*. Gleiches gilt für das Attribut *.sensitive*.

Tabelle 3: *Vererbung der Attribute .visible und .sensitive*

Vater.visible	Kind.visible	Vater.real_visible	Kind.real_visible
true	true	true	true
false	true	false	false
true	false	true	false
false	false	false	false

2.4 Ausnahmen bei der Vererbung von Attributen

Intern im Dialog Manager werden so gut wie alle setzbaren Attribute vom Modell auf die Instanz vererbt. Die Ausnahmen werden hier kurz vorgestellt:

Attribut	Ursache
.content einer ***Listbox***	Im Normalfall wird der Inhalt einer Listbox dynamisch von der Anwendung gefüllt und daher nicht in der Hierarchie vererbt.
export bei allen Objekten	Das Kennzeichnen eines Objektes zum Exportieren muss bei jeder Instanziierung eines Objektes wiederholt werden, wenn auch die Instanz exportiert werden soll.
reexport bei allen Objekten	Wird nur bei den Kindobjekten von Modellen vererbt. Am Vaterobjekt muss *reexport* bei jedem abgeleiteten Objekt (Modell oder Instanz) angegeben werden, wenn es exportiert werden soll.

2.5 Auswirkungen auf die Regelabarbeitung

Wenn ein Ereignis auftritt, beginnt der DM, nach für das Ereignis passenden Regeln zu suchen. Dabei wird bei dem zugehörigen Default des Objekts angefangen, nach Regeln mit "before" zu suchen. Vor dort aus wird bei dem oder den zu dem Objekt gehörenden Modellen nach diesen Regeln gesucht. Abschließend wird beim Objekt selbst überprüft, ob eine Regel mit "before" definiert worden ist. Dabei werden alle gefundenen Regeln ausgeführt.

Danach beginnt der DM bei dem Objekt nach einer "normalen" Regel zu suchen und geht - falls er bei dem Objekt keine findet - zu dem zugehörigen Modell. Falls aber an dem Objekt eine Regel für das eingetroffene Ereignis definiert worden ist, geht er nicht mehr zu dem Modell. Bei dem Modell überprüft er, ob hier eine passende Regel definiert ist. Ist das nicht der Fall, geht er zum zugehörigen Modell des Modells bzw. zum Default. Falls jedoch eine Regel am Modell vorhanden ist, wird diese ausgeführt und die Suche abgebrochen.

Abschließend beginnt der DM wieder bei dem Objekt, nach passenden Regeln für das Ereignis zu suchen, diesmal nach solchen, die mit dem Schlüsselwort "after" gekennzeichnet sind. Vom Objekt geht er zu dessen Modell und beginnt dort wieder mit der Suche. Vom Modell geht er dann zu dessen Modell bzw. dem Default und sucht dort auch noch nach passenden Regeln. Hierbei werden aber alle gefundenen Regeln ausgeführt.

Eine Abweichung von diesem Schema gibt es für Regeln, die an Tastaturereignisse ("key") und an das Hilfeereignis ("help") gebunden sind. Für diese Ereignisse gilt das obengenannte Schema auch. Zusätzlich dazu wird aber noch zum Vater des Objekts gegangen, falls auf dem gesamten Weg keine passende Regel gefunden worden ist. Damit kann man sehr einfach in Form einer Regel z.B. ein Hilfesystem anbinden, wenn man die entsprechende Regel im Dialog hinterlegt.

Das nachfolgende Schaubild veranschaulicht die Suche nach Regeln. Dabei wird in diesem Bild davon ausgegangen, dass dem Objekt ein Modell zugeordnet ist, das direkt vom Default abgeleitet ist.

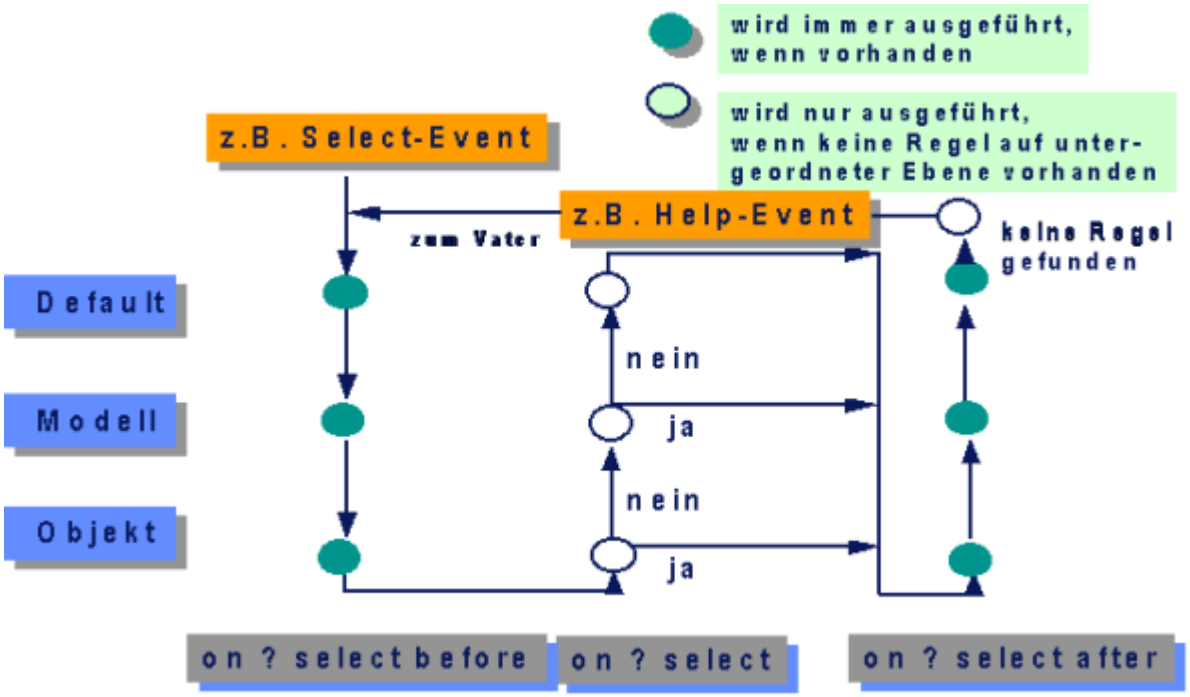

Abbildung 4: *Regelabarbeitungs-Reihenfolge*

Beispiel

Für ein Fensterobjekt sind folgende 4 Regeln definiert:

1. `on WINDOW close before`
2. `on WnFenster1 close`
3. `on WINDOW close`
4. `on WnFenster1 close after`

Damit ergibt sich folgende Reihenfolge: 1-2-4.

Die Regel 3 wird aufgrund der Existenz der Regel 1 nie ausgeführt.

Für einen Pushbutton sind folgende 5 Regeln definiert:

1. `on PUSHBUTTON select`
2. `on MPbOK select`
3. `on PUSHBUTTON select after`
4. `on MPbOK select before`
5. `on MPbOK select after`

Damit ergibt sich folgende Reihenfolge: 4-2-5-3.

Die Regel 1 wird aufgrund der Existenz der Regel 2 nie ausgeführt.

Als Anwendungsbeispiel kann hier ein System zur Dateneingabe genannt werden. In diesem System werden viele Fenster definiert, die über einen OK-Button den Inhalt verarbeiten und über einen Abbrechen-Button den Inhalt verwerfen. Bei beiden Buttons wird jeweils das Fenster geschlossen.

Die Regeln kann man nun so definieren, dass am Modell des Abbrechen-Buttons eine Regel hängt, die das jeweilige Fenster schließt. Im Fall vom OK-Button ist es schwieriger, da zunächst eine fensterspezifische Verarbeitung erfolgen muss. Deshalb ergibt sich folgende sinnvollere Möglichkeit:

An jeder Instanz des OK-Buttons hängt eine Regel, die die aktuelle Funktion aufruft und am Modell des OK-Buttons hängt eine "after" Regel, die immer das Fenster schließt. Damit wird erreicht, dass das Schließen des Fensters zentral erfolgt und nicht in jeder Instanzregel erneut programmiert werden muss.

3 Userdata und benutzerdefinierte Attribute

In diesem Kapitel soll gezeigt werden, wie das Attribut .userdata und die benutzerdefinierten Attribute sinnvoll zum Speichern von Werten genutzt werden können.

Um den Einsatz dieser Attribute sinnvoll zeigen zu können, soll ein Beispiel gebaut werden, das eine einfache Adressenverwaltung darstellen soll.

In einem Startfenster soll eine Tabelle dargestellt werden, in der der Name, der Vorname und der Wohnort enthalten ist. Durch Pushbuttons soll der Anwender in die Lage versetzt werden, die Einträge zu löschen, zu verändern oder neue Einträge hinzuzufügen. Das Ändern oder Hinzufügen erfolgt in einem weiteren Fenster, in dem weitere Daten wie die Straße und die Postleitzahl erfasst werden. Dabei wird gefordert, dass je Zeile in der Tabelle dieses Fenster genau einmal offen sein kann, damit der Anwender nicht durch doppelte Fenster mit identischen Daten auf dem Bildschirm verwirrt wird.

3.1 Realisierung der Fenster

Zunächst werden mit Hilfe des Editors die beiden im Beispiel geforderten Fenster angelegt. Das eine Fenster wird ganz normal als Instanz angelegt, da es nur einmal offen sein kann. Das Detailfenster hingegen soll je Zeile einmal geöffnet werden können. Aus diesem Grund wird es zunächst als Instanz definiert und dann in ein Modell umgewandelt.

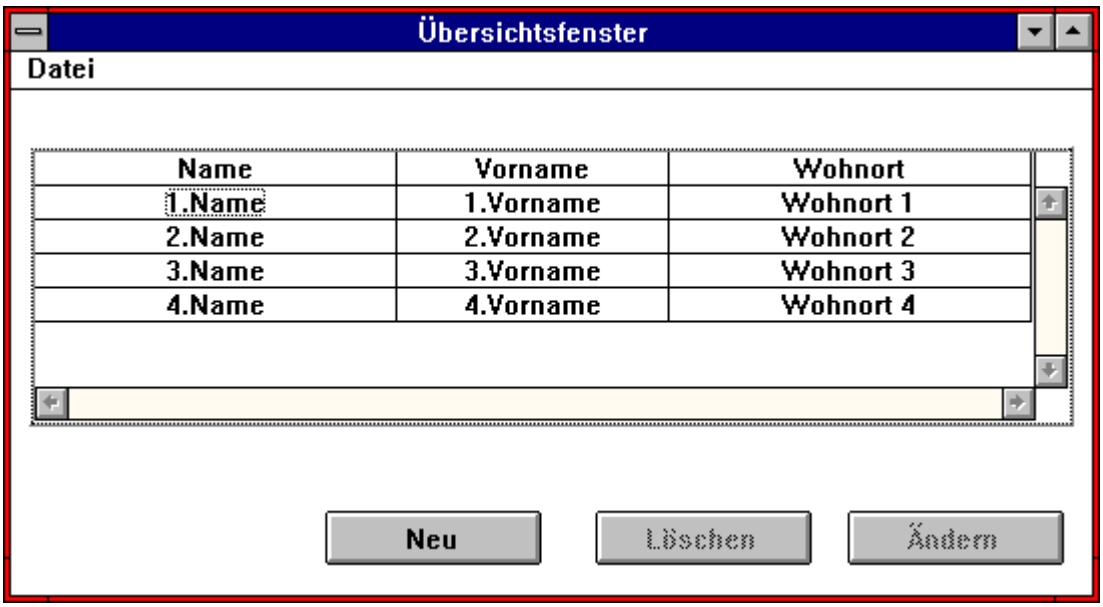

Abbildung 5: *Startfenster der Adressverwaltung*

Zusätzlich wird diesem Fenster ein Menü zugeordnet, so dass das Programm regulär durch die Selektion des Menüeintrags beendet werden kann. Die Attribute der Tabelle werden so gesetzt, dass die Tabelle 3 Spalten und keine Schatten hat. Zusätzlich werden alle Spalten zentriert ausgerichtet.

Bei den Objekten wurden teilweise gleich im Objekt die Regeln hinterlegt, wie das Objekt sich verhalten soll. Bei der Selektion des "Neu"-Pushbuttons wird eine neue Instanz des Detailfensters generiert und angezeigt. In dieses kann der Benutzer dann seine Daten eingeben. Durch die Selektion des "Beenden"-Menüeintrags wird das gesamte System durch den Aufruf der Funktion "exit" verlassen. Durch die Selektion des "Ändern"-Pushbuttons wird eine benannte Regel aufgerufen, da die gleiche Funktionalität auch durch einen Doppelklick auf das Tabellenelement ausgelöst werden kann.

Im Dialogskript sieht das Fenster wie folgt aus:

```
window WnUebersicht
{
  .width 61;
  .height 10;
  .title "\334bersichtsfenster";
  child tablefield TfAdressen
  {
    .xauto 0;
    .xleft 0;
    .ytop 0;
    .height 6;
    .fieldshadow false;
    .selection[sel_row] true;
    .selection[sel_header] false;
    .selection[sel_single] false;
    .colcount 3;
    .rowheader 1;
    .colfirst 1;
    .rowfirst 2;
    .colwidth[1] 20;
    .colalignment[1] 0;
    .colwidth[2] 15;
    .colalignment[2] 0;
    .colwidth[3] 20;
    .colalignment[3] 0;
    .content[1,1] "Name";
    .content[1,2] "Vorname";
    .content[1,3] "Wohnort";
    .content[2,1] "1.Name";
    .content[2,2] "1.Vorname";
    .content[2,3] "Wohnort 1";
    .content[3,1] "2.Name";
    .content[3,2] "2.Vorname";
    .content[3,3] "Wohnort 2";
    .content[4,1] "3.Name";
```

```
  .content[4,2] "3.Vorname";
  .content[4,3] "Wohnort 3";
  .content[5,1] "4.Name";
  .content[5,2] "4.Vorname";
  .content[5,3] "Wohnort 4";
  on dbselect
  {
    !! Wenn in der Tabelle wirklich was selektiert ist,
    !! anzeigen des Aenderungsdialogs
    if (first(this.activeitem) > 0) then
        RlAendern();
    endif
  }
}
child pushbutton PbNeu
{
  .xleft 17;
  .ytop 7;
  .text "Neu";
  on select
  {
    !! Generieren des neuen Objektes, zunaechst
    !! unsichtbar
     variable object Neuesfenster :=
        create(MWnDetail, this.dialog, true);

     !! Anzeigen des neuen Objektes
     Neuesfenster.visible := true;
  }

}
child pushbutton PbLoeschen
{
  .xleft 32;
  .ytop 7;
  .text "L\366schen";
}
child pushbutton PbAendern
{
  .xleft 46;
  .ytop 7;
  .text "\304ndern";
  on select
  {
     !! Wenn Aendern angewaehlt wird, anzeigen des
     !! Aendern-Dialogs
```

```
        RlAendern();
      }
    }
    child menubox MbDatei
    {
      .title "Datei";
      child menuitem MiSpeichern
      {
        .text "Speichern";
      }
      child menuitem MiEnde
      {
        .text "Beenden";
        on select
        {
             exit();
         }
      }
    }
}
```

Das Detailfenster sieht wie folgt aus:

Abbildung 6: *Adresseneingabefenster*

Da dieses Fenster mehrmals zur Laufzeit parallel benötigt wird, wird dieses Fenster als Modell abgelegt. Die beiden Pushbuttons sind von einem gemeinsamen Modell abgeleitet. Das Modell dient ausschließlich der Regelvereinfachung.

Im Dialogskript ist das Fenster wie folgt definiert:

```
model window MWnDetail
{
  .xleft 16;
  .width 58;
  .ytop 303;
```

```
.height 8;
.title "Detailansicht";
integer ZugehoerigeZeile := 0;
child MPbOkAbbrechen PbOK
{
  .xleft 44;
  .ytop 5;
  .text "OK";
}
child MPbOkAbbrechen PbAbbrechen
{
  .xleft 31;
  .ytop 5;
  .text "Abbrechen";
}
child statictext
{
  .xleft 1;
  .ytop 0;
  .text "Name:";
}
child statictext
{
  .xleft 1;
  .ytop 1;
  .text "Vorname:";
}
child statictext
{
  .xleft 1;
  .ytop 2;
  .text "Stra\337e:";
}
child statictext
{
  .xleft 1;
  .ytop 3;
  .text "Plz:";
}
child statictext
{
  .xleft 14;
  .ytop 3;
  .text "Ort:";
}
child edittext EtName
```

```
  {
    .active false;
    .xleft 12;
    .width 29;
    .ytop 0;
    .content "";
  }
  child edittext EtVorname
  {
    .active false;
    .xleft 12;
    .width 29;
    .ytop 1;
    .content "";
  }
  child edittext EtStrasse
  {
    .active false;
    .xleft 12;
    .width 29;
    .ytop 2;
    .content "";
  }
  child edittext EtOrt
  {
    .active false;
    .xleft 18;
    .width 29;
    .ytop 3;
    .content "";
  }
  child edittext EtPlz
  {
    .xleft 5;
    .width 4;
    .ytop 3;
  }
}
```

3.2 Das Modell MPbOkAbbrechen

Das Modell "MPbOkAbbrechen" ist wie folgt definiert:

```
model pushbutton MPbOkAbbrechen
{
    !! Allgemeine Regel fuer die
```

```
    !! Selektion der Pushbuttons
    !! im Detailfenster
    on select after
    {
      !! Beim Schliessen muss das Fenster
      !! zerstoert werden
      destroy(this.window, true);
    }
}
```

Durch die Einführung dieses Modells kann die Implementierung der Regeln bei dem "Abbrechen"- und "OK"-Pushbutton vereinfacht werden. Da beide Pushbuttons von diesem Modell abgeleitet sind und die Regel an diesem Modell mit dem Schlüsselwort "after" definiert worden ist, wird diese Regel immer bei der Selektion einer der beiden Pushbuttons durchlaufen. Für den "Abbrechen"-Pushbutton muss also keine weitere Regel definiert werden, für den "OK"-Pushbutton muss noch das Übernehmen der Änderungen implementiert werden.

3.3 Zuordnung der Detailfenster zu einer Zeile

Die Zuordnung der Detailfenster zu einer Zeile in der Tabelle erfolgt über folgenden Mechanismus:

In der Userdata der 1. Spalte jeder Zeile wird die ID des zugehörigen Fensters hinterlegt. Wenn nun zu einer Zeile ein Fenster geöffnet werden soll, wird zunächst in der Userdata überprüft, ob dort bereits ein Fenster gespeichert worden ist. Wenn das der Fall ist, ist die dort gespeicherte ID ungleich der ID *null*. Dann wird das Fenster nur noch aktiviert, in dem es erneut auf sichtbar gesetzt wird. Ist die in der Userdata gespeicherte ID *null*, gehört kein Fenster zu dieser Zeile. Es wird dann ein neues Fenster generiert, mit den entsprechenden Daten gefüllt und angezeigt. In diesem Fall ist der Einsatz der Userdata sehr sinnvoll, da jede Zelle der Tabelle über eine Userdata verfügt und der Dialog Manager das Verwalten dieser Einträge übernimmt. Wird also eine Zeile gelöscht oder hinzugefügt, wird die Userdata entsprechend korrigiert. Bei einem benutzerdefinierten Attribut wäre das nicht der Fall und müsste in der Anwendung verwaltet werden.

Bei der Zuordnung des Fensters zu einer Zeile sieht das anders auch. Auch hier könnte man die Userdata des Fensters benutzen, um sich die notwendige Information zu merken. Im Normalfall wird man dies aber über benutzerdefinierte Attribut lösen, da für die Zuordnung ein einziges Attribut am Objekt ausreicht und damit die Lesbarkeit des Dialogcodes wesentlich erhöht wird.

Damit sieht das Detailfenster im Dialogskript wie folgt aus:

```
model window MWnDetail
{
  .xleft 16;
  .width 58;
  .ytop 303;
  .height 8;
  .title "Detailansicht";
  integer ZugehoerigeZeile := 0;
}
```

In dem Attribut "ZugehoerigeZeile" wird die Nummer der Zeile gemerkt, aus der das Fenster geöffnet worden ist. Wird das Fenster über den "Neu"-Pushbutton geöffnet, wird hier kein neuer Wert hinterlegt; es bleibt also der definierte Wert *0* erhalten. Dadurch kann bei der Selektion des "OK"-Pushbutton erkannt werden, ob das Fenster über "Neu" oder "Ändern" geöffnet worden ist.

Damit sieht die Regel zum Ändern eines Datensatzes wie folgt aus:

```
rule void RlAendern
{
    !! Anlegen einer Variablen zum Zwischenspeichern
    !! des Fensters
    variable object NeuesFenster;

    !! Uebernehmen der Userdata aus Spalte 1 der
    !! aktiven Zeile des Tablefields
    NeuesFenster := TfAdressen.userdata
        [first(TfAdressen.activeitem),1];
    !! Wenn noch kein Fenster dazugehoert
    !! muss ein Fenster generiert werden
    !! und in die Userdata eingetragen werden
    if (NeuesFenster = null) then
      NeuesFenster := create(MWnDetail, this.dialog, true);
      !! Merken des neuen Fenster in der Userdata
      TfAdressen.userdata[first(TfAdressen.activeitem),1]
        := NeuesFenster;
      !! Uebernehmen der Werte aus der aktiven Zeile
      !! in das neue Fenster
      NeuesFenster.EtName.content :=
       TfAdressen.content[first(TfAdressen.activeitem),1];
      NeuesFenster.EtVorname.content :=
       TfAdressen.content[first(TfAdressen.activeitem),2];
      NeuesFenster.EtOrt.content :=
       TfAdressen.content[first(TfAdressen.activeitem),3];
      !! Jetzt muss das Fenster sich noch merken,
      !! aus welcher Zeile es geoeffnet worden ist.
      !! Dazu dient das Attribut ZugehoerigeZeile.
      NeuesFenster.ZugehoerigeZeile :=
        first(TfAdressen.activeitem);
    endif
    !! Zum Schluss wird das Fenster noch sichtbar gemacht.
    !! Wenn es schon sichtbar war, kommt es nur in den
    !! Vordergrund
    NeuesFenster.visible := true;
}
```

Durch die Selektion des "OK"-Pushbuttons werden die Daten aus dem Detailfenster in das Übersichtsfenster übernommen. Dabei wird an dem Wert des Attributes "ZugehoerigeZeile" erkannt, ob

das Fenster über "Neu" oder "Ändern" geöffnet worden ist. Wurde es zum Ändern geöffnet, werden die Daten in die Ursprungszeile zurückgeschrieben. Wurde es für einen neuen Datensatz geöffnet, wird die Tabelle um eine Zeile vergrößert und der neue Datensatz am Ende der Tabelle angefügt.

```
!! Wenn OK selektiert wird,
!! muessen die Daten in das Uebersichtsfenster
!! uebernommen werden.
on PbOK select
{
    !! In dem Attribut ZugehoerigeZeile des Fensters ist
    !! die Information gespeichert,
    !! wohin die Daten zurueckgeschrieben werden muessen;
    !! dabei gibt es eine Ausnahme: 0.
    !! Dieser Wert heisst, dass das Fenster ueber NEU geoeffnet
    !! worden ist.
    if (this.window.ZugehoerigeZeile = 0) then
      !! In diesem Fall muss eine Zeile am Ende eingefuegt
      !! werden
      TfAdressen.rowcount := (TfAdressen.rowcount + 1);
      this.window.ZugehoerigeZeile := TfAdressen.rowcount;
    endif
    !! Jetzt kann der Inhalt uebernommen werden.
    !! Vorsicht: Die Zugriffe im Detailfenster muessen
    !! alle relativ sein, da es ein Modell ist
    TfAdressen.content[this.window.ZugehoerigeZeile,1] :=
      this.window.EtName.content;
    TfAdressen.content[this.window.ZugehoerigeZeile,2] :=
      this.window.EtVorname.content;
    TfAdressen.content[this.window.ZugehoerigeZeile,3] :=
      this.window.EtOrt.content;
}
```

Bei dieser Realisierung wird zusätzlich ausgenutzt, dass ein Objekt, wenn es zerstört wird, alle Referenzen auf sich im Dialog entfernt. Damit muss beim Zerstören des Detailfensters der Wert in der zugehörigen Userdata nicht vom Programmierer auf *null* gesetzt werden; dies wird automatisch vom Dialog Manager übernommen.

3.4 Löschen einer Zeile

Beim Löschen einer Zeile kann die Methode ":delete" benutzt werden. Diese Methode löscht eine oder mehrere Zeilen (bzw. Spalten) aus einer Tabelle. Dabei werden alle Informationen zu der betroffenen Zeile der Spalte gelöscht, also auch die zugehörige Userdata.

Wenn nun eine Zeile gelöscht wird, müssen alle Fenster, die aus einer nachfolgenden Zeile geöffnet worden sind, informiert werden, dass sich ihre "ZugehoerigeZeile" verändert hat. Damit nicht alle Tabellenelemente betrachtet werden müssen, sollte diese Schleife mit der Zeilennummer der gerade gelöschten Zeile beginnen und bis zur Zeilenanzahl ablaufen.

```
!! Loeschen der aktiven Zeile in der Tabelle
!! Wenn dazu noch ein Fenster offen ist, zuerst das
!! Fenster zerstoeren, dann erst die Zeile loeschen
on PbLoeschen select
{
    !! Schleifenvariable
    variable integer I;
    !! Ueberpruefen, ob ein Fenster dazugehoert
    if (TfAdressen.userdata[first(TfAdressen.activeitem),1]
    <> null) then
     destroy(TfAdressen.userdata[first(TfAdressen.activeitem),
       1], true);
    endif
    !! Merken der aktiven Zeile. Nach dem Loeschen
    !! ist diese Information nicht mehr abfragbar
    I := first(TfAdressen.activeitem);
    !! Loeschen der Zeile
    TfAdressen:delete(first(TfAdressen.activeitem), 1, false);
    !! Verschieben der Zeileninformation
    !! in allen nachfolgenden Fenstern
    for I := I to TfAdressen.rowcount do
      if (TfAdressen.userdata[I,1] <> null) then
        TfAdressen.userdata[I,1].ZugehoerigeZeile := I;
      endif
    endfor
    TfAdressen.activeitem := [1,0];
}
```

3.5 Zusätzliche Regeln

In der Dialogstartregel wird dafür gesorgt, dass in der Tabelle nichts selektiert ist und alle Userdata der Tabelle mit dem Wert *null* initialisiert sind.

```
on dialog start
{
    !! Sicherstellen, dass in der Userdata nichts
    !! gespeichert wird
    TfAdressen.userdata[0,0] := null;
    !! Loeschen aller Selektionen
    TfAdressen.activeitem := [0,0];
}
```

Um die Selektierbarkeit der beiden Pushbuttons „Ändern“ und „Löschen“ nicht mehrmals zu implementieren, wird eine Regel definiert, die auf Änderungen des Selektionszustandes in der Tabelle reagiert. Nur wenn nach einer solchen Änderung noch eine Zeile selektiert ist, dürfen die beiden Pushbuttons selektierbar sein.

```
!! Immer wenn in der Tabelle die Selektion veraendert wird,
!! muss ueberprueft werden, ob die Pushbuttons
!! noch sensitiv bleiben duerfen bzw sensitiv werden muessen
on TfAdressen select, .activeitem changed
{
    !! Wenn keine Zeile selektiert ist,
    !! beide Pushbuttons auf insensitiv schalten;
    !! anderenfalls beide sensitiv schalten
    if (first(TfAdressen.activeitem) = 0) then
      PbLoeschen.sensitive := false;
      PbAendern.sensitive := false;
    else
      PbLoeschen.sensitive := true;
      PbAendern.sensitive := true;
    endif
}
```

4 Objektorientierte Programmierung

Dieses Beispiel richtet sich an Programmierer, die bereits Erfahrungen im Umgang mit dem Dialog Manager haben. Es werden Verfahren gezeigt, wie man in der Dialogbeschreibungssprache objektorientiert programmieren kann. Dazu wird die im System vorhandene Möglichkeit, eigene Attribute zu definieren, intensiv genutzt. In diesen benutzerdefinierten Attributen werden dann alle notwendigen Informationen abgelegt, die normalerweise in der Klassenstruktur eines Objektes zu finden sind.

4.1 Beschreibung des zu entwickelnden Systems

Mit Hilfe des Dialog Managers soll eine Verwaltung für Kunden und Aufträge prototypisch entwickelt werden. Dabei soll objektorientiert vorgegangen werden, so dass das System möglichst einfach erweitert und gewartet werden kann. Dabei ergeben sich für diese beiden Anwendungsobjekte folgende Strukturen:

Kunde

Liste aller Kunden

- Information über den Kunden
  - Rechnungen als Detailinformation
  - Offene Aufträge als Detailinformation

Auftrag

Liste aller Aufträge

- Information über den Auftrag
  - Fertigungsaufträge, die zu dem Auftrag gehören
  - Maschinenbelegung durch den Auftrag

Das nachfolgende Schaubild stellt die Struktur der Fenster in dem System dar.

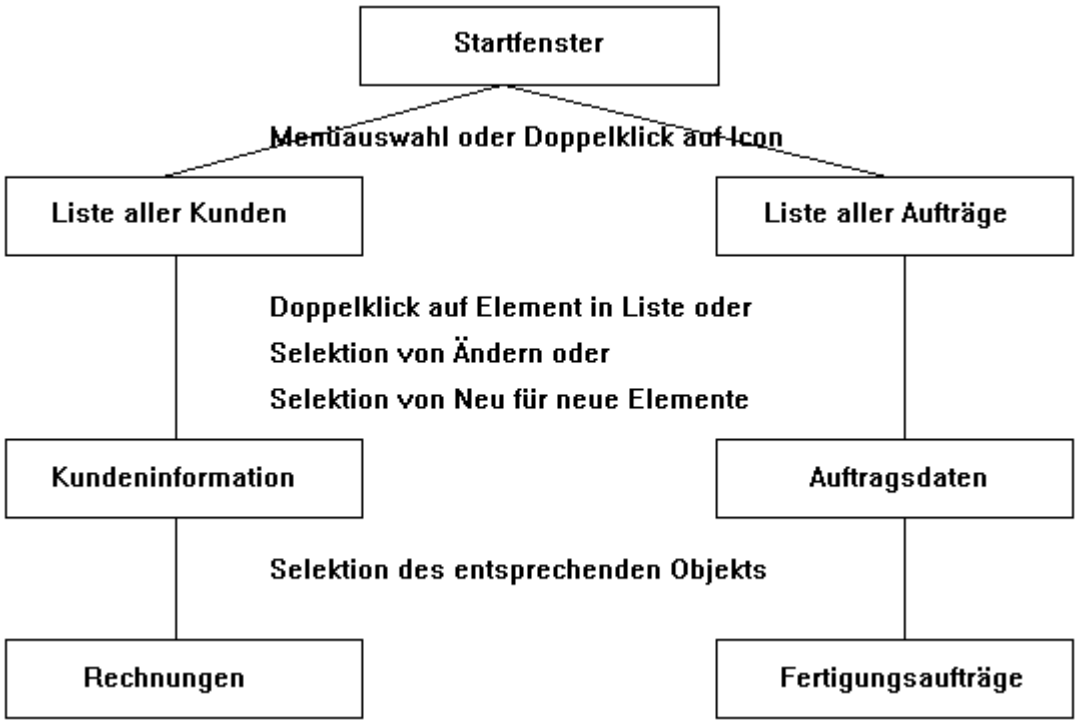

Abbildung 7: *Struktur des Kundensystems*

4.2 Realisierungsansatz

Bei der Implementierung wurde so vorgegangen, dass die verarbeitenden Teile (Regeln) eigentlich nicht wissen, mit welcher Art von Objekt sie gerade arbeiten. Die für die Regeln notwendigen Informationen werden alle in Form von Parametern übergeben. Um diese so allgemeingültig wie nur möglich gestalten zu können, wurden zunächst die realen Objekte untersucht und definiert, welche Funktionen auf diese Objekte angewendet werden können. Dabei wurde festgehalten, in welchen Teilen die einzelnen Methoden sich bei den Objekten in welcher Form unterscheiden. Diese Unterschiede wurden dann in geeignete DM-Strukturen übersetzt. Bei dieser Übersetzung wurden alle Teile, die mehrfach verwendet werden, in Vorlagen hinterlegt. Aus diesen Vorlagen wurden dann die entsprechenden Einzelstrukturen aufgebaut. Diese Einzelstrukturen beinhalten also alle Informationen, die zur Bearbeitung des Objekts notwendig sind.

Um nun diese Informationen in Fenstersystemen richtig verwalten zu können, trägt jedes offene Fenster einen Verweis auf die so geschaffene Grundstruktur eines Objektes und einen Verweis auf das wirklich in ihm dargestellte Objekt. Der Verweis auf das Grundobjekt ist notwendig, um die für das Objekt übliche Reaktionsweise auf Benutzerereignisse durchführen zu können. Der Verweis auf das wirklich dargestellte Objekt ist notwendig, um feststellen zu können, ob das Objekt schon in irgendeinem Fenster dargestellt wird. Neben den Fenstern beinhalten alle Objekte, die direkt mit den Objektstrukturen arbeiten müssen, Informationen wie diese Objekte behandelt werden müssen. Bei Eingabefeldern ist das Element der internen Datenstruktur hinterlegt, dessen Inhalt sie anzeigen sollen, bei Bildern ist hinterlegt, zu welcher Objektart sie gehören und wie sie auf Ereignisse reagieren sollen, bei Pushbuttons ist hinterlegt, welche Methode durch ihre Selektion ausgelöst werden soll.

Diese Hinterlegung von Informationen bei den einzelnen Objektarten ist natürlich nur für dieses Beispiel gültig. Sie soll nur zeigen, wie solche Informationen sinnvoll hinterlegt und verarbeitet werden können.

4.3 Implementierung im Dialogskript

4.3.1 Namensgebung

Im Dialogskript wurde eine Namensgebung für die Objekte angewendet, so dass aufgrund des Namens die einzelnen Objekttypen erkannt werden können. Dieses Schema ist wie folgt aufgebaut:

```
[M]<Objektkennzeichen>ObjektName
```

M

Ist der erste Buchstabe ein "M", so handelt es sich um ein Modell bzw. eine Vorlage. Danach folgen die Buchstaben, die den Objekttyp kennzeichnen.

Objektkennzeichen

Für das Objektkennzeichen wurden folgende Abkürzungen verwendet:

Pb	bezeichnet Pushbutton
Wn	bezeichnet Fenster
Rc	bezeichnet Records
Et	bezeichnet Eingabefelder
Lb	bezeichnet Listboxen
Im	bezeichnet Bilder

ObjektName

Beschreibender Name des Objekts

4.3.2 Datenstrukturen

Die Klassenstruktur wird in einer Dialogstruktur abgebildet. Dazu werden mehrere Unterstrukturen gebildet, um später möglichst einfach alle Elemente verwalten zu können. Diese Struktur besteht aus folgenden Einzelstrukturen: Struktur für Aufträge, Struktur für Kunden, Struktur zum Abspeichern der Objekte, Struktur für die Methoden.

4.3.2.1 Auftragsstruktur

In der Auftragsstruktur werden alle zu einem Auftrag gehörenden Informationen abgelegt. Dazu gehören die Auftragsnummer, die Artikelnummer, die Anzahl, die Start-, Ende- und Liefertermine. Diese

Struktur wird als Vorlage abgelegt, da beliebig viele Aufträge in das System eingegeben werden können.

```
!!Datenstruktur für den Auftrag
model record MRcAuftrag
{
  string AuftragsNr := "";
  string Artikel := "";
  integer Anzahl := 0;
  string FStart := "";
  string FEnde := "";
  string Liefer := "";
}
```

4.3.2.2 Kundenstruktur

Die Kundenstruktur umfasst alle Informationen, die den Kunden betreffen. Dazu gehören die Kundennummer, Firma, Anschrift und der Ansprechpartner.

```
!!Datenstruktur für den Kunden
model record MRcKunde
{
  string KundenNr := "";
  string Firma := "";
  string Strasse := "";
  integer Plz := 0;
  string Ort := "";
  string Partner := "";
  string Telefon := "";
  string Fax := "";
}
```

4.3.2.3 Struktur zum Abspeichern der Elemente

Damit beliebig viele Objekte (Kunden oder Aufträge) intern auch verwaltet werden können, gibt es eine Struktur, die zur Laufzeit in ihrer Größe an die Bedürfnisse angepasst werden kann. Dabei können in dem so realisierten Feld beliebig viele Objekte beliebiger Art abgespeichert werden

```
!!Datenstruktur zum Abspeichern der Werte. Die Struktur
!! wird bei Bedarf vergrößert.
model record MRcSpeicher
{
  object Elemente[5];
  .Elemente[1] := null;
  .Elemente[2] := null;
  .Elemente[3] := null;
  .Elemente[4] := null;
```

```
  .Elemente[5] := null;
}
```

Um nun in dieser Struktur Werte abzulegen, wurde die nachfolgende Regel definiert, die beliebige Elemente in diese Struktur übernehmen kann.

```
!!Aufnehmen eines neuen Elements in die Speicherliste
rule object Rl_NeuesElement (object ObjektInfo input)
{
  variable integer I;
  variable integer FreierPlatz := 0;

  !! Suchen nach einem freien Platz in der Liste
  for I:= 1 to ObjektInfo.MRcSpeicher.count[.Elemente] do
    if ((ObjektInfo.MRcSpeicher.Elemente[I] = null)
    and (FreierPlatz = 0)) then
      FreierPlatz := I;
    endif
  endfor
  !! Kein Platz gefunden: Speicherbereich muss
  !! vergrößert werden
  if (FreierPlatz = 0) then
    ObjektInfo.MRcSpeicher.count[.Elemente] :=
    (ObjektInfo.MRcSpeicher.count[.Elemente] + 10);
  !! Initialisierung der neuen Elemente mit null
  for I:=(ObjektInfo.MRcSpeicher.count[.Elemente] - 9) to
  ObjektIno.MRcSpeicher.count[.Elemente] do
    ObjektInfo.MRcSpeicher.Elemente[I] := null;
  endfor
  FreierPlatz:=(ObjektInfo.MRcSpeicher.count[.Elemente] -
    9);
  endif

  !! Generieren des neu zu speichernden Objekts
  ObjektInfo.MRcSpeicher.Elemente[FreierPlatz] :=
    create(ObjektInfo.ObjektArt, this.dialog);
  !! Rückgabe des neugenerierten Objekts
  return ObjektInfo.MRcSpeicher.Elemente[FreierPlatz];
}
```

4.3.2.4 Struktur zur Hinterlegung der Methoden

Dieses Objekt stellt das Kernstück des Beispiels dar. In der Methodenstruktur werden alle Methoden hinterlegt, die auf die unterschiedlichen Objekte im System angewendet werden können. Dazu wird zunächst bei allen Objekten untersucht, welche Methoden für sie realisiert werden müssen. In diesem Beispiel sind das folgende Methoden:

Create

In diesem Eintrag wird hinterlegt, welches Fenster zum Neuanlegen und zum Ändern der Haupt-tinformation zu einem Objekt genommen werden soll.

Init

Hier erfolgt die Hinterlegung der Initialisierungsfunktion, d.h. der Funktion, die eigentlich die notwendigen Daten aus der Datenbank laden soll. Da in diesem Beispiel keine Funktionen implementiert sind, ist hier jeweils nur eine Dummy-Regel hinterlegt.

Detail1-3

Hier erfolgt die Hinterlegung der Fenster, die die unterschiedlichen Detailinformationen zu den Objekten liefern können.

List

In diesem Element erfolgt die Hinterlegung des Fensters, in dem die Liste der Objekte dargestellt werden soll.

Rlist

Hier wird die Regel hinterlegt, die die Liste in dem Übersichtsfenster auffüllen kann.

Tlist

Dieser Eintrag enthält den Text, der als Überschrift für das Listenfenster dienen soll.

```
!!Dieses Objekt beschreibt die Methoden, die zu einem
!!Objekt vorhanden sind.
!!Create: Fenster zum Anlegen eines Objektes
!! (Grunddatenerfassung)
!!Init: Regel/Funktion zum Initialisieren der Datenstruktur
!!Detail[1]: Fenster für 1 Unterinformationen
!!Detail[2]: Fenster für 2 Unterinformationen
!!Detail[3]: ungenutzt
!!List: Fenster, in dem die Liste der Objekte angezeigt
!!werden soll
!!RList: Regel, die die Aufbereitung der Daten für die
!!Liste übernimmt
!!TList: Text, der als Titel im  Listenfenster erscheinen
!!soll
model record MRcMethoden
{
  object Create := null;
  object Init := null;
  object Detail[3];
  .Detail[1] := null;
  .Detail[2] := null;
  .Detail[3] := null;
  object List := null;
```

```
  object RList := null;
  string TList;
}
```

4.3.2.5 Objektstruktur

Die eigentliche Objektstruktur wird aus den anderen Strukturen zusammengesetzt. Dazu erhält diese Objektstruktur ein Element, in dem die Objektart (MRcAuftrag oder MRKunde abgespeichert wird. Daneben beinhaltet diese Struktur die beiden Unterstrukturen MRcSpeicher zum Abspeichern der Objekte und MRcMethoden zur Definition der Methoden, die zu dem Objekt gehören.

```
!!Struktur, die das Basisobjekt einer Objektklasse darstellt
model record MRObjekt
{
  object ObjektArt := null;
  child MRcSpeicher
  {
  }
  child MRcMethoden
  {
  }
}
```

4.3.2.6 Auftragsobjekt

Das Auftragsobjekt ist eine Instanz der Objektstruktur. Hier werden die entsprechenden Einträge für die Methoden entsprechend gesetzt.

```
!!Realisiertes Objekt für die Aufträge
MRObjekt RcAuftraege
{
  .ObjektArt := MRcAuftrag;
  .MRcMethoden.Create := MWnAuftrag;
  .MRcMethoden.Init := InitObjekt;
  .MRcMethoden.Detail[1] := MWnFertigung;
  .MRcMethoden.Detail[2] := Meldung;
  .MRcMethoden.List := MWnListe;
  .MRcMethoden.RList := RListAuftrag;
  .MRcMethoden.TList := "Auftragsliste";
}
```

4.3.2.7 Kundenobjekt

Das Kundenobjekt ist eine Instanz der Objektstruktur. Hier werden die entsprechenden Einträge für die Methoden entsprechend gesetzt.

```
!!Realisiertes Objekt für die Kunden
```

```
MRObjekt RcKunden
{
  .ObjektArt := MRKunde;
  .MRcMethoden.Create := MWnKunden;
  .MRcMethoden.Init := InitObjekt;
  .MRcMethoden.Detail[1] := MWnRechnung;
  .MRcMethoden.Detail[2] := Meldung;
  .MRcMethoden.List := MWnListe;
  .MRcMethoden.RList := RListKunde;
  .MRcMethoden.TList := "Kundenliste";
}
```

4.3.2.8 Übergeordnete Struktur

Um beim Programmstart und Programmende automatisch alle Datenstrukturen behandeln zu können, wurde eine übergeordnete Struktur gebildet, die nur Verweise auf die untergeordneten Strukturen beinhaltet. Dadurch kann z.B. beim Programmstart eine Schleife über alle intern vorhandenen Datenstrukturen gebildet werden, um diese entsprechend ihrer Initialisierungsmethode initialisieren zu lassen.

```
!!Die nachfolgende Datenstruktur dient nur dazu, alle
!!Strukturen beim
!!Programmstart zu initialisieren. Diese müssten natürlich
!!aus der Datenbank gelesen werden, was aber in diesem
!!Beispiel nicht implementiert worden ist. Damit Daten aber
!!erhalten bleiben, speichert der Dialog sich selbst wieder
!!ab. Dadurch bleiben auch die geänderten Werte erhalten.
record RcObjekttypen
{
  object Objekte[2];
  .Objekte[1] := RcAuftraege;
  .Objekte[2] := RcKunden;
}
```

4.3.3 Erweiterung der bestehenden Objekte

Die im DM vorhandenen Objekte wurden zum Teil durch eigene Attribute erweitert, so dass dort alle notwendigen Informationen abgespeichert werden können.

4.3.3.1 Erweiterungen beim Objekt window

Bei diesem Objekt wurden vier Attribute ergänzt, damit die Fenster alle mit denselben Regeln bearbeitet werden können. Über das Attribut "Dynamisch" wird gekennzeichnet, ob das Fenster im Fenster nur genau einmal offen sein darf oder ob das Fenster beliebig oft vom Anwender mit anderen Daten geöffnet werden darf. Die Attribute "MethodenObjekt" und "DargestelltesObjekt" sind

Verweise, die zur Identifikation des im Fenster dargestellten Inhalts dienen. Das "MethodenObjekt" ist dabei ein Verweis auf das Grundobjekt der Aufträge oder Kunden, das "Dargestelltes Objekt" ist ein Verweis auf die interne Datenstruktur, die zu diesem Fenster gehört. In dem Attribut "ZuAktivierendesObjekt" wird gemerkt, welches Objekt beim Öffnen des Fensters aktiviert werden soll.

```
!!Attribute MethodenObjekt, DargestelltesObjekt, Dynamisch
!!und ZuAktivierendesObjekt eingefügt
!!Bedeutung der Attribute
!!MethodenObjekt enthält alles, was zu der Objektart gehört
!!DargestelltesObjekt ist das aktuelle Objekt, das in dem
!!Fenster gerade bearbeitet wird
!!Dynamisch ist ein Kennzeichen dafür, wie das Fenster beim
!!Unsichtbarmachen behandelt werden soll.
!!ZuAktivierendesObjekt enthält das Objekt, das beim
!!Fensteröffnen in dem Fenster aktiviert werden soll.
default window
{
  ...
  object MethodenObjekt := null;
  object DargestelltesObjekt := null;
  boolean Dynamisch := true;
  object ZuAktivierendesObjekt := null;
}
```

Neben diesen Attributerweiterungen wurde eine globale Regel für das Defaultfenster definiert, die das Sichtbar- und Unsichtbarmachen des Fensters handhaben kann. Beim Unsichtbarmachen muss geprüft werden, ob das Fenster dynamisch generiert wurde. Ist dies der Fall muss das Fenster intern wieder zerstört werden. Beim Sichtbarmachen eines neuen Fensters wird über diese Regel das richtige Objekt im Fenster aktiviert, so dass der Anwender sofort mit der Bearbeitung des Fensters beginnen kann.

```
!!Allgemeine Regel zum Behandeln sichtbar / unsichtbar
!!gemachter Fenster
on Dialog.WINDOW close, .visible changed
{
  variable integer I;

  if ( not this.visible) then
    !! Fenster ist dynamisch generiert worden, d.h. es muss
    !! jetzt wieder gelöscht werden
    if this.Dynamisch then
      !! Zerstören des Fensters
      destroy(this, true);
    endif
  else
    !! Fenster ist sichtbar gemacht worden.
    !! Suchen nach dem Objekt, das aktiviert bzw.
    !! fokussiert werden soll
```

```
    for I := 1 to this.childcount do
      if (this.window.child[I].model =
      this.window.model.ZuAktivierendesObjekt) then
        if (this.window.child[I].class = edittext) then
          this.window.child[I].active := true;
        else
          this.window.child[I].focus := true;
        endif
        return ;
      endif
    endfor
  endif
}
```

Wenn ein Fenster zu einem bestimmten Datensatz geöffnet werden soll, wird zunächst gesucht, ob dieses Fenster nicht schon offen ist. Dazu wird über alle Fenster gegangen und geprüft, ob das aktuell betrachtete Fenster zu dem gesuchten Objektgrundtyp gehört, die gesuchte Art hat und das gesuchte Datenobjekt darstellt. Wird so ein Fenster gefunden, wird von der Regel dieses gefundene Fenster zurückgegeben, ansonsten wird *null* zurückgegeben.

```
!!Diese Regel sucht in allen geöffneten Fenstern nach einem
!!Fenster mit vorgegebenen Daten. Wird dieses Fenster
!!gefunden, wird es als Ergebnis nach außen zurück
!!geliefert, sonst wird null geliefert
rule object Rl_SucheNachFenster (object ObjektInfo input, object FensterArt
input, object DargObj input)
{
  variable integer I;

  !! Alle Kinder des Dialogs überprüfen
  for I := 1 to this.dialog.childcount do
    !! Schauen, ob das Objekt ein Fenster ist
    if (this.dialog.child[I].class = window) then
      !! Schauen, ob das Fenster zum gesuchten Methoden
      !! objekt gehört
      if(this.dialog.child[I].MethodenObjekt=ObjektInfo)
      then
        !! Schauen ob das Fenster zur gesuchten Fensterart
        !! gehört
        if (this.dialog.child[I].model = FensterArt) then
          !! Schauen, ob dieselbe Information dargestellt
          !! wird
          if (this.dialog.child[I].DargestelltesObjekt =
          DargObj) then
            return this.dialog.child[I];
          endif
        endif
```

```
      endif
    endif
  endfor
  !! Fenster nicht gefunden, null zurückliefern.
  return null;
}
```

Die nächste Regel legt ein Fenster einer definierten Art zu einem gegebenen Objekt an. Dabei wird das Fenster zunächst unsichtbar generiert, so dass die Daten noch im unsichtbaren Zustand in das Fenster gefüllt werden können

```
!!Anlegen eines Fensters für einen definierten Datensatz.
!!Zuerst wird überprüft, ob das Fenster nicht schon offen
!!ist und wieder verwendet werden kann. Ist das Fenster
!!als nicht dynamisch gekennzeichnet, so wird das
!! gefundene Fenster wiederverwendet
rule object FensterNeuAnlegen (object ObjektInfo input, object FensterArt
input, object DargObj input)
{
  variable object Obj := null;

  !! überprüfen, ob eine Fensterart angegeben ist
  if (FensterArt <> null) then
    !! überprüfen ob die Fensterart ein Fenster oder eine
    !! Messagebox ist
    if (FensterArt.class = window) then
      !! überprüfen, ob das Fenster nicht schon existiert
      Obj := Rl_SucheNachFenster(ObjektInfo, FensterArt,
      DargObj);
      !! überprüfen, ob das Fenster dynamisch generiert
      !! werden soll
      if ((FensterArt.Dynamisch = true) and (Obj = null))
      then
        !! Unsichtbares Generieren des Fensters
        Obj := create(FensterArt, this.dialog, true);
      else
        !! Fenster aktivieren
        if (Obj <> null) then
          Obj.visible := true;
        else
          !! Daten neu in das statische Fenster setzen
          FensterArt.visible := true;
          Obj := FensterArt;
        endif
      endif
      !! Werte in das gefundene / neu generierte Fenster
      !! eintragen
```

```
      Obj.MethodenObjekt := ObjektInfo;
      Obj.DargestelltesObjekt := DargObj;
      return Obj;
    else
      !! Meldungsfenster öffnen
      querybox(FensterArt);
    endif
  endif
  return null;
}
```

4.3.3.2 Erweiterungen beim Objekt edittext

Beim Eingabefeld wurde ein Attribut ergänzt, das beschreibt, aus welchem Element der internen Struktur der Inhalt in dem Objekt angezeigt werden soll. Dazu wurde ein Attribut vom Typ "attribute" hinzugefügt, um das Attribut der Datenstruktur aufnehmen zu können.

```
default edittext
{
  ...
  attribute Datenelement := .visible;
}
```

Aufgrund dieser Erweiterung kann jetzt eine allgemeine Regel definiert werden, die die Daten vom Darstellungsobjekt zu den internen Strukturen bzw. umgekehrt kopiert. Dazu werden von einem gegebenen Fenster alle Kinder betrachtet und überprüft, ob sie ein Äquivalent in der internen Struktur haben. Wenn dies der Fall ist, werden die Werte ihren Datentypen entsprechend kopiert.

```
!!Regel zum Handhaben der Informationsanzeige und Übernahme
!!der geänderten Daten in die interne Struktur
rule void Rl_AnzeigeHandhaben (object ObjektInfo input, object Fenster input,
boolean Display input)
{
  variable integer I;

  !! Daten sollen von der Anzeige in die interne Struktur
  !! übernommen werden
  if ( not Display) then
    !! Überprüfen, ob schon ein internes Objekt existiert
    if (Fenster.DargestelltesObjekt = null) then
      !! Anlegen eines internen Objekts
      Fenster.DargestelltesObjekt :=
      NeuesElement(ObjektInfo);
    endif
    if (Fenster.DargestelltesObjekt <> null) then
    !! Durchlaufen der Kinder des Fensters und Übernehmen
    !! der Werte in die interne Struktur
```

```
  for I := 1 to Fenster.childcount do
  !! Überprüfen, ob das Objekt ein Edittext ist
  if (Fenster.child[I].class = edittext) then
    !! Überprüfen, ob das Objekt zu einem sinnvollen
    !! Attribut gehört
    if (Fenster.child[I].Datenelement <> .visible) then
    !! Überprüfen des Datentyps der internen Struktur
    if (Fenster.DargestelltesObjekt.type
      [Fenster.child[I].Datenelement] = integer) then
      !! Überprüfen, ob wirklich eine Zahl eingetragen
      !! worden ist
      if fail(atoi(Fenster.child[I].content)) then
      !! Übernehmen des Werts in die interne Struktur
      setvalue(Fenster.DargestelltesObjekt,
        Fenster.child[I].Datenelement, 0);
      else
      !! Übernehmen des Werts in die interne Struktur
      setvalue(Fenster.DargestelltesObjekt,
        Fenster.child[I].Datenelement,
        atoi(Fenster.child[I].content));
      endif
    else
      !! Übernehmen des Werts in die interne Struktur
      setvalue(Fenster.DargestelltesObjekt,
        Fenster.child[I].Datenelement,
        Fenster.child[I].content);
    endif
    endif
  endif
  endfor
  !! Auswahlliste neu aufbauen
  ListeFuellen(Fenster.MethodenObjekt,
  InstanzVon(Fenster.MethodenObjekt.MRcMethoden.List,
    Fenster.MethodenObjekt));
endif
else
  !! Durchlaufen der Kinder des Fensters und Übernehmen
  !! der Werte in die interne Struktur
  for I := 1 to Fenster.childcount do
    !! Überprüfen, ob das Objekt ein Edittext ist
    if (Fenster.child[I].class = edittext) then
    !! Überprüfen, ob das Objekt zu einem sinnvollen
    !! Attribut gehört
    if (Fenster.child[I].Datenelement <> .visible) then
      if (Fenster.DargestelltesObjekt <> null) then
        if (Fenster.DargestelltesObjekt.type
```

```
            [Fenster.child[I].Datenelement] = integer)
          then
            !! Übernehmen des Werts in die Anzeige
            Fenster.child[I].content :=
              itoa(getvalue(Fenster.DargestelltesObjekt,
              Fenster.child[I].Datenelement));
          else
            !! Übernehmen des Werts in die Anzeige
            Fenster.child[I].content :=
              getvalue(Fenster.DargestelltesObjekt,
              Fenster.child[I].Datenelement);
          endif
        else
          !! Übernehmen des Werts in die Anzeige
          Fenster.child[I].content := "";
        endif
      endif
      endif
    endfor
  endif
}
```

Damit diese Regel richtig arbeiten kann, muss sie in der Lage sein, das für diesen Objekttyp zuständige Listenfenster zu bearbeiten. Dazu ist die nachfolgende Regel vorhanden.

```
!!Suchen nach einem Fenster, das die Instanz eines
!!vorgegebenen Modells ist
rule object InstanzVon (object Modell input, object Methode input)
{
  variable integer I;

  !! Alle Kinder des Dialogs betrachten
  for I := 1 to this.dialog.childcount do
    !! Überprüfen, ob das Objekt ein Fenster ist
    if (this.dialog.child[I].class = window) then
      !! Überprüfen, ob das Fenster Instanz des Modells
      !! ist
      if ((this.dialog.child[I].model = Modell)
      and (this.dialog.child[I].MethodenObjekt = Methode))
      then
        return this.dialog.child[I];
      endif
    endif
  endfor
  !! keine Instanz gefunden, Rückgabe von null.
  return null;
}
```

4.3.3.3 Erweiterungen beim Objekt image

Beim Bild wurde ein Attribut eingefügt, das einen Verweis auf das Methodenobjekt darstellt. Dadurch können bei Ereignissen auf Bildern die entsprechenden Methoden aufgerufen werden.

```
!!Attribut MethodenObjekt eingefügt.
!!Dieses Objekt enthält alle Informationen, die zu einem !!"Objekt" gemerkt
werden müssen.
default image
{
   ..
  object MethodenObjekt := null;
}
```

4.3.3.4 Erweiterungen beim Objekt pushbutton

Der Pushbutton wurde um ein Attribut erweitert, das die aufzurufende Methode kennzeichnet.

```
!!Attribut AttributIndex eingefügt:
!!In diesem Attribut wird der Index der Methode gemerkt,
!!die bei Selektion des Pushbuttons aufgerufen werden soll.
default pushbutton
{
  ...
  integer AttributIndex := 0;
}
```

4.3.4 Definitionen für die einzelnen Fenster

4.3.4.1 Aktionen beim Programmstart und Programmende

Bei Programmstart werden die internen Datenstrukturen initialisiert. Im Normalfall sollten diese Werte aus der Datenbank gelesen werden. Darauf wurde in diesem Beispiel aber verzichtet. Damit aber die eingegebenen Werte nicht verloren gehen, wird beim Programmende der gesamte Dialog mit den eingetragenen Werten abgespeichert.

```
!!Diese Regel sorgt für die Initialisierung aller
!! Objektgrundtypen
rule void InitialisierenObjekte
{
  variable integer I;

  !! Durchlaufen aller RcObjekttypen
  for I := 1 to RcObjekttypen.count[.Objekte] do
    !! Überprüfen, ob wirklich ein Objekt gespeichert ist.
    if (RcObjekttypen.Objekte[I] <> null) then
      !! Aufruf der Initialisierungsmethode, falls diese
```

```
      !! vorhanden ist
      if (RcObjekttypen.Objekte[I].MRcMethoden.Init
      <> null) then
        RcObjekttypen.Objekte[I].MRcMethoden.Init(
          RcObjekttypen.Objekte[I].MRcSpeicher,
          RcObjekttypen.Objekte[I].ObjektArt);
      endif
    endif
  endfor
}

!!Abspeichern des aktuellen Dialogzustandes beim Beenden des Dialogs
on dialog finish
{
  save(this, "kunden.sav");
}

!!Startregel für das System
on dialog start
{
  InitialisierenObjekte();
}

!!Diese Regel sollte die Objekte eines Types initia-
!! lisieren, also sinnvollerweise aus der Datenbank laden
rule void InitObjekt (object Anker input, object Typus input)
{
  print "Hier m\201\341te das Laden aus der DB realisiert werden";
}
```

4.3.4.2 Das Startfenster

Das Startfenster besteht aus zwei Icons, die die unterschiedlichen Objektarten darstellen und aus einem Menü zum Auslösen der Aktion und zum Beenden des Dialogs.

```
!!Das nachfolgende Fenster erscheint beim Programmstart.
!!Von ihm aus können
!!die Fenster zu Kunden und Aufträge geöffnet werden.
window WnStart
{
  .userdata null;
  .active false;
  .xleft 400;
  .width 38;
  .ytop 396;
  .height 8;
```

```
  .title "Auswahl";
  .MethodenObjekt := null;
  .Dynamisch := false;
  child image IKunden
  {
    .xleft 5;
    .ytop 2;
    .text "Kunden";
    .picture TiKunde;
    .MethodenObjekt := RcKunden;
  }
  child image IAuftraege
  {
    .xleft 15;
    .ytop 2;
    .text "Auftr\344ge";
    .picture TiAuftrag;
    .MethodenObjekt := RcAuftraege;
  }
  child menubox MbAuswahl
  {
    .title "Auswahl";
    child menuitem MiListe
    {
      .text "Liste anzeigen";
    }
    child menuitem MiBeenden
    {
      .text "Beenden";
    }
  }
}

!!Menüeintrag zur Anzeige der Objektliste
on MiListe select
{
  variable object Obj;
  variable object NeuesObjekt;

  Obj := this.window.focus;
  !! Schauen, zu welchem Objekttyp die Fenster gezeigt
  !! werden sollen
  if (Obj.MethodenObjekt <> null) then
    NeuesObjekt := FensterNeuAnlegen(Obj.MethodenObjekt,
      Obj.MethodenObjekt.MRcMethoden.List, null);
    ListeFuellen(Obj.MethodenObjekt, NeuesObjekt);
```

```
    NeuesObjekt.visible := true;
  endif
}

!!Beenden des Systems
on MiBeenden select
{
  exit();
}
```

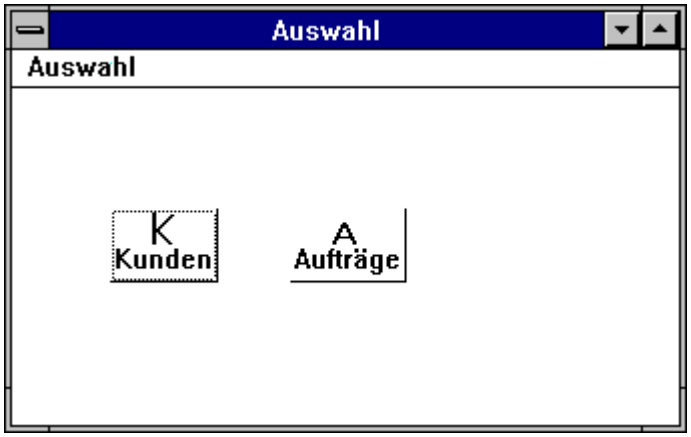

Abbildung 8: *Startfenster*

4.3.4.3 Das Übersichtsfenster

Im Übersichtsfenster werden die Listen aller Kunden oder Aufträge dargestellt. Über die Pushbuttons am unteren Ende des Fensters können die Aktionen ausgelöst werden. Dabei ist folgende Funktionsweise implementiert worden: Ein Doppelklick auf einen Eintrag in der Liste oder die Selektion des "Informations"-Pushbutton zeigt die Hauptdaten zum entsprechenden Datensatz an. Das sind dann entweder die Anschrift oder die Fertigungsdaten. Über den Pushbutton "Löschen" können Einträge in der Liste und damit auch in den internen Strukturen gelöscht werden. Durch Selektion des Pusbuttons "Neu" können neue Datensätze angelegt werden. Dieses Fenster ist als Vorlage hinterlegt, da es mehrmals zur gleichen Zeit mit unterschiedlichen Daten offen sein kann.

```
!!Modell für die Auflistung aller Objekte
model window MWnListe
{
  .userdata null;
  .active false;
  .xleft 78;
  .ytop 9;
  .title "Liste";
  .MethodenObjekt := null;
  .DargestelltesObjekt := null;
  .Dynamisch := true;
  .ZuAktivierendesObjekt := PNeu;
```

```
  child listbox LListe
  {
    .xauto 0;
    .xleft 3;
    .xright 3;
    .yauto 0;
    .ytop 0;
    .ybottom 1;
    .firstchar 1;
  }
  child pushbutton PNeu
  {
    .xleft 2;
    .yauto -1;
    .text "Neu";
  }
  child pushbutton PInfos
  {
    .sensitive false;
    .xleft 16;
    .yauto -1;
    .text "Informationen";
  }
  child pushbutton PLoeschen
  {
    .sensitive false;
    .xauto -1;
    .xright 13;
    .yauto -1;
    .text "L\366schen";
  }
  child MPOK
  {
  }
}
```

Abbildung 9: *Auftragsliste*

Zu diesem Fenster gehören die Regeln, die die Liste füllen können. Dazu wird über die jeweilige Objektstruktur gegangen und aus den vorhandenen Objekten die gewünschte Textinformation extrahiert und zur Anzeige gebracht. Dazu wird von den Objekten jeweils die Methode Rlist aufgerufen, die aus den internen Datenstrukturen die externe Repräsentation des Objektes generiert.

```
!!Aufbau des Anzeigestrings für die Auftragsliste
rule string RListAuftrag (object Auftrag input)
{
    return ((Auftrag.AuftragsNr + "    ") +
      Auftrag.Artikel);
}

!!Aufbau des Anzeigestrings für die Kundenliste
rule string RListKunde (object Kunde input)
{
    return ((((Kunde.KundenNr + "  ") + Kunde.Firma) +
      "   ") + Kunde.Ort);
}

!! Regel zum Füllen der Liste der Objekte
rule void ListeFuellen (object ObjektInfo input,
object Target input)
{
  variable string String;
  variable integer I;
!! Setzen des Fenstertitels
  Target.title := ObjektInfo.MRcMethoden.TList;
  !! Löschen aller vorhandenen Einträge
  Target.LListe.itemcount := 0;
  !! Insensitiv Schalten der Pushbuttons für Löschen und
  !! Detailinformationen
  Target.PLoeschen.sensitive := false;
```

```
  Target.PInfos.sensitive := false;
  !! Durchlaufen der Liste von Objekten und Einträgen in
  !! die Auswahlliste
  for I := 1 to ObjektInfo.MRcSpeicher.count[.Elemente] do
    !! Überprüfen, ob überhaupt ein Objekt gespeichert ist
    if (ObjektInfo.MRcSpeicher.Elemente[I] <> null) then
      !! String berechnen lassen
      String := ObjektInfo.MRcMethoden.RList
        (ObjektInfo.MRcSpeicher.Elemente[I]);
      !! String übernehmen
      if (String <> "") then
        Target.LListe.content[(Target.LListe.itemcount +
           1)] := String;
      endif
    endif
  endfor
}
```

Neben dieser Auflistungsfunktionalität gehören zu diesem Fenster noch Regeln, die die Selektierbarkeit der Pushbuttons steuern. Die Pushbuttons sollen nur selektierbar sein, wenn ein Eintrag in der Liste selektiert ist.

```
!!In Liste wird etwas selektiert, darum werden die
!!Pushbuttons "Löschen" und "Informationen" freigeschaltet
on LListe select
{
  this.window.PLoeschen.sensitive := true;
  this.window.PInfos.sensitive := true;
}
```

Zum Neuanlegen von Objekten wird das zugehörige Fenster ohne Inhalt auf den Bildschirm gebracht. Diese Aktion wird durch die Selektion des "Neu"-Pushbuttons ausgelöst.

```
!!Regel zum Neuanlegen eines beliebigen Objektes
rule void ObjektNeuAnlegen (object ObjektInfo input, object Info input)
{
  variable object Obj;

  !! Generieren des Fenster / Suchen nach vorhandenem
  !! Fenster
  Obj := FensterNeuAnlegen(ObjektInfo,
    ObjektInfo.MRcMethoden.Create, Info);
  !! Daten in Fenster setzen
  Rl_AnzeigeHandhaben(Obj.MethodenObjekt, Obj, true);
  !! Objekt sichtbar machen
  Obj.visible := true;
}
```

```
!!Regel zum Neuanlegen eines beliebigen Objekts
on PNeu select
{
  ObjektNeuAnlegen(this.window.MethodenObjekt, null);
}
```

Um bestehende Werte ändern zu können, wird auf die Selektion in der Listbox mit einem Doppelklick bzw. durch einfache Selektion des "Informationen"-Pushbutton durch Anzeige der entsprechenden Daten reagiert.

```
!!Information soll angezeigt werden.
on LListe dbselect
{
  InfosZeigen(this.window);
}

!!Information soll angezeigt werden.
on PInfos select
{
  InfosZeigen(this.window);
}

!!Regel zum Anzeigen der Information zu einem Objekt
rule void InfosZeigen (object Fenster input)
{
  variable object Obj;

  !! Überprüfen, ob ein Eintrag in der Liste selektiert
  !! ist.
  if (Fenster.LListe.activeitem <> 0) then
    !! Fenster anlegen und anzeigen
    Obj := FensterNeuAnlegen(Fenster.MethodenObjekt,
    Fenster.MethodenObjekt.MRcMethoden.Create,
      Fenster.MethodenObjekt.MRcSpeicher.Elemente
       [Fenster.LListe.activeitem]);
  Rl_AnzeigeHandhaben(Obj.MethodenObjekt, Obj, true);
  Obj.visible := true;
  endif
}
```

Beim Löschen über den "Löschen" Pushbutton werden auch die internen Strukturen entsprechend verändert.

```
!!Ein Objekt aus der Liste soll gelöscht werden
on PLoeschen select
{
  variable integer I;
```

```
  !! Durchsuchen aller Fenster, ob das zu löschende Objekt
  !! noch irgendwo angezeigt wird.
  for I := 1 to this.dialog.childcount do
    !! überprüfen, ob das Kind ein Fenster ist
    if this.dialog.child[I].class = window then
      !! Vergleichen des MethodenObjekts und des
      !! DargestelltenObjekts
      if this.dialog.child[I].MethodenObjekt =
        this.window.MethodenObjekt then
          if this.dialog.child[I].DargestelltesObjekt =
          this.window.MethodenObjekt.MRcSpeicher.Elemente
            [this.window.LListe.activeitem] then
            !! Ausblenden des Fensters
            this.dialog.child[I].visible := false;
          endif
      endif
    endif
  endfor
  if (this.window.LListe.activeitem > 0) then
  !! Zerstören der internen Struktur zu dem Fenster
  destroy(this.window.MethodenObjekt.MRcSpeicher.Elemente
    [this.window.LListe.activeitem], true);
  !! Schließen des zugehörigen Fensters
  this.window.MethodenObjekt.MRcSpeicher.Elemente
    [this.window.LListe.activeitem] := null;
  !! Neuaufbau der Liste
  ListeFuellen(this.window.MethodenObjekt, this.window);
  endif
}
```

4.3.4.4 Das Kundenfenster

Im Kundenfenster werden die direkt zu einem Kunden gehörenden Informationen wie Firmenname, Anschrift, Ansprechpartner und Telefon dargestellt. Dabei können die Daten in das Fenster neu eingegeben oder bestehende Daten verändert werden. Bevor dieses Fenster definiert wird, werden noch Vorlagen für die auch bei den Aufträgen auftretenden Pushbutton "Detailinformationen", "OK" und "Abbruch" definiert.

```
!!Modell für den Pushbutton, der Detailinformationen anfordert
model pushbutton MPDetail
{
  .yauto -1;
}

!!Modell für den OK Button
model pushbutton MPOK
```

```
{
  .xauto -1;
  .yauto -1;
  .text "OK";
}

!!Modell für den Abbrechen Pushbutton
model pushbutton MPAbbruch
{
  .xauto -1;
  .xright 13;
  .yauto -1;
  .text "Abbruch";
}

!!Meldungsfenster für noch nicht realisierte Teile im
!! Dialog.
messagebox Meldung
{
  .text "Noch nicht implementiert!";
  .title "Meldungsfenster";
  .button[2] nobutton;
}

!!Modell für das Kundeninformationsfenster
model window MWnKunden
{
  .userdata null;
  .active false;
  .xleft 467;
  .ytop 113;
  .height 13;
  .title "Kundeninformation";
  .MethodenObjekt := null;
  .DargestelltesObjekt := null;
  .Dynamisch := true;
  .ZuAktivierendesObjekt := EtKundenNr;
  child statictext
  {
    .xleft 2;
    .ytop 0;
    .text "Kunden-Nummer:";
  }
  child statictext
  {
    .xleft 2;
```

```
  .ytop 1;
  .text "Firma:";
}
child statictext
{
  .xleft 2;
  .ytop 2;
  .text "Strasse:";
}
child statictext
{
  .xleft 2;
  .ytop 3;
  .text "PLZ:";
}
child statictext
{
  .xleft 24;
  .width 0;
  .ytop 3;
  .height 0;
  .text "Ort:";
}
child statictext
{
  .xleft 2;
  .ytop 5;
  .text "Ansprechpartner:";
}
child statictext
{
  .xleft 2;
  .ytop 6;
  .text "Telefon:";
}
child statictext
{
  .xleft 2;
  .ytop 7;
  .text "Fax-Nummer:";
}
child MPDetail PAuftraege
{
  .xleft 2;
  .yauto -1;
  .text "Auftr\344ge";
```

```
  .AttributIndex := 2;
}
child MPDetail
{
  .xleft 20;
  .text "Rechnungen";
  .AttributIndex := 1;
}
child MPOK
{
}
child edittext EtKundenNr
{
  .xleft 15;
  .width 41;
  .ytop 0;
  .height 0;
  .Datenelement := .KundenNr;
}
child edittext EtFirma
{
  .xleft 15;
  .width 41;
  .ytop 1;
  .height 0;
  .Datenelement := .Firma;
}
child edittext EtStrasse
{
  .xleft 15;
  .width 41;
  .ytop 2;
  .height 0;
  .Datenelement := .Strasse;
}
child edittext EtPlz
{
  .xleft 15;
  .width 7;
  .ytop 3;
  .height 0;
  .Datenelement := .Plz;
}
child edittext EtOrt
{
  .xleft 28;
```

```
    .width 27;
    .ytop 3;
    .Datenelement := .Ort;
  }
  child edittext EPartner
  {
    .xleft 18;
    .width 38;
    .ytop 5;
    .Datenelement := .Partner;
  }
  child edittext EtTelefon
  {
    .xleft 18;
    .width 24;
    .ytop 6;
    .Datenelement := .Telefon;
  }
  child edittext EtFax
  {
    .active false;
    .xleft 18;
    .width 24;
    .ytop 7;
    .content "";
    .Datenelement := .Fax;
  }
  child MPAbbruch
  {
  }
}
```

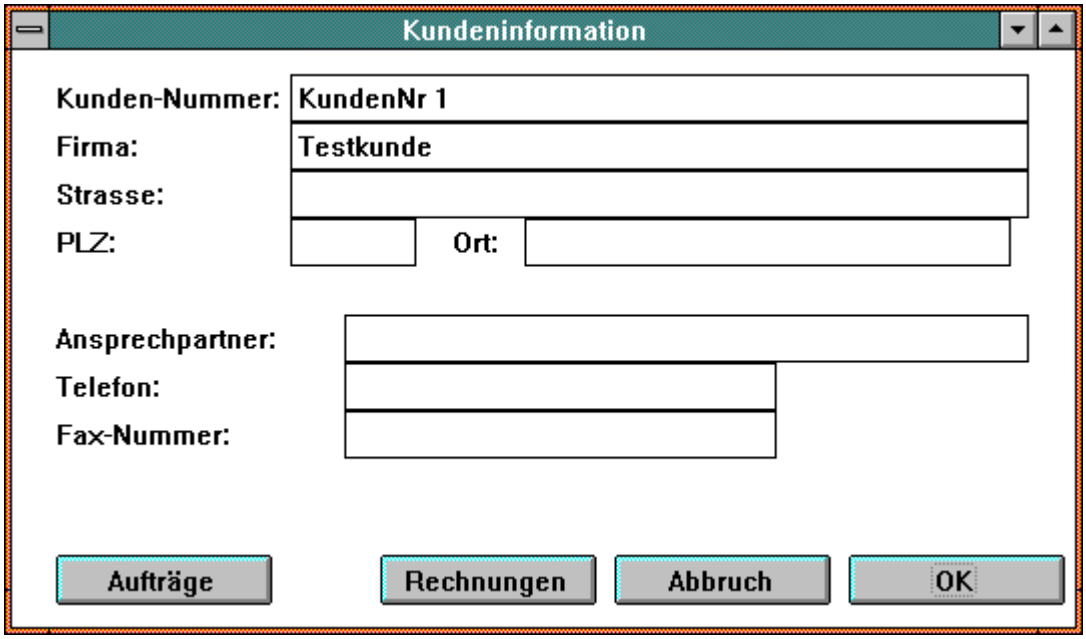

Abbildung 10: *Kundeninformation*

Durch Selektion des Pushbutton "Rechnungen" werden die zu dem Kunden gehörenden Rechnungen in einem Fenster dargestellt. Die Querverbindung zu den Aufträgen ist in diesem Prototyp nicht realisiert, deshalb erscheint bei der Selektion des Pushbutton "Aufträge" eine entsprechende Meldung.

```
!!Regel zum Handhaben der Detailinformation 1
rule void Detail1Handhaben (object ObjektInfo input, object Fenster input,
boolean Display input)
{
  variable integer I;
  variable object TFObj;

  !! Suchen nach dem Tablefield
  for I := 1 to Fenster.childcount do
  if (Fenster.child[I].class = tablefield) then
    TFObj := Fenster.child[I];
  endif
  endfor
  !! Jetzt sollte das Tablefield aus der Datenbank gefüllt
  !! werden.
  if (TFObj <> null) then
    if ( not Display) then
    endif
  endif
}

!!Anzeigen der Detailinformationen zu den Objekten
on MPDetail select
```

```
{
  variable object Obj;
  !! Fenster anlegen und anzeigen
  Obj := FensterNeuAnlegen(this.window.MethodenObjekt,
    this.window.MethodenObjekt.MRcMethoden.Detail
      [this.AttributIndex],
    this.window.DargestelltesObjekt);
  Detail1Handhaben(Obj.MethodenObjekt, Obj, true);
  Obj.visible := true;
}
```

Wenn der "OK"-Pushbutton selektiert wird, werden die im Fenster dargestellten Daten in die internen Strukturen übernommen.

```
!!Selektion des OK Buttons soll die Werte übernehmen
on MPOK select
{
  Rl_AnzeigeHandhaben(this.window.MethodenObjekt,
    this.window, false);
  this.window.visible := false;
}
```

Wenn der "Abbruch"-Pushbutton selektiert wird, werden die geänderten Daten verworfen und das Fenster geschlossen.

```
!!Die Selektion von Abbruch bewirkt das Unsichtbarmachen
!!des zugehörigen Fensters
on MPAbbruch select
{
  this.window.visible := false;
}
```

4.3.4.5 Das Auftragsfenster

Im Auftragsfenster werden die direkt zu einem Kunden gehörenden Informationen wie Firmenname, Anschrift, Ansprechpartner und Telefon dargestellt. Dabei können die Daten in das Fenster neu eingegeben oder bestehende Daten verändert werden. Dieses Fenster verfügt nicht über eigene Regeln, die notwendigen Regeln sind bereits alle im Zusammenhang mit dem Kundenfenster entstanden.

```
!!Modell für das Auftragsfenster
model window MWnAuftrag
{
  .userdata null;
  .active false;
  .xleft 305;
  .ytop 44;
  .title "Auftragsinformation";
  .MethodenObjekt := null;
```

```
.DargestelltesObjekt := null;
.Dynamisch := true;
.ZuAktivierendesObjekt := EtAuftragsNr;
child statictext
{
  .xleft 2;
  .ytop 0;
  .text "Auftrags-Nummer:";
}
child statictext
{
  .xleft 2;
  .ytop 1;
  .text "Artikel:";
}
child statictext
{
  .xleft 2;
  .ytop 2;
  .text "Anzahl:";
}
child statictext
{
  .xleft 2;
  .ytop 3;
  .text "Fertigungs-Starttermin:";
}
child statictext
{
  .xleft 2;
  .ytop 4;
  .text "Fertigungs-Endetermin:";
}
child statictext
{
  .xleft 2;
  .ytop 5;
  .text "Liefertermin:";
}
child edittext EtAuftragsNr
{
  .xleft 16;
  .width 39;
  .ytop 0;
  .height 0;
  .Datenelement := .AuftragsNr;
```

```
}
child edittext EtArtikel
{
  .xleft 16;
  .width 38;
  .ytop 1;
  .height 0;
  .Datenelement := .Artikel;
}
child edittext EtAnzahl
{
  .xleft 16;
  .width 9;
  .ytop 2;
  .height 0;
  .Datenelement := .Anzahl;
}
child edittext EtFStart
{
  .xleft 21;
  .width 17;
  .ytop 3;
  .Datenelement := .FStart;
}
child edittext EtFEnde
{
  .xleft 21;
  .width 17;
  .ytop 4;
  .Datenelement := .FEnde;
}
child edittext EtLiefer
{
  .xleft 21;
  .width 17;
  .ytop 5;
  .Datenelement := .Liefer;
}
child MPAbbruch
{
}
child MPOK
{
}
child MPDetail PMaschinen
{
```

```
    .xleft 2;
    .yauto -1;
    .text "Maschinen";
    .AttributIndex := 2;
  }
  child MPDetail
  {
    .xleft 17;
    .text "Fert. Auftr\344ge";
    .AttributIndex := 1;
  }
}
```

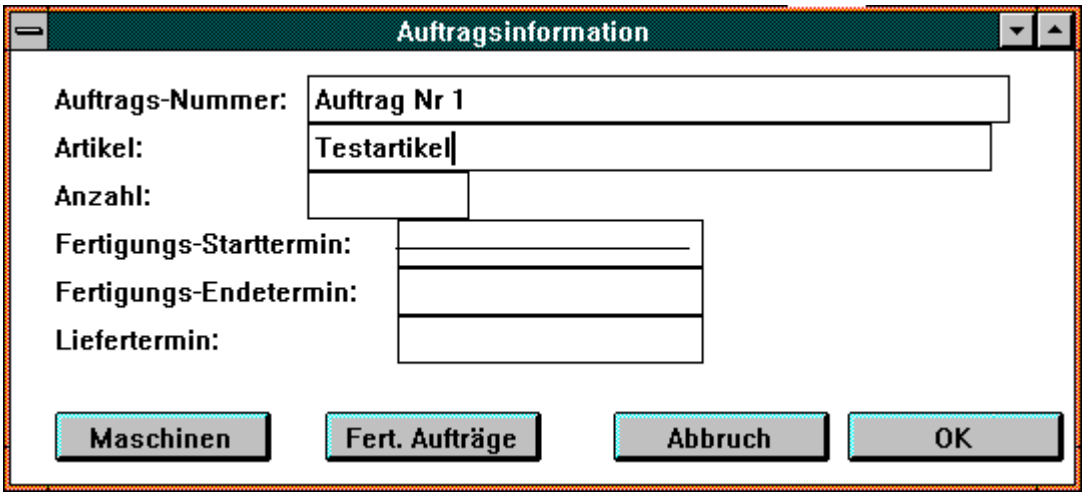

Abbildung 11: *Auftragsinformation*

4.3.4.6 Das Rechnungsfenster

Im Rechnungsfenster sollen die zu einem Kunden gehörenden Rechnungen in einem Tablefield dargestellt werden. Dieses ist in diesem Prototyp nicht implementiert worden. Durch die Verwendung von Modellen verfügt dieses Fenster über keine eigenen Regeln.

```
!!Modell für das Rechnungsfenster
model window MWnRechnung
{
  .userdata null;
  .active false;
  .xleft 414;
  .ytop 156;
  .title "Rechnungen";
  .MethodenObjekt := null;
  .DargestelltesObjekt := null;
  .Dynamisch := true;
  .ZuAktivierendesObjekt := MPOK;
  child tablefield TfRechnungen
```

```
  {
    .xauto 0;
    .xleft 3;
    .xright 3;
    .yauto 0;
    .ytop 0;
    .ybottom 2;
    .fieldshadow false;
    .borderwidth 2;
    .colcount 4;
    .rowcount 5;
    .rowheadshadow false;
    .rowheader 1;
    .colheadshadow false;
    .colheader 0;
    .colfirst 1;
    .colwidth[2] 8;
    .colalignment[2] 0;
    .colalignment[3] 1;
    .colwidth[4] 5;
    .colalignment[4] 0;
    .content[1,1] "Rechnungs-Nr";
    .content[1,2] "Datum";
    .content[1,3] "Betrag";
    .content[1,4] "Status";
  }
  child MPOK
  {
  }
  child MPAbbruch
  {
  }
}
```

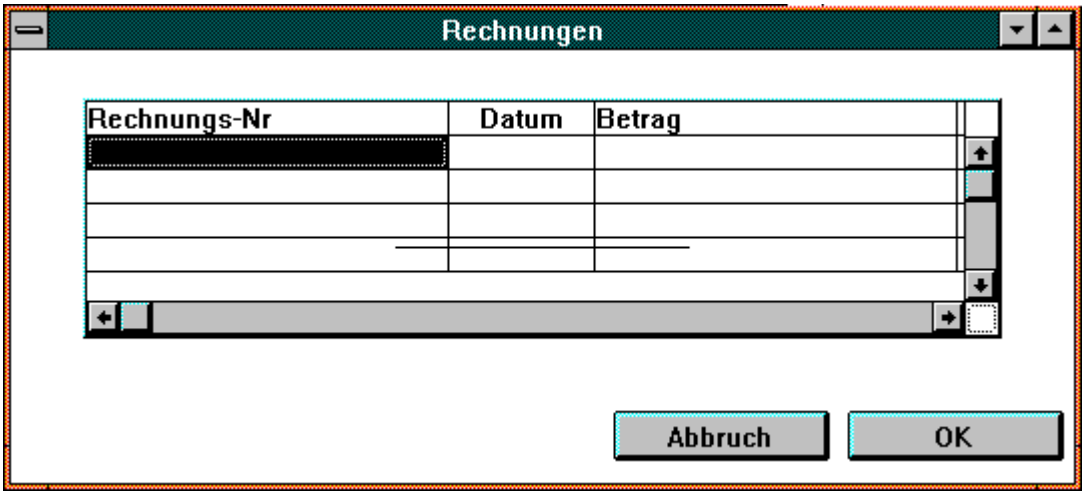

Abbildung 12: *Rechnungsfenster*

4.3.4.7 Das Fertigungsauftragsfenster

Im Fertigungsauftragsfenster sollen die zu einem Auftrag gehörenden Fertigungsaufträge in einem Tablefield dargestellt werden. Dieses ist in diesem Prototyp nicht implementiert worden. Durch die Verwendung von Modellen verfügt dieses Fenster über keine eigenen Regeln.

```
!!Modell für das Fenster für die Fertigungsaufträge
model window MWnFertigung
{
  .userdata null;
  .active false;
  .xleft 220;
  .ytop 55;
  .title "Fertigungsauftr\344ge";
  .MethodenObjekt := null;
  .DargestelltesObjekt := null;
  .Dynamisch := true;
  .ZuAktivierendesObjekt := MPOK;
  child MPAbbruch
  {
  }
  child MPOK
  {
  }
  child tablefield TfFertigung
  {
    .xauto 0;
    .xleft 3;
    .xright 3;
    .yauto 0;
    .ytop 0;
```

```
    .ybottom 2;
    .fieldshadow false;
    .borderwidth 2;
    .colcount 5;
    .rowcount 5;
    .rowheader 1;
    .colfirst 1;
    .colwidth[2] 10;
    .colwidth[3] 10;
    .colwidth[5] 15;
    .content[1,1] "Fertigungsauftrag";
    .content[1,2] "Starttermin";
    .content[1,3] "Endetermin";
    .content[1,4] "Maschine";
    .content[1,5] "Materialstatus";
  }
}
```

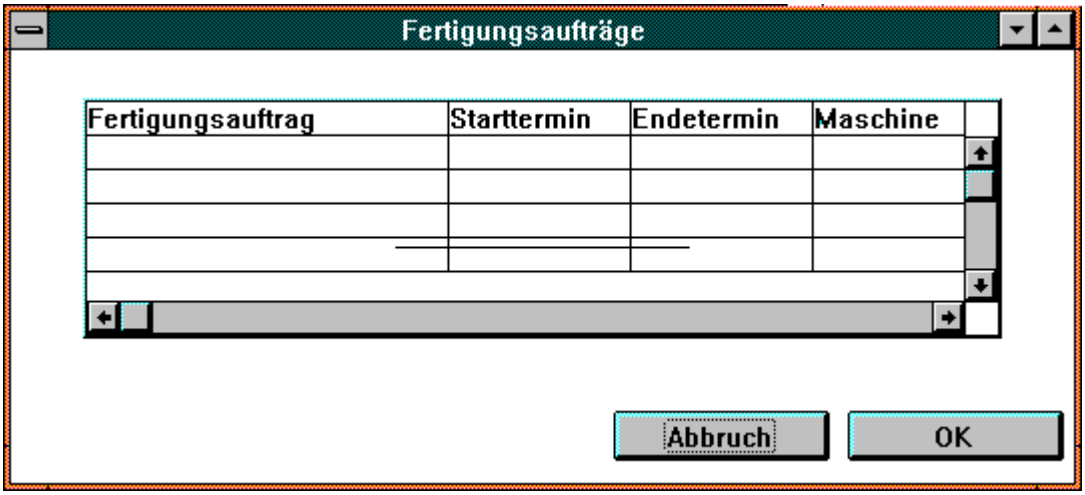

Abbildung 13: *Fertigungsfenster*

5 Modularisierung

Für die Entwicklung von Dialogen werden Sie mit der Modularisierung in die Lage versetzt, in Teams an einem Dialog zu arbeiten, wenn dieser sinnvoll in Teildialoge (sogenannte „Module“) zerlegt wird. Zusätzlich können Ressourcen, wie z.B. die Farben und Zeichensätze, von einer zentralen Stelle bereitgestellt und von allen Anwendungsentwicklern verwendet werden. Darüber hinaus können in solchen Teildialogen Vorlagen enthalten sein, die von allen Anwendern genutzt werden, so dass nicht mehr jeder Anwender eigene Basismodelle schaffen muss, sondern die durch eine zentrale Stelle geschaffenen Vorlagen einsetzen kann. Dadurch lässt sich auch die Realisierung eines Firmen-Styleguides wesentlich vereinfachen. Natürlich können in solchen Teildialogen auch Funktionalitäten hinterlegt werden, die immer wieder in den Dialogen auftauchen. Damit können solche Teilfunktionalitäten einfach wiederverwendet und zentral gepflegt werden. Weitere Anwendungsmöglichkeiten werden ausführlich im letzten Abschnitt beschrieben.

Neben diesen Aspekten bei der Dialogentwicklung hat die Modularisierung auch Auswirkungen auf das Laufzeitverhalten von Dialogen. Teildialoge können einzeln bei Bedarf geladen und wieder entladen werden. Dadurch werden Ladezeiten für den Benutzer scheinbar reduziert, da sie wesentlich besser über die Gesamtarbeitszeit mit dem Dialog verteilt werden können. Funktionen, die nur sehr selten benutzt werden, können so bei Bedarf geladen und nach ihrem Ende wieder entladen werden, um so den Speicherplatzbedarf des Dialogprozesses zu reduzieren.

Wenn große Dialoge jetzt mit Hilfe von Teildialogen realisiert werden, erhöht sich aufgrund der wesentlich geringeren Größe der Dialogdateien die Wartbarkeit solcher modularisierten Systeme. Die kleineren Dialogeinheiten sind für den Einzelnen wesentlich leichter zu überschauen und zu pflegen.

Neben diesen eben beschriebenen Vorteilen der Modularisierung gibt es noch einen weiteren Aspekt beim Einsatz von Modulen in Verbindung mit dem „Verteilten Dialog Manager“ (DDM): Man kann mit Hilfe solcher Teildialoge die Anzahl der Netzwerkverbindungen zwischen Client und Server reduzieren. Wenn man mehrere eigenständige Dialoge hat, besitzt jeder dieser Dialoge eine Netzwerkverbindung, falls man den verteilten Dialog Manager einsetzt. Werden diese Dialoge in Module umgestaltet, so kommt der PC mit einer Verbindung zum Server und damit auch mit einem einzigen Serverprozess aus.

5.1 Umsetzung der Modularisierung

Ein Dialog kann in mehrere Module zerfallen. Wenn man sich nicht-modularisierte Dialoge betrachtet, haben diese ungefähr den folgenden Aufbau:

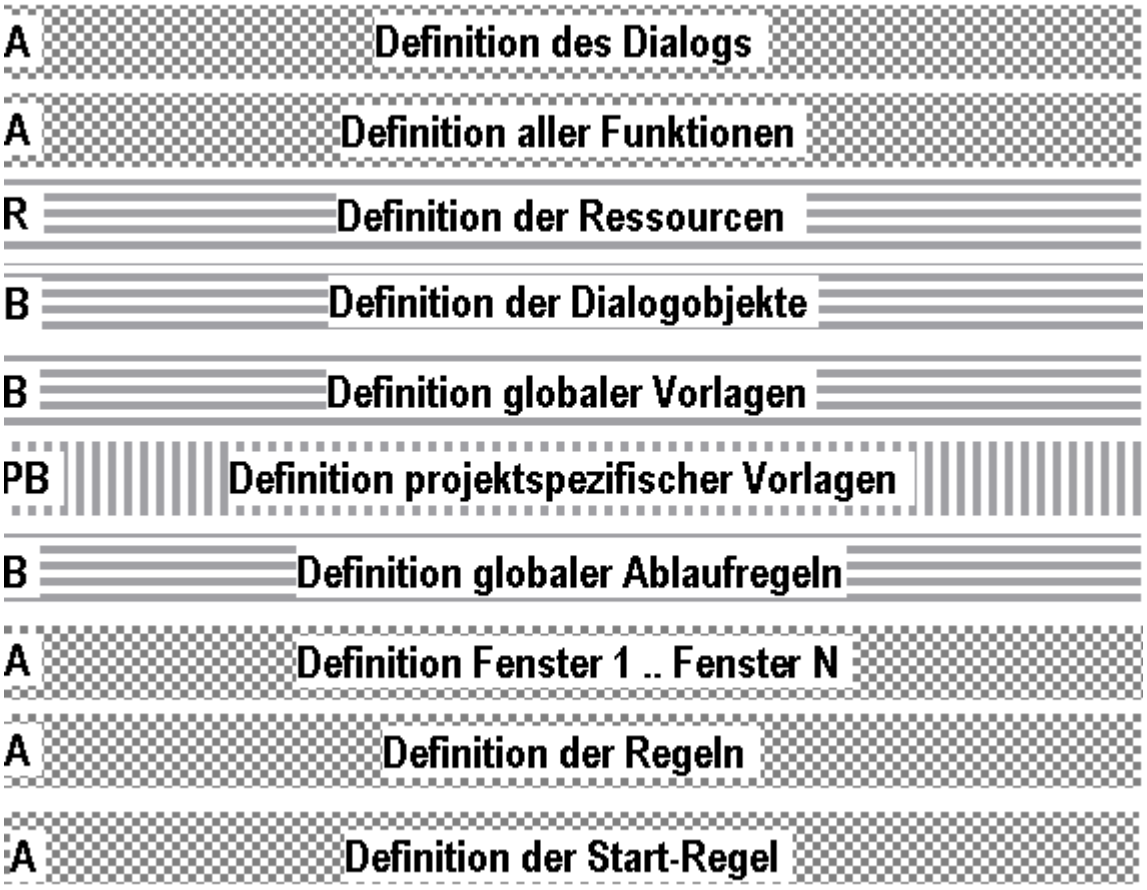

Abbildung 14: *Struktur eines Dialogs*

Dabei bedeutet

- A: Anwendungsspezifische Dialogteile
- B: Immer wieder auftauchende Vorlagenteile
- PB: Projektspezifische Dialogteile, die in mehreren Dialogen auftauchen
- R: Ressourcen, die immer auf der Plattform eingesetzt werden

Damit können also die Teile, die mit B, PB und R gekennzeichnet sind, als Teile bezeichnet werden, die im Normalfall wiederverwendet bzw. global zur Verfügung gestellt werden sollten.

Genau dieses wird jetzt durch die Modularisierung unterstützt. Aus diesen gekennzeichneten Teilen werden in sich abgeschlossene Teildialoge (Module) gebaut, die auch von verschiedenen Entwicklern bearbeitet werden können.

Dann hat der Dialog folgendes internes Aussehen:

Abbildung 15: *Struktur eines Dialogs mit Modulen*

Jedes dieser Module wird in einer Datei abgespeichert (binär oder als Skript). Dadurch werden die einzelnen Dateigrößen reduziert und können so besser von den Entwicklern gewartet werden.

Wenn ohne Module gearbeitet wird, ergibt sich eine Objekthierarchie, die mit dem Dialog als Wurzel beginnt und mit den Objekten in den einzelnen Fenstern endet. Diese Struktur wird dann direkt in einer Datei gespeichert.

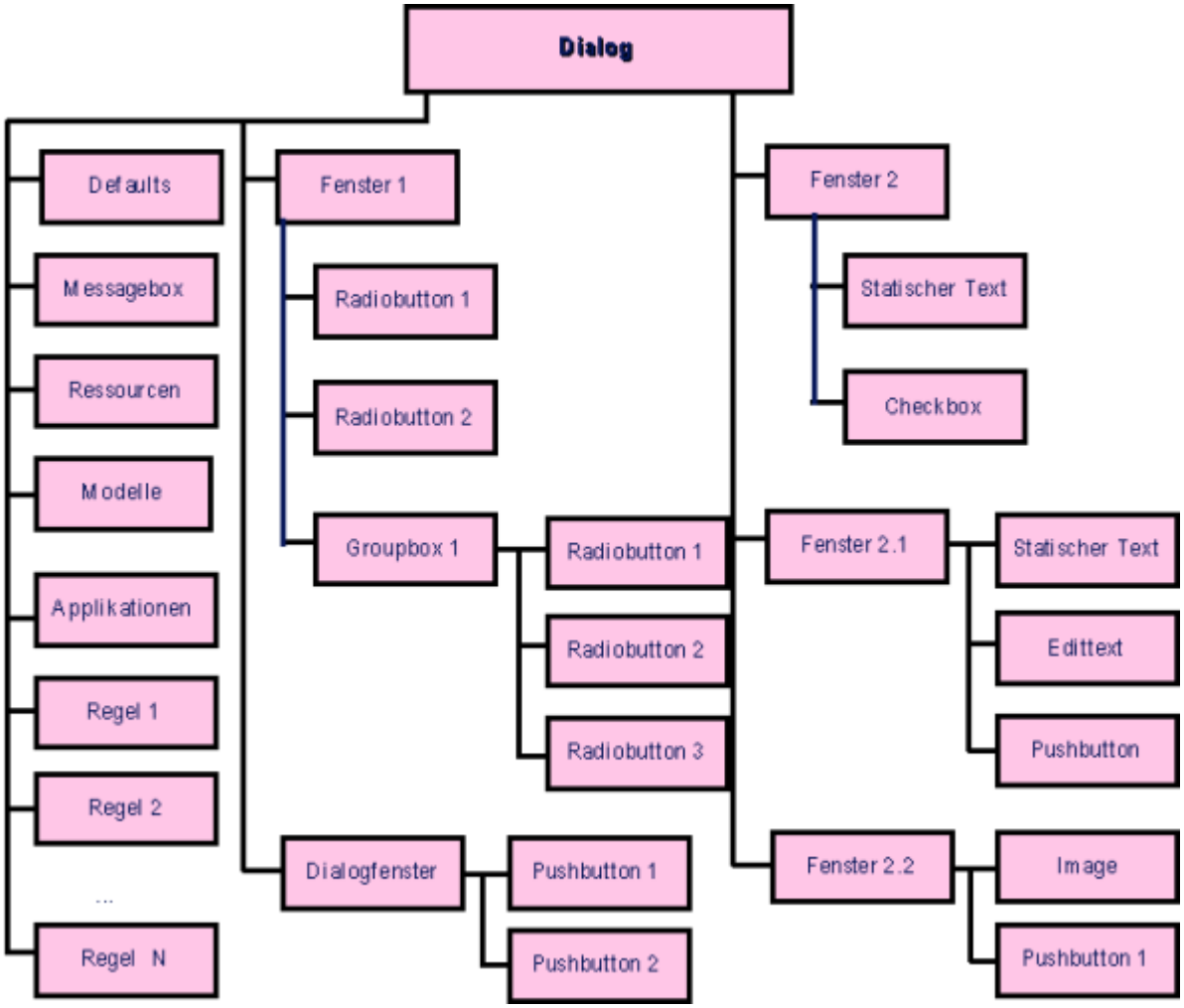

Abbildung 16: *Hierarchie eines Dialogs ohne Module*

Da dieser Dialog in verschiedene Module aufgeteilt wird, werden die Hierarchieebenen vergrößert, denn zusätzlich kommt die Hierarchieebene des Moduls hinzu. Die Module werden aber in getrennten Dateien gespeichert, so dass dadurch die ganze Struktur wesentlich übersichtlicher wird.

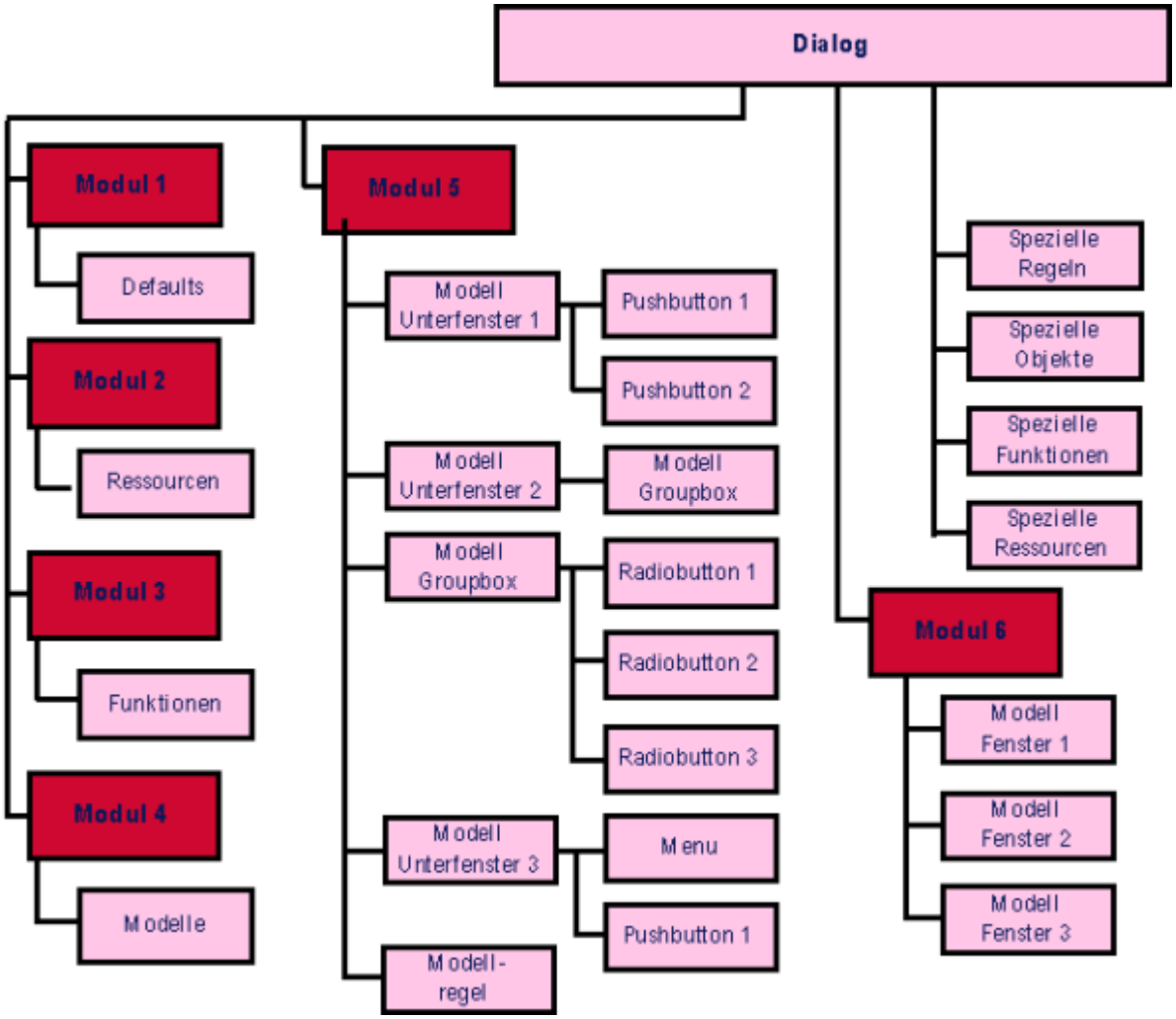

Abbildung 17: *Hierarchie eines Dialogs mit Modulen*

5.2 Sprachbeschreibung

5.2.1 Schlüsselwörter

Für die Modularisierung existieren in der Dialogbeschreibungssprache einige Elemente. Diese werden im Folgenden hier kurz vorgestellt.

Im Einzelnen sind folgende Objekte bzw. Attribute in der Regelsprache verfügbar:

Objekte

- **module**
- **import**
- **use** (ab IDM-Version A.06.02.g)

Attribute

- **export**
- **reexport** (ab IDM-Version A.05.02.j)
- **use**

Mit Hilfe dieser Schlüsselwörter wird dem Modul bekanntgegeben, welche Objekte nach außen transparent sind und welche nur für modul-interne Zwecke definiert sind. Die Erklärung dieser Attribute folgt in den nachfolgenden Kapiteln.

5.2.2 Das Modul

Ein Modul ist eine für sich geschlossene Einheit und kann mit dem DM-Objekt Dialog verglichen werden. Es bildet die Klammer für alle in ihm enthaltenen Objekte, so dass diese in einer Datei abgespeichert werden können. Die Datei, die eine Definition eines Moduls beinhaltet, wird durch das Schlüsselwort **module** eingeleitet - wie eine Dialogdatei mit **dialog**. Die Syntax ist identisch mit der Definition eines Dialogs.

Auf das Schlüsselwort **module** folgt der eindeutige Name des Moduls.

In einem Modul können nun beliebige Ressourcen, Regeln, Defaults, Vorlagen oder Instanzen definiert werden, die die Funktionalität des Moduls darstellen. Diese werden wie beim Objekt ***dialog*** definiert.

In den folgenden Beispielen wurde weitgehend auf Attribute verzichtet, die mit dem eigentlichen Verständnis des Beispiels nichts zu tun haben. Fehlende Attribute wurden durch 3 Punkte (...) dargestellt.

Beispiel

```
module TestExample

model window MyModel
{
  child pushbutton MyPushbutton
  {
  ...
  }
}
```

In diesem Beispiel wird ein Modul definiert, das lediglich eine in dem Modul verfügbare Vorlage definiert.

5.2.2.1 Ereignisse des Objekts module

Genau wie bei einem Dialog können für das Modul eine Start- und eine Ende-Regel definiert werden. Diese werden wie beim Dialog durch die Schlüsselworte **start** und **finish** eingegeben.

Regeln für das Hilfe-Ereignis (help) und das Key-Ereignis (key), die beim Dialog möglich sind, sind an diesem Objekt nicht verfügbar.

Beispiel

```
module SimpleModule

on SimpleModule start
{
  print "start";
}

on SimpleModule finish
{
  print "finish";
}
```

Wie aus dem Beispiel ersichtlich ist, können Ereignisse für Module definiert werden.

Für Module gelten die gleichen Ereignisse wie für Dialoge - bis auf die o.g. Ereignisse help und key.

Statt dem logischen Namen in Regeln kann auch das Schlüsselwort **module** verwendet werden. Damit sieht das obige Beispiel wie folgt aus:

Beispiel

```
module SimpleModule

on module start
{
  print "start";
}
on module finish
{
  print "finish";
}
```

Die beiden Beispiele haben die identischen Aktionen zur Folge.

5.2.2.2 Kinder des Objekts module

Im Gegensatz zum Dialog kann das Modul beliebige Objekte als Kinder aufnehmen. Ob das syntaktisch richtig ist, kann erst durch die Verwendung der Kinder in dem aufrufenden Modul festgestellt werden. Damit ist es also möglich, auch nicht-Top-Level-Objekte als Kinder des Moduls zu definieren, wenn diese bei ihrer Verwendung in den Objektbaum richtig eingepasst werden.

Beispiel

```
module Objectcollection
```

```
pushbutton MyPush1 {}

window MyWindow1 {}

listbox Myist1 {}
```

5.2.2.3 Attribute des Objekts module

Das Objekt ***module*** verfügt über keine Attribute, die in der Regelsprache abgefragt werden können.

5.2.3 Exportieren von Objekten

Damit Objekte außerhalb eines Moduls, in dem sie definiert wurden, ansprechbar sind, müssen diese durch spezielle Schlüsselwörter gekennzeichnet werden. Nur Objekte, die mit den Schlüsselwörtern **export** oder **reexport** gekennzeichnet sind, sind nach außen transparent und stellen somit die Schnittstelle zu diesem Modul dar. Export und reexport unterscheiden sich darin, wie die Export-Eigenschaft bei hierarchischen Kindern vererbt wird. Im Unterschied zu export wird bei reexport die Export-Eigenschaft der Kindobjekte eines Modells vererbt. Bei export muss ein Kindobjekt, das nach außen sichtbar sein soll, auf jeder Stufe der Vererbungshierarchie explizit exportiert werden. Reexport muss beim Modell am Kind- und Vaterobjekt angegeben werden. Bei einem aus dem Modell abgeleiteten Objekt muss es dann nur noch am Vater angegeben werden. Am geerbten Kind des abgeleiteten Objekts ist – im Gegensatz zu export – kein reexport mehr notwendig.

Mit Hilfe dieser Schlüsselwörter können alle Objekte im Sinne des Dialog Managers, also Ressourcen, Defaults, Vorlagen, Instanzen und benannte Regeln, nach außen verfügbar gemacht werden. Damit ist es möglich, alle Objekte, die einen Namen tragen, aus dem Modul zu exportieren und damit für die Benutzer des Moduls verwendbar zu machen.

Auf einzelne Attribute können **export** und **reexport** nicht angewendet werden, d.h. es werden immer ganze Objekte und nicht einzelne Attribute dieser Objekte verfügbar gemacht. In Dialogen selbst dürfen die Schlüsselwörter **export** und **reexport** nicht vorkommen, da Dialoge nicht in anderen Modulen verwendet werden können. Exportierte Objekte müssen einen eindeutigen Namen besitzen. Eine Ausnahme bilden hiervon die Defaults, diese können auch ohne Namen exportiert werden.

Beispiel 1

```
!! Colordefinition
module Color
export color Red rgb(255, 0, 0 );
export color Green rgb(0, 255, 0 );
export color Blue rgb(0, 0, 255 );
export color Yellow rgb(0, 255, 255 );
color TestRed rgb(240, 40, 50 );
```

Im Beispiel macht der Programmierer des Moduls Color nur vier Farben für den Anwender des Moduls transparent. Die fünfte Farbe ist zwar im Modul selbst vorhanden und wäre dort auch anwendbar, aber außerhalb sind nur die mit **export** gekennzeichneten Ressourcen und Objekte bekannt.

Beispiel 2

```
module WindowWithButton

export window Window
{
  child pushbutton PushMe { }
  child pushbutton Button { }
}
```

In diesem Beispiel ist für den Anwender nur das Fenster selbst bekannt; die Kinder (hier: PushMe, Button) sind nicht bekannt. Deshalb können die Kinder außerhalb dieses Moduls nicht verwendet werden. Soll ein Kind auch außerhalb des Moduls benutzt werden dürfen, muss ebenfalls ein **export** vor dieses Kind geschrieben werden:

```
module WindowWithButton

export window Window
{
  child pushbutton PushMe        { }
  export child pushbutton Button { }
}
```

In der modifizierten Version des Moduls WindowWithButton - jetzt stimmt der Name: es ist ein Fenster mit einem Pushbutton - sind für den Anwender die Objekte Window und Button transparent, d.h. er kann es in seinem eigenen Dialog (oder Modul) benutzen. Auf den Pushbutton PushMe hat er aber immer noch keinen Zugriff.

Um ein Kind zu exportieren, muss auch dessen Vater exportiert sein. Ist das nicht der Fall, wird auch das Kind nicht exportiert und die Schlüsselwörter **export** und **reexport** werden ignoriert.

Im folgenden Beispiel wird **export** ignoriert:

```
!! export is ignored
module WindowWithButton

window Window
{
  child pushbutton PushMe        { }
  export child pushbutton Button { }
}
```

Regeln, die direkt an ein Objekt gebunden sind, also mit dem Schlüsselwort **on** beginnen, können genau wie das Objekt ***module*** nicht exportiert werden.

5.3 Import mit use

Verfügbarkeit

Ab IDM-Version A.06.02.g

5.3.1 Der alternative Import-Mechanismus

Neben dem Weg, auf ein Modul über ein ***Import***-Objekt zuzugreifen, kann das Schlüsselwort **use** genutzt werden um den Umgang mit Imports, Modulen, Interface- und Binärdateien zu erleichtern.

Im Gegensatz zum Import erfolgt die Auswahl des Moduls nicht durch einen Dateipfad auf die Interface-Datei, sondern über einen Bezeichner-Pfad – genauer gesagt dem **Use-Pfad**.

Der letzte Bezeichner im Use-Pfad ist, analog zum Bezeichner eines Imports, der Vater-Bezeichner mit dem die exportierten Objekte im importierenden Modul bekannt gemacht werden. Die vorderen Bezeichner im Use-Pfad definieren eine „Paket-Hierarchie“ und können somit zur Untergliederung der Module genutzt werden. Bei der Evaluation von Objekt-Pfaden bleiben diese Teile unberücksichtigt.

Beispiel

```
!! file: Main.dlg

dialog MainDlg

use Models;
use Base.Fonts;

MWiMain WiMain {
  .font Fonts.FnBig;
}
```

```
!! file: Models.mod

module ModModels

export window MWiMain {
  on close {
    exit();
  }
}
```

```
!! file: Base/Fonts.mod

module ModFonts

export font FnBig "24.Arial";
```

Die Verzeichnisse, in denen die eigentliche Modul- und Interface-Datei gesucht werden, können als Startoption oder eine spezielle Umgebungsvariable, über einen Auf ruf von **DM_ControlEx()** oder in der Regelsprache gesetzt werden.

Das Vorhandensein einer Interface-Datei ist dabei nicht zwingend notwendig, aber sinnvoll um das Laden von Modulen ohne direkte Nutzung zu erleichtern und das Nachladen der Implementierung bei Bedarf zu unterstützen.

5.3.1.1 Besonderheiten

Module die über **use** angezogen werden, sind im Dialog immer nur einmal geladen. Dieser Umstand erleichtert die Verwendung von Basismodulen, da nicht versehentlich durch eine fehlerhafte Import-Kette ein Modul mehrfach geladen wird.

Die Entkopplung von einem Interface- oder Modul-Dateinamen hat mehrere Vorteile:

1. Die Verwendung von eigenen Umgebungsvariablen für den Ort der Interface- und Binärmodule entfällt.
2. Die Erzeugung der Interface- und Binärdateien wird vereinfacht.

3. Das Laden von Modulen ohne Interface-Datei ist nun möglich, da ein zentraler Suchpfad für Interfaces, Module und Binärmodule vorhanden ist.

Eine weitere Besonderheit ist die Vorgabe von Dateiendungen (Datei-Suffixe) für Quelltext-Dateien (*.dlg* und *.mod*), Interface-Dateien (*.if*) und Binärdateien (*.bin*).

Die Umsetzung von einem Use-Pfad zu einem Dateipfad geschieht dabei wie folgt: Die vorangestellten Paketbezeichner entsprechen einem Verzeichnispfad, der letzte Bezeichner dem Basisnamen des Moduls.

Das heißt im Beispiel oben ergibt sich für den Use-Pfad `Base.Fonts` der Basis-Dateiname „Base\Fonts“. Das Quelltext-Modul heißt *Base\Fonts.mod*, die dazugehörige Interface-Datei *Base\Fonts.if* und das Binärmodul *Base\Fonts.bin*.

Wird nun über `use Base.Fonts` ein Modul angezogen, dann wird im Suchpfad nach diesen drei Dateinamen gesucht um das Interface bzw. die Implementierung zu laden. In der Entwicklungsversion des IDM wird dabei in der Reihenfolge Interface-Datei → Quelltextdatei → Binärmodul vorgegangen. Die Laufzeitversion des IDM kann nur Binärdateien laden.

Einige weitere Unterschiede zum ***Import***-Objekt sind noch vorhanden: Es gibt keine Steuermöglichkeiten über die Attribute *.application*, *.static* und *.load*. Es wird immer versucht, erst das Interface heranzuziehen und dann die Implementierung (Quelltext, Binär). Die Implementierung wird nur wenn benötigt nachgeladen. Ein direkte Steuerung des Ladens entfällt somit.

Für ein dynamisches Laden und Entladen gibt es die neuen Methoden **:use()** und **:unuse()**.

Grundsätzlich können Binärmodule und Interface-Dateien, die für die Nutzung durch ein ***Import***-Objekt erzeugt wurden, unverändert auch mit **use** genutzt werden. Umgekehrt gilt dies nicht zwangsläufig da hier meist der Hinweis auf den Modulpfad im Interface fehlt.

5.3.1.2 Groß- und Kleinschreibung in Dateinamen

Sie ist im IDM bei Bezeichnern und somit auch beim Use-Pfad sowie für die Dateinamen von Bedeutung!

5.3.1.3 Empfehlungen

Zwar ist eine gemischte Nutzung von **import** und **use** möglich, aber **nicht** empfehlenswert, da das ***Import***-Objekt ein mehrfaches Laden von Modulen provoziert. Deshalb wird empfohlen, komplett von **import** auf **use** umzustellen.

Außerdem wird empfohlen , für Datei- und Verzeichnisnamen grundsätzlich die gleiche Schreibweise wie im Use-Pfad zu wählen. Am besten sollten Module einen eindeutigen Namen besitzen, der auch nicht in einer abgewandelten Form mit anderer Groß- und Kleinschreibung auftaucht.

Statt ein Modul über **import** zu laden, sollte man Modelle verwenden, die problemlos mehrfach instanziiert werden können.

5.3.2 Sprachbeschreibung und Use-Pfad

Folgende Sprachbeschreibung gilt für das Schlüsselwort **use**:

```
{ export | reexport } use { <Use-Pfad> }

<Use-Pfad>   ::= [ <Paket-Pfad> ] <Bezeichner>
<Paket-Pfad> ::= [ <Paket-Pfad> ] <Bezeichner> .
```

Die **Bezeichner** sollten mit einem Großbuchstaben anfangen ('A' – 'Z') und kann danach auch Kleinbuchstaben, Ziffern und Unterstriche ('_') enthalten. Ein Bezeichner darf maximal 31 Zeichen haben.

Beispiele für gültige use-Verwendungen

```
use Defaults;
use DEFAULTS;
use Max_Wi09_;
use Modul78.Aa.MasterA__1z;
use BaseMod.M_ods.Wins.MWiEnter1_9;
```

Der **Use-Pfad** besteht dabei aus einem Bezeichner (Identifikator) am Ende und repräsentiert den Basisnamen des Moduls, sowie den Vater-Bezeichner für den weiteren Zugriff auf die exportierten Objekte (falls diese nicht eindeutig aufzulösen sind). Dieser Bezeichner muss nicht dem Modulbezeichner entsprechen.

Der **Paket-Pfad** wiederum dient zur logischen Untergliederung von mehreren Modulen und wird bei der Suche nach dem Modul als Verzeichniskette vorangestellt.

5.3.3 Use-Pfad, Dateinamen und Namensrestriktionen

Der IDM konvertiert intern einen Use-Pfad in einen Dateipfad, um diesen bei der Suche von Interface-Datei, Quelltextdatei oder Binärmodul zu verwenden.

So wird der Use-Pfad `Modul78.Aa.MasterA__1z` zum Dateipfad *Modul78\Aa\MasterA__1z*.

Außerdem gibt es einen vorgegebenen Satz von Dateiendungen, die bei Suche und Zugriff relevant sind und intern an den Basis-Dateipfad angehängt werden:

Dateiendung	Bedeutung
.dlg	Quelltext eines Dialogs
.mod	Quelltext eines Moduls
.if	Interface-Datei
.bin	Binärdatei eines Dialogs oder Moduls

Da der Use-Pfad über IDM-Bezeichner aufgebaut ist, bedeutet dies natürlich, dass Datei- und Verzeichnisnamen mit einem Großbuchstaben beginnen können sowie die Groß- und Kleinschreibung eine Bedeutung hat und wichtig ist! Dies ist bei der Nutzung auf Dateisystemen, die dies nicht unterscheiden, zu beachten.

Das heißt im IDM gilt: `use Default;` ist etwas anderes als `use DEFAULT;`. Unter Windows kann das Modul aber durchaus bei Vorhandensein der Modul-Datei „DEFault.Mod“ gefunden werden.

Grundsätzlich kann auch der Dialog über einen Use-Pfad angesprochen (geladen) werden.

Falls unüberbrückbare Probleme bei der Umwandlung zwischen Use-Pfad und Dateipfad bestehen, kann das interne Konvertierungsverhalten über die Option **-IDMusepathmodifier** gesteuert werden.

5.3.4 Suchpfad für Interface-, Modul-, Dialog- und Binärdateien

Der IDM besitzt einen zentralen Suchpfad für das Auffinden von IDM-Dateien. Dieser Suchpfad kann eine oder mehrere absolute oder relative Verzeichnisangaben mit Semikolon („;“) getrennt beinhalten.

Standardmäßig ist der Suchpfad auf „~;“ gesetzt, also auf den Sonderpfad „~“ und den Leerpfad *""*. Diese haben folgende Bedeutung:

~ oder *~:*	Sucht unterhalb des Verzeichnisses, in dem sich die Applikation befindet.
"" (Leerpfad)	Sucht im aktuellen Arbeitsverzeichnis (Verhalten wie in den Vorgängerversionen).
<ENVNAME>:	Sucht in den Pfaden, die in der Umgebungsvariablen `<ENVNAME>` definiert sind.

Bei der Suche nach einer IDM-Datei, z.B. über **DM_LoadDialog()** oder die eingebaute Funktion **load ()**, wird wie folgt vorgegangen:

1. Wenn der angegebene Pfad absolut ist, wird die Datei an dem exakt angegebenen Pfad gesucht ohne das Anhängen einer Erweiterung (kompatibles Verhalten wie bisher).
2. Handelt es sich um einen relativen Pfad mit Dateierweiterung, wird dieser im Suchpfad mit exakt dieser Erweiterung gesucht. Dabei stellt der Standard-Eintrag *""* (Leerpfad) das kompatible Verhalten wie bisher dar.
3. Ist am Dateinamen keine Erweiterung vorhanden, werden je nach Aktion die möglichen Erweiterungen bei allen Suchpfaden durchprobiert. Zum Beispiel werden bei **DM_LoadDialog()** die Erweiterungen *.dlg*, *.mod*, *.bin* durchprobiert. Bei einem „use“ wird versucht, mit Vorrang das Interface zu bekommen, also wird die Reihenfolge *.if*, *.bin*, *.mod* durchprobiert.

Folgende Einstellmöglichkeiten für den Suchpfad sind vorhanden:

- Beim Start einer IDM-Anwendung über die Option **-IDMsearchpath <Suchpfad>**.
- Über die Umgebungsvariable `IDM_SEARCHPATH`.
- Durch den Aufruf von **DM_ControlEx()** mit *DMF_SetSearchPath* und dem Suchpfad als *data*-

Argument.

» Über das Attribut *.searchpath* am ***Setup***-Objekt.

Für eine selbst kompilierte IDM-Anwendung empfehlen wir die Nutzung von **DM_ControlEx()** in der **AppMain()**-Routine mit Angabe eines Applikations-relativen Pfades über „~“ und ohne andere relative Angaben wie den Leerpfad oder *"."*. So würde die Ausführung von *DM_ControlEx((DM_ID)0, DMF_SetSearchPath, "~:dlg")* sicherstellen, dass die Dialogdateien nur im Unterverzeichnis *dlg* parallel zur Anwendung gesucht werden. Ist der Suchpfad erst einmal gesetzt, kann dann auch über **DM_LoadDialog()** das initiale Laden des Hauptdialogs über einen relativen Dateipfad ohne Dateiendung erfolgen.

Beispiele für korrekte und erlaubte Suchpfade

"" "."	Sucht im aktuellen Arbeitsverzeichnis.
"~"	Sucht im Verzeichnis in dem sich die Applikation befindet.
"if;bin;mods;customer/modules"	
"~:dlg;~:../mods;."	
".;MODPATH:if;MODPATH:bin"	Sucht im Arbeitsverzeichnis sowie in den Verzeichnissen *bin* und *if* innerhalb der Verzeichnisse, die in der Umgebungsvariablen `MODPATH` angegeben sind.

Beim Öffnen und Laden von IDM-Dateien versucht der IDM den Dateityp festzustellen, um zu erkennen, ob die Datei eine Interface-Datei, ein Dialog, Modul oder eine Binärdatei ist. Zu diesem Zweck untersucht er die ersten 1.024 Bytes der Datei, ob sie die Signatur für eine IDM-Binärdatei hat, oder entsprechende Schlüsselwörter die ein Interface, Dialog oder Modul kennzeichnen. Andernfalls wird die Dateiendung zur Bestimmung herangezogen.

5.4 Vergleich zwischen import und use

Funktion, Konzept	use	import
Mehrfaches Laden von Modulen.	**Nicht** möglich. Modul ist in einem Dialog immer nur einmal geladen (gilt nur für den Zugriff mit „use“).	Mehrfaches Laden des gleichen Moduls durch anderen Import-Bezeichner bzw. bei Lücken in der Import-Hierarchie der importierenden Module bis hoch zum Dialog möglich.
Ankopplung von Funktionen an ein Applikationsobjekt.	Über das *.masterapplication[enum]*-Attribut möglich.	Über das *.application*-Attribut möglich.

Funktion, Konzept	use	import
Erzeugung von Interface- und Binärdateien in einem Schritt.	Über **-compile** bzw **-recompile** möglich. Der Dialog sollte aber alle Module über „use“ angezogen haben.	**Nicht** möglich.
Quelltext, Interface und Binärmodul im gleichen Verzeichnis.	**Ja**, möglich durch die unterschiedlichen Dateiendungen.	**Nicht** möglich.
Verteilung der Module in Unterverzeichnisse.	**Ja**, über Paketpfad (Teil des Use-Pfads) oder über den Suchpfad.	**Ja**, durch Suchsymbol, Pfadlisten in Such-Variablen und als Auflistung in Interface- und Moduldatei.

5.5 Interface und Binärdateien bei Verwendung von import

5.5.1 Vom Modul zum Interface

Will ein Programmierer die Funktionalität eines Moduls in seinem Dialog (oder Modul) benutzen, muss er die Namen der darin enthaltenen exportierten Objekte kennen.

Um dies zu vereinfachen, ist die Simulation des Dialog Managers in der Lage, aus der Beschreibung eines Moduls eine sogenannte "Interface-Datei" zu generieren. Damit werden dem Benutzer des Moduls die Namen der in dem Modul exportierten Objekte bekannt gemacht. Der Anwender kann auf diese Objekte per Namen zugreifen. Die eigentliche Realisierung dieser Objekte bleibt vor dem Anwender verborgen. Damit ist der Moduldesigner in der Lage, die reale Implementierung seiner Objekte beliebig zu ändern, solange die nach außen bekannte Schnittstelle - diese Interface-Datei - nicht verändert wird.

Die eigentlichen Objekte werden erst zur Laufzeit bzw. beim Erstellen der Binärdatei(en) durch den Dialog Manager dazu gelinkt, so dass erst ab diesem Zeitpunkt die Ausprägung der Objekte bekannt ist.

Mit der Startoption

+writeexport

kann der Entwickler automatisch die Interface-Datei aus seinem Modul erstellen. Dabei werden Kommentare, die mit "!!" vor den eigentlichen Objekten stehen, mit in die Interface-Datei übernommen. Auf diese Art und Weise kann eine Kommentierung der Dialogsourcen auch für den Anwender eines Moduls verfügbar gemacht werden.

Die Syntax für diesen Aufruf sieht wie folgt aus:

```
idm +writeexport <Exportdatei-Name> <Modul-Name>
```

Beispiel (die Datei "color.mod")

```
module Color
!! Provision of the color red
export color Red rgb(255, 0, 0 );
export color Green rgb(0, 255, 0 );
export color Blue rgb(0, 0, 255 );
!! Provision of the color yellow
export color Yellow rgb(0, 255, 255 );
color TestRed rgb(240, 40, 50 );
```

Um die Interface-Datei zu erzeugen, wird der Simulator mit der Option **+writeexport** aufgerufen:

```
$idm +writeexport color.if color.mod
```

Die daraus generierte Datei "color.if" hat nach der Ausführung des Kommandos **+writeexport** folgendes Aussehen:

```
interface of module Color "color.mod"
!! Provision of the color red
color Red;
color Green;
color Blue;
!! Provision of the color yellow
color Yellow;
```

Aus der Datei "color.if" kann der Anwender die nötige Information vom Modul "color.mod" holen, ohne zu wissen, wie die Farben definiert wurden. Damit steht zum Beispiel einer firmenweiten, standardisierten Farbenbibliothek nichts im Wege.

Die Kinderhierarchie bleibt ebenfalls erhalten, wie auch im folgenden Ausschnitt eines Moduls zu erkennen ist:

Ausschnitt aus dem Modul mit einer Fenstervorlage:

```
...
export model window W
{
  export child pushbutton P { }
}
...
```

Ausschnitt aus dessen Interface-Datei:

```
...
model window W
{
  child pushbutton P;
}
...
```

5.5.2 Vom Interface zum Modul

Die Dateinamen der Interface-Dateien müssen eindeutig sein, sonst überschreiben nachfolgend erstellte Interface-Dateien vorangegangene Interface-Dateien.

Die Dateien werden im aktuellen Verzeichnis gesucht. Werden sie nicht gefunden, bricht der Dialog Manager mit einer Fehlermeldung ab. Um die Module selbst unabhängig vom aktuellen Verzeichnis zu verwalten, können bei der Erstellung von Interface-Dateien Suchsymbole angegeben werden. Diese Suchsymbole sind eventuell schon aus den Funktionen zum Laden von Dialogen oder Profiles sowie bei der Lokalisierung von Bildern geläufig. Die Symbole werden - durch einen Doppelpunkt getrennt - dem eigentlichen Dateinamen vorangestellt. Der Dialog Manager interpretiert diese Symbole als Umgebungsvariablen und deren Inhalt als Suchpfad für die angegebene Datei. Unter Betriebssystemen, die einzelne Laufwerke mit Buchstaben gefolgt von einem Doppelpunkt unterscheiden (wie z.B. Microsoft Windows), müssen die Umgebungsvariablen aus mindestens zwei Buchstaben bestehen.

Das Modul wird dann in dem Pfad gesucht, der über solche Umgebungsvariablen definiert wird. Wird in einem Verzeichnis die entsprechende Datei gefunden, wird diese geladen und die weitere Suche abgebrochen.

Beispiel

```
$set IDMLIB=/usr/idm/lib:/usr/idm/ISAstandard:
/usr/idm/lib/dia:/home/project/lib
```

Damit nun diese Umgebungsvariable beim Laden der Module beachtet wird, muss beim Erstellen der Interface-Datei zusätzlich die Option

+searchsymbol IDMLIB

angegeben werden, damit das Symbol IDMLIB allen Dateinamen vorangestellt wird.

Die Syntax lautet also allgemein:

```
idm +writeexport <Exportdatei-Name> +searchsymbol
  <Umgebungsvariable> <Modul-Name>
```

Beispiel

```
idm +writeexport color.if +searchsymbol IDMLIB color.mod
```

Das Ergebnis (die Interface-Datei "color.if") sieht dann mit dem obigen Modulbeispiel Color folgendermaßen aus:

```
interface of module Color "IDMLIB:color.mod"

color Red;
color Green;
color Blue;
color Yellow;
```

Der Dialog Manager durchsucht die in der Umgebungsvariablen angegebenen Verzeichnisse nach dem Modul "color.mod".

5.5.3 Module in Module importieren

Die Verwendung eines Moduls in einem anderen Modul oder Dialog wird mit dem Schlüsselwort

import

angekündigt. Auf das Schlüsselwort folgt der logische Name des Moduls, unter dem es bekannt sein soll. Der logische Name ist im Allgemeinen nicht der Name des Moduls selbst. Unter Umständen kann sogar ein und dasselbe Modul unter verschiedenen logischen Namen in einem Modul verwendet werden, obwohl dies allgemein besser und eleganter durch Vorlagen beschrieben werden kann. Auf den logischen Namen des importierten Moduls folgt dann ein String mit dem Namen der Interface-Datei.

Beispiel aus einem Modul

```
module Models

import StandardColor "color.if";
import Colors "mycolor.if";

export model pushbutton MPB1
{
  .fgc Green;   // Farbe aus StandardColors
  .bgc MyGreen; // Farbe aus Colors
}
export model pushbutton MPB2
{
  // Doppelte Namen können durch Voranstellung des
  // Modulnamens unterschieden werden
  .bgc StandardColors.Red;
  .fgc Colors.Red;
}
```

Module, die Module importieren, können auch von anderen Modulen wieder importiert werden. Dadurch entsteht eine Art Baum mit dem Dialog selbst als Wurzel.

Die Imports eines Moduls können exportiert werden. Dies hat zur Folge, dass Module, die dieses Modul mit den exportierten Imports importieren, ebenfalls diese Imports besitzen.

```
!! Interfacefile: module.if
module Module
export import StandardColor "color.if";
...
module Main
/*
  import StandardColor "color.if";
  Dieses Modul muss hier nicht importiert werden, da es
  indirekt durch die Verwendung des Farbenmoduls geladen
  wird
*/
```

```
import ImportModule "module.if";
...
```

Sollen die Interface-Dateien nicht im gleichen Verzeichnis wie die Module stehen, können die Dateinamen am Import mit einem Suchsymbol versehen werden. Dieses Suchsymbol steht für eine Umgebungsvariable, die einen Suchpfad enthält (siehe auch Abschnitt oben).

Beispiel aus einem Modul, das seine Interface-Dateien in einem anderen Verzeichnis suchen soll. Die Umgebungsvariablen sind auf folgende Werte gesetzt:

```
$set IDMLIB=/usr/idm/lib:/usr/idm/ISAstandard:
  /usr/idm/lib/dia
```

und

```
$set PRIVLIB=/home/project/lib
```

```
module Models

import StandardColor "IDMLIB:resrc/color.if";
  // Die Datei color.if wird in folgenden Verzeichnissen
  // gesucht
  // /usr/idm/lib/resrc
  // /usr/idm/ISAstandard/resrc
  // /usr/idm/lib/dia/resrc
import Colors "PRIVLIB:mycolor.if";
  // Die Datei mycolor.if wird nur im Verzeichnis
  // /home/project/lib gesucht

export model pushbutton MPB1
{
  .fgc Green; // Farbe aus StandardColors
  .bgc MyGreen;// Farbe aus Colors
}
export model pushbutton MPB2
{
.bgc StandardColors.Red;
  .fgc Colors.Red;
}
```

5.5.4 Benutzen der Objekte – use

Objekte aus Modulen können nicht nur angesprochen oder Attributen zugewiesen werden, sondern auch als Kinder von Objekten aus anderen Modulen benutzt werden. Dies wird mit dem Schlüsselwort

use

angekündigt. Die Attribute dieses Kindes können nicht mehr direkt bei der Verwendung des Objektes geändert werden. Zur Laufzeit ist dies natürlich weiterhin in den Regeln möglich.

Beispiel

```
module Childlibrary

export pushbutton Exit
{
  .text "End";
}
on Exit select { exit(); }

dialog Main
import ChildLib "childlib.if" { .load false; }

window Window
{
  child pushbutton Action { .text "Action ..."; }
  use child Exit;  // change of attributes impossible
}
on Action select
{
  Exit.text := "Exit";  // change of attributes at runtime
}
...
```

Ein Kind kann immer nur **einmal** verwendet werden, d.h. Objekte können nur an maximal einer Stelle über das Schlüsselwort **use** in den bestehenden Objektbaum eingefügt werden. Sollen Objekte mehrfach verwendet werden, müssen wie bisher auch Vorlagen benutzt werden.

Hinweis

Im allgemeinen sind Vorlagen die bessere Wahl, in Ausnahmefällen können aber über **use** eingebundene Objekte von Vorteil sein, z.B. für die Aufteilung der Kinder eines Fensters an verschiedene Programmierer (z.B. Fenster als solches und ein Tablefield-Objekt).

5.5.5 Binärdateien

Dialoge und Module liegen dem Entwickler als ASCII-Dateien vor. An den Endkunden wird nur die binär kodierte Form der Module ausgeliefert. Die Laufzeit-Version (Runtime) kann nur diese binär kodierten Module, d.h. die Binärdateien lesen.

Die Module werden einzeln in Binärdateien übersetzt. Die Übersetzung erfolgt mit der Option

+writebin

des Dialog Managers:

```
idm +writebin <Binärdatei-Name> <Dialogdatei-Name>
```

Die Binärdatei ist eine identische Übersetzung des Moduls in eine maschinenlesbare Form. Der Ladevorgang des Dialogs wird dadurch wesentlich beschleunigt.

Binärdateien und Module in ASCII-Format können durchaus gemischt werden. Ändert sich aber eine Schnittstelle (Interface-Datei), müssen die abhängigen Module erneut in Binärform übersetzt werden. Ändern sich in Modulen Vorlagen oder Defaults, müssen die abhängigen(!) Module ebenfalls erneut in Binärform übersetzt werden. Man kann daher solche Modul-Dateien mit C-Header-Dateien vergleichen. Wenn sich eine C-Header-Datei ändert, müssen alle davon abhängigen Dateien auch neu übersetzt werden, damit wieder eine einheitliche Version vorliegt. Genauso verhält es sich auch mit den Binärdateien der Module.

Empfehlung

Die oben genannten Abhängigkeiten beim Binärschreiben können durch sogenannte "Makefiles" oder ähnliche Instrumente der Softwareentwicklung am besten beschrieben und automatisch erfasst werden.

5.6 Kompilieren von Interface- und Binärdateien für Imports mit use

Während der Entwicklung einer IDM-Applikation und der Nutzung mit Quelltext-Dateien dienen die Interface-Dateien als Repräsentation der extern verfügbaren (exportierten) Objekte eines Moduls, also der Schnittstelle nach außen. Durch die Trennung von Schnittstellen-Definiton und Implementierung ist der IDM in der Lage, das Laden der Implementierung (also des Moduls) nur bei Bedarf durchzuführen.

Durch „use“ kann diese Schnittstellen-Definition auch direkt aus dem Modul, also der Implementierung erfolgen. Aus oben genannten Grund macht es aber dennoch Sinn Interface-Dateien zu nutzen.

Für die Auslieferung einer IDM-Anwendung an den Endkunden bedarf es einer Binärfassung der Dialoge und Module.

Grundsätzlich können Interface- und Binärdateien auf dem herkömmlichen Weg über die Aktionen **-writeexport** und **-writebin** erzeugt wegen. Solange diese Dateien mit der richtigen Dateiendung an der richtigen Stelle im Suchpfad stehen, hat die Anweisung „use“ damit kein Problem.

Für modularisierte Dialoge die „use“ verwenden gibt es viel einfachere Aktionen, um alle Interface- und Binärdateien in einem Schritt zu erzeugen.

-compile	Erneuert alle Interface- und Binärdateien für geänderte Module und Dialoge.
-recompile	Erzeugt alle Interface- und Binärdateien neu.
-cleancompile	Löscht alle Interface- und Binärdateien.

Dazu muss der Dialog fehlerlos geladen werden können. Erzeugt werden Interface- und Binärdateien nur für den angegebenen Dialog und alle per „use“ geladenen Module. Für Module, die per „import“ angezogen werden, wird **keine** Interface- und Binärdatei erzeugt!

Die zu ladenden Modul- und Dialogdateien müssen als Quelltext vorliegen. Bei einem Ladefehler werden keine Dateien erzeugt.

Beispiel

```
!! file: Main.dlg

dialog MainDlg

use Models;
use Base.Fonts;

MWiMain WiMain {
  .font Fonts.FnBig;
}
```

```
!! file: Models.mod

module ModModels

export window MWiMain {
  on close {
    exit();
  }
}
```

```
!! file: Base/Fonts.mod

module ModFonts

export font FnBig "24.Arial";
```

Über folgendes Kommando kann man in einem Schritt alle Interface- und Binärdateien erstellen:

```
pidm -compile Main
```

Nun sollten die folgenden Dateien parallel zu den Quelltext-Dateien stehen:

```
Main.bin
Models.bin
Models.if
Base/Fonts.bin
Base/Fonts.if
```

Fügt man nun in der Datei *Base/Fonts.mod* noch die Zeile *export font FnSmall…* hinzu, so aktualisiert man einfach mit folgendem Kommando:

```
pidm -compile Main
```

Was zum erneuten Schreiben der folgenden Dateien führt:

```
Base/Fonts.bin
Base/Fonts.if
```

Die anderen Interface- und Binärdateien werden nicht neu geschrieben, da das Dateidatum dieser Dateien neuer ist als das der entsprechenden Quelldatei.

Möchte man auf jeden Fall alle Interface- und Binärdateien neu erzeugen lassen, so geht dies über:

```
pidm -recompile Main
```

Möchte man alle Interface- und Binärdateien löschen, so geht dies über:

```
pidm -cleancompile Main
```

Um die Interface- und Binärdateien in ein anderes Verzeichnis hinein zu erzeugen, dienen die Optionen **-ifdir <Verzeichnispfad>** und **-bindir <Verzeichnispfad>**.

5.7 Dynamische Modulverwaltung

5.7.1 Bei Verwendung von import

5.7.1.1 Ladevorgang

Ein Modul kann in ein anderes Modul über mehrere Arten geladen werden. Unter Laden versteht man hier, dass nicht nur die Namen der Objekte, sondern auch deren Definition dem Modul bekannt sind und damit darstellbar und veränderbar sind. Dabei werden intern drei verschiedene Arten des Laden unterschieden, die zum Teil bereits durch die im Modul definierten bzw. exportierten Objekte bestimmt werden.

- **load on use**
  Das Modul wird automatisch sofort geladen. Der Programmierer hat keinen Einfluss darauf, wann das Modul geladen wird, da Objekte aus dem Modul sofort (Defaults) und andere beim Einlesen des übergeordneten Moduls gebraucht werden. Dies ist der Fall, wenn das Modul exportierte benötigte Vorlagen oder Ressourcen enthält.
- **implicit load**
  Das Modul wird erst geladen, wenn ein exportiertes Objekt aus dem Modul wirklich benötigt wird. Dies ist zum Beispiel dann der Fall, wenn auf das Objekt in einer Regel per Zuweisung oder Abfrage zugegriffen wird.
- **explicit load**
  In diesem Fall entscheidet der Dialogdesigner selbst, dass er jetzt das Modul geladen haben möchte. Er setzt das Attribut *.load* am logischen Modulnamen (Importname) auf *true*. Daraufhin wird das Modul vom Dialog Manager geladen und verfügbar gemacht.

Beispiel

```
module Modul
import Colors "color.if";

window W
{
  .bgc Green;           // load on use
}
on W focus
{
  this.bgc := Red;      // implicit load
}
on W select
{
```

```
  Colors.load := true;  // explicit load
}
```

Load on use hat Vorrang vor den beiden anderen Arten des Ladens. Ebenso hat **implicit load** Vorrang vor **explicit load**. D.h. im obigen Beispiel wird ein **load on use** durchgeführt, die beiden anderen haben keine Auswirkungen mehr, da das Modul sich bereits im Speicher befindet und die Regeln direkt ausgeführt werden können.

Ein **explicit load** kann bereits als Attribut beim Import angegeben werden.

```
module Modul
import Colors "color.if"
{
  .load true;  // explicit load
}
```

Damit nicht bei jedem einzelnen Import das Modul neu in den Speicher geladen wird, kann ein Modul mit anderen Imports geteilt werden. Dies sollte die Regel sein und nur in begründeten Ausnahmen sollte der Programmierer von diesem Vorgehen abweichen. In diesen Fällen scheint eine Überarbeitung des Designs angebracht. Wie oben bereits erwähnt, besitzt jeder Import einen logischen Namen. Besitzt ein übergeordnetes Supermodul (d.h. in der Importhierarchie näher zum Dialog) einen Import mit dem gleichen logischen Namen wie ein untergeordnetes Submodul, wird das Modul nur einmal geladen.

```
module M1
import Colors "color.if";
...
module M2
import Colors "color.if";
  // "M1.if" is the interface-file of module M1
import Modul1 "M1.if";
```

Im Beispiel oben wird das Modul „color“ nur einmal unter dem logischen Namen „Colors“ geladen. Im unteren Beispiel hingegen wird das Modul (unnötigerweise!) zweimal geladen. Und zwar einmal unter dem logischen Namen „Colors“ und unter „Colros“ – man beachte die große Auswirkung dieses (hier: absichtlichen) Tippfehlers.

```
module M1
import Colors "color.if";
...

module M2
import Colros "color.if";
import Modul1 "M1.if";
  // "M1.if" ist die Interface-Datei von Modul M1
```

Eine ungewollte Verdoppelung kann aber bereits durch die Reihenfolge der Imports passieren. Wird im folgenden Beispiel (unten) das Modul M1 sofort geladen (**load on use**), kann unter Umständen der Import Farben von M1 vor(!) den Farben von M2 gelesen werden. Da aber M2 auch M1 lädt, ist

der Import von Farben zu diesem Zeitpunkt nur M1 bekannt und somit wird das Modul Color zweimal geladen.

```
module M1
import Colors "color.if";
...

module M2
  // "M1.if" is the interface-file of module M1
import Modul1 "M1.if;
import Colors "color.if";
```

Anmerkung

Es darf kein **start()** auf Module bzw. deren logischen Namen ausgeführt werden.

5.7.1.2 Entladevorgang

Ein nicht mehr benutztes Modul kann über seinen Import (logischer Name) wieder aus dem Speicher entfernt werden. Dies kann nur über das Attribut *.load* am Import geschehen. Der Wert des Attributs wird auf *false* gesetzt.

Man beachte dabei aber, dass nicht jedes Modul wieder entladen werden kann. Im Allgemeinen ist dies nur dann möglich, wenn das geladene Modul nur exportierte Objekte, aber keine exportierten Ressourcen, Vorlagen oder Defaults enthält. Physikalisch aus dem Speicher entfernt wird das Modul jedoch erst, wenn jeder Import mit dem entsprechenden logischen Namen dieses Modul entladen hat.

Ein Modul kann nicht mehr entladen werden, wenn das Attribut

```
.static true;
```

am Objekt ***import*** gesetzt ist. Dieses Attribut hat auch Auswirkungen auf die Erstellung der Binärdateien. Die über *.static true* angezogene Datei wird beim Binärschreiben in die aufrufende Datei mit hineinkopiert.

Der Entladevorgang ist rechenintensiv und sollte nur gezielt eingesetzt werden, um z.B. Speicherplatzprobleme zu lösen.

Anmerkung

Die Funktion **stop()** kann auf ein Modul nicht ausgeführt werden.

5.7.2 Bei Verwendung von use

Für die dynamische Nutzung von Modulen gibt es zwei Methoden:

- **:use()**
  Um ein Modul zugänglich zu machen und wenn nötig zu laden. Dies entspricht dem `use`-Kommando in einem Dialog oder Modul.
- **:unuse()**

Dient dazu die use-Beziehung wieder aufzulösen, also das Modul zu „entladen“, wobei das Modul erst dann komplett entfernt wird, wenn es keine weiteren use-Beziehungen (z.B. von anderen importierenden Modulen) gibt.

Beispiel

```
!! file DynLoad.dlg
dialog Dlg

window Wi
{
  pushbutton PbUnload
  {
    .yauto -1;
    .text "Unload";
    .sensitive false;

    on select
    {
      this.dialog:unuse("DynMod");  // Entladen des Moduls DynMod
      this.sensitive := false;
    }
  }

  on close { exit(); }
}

on dialog start
{
  variable object Module, Model, Child;

  Module := this:use("DynMod");  // Laden des Moduls DynMod
  if Module<>null then
    PbUnload.sensitive := true;
    Model := parsepath("MPbBeep");
    if Model<>null then
      Child := create(Model, Wi);
    endif
  else
    print "Cannot load DynMod module!";
    exit();
  endif
}

!! file DynMod.mod
module ModDyn

export model pushbutton MPbBeep
```

```
{
  .text "Beep";

  on select
  {
    beep();
  }
}
```

5.8 Objekt Application

Das Objekt ***application*** dient zur Anbindung von Clients an Server. Dieses Verfahren kann auch mit Modulen eingesetzt werden. Ein Problem besteht jedoch darin, dass ein und dasselbe Modul bei verschiedenen Dialogen verwendet werden kann. Dabei kann es passieren, dass diesen Dialogen verschiedene Serverprozesse zugeordnet sind. Aus diesem Grund kann bei der Verwendung eines Moduls definiert werden, wo dessen Funktionen zu suchen sind. Dies wird dem Modul über das Objekt ***import*** mitgeteilt. Dort wird definiert, welches Applikationsobjekt für die in dem Modul vorhandenen Funktionen zuständig ist.

Beispiel:

```
dialog Main
application MyServer
{
  ...
}
import Modul_X "modulX.if"
{
  .application MyServer;
}
...
```

Jede Funktion des Moduls Modul_X wird nun auf dem Server aufgerufen. Damit kann das Modul, das in der Interface-Datei "modulX.if" beschrieben ist, unabhängig von der jeweiligen Serverseite programmiert werden.

Des Weiteren können Funktionen aus Modulen exportiert werden, ohne dass die dazugehörige Applikation mit exportiert werden muss. Dies ist auch die einzige Ausnahme, bei der der Vater eines Objekts nicht mit exportiert werden muss. Diese Vorgehensweise hat den Vorteil, dass der Designer des Moduls immer weiß, auf welchem Rechner seine Funktionen ablaufen - damit verwehrt man aber auch jegliche Chance, von außen Änderungen durchführen zu können.

Beispiel

```
module Modul

application MyServer
```

```
{
  ...
  export function void Calculation();
}
...
```

Die Funktion "Calculation" kann von dem importierenden Modul aus aufgerufen werden, ohne dass dieses Modul über die Applikation "MyServer" Kenntnis hat. Dem Programmierer stehen durch diese beiden Mechanismen Möglichkeiten offen, unabhängig einsetzbare Module zu implementieren.

5.8.1 Applicationszuordnung von Modul-Funktionen

Ein spezieller Fall für die Benutzung von Funktionen in modularisierten Dialogen stellt die Verwendung des *.application*-Attribut am ***import***-Objekt dar um alle Funktionen eines Modules von „außen“ an eine spezielle Applikation zu koppeln. Diese Vorgehensweise ist im Kapitel „Programmiertechniken“/ „Modularisierung“ / „Objekt Application“ beschrieben.

Beim Importieren von Modulen über **use** steht diese Vorgehensweise allerdings nicht zur Verfügung. Als Alternative bzw. Ergänzung kann jedoch die Koppelung von Funktionen über das *.masterapplication*-Attribut am Modul bzw. Dialog verwendet werden. Damit sind Funktionen die direkt unter einem Modul/ Dialog definiert sind einer Applikation zuordenbar. Falls keine Zuordnung am Modul vorhanden ist wird die Zuordnung vom Dialog herangezogen. Das Attribut kann auch mit einem enum-Wert indiziert werden. Dieser Index-Wert sollte aus dem Bereich *lang_default…lang_java* oder *func_normal..func_data* sein. Dadurch lassen sich Funktionen entsprechend ihrer Sprache- oder Funktionsart unterschiedlichen Applikationen zuordnen. Eine explizite Zuordnung (auch eine null-Setzung) am Modul überschreibt/ überdeckt immer die Zuordnung des Dialogs.

Beispiel

Modularisierter Dialog mit den Funktionen im Modul „Functions.mod“, bei dem COBOL-Funktionen der Server-Applikation „ApplServer“ zugeordnet werden und alle anderen Funktionen der „dynlib“-Applikation „ApplLocal“.

```
// Dialog.dlg
dialog Dlg
{
  .masterapplication ApplLocal;
}

application ApplLocal {
  .transport "dynlib";
  .exec "libusercheck.so";
  .active true;
}

use Functions;
```

```
on dialog start
{
  variable string Username := setup.env["USER"];
  if  ValidateUser(Username) then
    print "Age = "+GetAge(Username);
  endif
  exit();
}

// Functions.mod
module Functions
{
   .masterapplication[lang_cobol] ApplServer;
}

use Server;

record RecUser
{
  string[50] FirstName;
  string[50] LastName;
  integer Age;
}

export function boolean ValidateUser(string Username);
function cobol integer GetUserProfile(string[50] Username,
                                      record RecUser output);

export rule integer GetAge(string Username)
{
  if GetUserProfile(Username, RecUser)>0 then
    return RecUser.Age;
  endif
  return 0;
}

// Server.mod
module Server

export application ApplServer {
  .connect "server:4712";
  .active true;
}
```

Siehe auch

Attribut .masterapplication[enum]

5.9 Anwendungsbeispiele für Modularisierung

Wie bereits in der Einleitung beschrieben, besteht die Möglichkeit einheitliche Standards zu entwickeln und diese in zentralen Modulen abzulegen (siehe Abschnitte über Ressourcenbasis und Vorlagenbasis). Des Weiteren besteht die Möglichkeit, Basisdialoge zu bauen und diese über Module zu erweitern bzw. abzuändern (siehe Abschnitt über differenzierte Dialoge). Anwendungen, die sich in mehrere Teile abgrenzen lassen, können über Module aufgeteilt und nur zur Laufzeit dynamisch geladen werden (siehe Abschnitt über aufteilbare Dialoge). Mit dem Einsatz modularisierter Dialoge lassen sich auf einfache Weise Prototypen bauen, die dem Kunden vorgestellt werden können. Testfälle lassen sich über Module leichter realisieren (siehe Abschnitt über Prototyping & Testing).

5.9.1 Ressourcenbasis

Einheitliche Standards der Softwareprodukte fördern die Benutzbarkeit, Anwenderfreundlichkeit, aber auch den Wiedererkennungswert der Software. Einheitliche Farben, Schriften, Texte, Formatroutinen und weitere Ressourcen können in einem oder mehreren Modulen zusammengefasst werden und zentral den einzelnen Entwicklungsabteilungen zu Verfügung gestellt werden. Dies erleichtert die Verwaltung der einheitlichen Ressourcen bei Änderungen und eventuellen Erweiterungen.

Beispiel zu einem Textressourcen-Modul:

```
module TextModul

!! Name der Firma in Kurzform
export text FirmaKurz  "ISA GmbH";
!! Name der Firma
export text FirmaLang  "ISA Informationssysteme GmbH";
!! Text für Pushbuttons
export text EndeText   "Exit"
{
  1:  "Beenden";
  // weitere Sprachen
}
```

Weitere Anwendungen für Standardfunktionen sind möglich.

Beispiel

```
module GlobalFunctions
export function string GetDateString(string Format input);
export function string GetTimeString(string Format input);
export Function boolean GetDate( integer Day output,
        integer Month output, integer Year output);
export Function boolean GetTime( integer Seconds output,
      integer Minutes output, integer Seconds output);
```

Weitere und weitaus detailliertere Beispiele sind hier denkbar.

5.9.2 Vorlagenbasis

Ähnlich wie die Ressourcenbasis lassen sich Defaults und Vorlagen in Modulen definieren und zentral verwalten bzw. bereitstellen. Mit diesen Modulen können ganze Bibliotheken aufgebaut werden, die ein einheitliches Aussehen der gesamten Produktpalette garantieren. In diesen Vorlagenbibliotheken kann das Aussehen mit den zugehörigen Regeln oder können ganze Objektgruppen (wie etwa Fenster mit bestimmten Menüs) festgelegt werden und in den verschiedensten Projekten wiederverwendet werden. Gerade im Bereich des Oberflächendesigns bietet sich die Wiederverwendbarkeit ganzer Objektgruppen an.

Beispiel für eine Defaultbasis ("default.mod"):

```
!! Zentrale defaults für alle Vorlagen und Instanzen
module Defaults
export default window  // Bei defaults ist kein Name notwendig
{
  .visible false
  ...  // weitere Attribute
  object CloseRule := null;// benutzerdefiniertes Attribut
            // soll eine Regel beinhalten, die
            // beim Schließen aufgrufen wird
}
on WINDOW close
{
  if ( CloseRule <> null )
  then
    this.CloseRule( this );
  endif
}
...
```

Eine Anwendung ("model.mod") der Defaultbasis könnte z.B. so aussehen:

```
module Models

import DefaultBase "MODLIB:default.if";
export model window MMainWin
{
  .CloseRule := MainWinClose;
  child menubox System
  {
    .title "System";
    ...
  }
}
!! Diese Regel braucht nicht exportiert zu werden!
rule void MainWinClose( object ThisObj input )
{
  exit;
```

```
}
...
```

Mit dem obigen Beispiel wäre auch dann schon ein Anfang einer ausbaubaren Vorlagenbasis geschaffen. Damit kann dem Anwendungsprogrammierer eine breite Basis von erweiterten Standardobjekten mit ihrer zugehörigen Funktionalität bereitgestellt werden. Diese muss dann noch um weitere produktspezifische Funktionalität ergänzt werden.

Beispiel

```
dialog Main

import DefaultBase "MODLIB:default.if";
import ModelBase "MODLIB:model.if";

MMainWin MainWindow
{
  .System.title "Datei";
    // ändern der Vorlagenvoreinstellung
  child groupbox
  {
    // produktspezifische Objekte ...
  }
}
on dialog start
{
  // produktspezifische Regeln
}
```

5.9.3 Austauschbare Teile einer Anwendung

Oft müssen nur Teile eines Dialoges ausgetauscht werden, um eine bestehende Branchenanwendung für eine andere, aber ähnliche Branche umzuschreiben. Stehen diese austauschbaren Teile im Voraus fest (siehe auch Abschnitt über Prototyping), kann mit Hilfe der Modularisierung der Dialog so gestaltet werden, dass nur einzelne Module ausgetauscht werden müssen, um eine "neue" Anwendung zu erhalten. Ebenso können hiermit kundenspezifische Wünsche in Aussehen und Funktionalität mitberücksichtigt werden.

Beispiel eines Tankstellenverwaltungsprogramms:

Im Großen und Ganzen wird die Verwaltung einer Tankstelle auch bei verschiedenen Konzernen übereinstimmen. Lagerverwaltung, Kassenbuch und Einkauf werden identisch sein. Doch werden bei der Abrechnung oder im Bestellwesen mit den Konzernen Unterschiede auftreten, die als Komplettlösung schwierig zu bedienen sind oder zu viele umständliche Einstellungen nötig machen. Der Entwickler kann ein einheitliches Programm für die allgemeinen Aufgaben eines Tankstellenbesitzers entwerfen und nur bei den konzernbedingten Unterschieden austauschbare Module entwerfen (d.h.

die gleiche Schnittstelle). Diese austauschbaren Module basieren im Normalfall wieder auf einer einheitlichen Vorlagenbasis.

Bei austauschbaren Modulen muss darauf geachtet werden, dass der Aufwand der Modulverwaltung im Verhältnis zur Ersparnis an Entwicklungszeit und natürlich im Verhältnis zur Benutzerakzeptanz steht.

5.9.4 Aufteilbare Anwendungen

Viele Anwendungen lassen sich in verschiedene Teilanwendungen aufgliedern. Einige dieser Teilanwendungen sind ständig in Benutzung, andere wiederum werden selten, manche gar von einigen Benutzern nie benutzt. Diese selten oder gar nie benutzten Teile einer Anwendung brauchen nicht die Ressourcen des Computers zu belegen. Über die Modularisierung ist es möglich, diese Teilanwendungen in Modulen zu beschreiben, diese aber nicht ständig im Speicher zu halten. Dadurch lassen sich Ressourcen des Computers sparen, da nicht benutzte Teile nicht im Speicher sind. Weiterhin wird die Zeit, in der der komplette Dialog in den Speicher geladen ist, reduziert, indem nur der Hauptteil bzw. der ständig benutzte Teil des Dialogs geladen wird. Die selten benutzten Module können zur Laufzeit des Dialogs geladen werden. Somit kann die eigentliche komplette Ladezeit in mehrere Zeitspannen aufgeteilt werden und damit den Benutzern unnötige Wartezeiten am Beginn des Programms ersparen.

Manche Dialogteile werden nur kurzfristig benötigt und dann für längere Zeit nicht mehr. Hier empfiehlt es sich dann, diese Dialogteile in Modulen zu definieren. Diese können dann bei Bedarf in den Speicher geladen, ausgeführt und falls gewünscht auch wieder aus diesem entfernt werden.

(Anmerkung: Bei den heutigen Betriebssystemen verbleibt der freigegebene Speicher beim Programm und kann nur von diesem wieder benutzt werden, aber auch durch andere wiederum nachgeladene Module).

Beispiele hierzu lassen sich in fast jeder Anwendung finden.

5.9.5 Prototyping & Testing

Bei vielen Softwareprojekten steht am Anfang die Aufgabe, dem Kunden einen Prototyp vorzustellen. Dieser Prototyp beinhaltet nicht die volle Funktionalität des späteren Produktes, doch aber so viel, dass der Kunde Missverständnisse oder Unklarheiten vor der Entwicklungsarbeit mit dem Hersteller zusammen aus dem Weg räumen kann. Oftmals können durch einen Prototypen die Machbarkeit oder Bedienbarkeit eines Programms am Anfang evaluiert werden, ohne dass dadurch wertvolle Zeit und Arbeit verlorengeht, die am Ende des Projektes wiederum fehlt.

Da bei einem Prototyp, der mit dem Dialog Manager entwickelt wird, natürlich die Oberfläche und ihre Funktionalität im Vordergrund stehen, wird hier verstärkt auf diesen Gesichtspunkt beim Prototyping eingegangen.

Grob kann das entstehende Programm aus der Sicht des Dialog Managers in zwei Teile aufgegliedert werden: die Oberfläche mit ihrer Funktionalität und die eigentliche Anwendung. Mit dem Dialog Manager kann mit Hilfe der Modularisierung schnell ein wiederverwendbarer Prototyp erstellt werden, der

ohne die eigentliche Anwendung auskommt, und somit dem Kunden schnell ein Prototyp vorgestellt werden.

Die Schnittstelle zwischen Dialog Manager und Anwendung erfolgt über Funktionen. Diese Funktionen können oft durch benannte Regeln ersetzt werden. Der Entwickler kann diese Funktionen zwar durch Regeln ersetzen, muss aber jedes Mal Änderungen am eigentlichen Dialog vornehmen. Mit verschiedenen Modulen, die einmal die eigentlichen Funktionsdefinitionen enthalten und einmal die identisch benannten Regeln beinhalten, kann eine Simulation eines Prototyps leicht erfolgen. Hier muss aber nicht auf eine Änderung des jeweiligen Dialoges zurückgegriffen werden.

Im folgenden Beispiel wird nur auf die dem Verständnis helfenden Objekte und Attribute eingegangen.

Beispiel

```
dialog Main

import Functions "PROTOLIB:func.if";
variable boolean OnLine := false;
on dialog start
{
  OnLine := CheckOnline();
  if ( not OnLine )
  then
    // eine Meldung, dass das Programm nur online abläuft
    exit;
  endif
  // Programm darf laufen
}
...
```

Das Modul func.mod wird wie folgt definiert:

```
module FunctionModule

export function boolean CheckOnline();
...
```

Hier im Beispiel müsste der Dialog so modifiziert werden, dass die Startregel des Dialoges abgeändert wird, so dass das Programm nicht sofort abbricht. Diese Änderung ist nicht mehr nötig, wenn dieses Modul durch ein Modul mit identisch benannten Regeln ersetzt wird.

Das Modul "rulefunc.mod" wird dann wie folgt definiert:

```
module FuntionRuleModule

export rule boolean CheckOnline()
{
  return( true );
}
...
```

Um am Dialog wirklich nichts zu ändern, muss aus dem Modul "rulefunc.mod" eine Interface-Datei erzeugt werden, die den gleichen Namen wie die des Moduls "func.mod" besitzt ("func.if"). Um die Interface-Dateien nicht jedes Mal neu zu erzeugen, wenn zwischen Prototyp und voranschreitender Entwicklungsarbeit umgeschaltet werden muss, kann dies über eine Umgebungsvariable gesteuert werden (hier: PROTPLIB). Damit können die Interface-Dateien in verschiedenen Verzeichnissen abgelegt werden.

Dieses Schema lässt sich mit Hilfe des Dialog Managers noch weiter ausbauen, indem der Benutzer während des Ablaufs über die Reaktionen der eigentlichen Anwendung Auskunft geben kann.

Erweiterung des Moduls "rulefunc.mod" um eine Abfrage:

```
module FuntionRuleModule

messagebox MBCheckOnline
{
  .title "Online";
  .text "Is your program online?";
  .button[1] button_yes;
  .button[2] button_no;
  .icon icon_query;
}
export rule boolean CheckOnline()
{
  variable boolean Result;
  if ( querybox(MBCheckOnline) = button_yes )
  then
    Result := true;
  else
    Result := false;
  endif
  return( Result );
}
...
```

In dem späteren Modul (hier: "func.mod") ist diese Messagebox nicht vorhanden, sie dient nur der direkten Steuerung des Dialogablaufs, da dieser durch die eigentliche Anwendung gesteuert wird, die in diesem Fall noch nicht vorhanden ist.

Ein weiterer Vorteil dieser Methode ist, dass damit auch Dialoge gezielt getestet werden können, indem der Entwickler oder die Qualitätssicherung damit auch unwahrscheinliche Testfälle simulieren kann, ohne dass an der Anwendung manipuliert werden muss. Hier im Beispiel könnte es im Test-betrieb Probleme bereiten, die Anwendung wirklich Offline zu schalten und damit wäre der Teil des Dialogs, der sich mit dem Offline-Betrieb beschäftigt, nur umständlich zu testen.

5.10 Beispiel

Das nachfolgende Beispiel finden Sie in Ihrem Installationsverzeichnis im Verzeichnis "modular". Dort können Sie je nach Umgebung "make" oder "nmake" aufrufen. Dadurch erhalten Sie ein ablauffähiges Programm.

In diesem Beispiel sind die Module bewusst klein gehalten, um den Einsatz der Modularisierung zu zeigen.

5.10.1 Das Defaultsmodul

In diesem Modul sind alle Defaultobjekte definiert. Dieses Modul wird dann in alle anderen Module importiert, damit auf dieselben Defaultwerte zugegriffen werden kann. In diesem Modul werden alle Objekte exportiert.

```
!! EXAMPLE FOR ISA DIALOG MANAGER
!!
!! This module defines all defaults and should be
!! imported in every
!! module that defines objects (instances or models)

module Default
{
}

!! We export this font, because it is used also outside
!! this module
export font FontNormal
{
    0: "*helvetica-medium-r-normal--18-*-iso8859-1";
    1: "SYSTEM";
    2: "12.System Proportional";
}

!! THE DEFAULTS
!!
!! This default might be used *outside* this module hence we
!! have to export it.
export default menubox
{
}

export default menuitem
{
}

export default menusep
```

```
{
}

export default scrollbar
{
}

export default window
{
}

export default listbox
{
}

export default edittext
{
}

export default statictext
{
}

export default pushbutton
{
}
export default checkbox
{
}

export default radiobutton
{
}

export default poptext
{
}

export default groupbox
{
}

export default image
{
}
```

```
export default canvas
{
}

export default rectangle
{
}

export default messagebox
{
}

export default tablefield
{
}
```

5.10.2 Das Modul für die Pushbutton-Modelle

In diesem Modul wird ein Modell für einen Pushbutton definiert. Zu diesem Modell wird dann eine Regel hinterlegt. Das Modul selber benötigt das Defaultsmodul.

```
!! EXAMPLE FOR ISA DIALOG MANAGER
!!
!! define a model for the pushbuttons that are loading
!! the corresponding module (if selected)
!! we define a userdefined attribute to store the
!! corresponding import/module that contains our window
!!
!! rule for every instance of the model
!! the 'if'-statement isn't necessary, the builtin-function
!! 'load' checks if the module is already loaded
module PushbuttonLibrary

export import Defaults "MODLIB:default.if";

!! only an internal model
!! you can't use it outside this module
model pushbutton MPImport
{
    object Import := null;
}

!! A pushbutton with "load" label. You have to initialize
!! the instance with the attribute. Import with an import
!! object. If the pushbutton is selected then the
!! corresponding module (to .Import) is loaded.
export model MPImport MPLoad
```

```
{
    .text "load";
}
on MPLoad select
{
    if ( not this.Import.load )
    then
      this.Import.load := true;
    else
      print "Cannot load again";
    endif
}
!! see above model, the label is "unload" and the instance
!! has to be initialized again (.Import).
!! On selection the pushbutton unload the corresponding
!! module.
export model MPImport MPUnload
{
    .text "unload";
    .xleft 16;
}

on MPUnload select
{
    if ( this.Import.load )
    then
      this.Import.load := false;
    else
      print "Module not loaded";
    endif
}
```

5.10.3 Weitere Module

Für dieses Beispiel sind noch weitere Module definiert worden, die Sie bitte Ihrem Beispielverzeichnis entnehmen.

5.10.4 Der Dialog LoadExample

```
!! EXAMPLE FOR ISA DIALOG MANAGER
!!
!! This dialog demonstrates the use of the dynamic load of
!! modules
dialog LoadExample
{
  .reffont FontNormal;
```

```
}

!! dialog wide defaults
import Defaults "MODLIB:default.if";
import ModelLib "MODLIB:modellib.if";
import PushbuttonLib "MODLIB:pushblib.if";

!! module window 1, but do not load it right now
!! see the attribute '.load',
!! '.load' is set to false ==> not loaded yet
import Window1 "MODLIB:window1.if"
{
  .load false;
}
!! see comment above
import Window2 "MODLIB:window2.if"
{
  .load false;
}
!! see comment above
import Window3 "MODLIB:window3.if"
{
  .load false;
}

!! this window controls the 'load' of the three modules
MWinMain ControlWindow
{
  .title "Control";
  .width 20;
  .height 7;

  child MPLoad
  {
  .ytop 0;
  .text "Load Module 1";
  .Import := Window1;
  }
  child MPLoad
  {
  .ytop 1;
  .text "Load Module 2";
  .Import := Window2;
  }
  child MPLoad
  {
```

```
   .ytop 2;
   .text "Load Module 3";
   .Import := Window3;
   }
}
```

Das Startfenster dieser Anwendung sieht wie folgt aus:

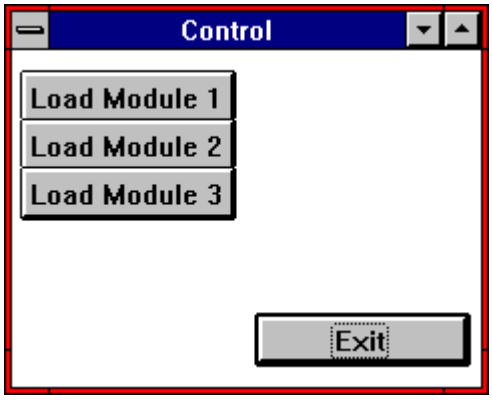

Abbildung 18: *Startfenster einer modularisierten Anwendung*

Durch die Selektion der Pushbuttons werden die einzelnen Module geladen. Man erhält dadurch in etwa folgendes Aussehen:

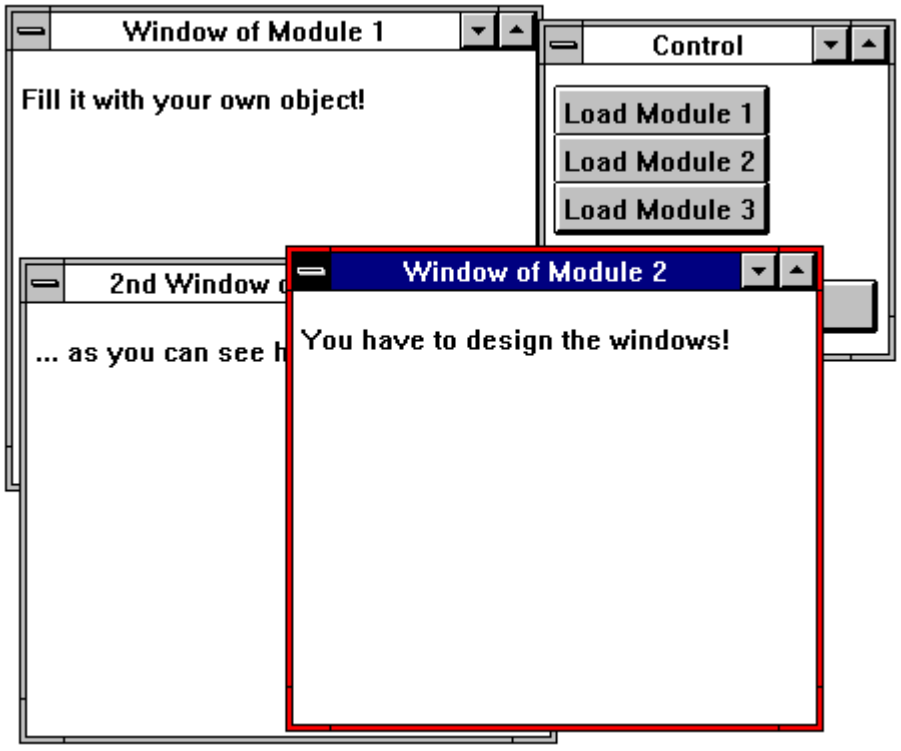

Abbildung 19: *Anwendung mit geladenen Modulen*

5.10.5 Beispiel für den USE-Operator

Als ein Beispiel für den Einsatz des USE-Operators kann das Beispiel "use.dlg" dienen. Hier wird ein Tablefield mit Hilfe des USE-Operators in ein Fenster übernommen.

```
!! EXAMPLE FOR ISA DIALOG MANAGER
!!
!! This example shows a dialog with a window and a
!! tablefield inside this
!! window. The tablefield is defined outside this module,
!! just take a
!! look in the table.mod module.
dialog UseExample
{
    .reffont FontNormal;
}

import Defaults "MODLIB:default.if";
import ModelLib "MODLIB:modellib.if";

!! import an object and *use* it as a child in another
!! object
import Tablefield "MODLIB:table.if";

MWinMain Win
{
    .title "Use Example";
    .width 50;
    .height 11;

    !! use the tablefield, we cannot change attributes here
    use child Table;

}
```

Nach dem Start der Anwendung sieht das Fenster wie folgt aus:

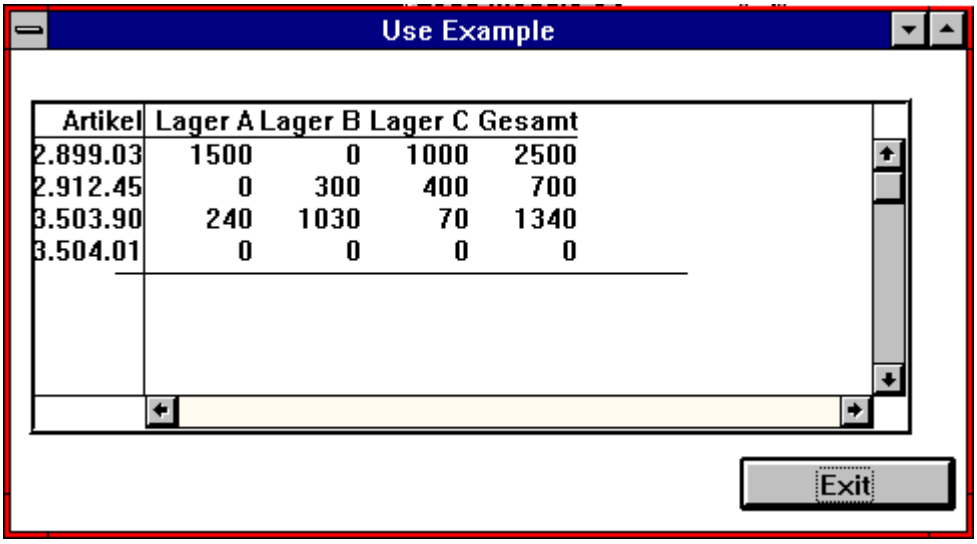

Abbildung 20: *Einsatz des USE-Operators*

5.11 Aufbau einer Entwicklungsumgebung

Wenn unter dem Einsatz der Modularisierung eine Entwicklungsumgebung aufgebaut werden soll, müssen folgende Aspekte beachtet werden:

- Die Interface-Dateien müssen immer dem aktuellen Stand der Moduldateien entsprechen, sonst kommt es beim Laden des entsprechenden Moduls zum Abbruch des Systems.
- Die Binärdateien der Module tragen im Normalfall dieselbe Endung wie die zugehörigen ASCII-Dateien der Module. Das kommt daher, da in den Interface-Dateien die Namen der zugehörigen Implementierungsdateien hinterlegt sind und keine Unterscheidung gemacht wird, ob die Datei die Daten in ASCII- oder Binärform enthält.
- Jeder Entwickler sollte zunächst in seiner privaten Umgebung Änderungen durchführen und austesten können, bevor er sein Modul seinen Kollegen zugänglich machen muss. Dadurch können auch weitergehende Änderungen erst lokal ausgetestet und dann erst dem gesamten Projektteam zugänglich gemacht werden.
- Beim Aufbau der Module muss darauf geachtet werden, dass Module sich nicht rekursiv importieren dürfen (auch nicht indirekt über mehrere Stufen!). Diese Rekursivität lässt sich am leichtesten erkennen, wenn man alle Interface-Dateien löscht und diese dann erneut bauen lässt. Hat man nun aus Versehen eine Rekursivität eingebaut, so lassen sich die Interface-Dateien nicht generieren. Die Module müssen dann umgebaut werden.

Unter diesen Aspekten kann eine Entwicklungsumgebung unter Einsatz der Modularisierung wie folgt aussehen:

Zunächst wird ein Projektverzeichnis angelegt, in dem alle freigegebenen Module und Dialoge abgelegt werden. Dort werden dann auch die zugehörigen Interface-Dateien hinterlegt.

Zusätzlich wird ein paralleles Verzeichnis erstellt, in dem die Binärversionen der Module abgespeichert werden. Diese Binärdateien werden immer dann automatisch erstellt, wenn sich am zugrundeliegenden Modul oder an den Interface-Dateien der benutzten Module etwas geändert hat.

Jedes Projektmitglied erhält für sich privat dieselbe Umgebung wie im Projektverzeichnis. Zunächst werden dort aber keine Dateien abgelegt. Wenn nun Änderungen an einem Modul durchgeführt werden sollen, wird die Datei aus dem Projektverzeichnis in das private Verzeichnis kopiert und dort solange verändert, bis sie den Anforderungen genügt. Dabei wird selbstverständlich auch die zugehörige Interface-Datei im privaten Verzeichnis abgelegt. Wenn die Änderungen ausgetestet sind, wird die Datei in das Projektverzeichnis zurückgestellt und im privaten Verzeichnis gelöscht. Dadurch können dann alle Projektmitglieder auf dieses Modul in seiner geänderten Form zugreifen.

Wenn aus Performanzgründen auch privat Binärdateien erstellt werden, so muss dieses wieder durch dieselbe Struktur wie im Projektverzeichnis erfolgen.

Die Auswahl der aktuellen Source beim Programmablauf wird über die beim Generieren der Interfacedateien angegebene Umgebungsvariable gesteuert. Ohne diese ist das Verhalten nicht sinnvoll steuerbar. Ein sinnvolles Verhalten wird erreicht, wenn die Umgebungsvariable für die ASCII-Version auf

PrivatesVerzeichnis;ProjektVerzeichnis

gesetzt wird. Dadurch wird zuerst im privaten Verzeichnis nach jeder Moduldatei gesucht und nur wenn dort keine gefunden wird, wird die Datei aus dem Projektverzeichnis genommen. Für die Binärversion sieht das genauso aus. Ein Mischen der ASCII- und Binärform kann fatale Folgen haben, wenn die Abhängigkeiten zwischen den Modulen in der Entwicklungsumgebung nicht richtig erfasst und umgesetzt werden. Daher sollte man entweder auf den Binär- oder auf den ASCII-Dateien arbeiten.

Die nachfolgende Abbildung soll eine mögliche Modulstruktur verdeutlichen. In diesem Beispiel werden im Modul "Anwendungsteil 1" direkt Ressourcen verwendet, während im Modul "Anwendungsteil 2" Ressourcen nicht direkt verwendet werden. Aus diesem Grund wird in das Modul "Anwendungsteil 1" das Modul "Ressourcen" importiert, in das Modul "Anwendungsteil 2" hingegen nicht.

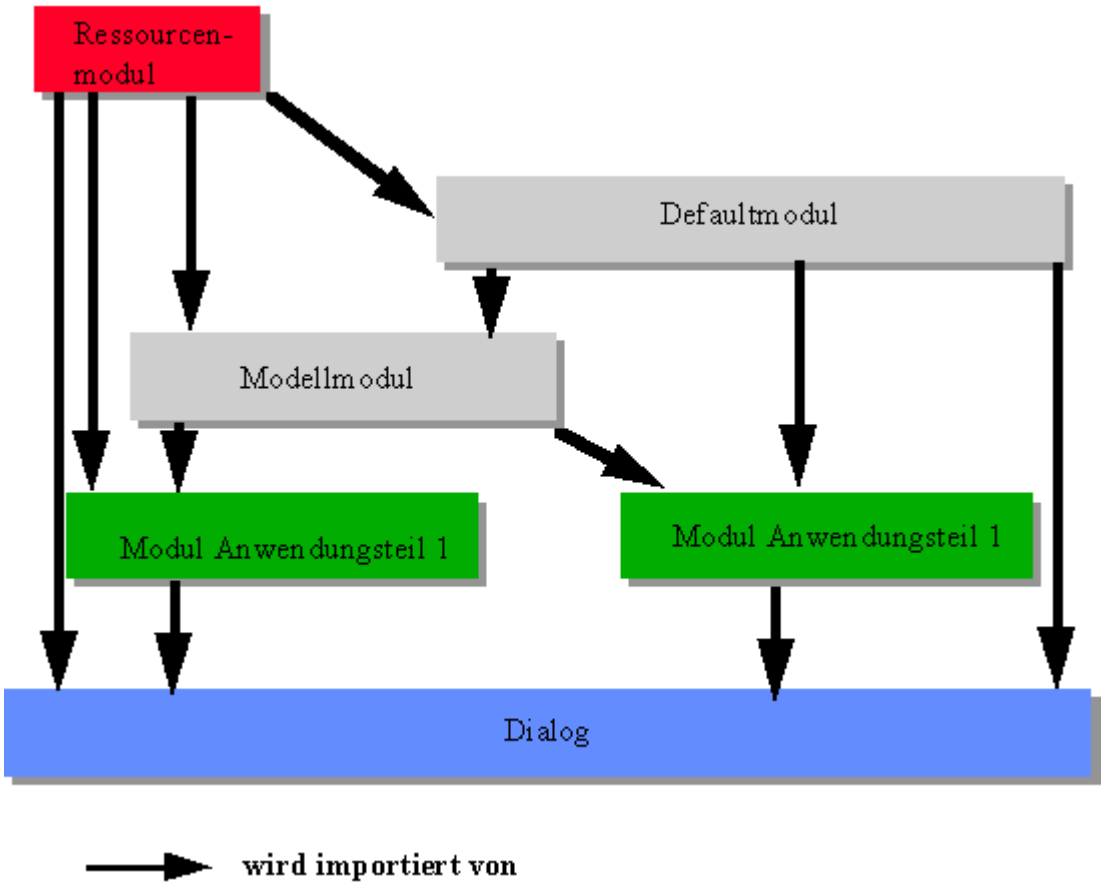

Abbildung 21: *Mögliche Modulstruktur*

Von diesen Voraussetzungen ausgehend kann eine sinnvolle Projektstruktur wie folgt aussehen:

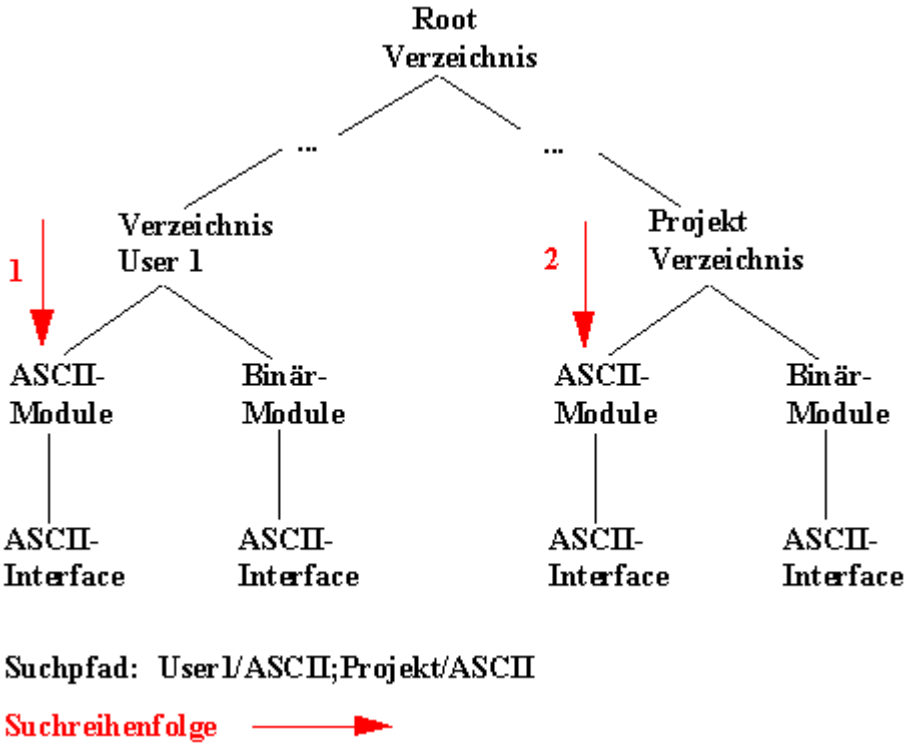

Abbildung 22: *Sinnvolle Projektstruktur*

Für diese Struktur ist Voraussetzung, dass jeder Benutzer auf die Projektverzeichnisse zugreifen kann. Auf MICROSOFT WINDOWS Rechnern können diese Verzeichnisse selbstverständlich auf verschiedenen logischen Platten liegen, solange nur sichergestellt ist, dass jeder Benutzer auf diese Platte zugreifen kann.

Zusätzlich kann der Einsatz von Source-Code Kontrollsystemen nur dringend angeraten werden, da dadurch paralleles Ändern an ein und derselben Sourcedatei entweder verhindert wird oder die parallelen Arbeiten wieder zusammengeführt werden können.

6 Datenmodell

Die **Motivation** für das „Datenmodell“ im IDM ist die damit erreichbare bessere Entkopplung zwischen Oberflächenobjekten und der Applikationsschicht, welche die Daten für die Darstellung liefert.

Die Hauptmerkmale für dessen Design sind:

- Trennung der Benutzerschnittstelle und der Applikation auf Basis des Entwurfsmusters Model-View-Presenter (MVP).
- Der Dialogentwickler definiert lediglich die Kopplung zwischen Oberflächenobjekten (View) zur Datenhaltung in der Applikationsschicht (Model).
- Möglichst kein bzw. wenig Regelcode durch den steuerbaren Einsatz einer automatischen Synchronisierung sowie durch automatische Typkonvertierungen.
- Beibehaltung des schon bekannten Datenschemas und Zugriffskonzeptes für indizierte Attribute sodass Daten in skalarer oder vektorieller Form verwendbar sind.
- Nutzung der bestehenden IDM-Möglichkeiten für die Datenhaltung bzw. für die Implementierung der Model-Schicht: Objekte des IDM (***Record***-Objekt) wie auch Applikationsfunktionen die ganz transparent auch auf der DDM-Serverseite liegen können.

6.1 Einführung

Der IDM erlaubt mit dem Datenmodell die Entkopplung in drei Komponenten[1] Model, View und Presenter auf Basis des MVP-Entwurfsmusters mit Anpassung an die Eigenheiten und Erfordernisse der Regelsprache.

Model

Verwaltet die Daten und die dahinterstehende Logik. Ist typischerweise nahe bzw. in der Applikationsschicht. Ein Model kann dabei mehrere Daten verwalten, welche vom IDM als indizierte Attribute betrachtet werden und somit Skalare, Vektoren und Matrizen beinhalten können.

Zum Beispiel ein ***Record***-Objekt mit benutzerdefinierten Attributen und Methoden.

View

Visuelle Repräsentation der Daten für die Darstellung, Eingabe oder Modifikation. Das Datenschema ist das gleiche wie beim Model: indizierte Attribute die somit Skalare, Vektoren oder Matrizen beherbergen.

Zum Beispiel ein ***Edittext***-Objekt das ein Format für die Ein- und Ausgabe verwendet.

[1] Die Komponenten werden mit ihrer englischen Bezeichnung verwendet um damit eine Unterscheidung zum Begriff Modell zu ermöglichen.

Presenter

Die Steuerlogik um die Bindung zwischen Model und View zu regeln oder um mehrere Views zu koppeln.

Zum .Beispiel in Form von Ereignisregeln um beim Beenden der Editierung den neuen Datenwert am View dem Model zuzuweisen.

Kurzer Begriffshinweis: In dieser Dokumentation gibt es für den Begriff „Datenmodell“ quasi zwei Bedeutungen. Erstens das vom MVP abgeleitete Konzept sowie zweitens ein konkretes Model.

Im Gegensatz zu bestehenden MVP-Konzepten in JAVA oder QT erlaubt der Ansatz im IDM die transparente Behandlung ohne eine Kopplung zu reglementieren oder die Unterscheidung zwischen Inhalt bzw. Selektion vorzuschreiben. Ob ein Model also den Inhalt eines Edittextes, die Farbe einer speziellen Tabellenzelle oder das aktive Element einer Liste steuert, bleibt dem Anwendungs- bzw. Dialogprogrammierer überlassen.

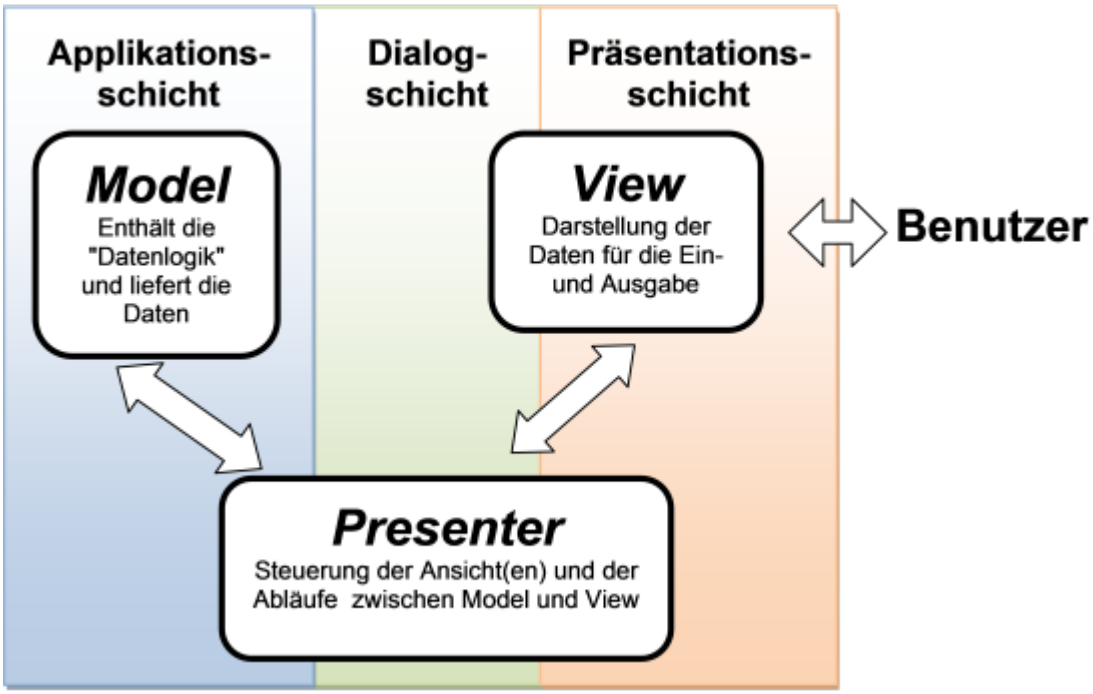

Abbildung 23: *Architekturmuster Model-View-Presenter*

Im IDM werden dabei die Kopplung zwischen Model und View sowie die wichtigsten Steuerfunktionen des Presenters durch einige wenige Attribute geregelt:

Tabelle 4: *Datenmodellattribute*

Attribut	Bedeutung
.datamodel	Verweis an der View auf das Model

Attribut	Bedeutung
.dataget	Definiert die Kopplung eines View-Attributs mit einem Model-Attribut für die Anzeige
.dataset	Definiert die Kopplung eines View-Attributs mit einem Model-Attribut für die Zuweisung
.dataindex	Erlaubt die Steuerung der Aggregation zwischen View- und Model-Attributen
.datamap	Erlaubt die Zuordnung von Attributen zwecks Auflösung von „Mehrdeutigkeiten“
.dataoptions	Steueroptionen der Presenter-Logik

Ein **Beispiel** für die Kopplung einer Dialogoberfläche (View) mit einem Datenmodell (Model) für die Anzeige von Personeninformationen mit Name, Geschlecht und Kindernamen kann etwa wie folgt aussehen:

```
dialog D
record RecPerson {
  string  Name   := "Ina Mustermann";
  boolean Female := true;
  string Children[integer];
  .Children[1] := "Anja";
  .Children[2] := "Peter";
}
window Wi {
  .title "Datamodel Example RecPerson";
  .width 200;
  .height 200;
  /* Kopplung aller View-Objekte mit dem Model RecPerson */
  .datamodel RecPerson;

  edittext EtName
  {
    .xauto 0;
    /* .content wird mit RecPerson.Name verknuepft */
    .dataget .Name;
  }

  checkbox CbGender
  {
    .ytop 30;
    .text "Female";
    /* .active wird aus RecPerson.Female befuellt */
    .dataget .Female;
  }

  listbox LbChildren
```

```
  {
    .ytop 60;
    .xauto 0;  .yauto 0;
    /* .content[] wird aus RecPerson.Children[] befuellt */
    .dataget .Children;
  }

  on close { exit(); }
}
```

Der Datenmodell-Ansatz im IDM nimmt dem Anwendungsprogrammierer einen Teil der Presenter-Logik ab, weshalb das obige Beispiel auch ohne weitere Regeln problemlos funktioniert. Das Befüllen der Oberflächenobjekte „EtName“, „CbGender“ und „LbChildren“ übernimmt der IDM durch die gesetzte Kopplung über die Attributdefinitionen *.datamodel* und *.dataget*. Weiter macht sich das obige Beispiel die Eigenschaft der zusätzlichen Wertweitergabe vom Vater zum Kind zunutze, welche für die Attribute *.datamodel* und *.dataoptions* gilt.

Das folgende Schaubild dient dazu diese Automatismen zu verstehen, sowie die Steuerungs- und Einflussmöglichkeiten aufzuzeigen, welche der Dialogprogrammierer für die Komponenten View und Presenter hat sowie der Applikationsprogrammierer für die Model-Komponente.

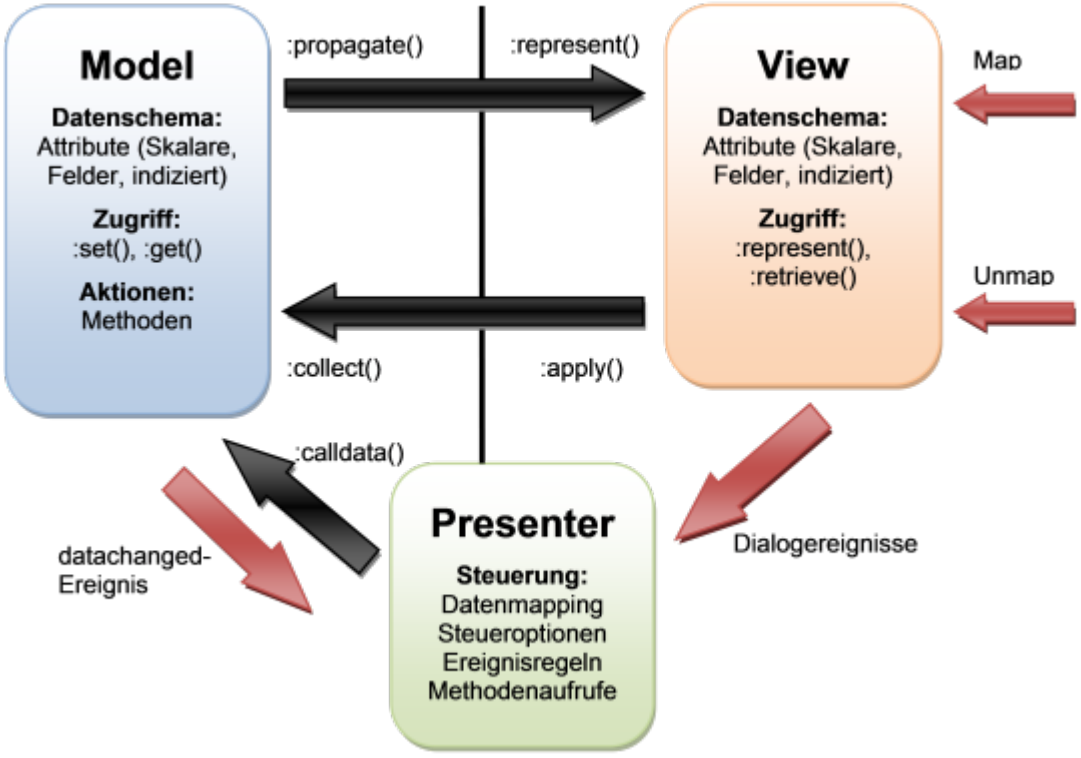

Abbildung 24: *Kopplungen zwischen Model, View und Presenter*

Tabelle 5: *Weitergabe von Änderungen zwischen Modell, View und Presenter*

Methode, Funktion	Komponente	
:represent()	View	View-Objekt wird mit den Daten des Model befüllt und ggf. wird ein Datenwert wird für die Anzeige aufbereitet
:apply()	View	View-Objekt weist die Daten an das Model zurück, ggf. werden Datenwerte für Zuweisung aufbereitet
Dialogereignis	View, Presenter	Änderungen der View werden signalisiert
:propagate()	Model	Daten der Model-Komponente werden an die verknüpften View-Objekte zur Anzeige weitergegeben
:collect()	Model	Die Model-Komponente holt sich alle Werte aus den verknüpften View-Objekten
DM_DataChanged() bzw. Attributzuweisung	Model	Die Model-Komponente signalisiert eine Änderung an den Daten
.dataoptions[]	Presenter, Model	Steuerungsoptionen für die automatische Synchronisation zwischen Model und View

Die Presenter-Logik im IDM umfasst aktuell vier Automatismen:

1. Wird ein View-Objekt mit einer Kopplung zu einem Model sichtbar („Map“), so holt es sich die Daten kurz bevor es sichtbar wird (standardmäßig aktiv).
2. Wird ein View-Objekt mit einer Kopplung zu einem Model unsichtbar („Unmap“), so weist es die Daten dem zugeordneten Model zu (standardmäßig inaktiv).
3. Ein Dialogereignis an einem View-Objekt mit einer Kopplung zu einem Model, welches eine Änderung an einem gekoppelten Attribut anzeigt, weist die Daten dem zugeordneten Model zu (standardmäßig inaktiv).
4. Änderungen an Daten eines Model, z.B. über ein *datachanged*-Ereignis signalisiert, führen zu einer Propagierung der Änderung an die gekoppelten View-Objekte wenn diese sichtbar sind (standardmäßig aktiv).

6.2 Kopplung zwischen Model und View

Grundsätzlich ist das Datenmodell so konzipiert, dass dem Model nicht bekannt ist, von welchen View-Objekten es genutzt wird. Das Datenschema ist für Model und View gleich: indizierte Attribute um somit Skalare, Vektoren und Matrizen zu erlauben.

Die Kopplung einer View mit einem Model geschieht über das *.datamodel*-Attribut:

```
object .datamodel[<View-Attribut>] <Model-ID>
```

Für die Datenübertragung vom Model zur View muss zusätzlich noch definiert werden, welches Attribut des Models auszulesen ist.

```
attribute .dataget[<View-Attribut>] <Model-Attribut>
```

Um die Rückübertragung von Daten von der View zum Model zu definieren dient das folgende Attribut:

```
attribute .dataset[<View-Attribut>] <Model-Attribut>
```

So kann exakt festgelegt werden, was im View-Objekt durch ein Model befüllt bzw. zurückreflektiert wird. Durch die Indizierung am *.datamodel*-Attribut ist auch die Nutzung von mehreren Datenmodellen für unterschiedliche Attribute möglich.

View-Attribute sind typischerweise, aber nicht zwangsläufig, reale Attribute des View-Objekts, also z.B. *.content* für einen ***Edittext*** als View-Objekt. Es empfiehlt sich, als Model-Attribute benutzerdefinierte Attribute zu verwenden, um so eine bessere Unterscheidbarkeit zwischen View- und Model-Attributen zu erreichen.

Um die Kopplung zu „aktivieren“ bzw. wirksam werden zu lassen, muss immer das *.datamodel*-Attribut ohne Index gesetzt sein, sowie jeweils eine Kopplung über *.dataget* bzw. *.dataset* (mit oder ohne Index).

Hier ein **Beispiel** für die Nutzung von mehreren Models (Datenmodelle: „RecUsers“, „RecManager“, „VarValid“, „LbUsers“) durch mehrere Views („listbox LbUsers“, „edittext EtName“, „pushbutton PbKill“).

```
dialog D
record RecUsers
{
  string Name[integer];
  .Name[1] := "mueller";
  .Name[2] := "schrade";
  .Name[3] := "maier";
  string Rights[integer];
  .Rights[1] := "guest";
  .Rights[2] := "user";
  .Rights[3] := "root";
}

record RecManager
{
  string CurrUser := "maier";
  boolean IsAdmin := true;
  rule boolean ChangeUser(string Name)
  {
    variable anyvalue Idx;
    Idx := RecUsers:find(.Name, Name);
    if typeof(Idx) = integer then
```

```
      this.CurrUser := Name;
      this.IsAdmin := stringpos(RecUsers.Rights[Idx], "root") > 0;
      return true;
    endif
    return false;
  }
}

color CoError "red";
variable object VarError := true;

window Wi
{
  .title "Datamodel Model-View Coupling";
  .width 200;  .height 200;
  boolean Valid := false;

  listbox LbUsers
  {
    .xauto 0;  .yauto 0;
    .ybottom 60;
    .datamodel RecUsers;
    .datamodel[.activeitem] RecManager;
    .dataget .Name;
    .dataget[.activeitem] .CurrUser;
    on select, .activeitem changed
    {
      EtName.dataindex[.content] := this.activeitem;
      VarError := null;
      RecManager:ChangeUser(EtName.content);
    }

    :represent()
    {
      if Attribute = .activeitem then
        Value := this:find(.content, Value);
      endif
      pass this:super();
    }
  }

  edittext EtName
  {
    .yauto -1;  .xauto 0;
    .ybottom 30;
    .datamodel LbUsers;
    .datamodel[.bgc] VarError;
```

```
    .dataget .content;
    .dataget[.bgc] .value;
    .dataindex[.content] 0;
    on deselect_enter
    {
      if not RecManager:ChangeUser(this.content) then
        VarError := CoError;
      endif
    }
  }

  pushbutton PbKill
  {
    .yauto -1;
    .text "Kill All Processes";
    .sensitive false;
    .datamodel RecManager;
    .dataget[.sensitive] .IsAdmin;
  }

  on close { exit(); }
}
```

Um gebräuchliche Kopplungen zwischen View- und Model-Attributen möglichst einfach zu definieren, kann auf die **Standardvorgaben** des IDM zurückgegriffen werden (im Beispiel bei „EtName“ zu sehen, hier werden *.datamodel* und *.dataget* ohne Index verwendet). Diese Vorgaben sind in folgender Tabelle definiert:

Tabelle 6: *Standardkopplungen von View-Attributen*

Objektklasse	Standard-View-Attribut für *.dataget*	Standard-View-Attribut für *.dataset*
pushbutton statictext messagebox	*.text*	–
checkbox radiobutton timer	*.active*	*.active*
image menuitem	*.text*	*.active*
filereq	*.value*	*.value*

Objektklasse	Standard-View-Attribut für *.dataget*	Standard-View-Attribut für *.dataset*
listbox ***treeview*** ***tablefield***	*.content*	*.activeitem*
notepage	*.title*	–
edittext	*.content*	*.content*
poptext	*.text*	*.activeitem*
progressbar	*.curvalue*	–
scrollbar ***spinbox***	*.curvalue*	*.curvalue*
menubox ***toolbar*** ***window***	*.title*	–

Obiges Beispiel weist noch eine Besonderheit auf. Einmal wird ein sichtbares Objekt als Datenmodell verwendet (die View-Komponente „EtName“ ist gekoppelt mit dem Model „LbUsers“).

Grundsätzlich unterstützt der IDM als Datenmodell alle Objektklassen die benutzerdefinierte Attribute erlauben, außerdem noch globale Variablen sowie Funktionen mit dem Funktionstyp **datafunc**. Eine Kopplung von mehreren Modellen ist ebenso möglich. Bei sichtbaren (visuellen) Objekten sollte man aber eines bedenken: die Benutzerinteraktion bewirkt kein *changed*-Ereignis für Attribute und somit auch keine automatische Änderungsweitergabe an das Model. Dies muss explizit bzw. durch die entsprechende Synchronisationseinstellung geschehen.

Als View-Objekt sind ebenso alle Objektklassen erlaubt, die benutzerdefinierte Attribute erlauben. Allerdings sind die Automatismen für die Synchronisation zwischen Model und View dafür ausgelegt, dass es sich bei den View-Objekten um Instanzen mit visueller Ausprägung handelt.

6.3 Reihenfolge und Werteaggregation

Für die Befüllung eines View-Objekts ist eine korrekte Reihenfolge beim Setzen der View-Attribute notwendig. Sie werden vom IDM in einer vordefinierten, klassenspezifischen Reihenfolge (wie in der Attributliste des Objekts) gesetzt, um die Abhängigkeiten der Attribute untereinander zu berücksichtigen. Für einen ***Poptext*** gilt zum Beispiel die Reihenfolge

…, *.itemcount*, …, *.text[]*, …, *.activitem*, …, *.userdata[]*, …

um so die Abhängigkeit von *.text[]*, *.activeitem* und *.userdata[]* vom *.itemcount*-Attribut zu berücksichtigen. Diese Reihenfolge kann aber nur bei der Repräsentation (also einer Synchronisierung die an der View-Komponente ausgelöst wird, z.B. über **:represent()**) berücksichtigt werden. Der IDM hat

keinen Einfluss auf die Reihenfolge wenn die Synchronisierung von der Model-Komponente ausgelöst wird, z.B. bei Nutzung der **DM_DataChanged()**-Funktion.

Falls ein Datenmodell mit allen vier Attributen gekoppelt ist, so erfolgt zuerst das Setzen der Elementanzahl, dann der Texte, dann die Umsetzung des aktiven Elements und zum Schluss das Setzen des *.userdata[]*-Felds.

Grundsätzlich verbietet der IDM nicht die Nutzung von benutzerdefinierten Attributen als View-Attribute, kann aber für diese keine konsistente Reihenfolge gewährleisten. Benutzerdefinierte Attribute und vordefinierte Attribute die nicht zur Objektklasse gehören, werden auf jeden Fall nach den vordefinierten Attributen behandelt, für welche die Objektklasse eine Reihenfolge vorgibt.

Für die Werteaggregation, also die Beziehung oder Relation von Datenwerten zwischen Model und View gibt es dabei unterschiedliche Arten:

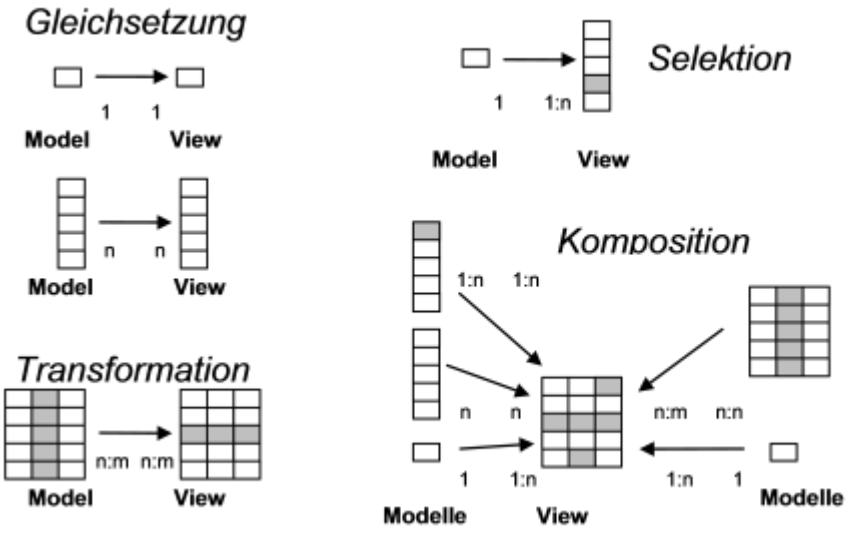

Abbildung 25: *Relationen zwischen Model und View*

Um alle diese Relationsarten zu ermöglichen dienen zwei Attribute:

```
anyvalue .dataindex[<View-Oder-Model-Attribut>] <Index>
attribute .datamap[<View-Attribut>] <View-Attribut>
```

Das *.dataindex[]*-Attribut dient dabei zur Selektion eines Wertes aus einer Werteliste sowie zur Transformation bei indizierten Werten. Dabei sollte beachtet werden, dass die Attribute als Skalar, Feld, assoziatives Feld oder Matrix (zweidimensionales Feld) vorkommen, allerdings der Zugriff immer nur einen Vektor (also ein eindimensionales Feld) ermöglicht.

Bei der Zuweisung von Sammlungen (Wertanzahl *<n>*) an ein vektorielles Attribut (eindimensional oder zweidimensional) finden dabei folgende Transformationen entsprechend dem Index statt:

Tabelle 7: *Indextransformationen bei der Zuweisung von Sammlungen*

Index	Zuweisung
[0,<col>]	*[1,<col>] … [<n>,<col>]*
[<row>,0]	*[<row>,1] … [<row>,<n>]*
[<row>,<col>]	*[<row>,<col>]*
0	*1 … <n>*
<row>	*<row>*
void	*void*
andere	*void*

Das *.datamap[]*-Attribut erlaubt die Vermischung von Datenwerten (aus einem oder verschiedenen Datenmodellen) die genau einem View-Attribut zugeordnet werden sollen. Das hauptsächliche Anwendungsgebiet wird für zweidimensionale Attribute sein, wie man sie am ***Tablefield***-Objekt findet. Man verwendet dazu ein „virtuelles" View-Attribut, welches dann mittels *.datamap[]*-Attribut einem „realen" View-Attribut zugeordnet wird. Als „virtuelles" Attribut kann ein beliebiges vordefiniertes oder benutzerdefiniertes Attribut verwendet werden, es empfiehlt sich unter Umständen aber die Nutzung eines am View-Objekt vorhandenen Attributs um die Reihenfolge bei der kompletten Repräsentation aller Model -Werte zu beeinflussen.

Beispiel

Folgendes Beispiel soll diese verschiedenen Relationsarten veranschaulichen. Beispielsweise wird das View-Objekt „PtMonth" mit dem gesamten Inhalt des Attributs „.MonthName[]" aus dem Model-Objekt „RecDate" gekoppelt. Hingegen zeigt das View-Objekt „StCurMonth" lediglich den 3. Eintrag aus dem Feld „RecDate.FullMonthName[]" an.

Weitaus komplexer ist die Datenkopplung für das View-Objekt „TfCurWeek". Die Wochentagsnamen aus „RecDate.DayName[7]" werden über die Kopplung `.dataindex[.content] [1,0];` auf die Titelzeile (Elemente *[1,1] … [1,7]*) verteilt. Um eine Kopplung der Wochennummern in die Tabelle zur erreichen, nutzen wir das *.field*-Attribut mittels der Definitionen `.dataget[.field] .WeekNr;` und `.dataindex[.field] [0,8];`. Da eine Kopfzeile mittels `.rowheader 1;` gesetzt ist, werden die Wochennummern in die Elemente *[2,8] … [6.8]* geschrieben. Für die Markierung des aktuellen Tages nutzen wird das „virtuelle" Attribut „.text", wobei eine Abbildung `.datamap[.text] .content;` eingerichtet werden muss, um den Markierungstext in das Feld „.content[4,3"] zu leiten.

```
dialog D
record RecDate
{
  integer CurMonth := 3;
  integer CurDay := 23;
  integer CurWeekDay := 3;
```

```
  string DayName[7];
  .DayName[1] := "Mon";
  ...
  string MonthName[12];
  .MonthName[1] := "Jan";
  ...
  string FullMonthName[12];
  .FullMonthName[1] := "January";
  ...
  integer WeekNr[5];
  .WeekNr[1] := 9;
  ...
}

window WiData
{
  .title "Datamodel: index-coupling";
  .width 600;  .height 400;
  .datamodel RecDate;

  poptext PtMonth
  {
    .xauto 0;
    .dataget .MonthName;  /* fills poptext.text[] with "Jan","Feb","Mar",...
*/
    .dataget[.userdata] .FullMonthName;  /* fill full names into .userdata[]
*/
    .dataget[.activeitem] .CurMonth;
  }

  statictext StCurMonth
  {
    .ytop 30;
    .xauto 0;
    .alignment 0;
    .dataget .FullMonthName;
    /* fills statictext.text with "Mar" by Model index */
    .dataindex[.FullMonthName] 3;
  }

  tablefield TfCurWeek
  {
    .xauto 0;  .yauto 0;
    .ytop 60;
    .rowheight[0] 30;  .colwidth[0] 60;
    .colcount 8;  .rowcount 6;
```

```
    .content[1,8] "Week";
    .rowheader 1;
    .dataget[.content] .DayName;
    .dataindex[.content] [1,0];  /* fill "Mon"... to first row via View index
*/
    .dataget[.field] .WeekNr;  /* map to .field attribute to fill week no.
*/
    .dataindex[.field] [0,8];  /* vertical into last column below the header
*/
    .datamodel[.text] Marker;  /* mark a specific day using the
*/
    .dataget[.text] .value;    /* .datamap attribute, because it goes into
*/
    .datamap[.text] .content;  /* the already used View attribute .content!
*/
    .dataindex[.text] [4,3];
  }

  on close { exit(); }
}
```

Das Bildschirmfoto dazu sieht folgendermaßen aus:

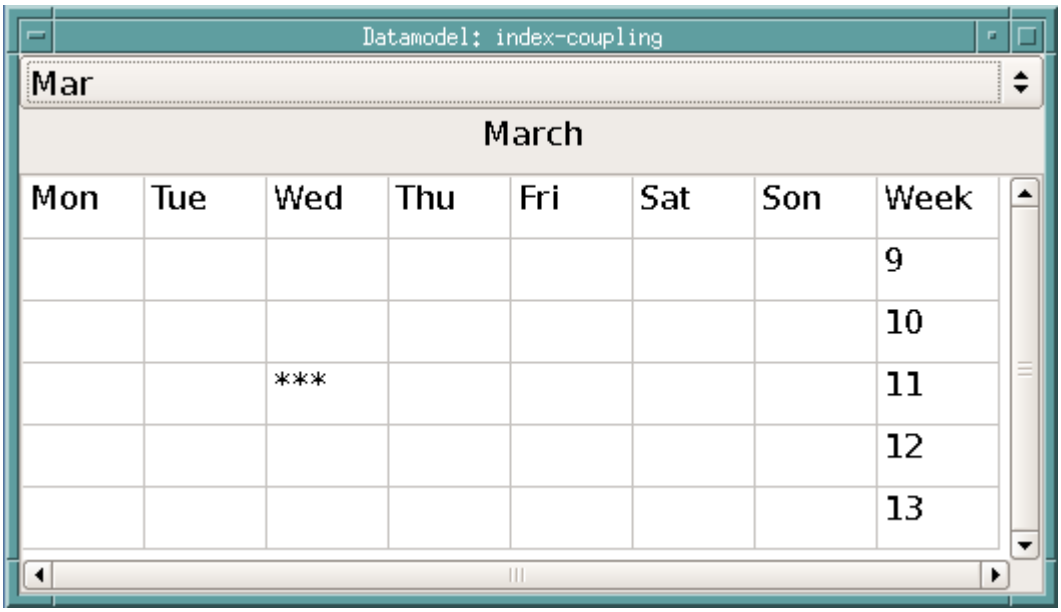

Abbildung 26: *Aus verschiedenen Datenmodellen befülltes Tablefield*

Sehr wichtig ist es folgenden Umstand zu verstehen:

Der Datenaustausch zwischen Model und View funktioniert über die bestehenden Mechanismen des IDM, also **setvalue()**, **getvalue()**, **setvector()** und **getvector()**. Dies impliziert, dass die ausgetauschten Datenwerte entweder skalar oder vektoriell sind.

Wird die Änderung eines Model-Attributs signalisiert (z.B. in einer eigenen Datenfunktion über den Aufruf der **DM_DataChanged()**-Funktion), so liefert diese normalerweise einen Model-Index als Hinweis auf das geänderte Detail mit (typischerweise vom Typ *void*, *integer* oder *index*). Da diese Änderungssignalisierung als *datachanged*-Ereignis vom IDM verarbeitet wird, werden mehrere Änderungen zusammengefasst zur Gesamtänderung (*void*-Index) bzw. überflüssige Ereignisse weggelassen.

Im Zusammenspiel mit der Beeinflussbarkeit der Werteaggregation über das *.dataindex[]*-Attribut hat dies mehrere Konsequenzen und Besonderheiten, die zu beachten sind:

1. Die komplette Befüllung eines zweidimensionalen Attributs (z.B. *.content[]* beim ***Tablefield***) ist problemlos möglich, wenn die Ausrichtung sowie die Spalten- und Zeilenzahl passend zur Datenlänge sind. Eine Anpassung der Tabellengröße erfolgt über die **setvector()**-Funktionalität.
   Die Aktualisierung von Einzelwerteänderungen kann aber nur an der korrekten Zelle im Tablefield erfolgen, wenn das Daten-Model eine Änderungsmeldung mit dem entsprechenden Index abgibt. Eine beliebige Übertragung eines zweidimensionalen Feldes in seiner Zeilen- und Spaltenform an eine bestimmte Stelle in einem Tablefield ist aber nicht möglich, lediglich die Signalisierung von Wertänderungen an einer beliebigen Stelle.
2. Durch die Nutzung von *.dataindex[]* am ***Tablefield*** kann ein skalarer oder vektorieller Datenwert an eine explizite Zelle, Zeile oder Spalte gebracht werden. Eine automatische Zellenerweiterung erfolgt nach den Regeln von **setvalue()** bzw. **setvector()** und hängt dementsprechend auch von der Indextransformation ab. Dabei ist zu beachten, dass benutzerdefinierte Attribute (z.B. Felder) keine automatische Erweiterung erlauben.
3. Analoges gilt auch für Vektor- und Matrixattribute wie sie an ***Listbox***, ***Treeview*** und ***Poptext*** zu finden sind.
4. Der Index für ein Model-Attribut wird genutzt um die unnötige Propagierung von Werteänderungen zu vermeiden (z.B. ist eine Änderung von „Data.Attr[5]“ für View-Objekte uninteressant, die nur mit „Data.Attr[2]“ gekoppelt sind).
5. Bei Nutzung von *void* (Standardwert für *.dataindex[]*) für die Indextransformation erfolgt die Weitergabe des Model-Index, signalisiert durch ein *datachanged*-Ereignis, je nach Attributtyp (skalar, vektoriell, zweidimensional) bis hin zur Präsentation durch **:represent()**. Genutzt unter anderem im Beispiel **randomcolors.dlg** um Farbwerte aus einem assoziativen Feld mit dem *.bgc[]*-Attribut eines ***Tablefield***-Objekts zu koppeln. Allerdings bedeutet dies auch, dass bei Kopplungen mit unterschiedlicher Wertedimension (z.B. ein Feld wird auf einen Skalar abgebildet) sich eine Werteaktualisierung entsprechend dem Index der letzten Änderung ergibt – was meist nicht überraschend ist. Abhilfe schafft hier die Nutzung des korrekten Index für das View- und das Model-Attribut mittels des *.dataindex[]*-Attributs.
6. Ein Setzen von Standardwerten (Index *[0]*, *[0,0]*, *[0,*]* oder *[*,0]*) der beteiligten Model- und View-Attribute ist nicht möglich.

Bei folgenden Beispielen (zu finden im Unterverzeichnis *examples/datamodel/* des IDM-Installationsverzeichnis) lohnt dabei ein Blick in den Quelltext wie auch in das Tracefile bei der Ausführung:

relations.dlg

Zeigt diverse Werteaggregationen von skalaren oder vektoriellen Datenwerten in ein ***Listbox***-Objekt mit und ohne Anpassung des *.content[]*-Felds sowie die Auswirkung bei Einzeländerungen.

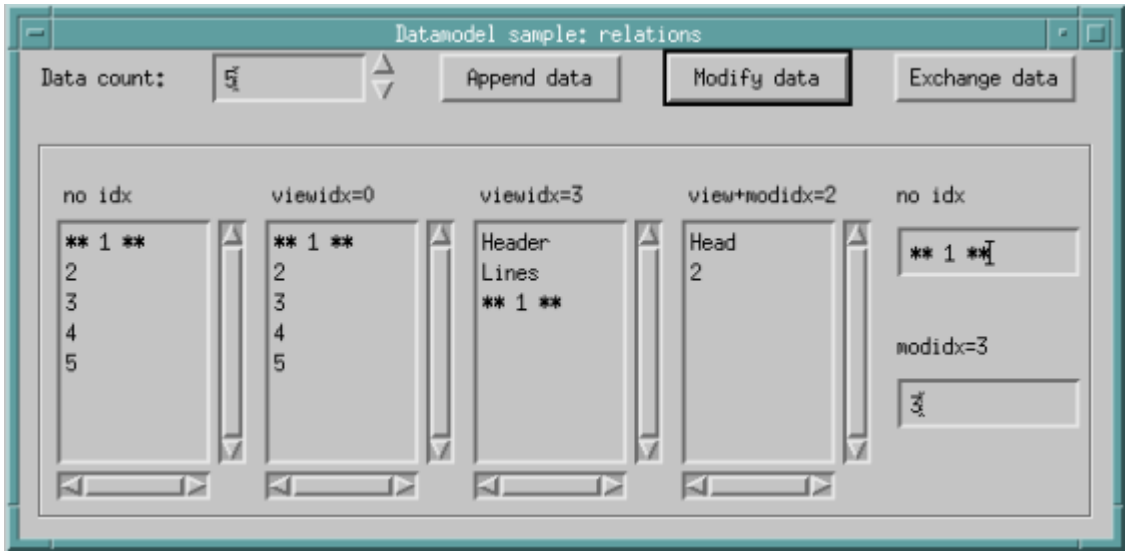

Abbildung 27: *Werteaggregationen für Listenobjekte, Beispiel relations.dlg*

matrixrel.dlg

Zeigt diverse einfache und komplexe Werteaggregationen von verschiedenen Datenwerten in ein ***Tablefield***-Objekt auf.

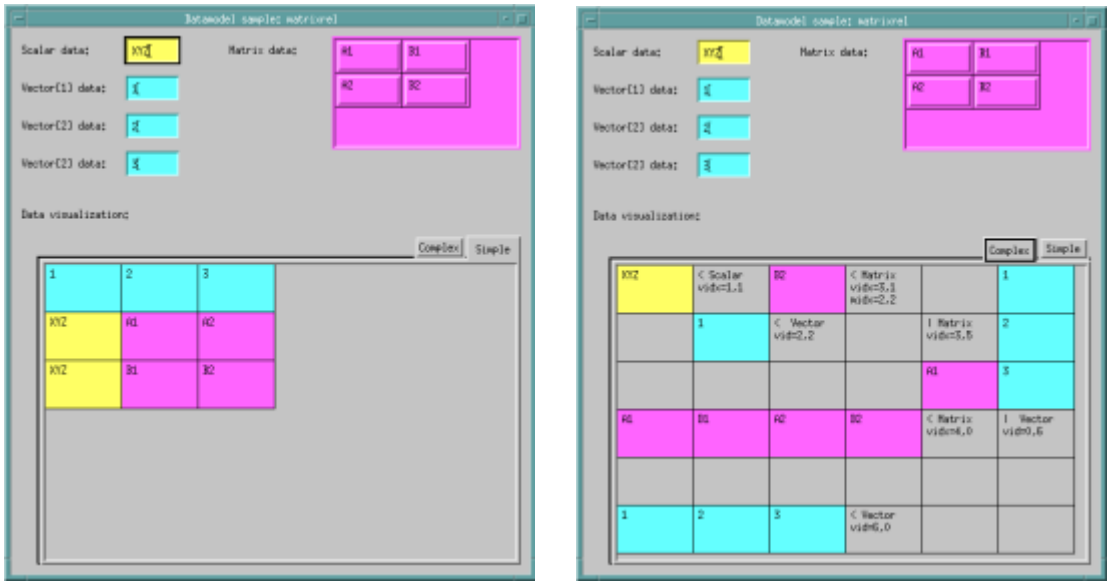

Abbildung 28: *Wertaggregationen für Tablefields, Beispiel matrixrel.dlg*

numbers.dlg

Zeigt die komplette Befüllung eines ***Tablefield***-Objekts mit einer Kopfzeile und inklusive automatischer Anpassung der Zeilenzahl auf.

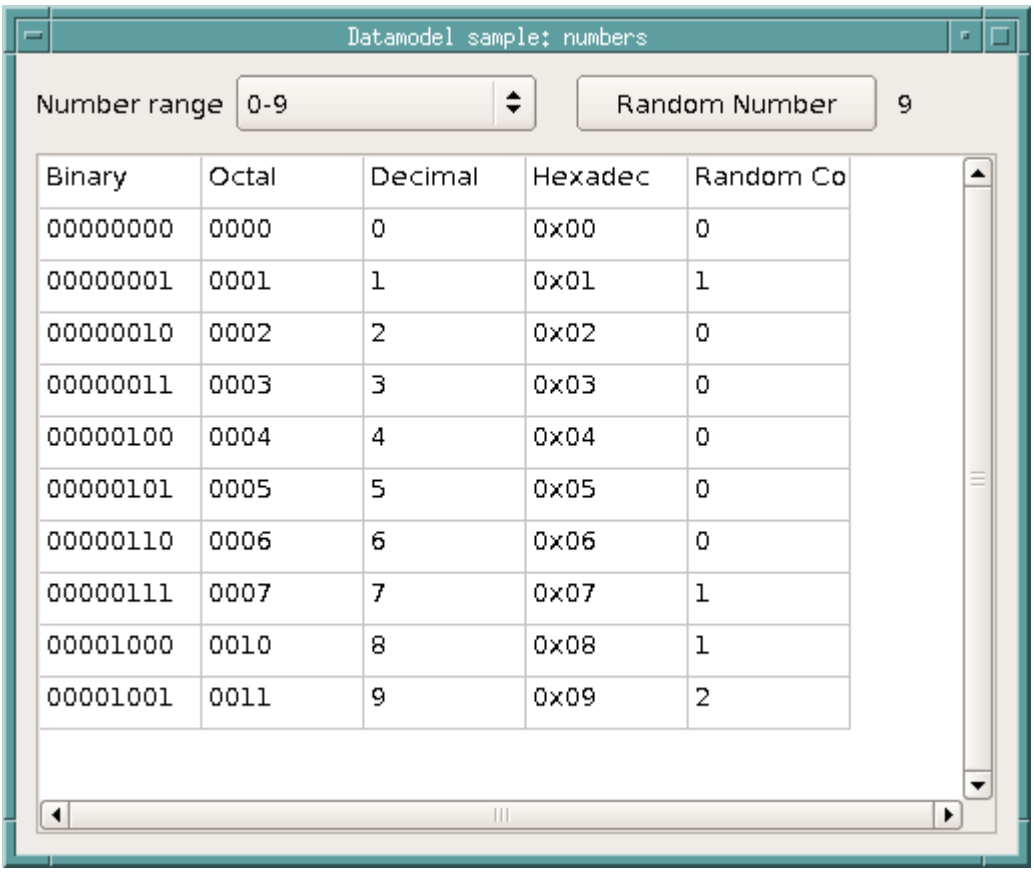

Abbildung 29: *Komplettes Befüllen eines Tablefields, Beispiel numbers.dlg*

puzzle.dlg

Zeigt die Befüllung eines ***Tablefields*** aus einem ***Record*** als Datenmodell oder einer Datenfunktion (lokal bzw. remote) auf.

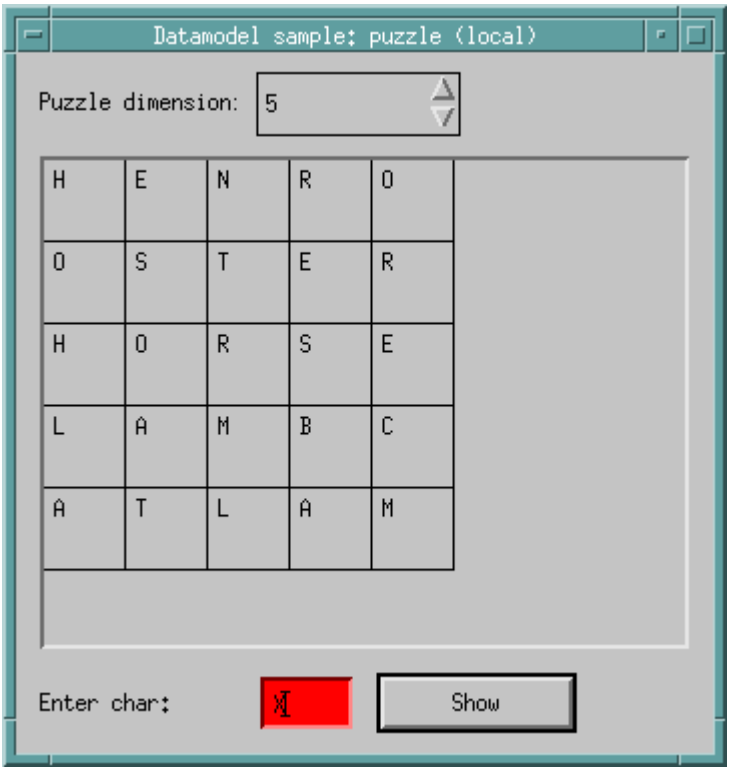

Abbildung 30: *Befüllung eines Tablefields aus Record und Datenfunktion, Beispiel puzzle.dlg*

6.4 Synchronisierung zwischen Model und View

Die automatische Synchronisierung zwischen Model und View wird über das *.dataoptions[]*-Attribut gesteuert.

Tabelle 8: *Optionen zur Synchronisierung zwischen Model und View*

Attribut-Index	Standard	Komponente	Bedeutung
dopt_represent_on_map	***true***	View	Direkt vor der Sichtbarschaltung werden von den Model-Komponenten die Datenwerte geholt und dem View-Objekt gesetzt.
dopt_represent_on_init	***false***	View	In der Objektinitialisierung (**:init**-Methode) werden die Datenwerte geholt und am View-Objekt gesetzt.
dopt_apply_on_unmap	***false***	View	Direkt vor der Unsichtbarschaltung werden die Datenwerte vom View-Objekt geholt und den gekoppelten Model-Komponenten zugewiesen.

Attribut-Index	Standard	Komponente	Bedeutung
dopt_apply_on_event	***false***	View	Wird durch eine Benutzerinteraktion ein Dialogereignis erzeugt, welches auf eine mögliche Änderung eines View-Attributs hinweist, so wird dieses den gekoppelten Model-Komponenten zugewiesen.
dopt_propagate_on_start	***true***	Model	Beim Start des Dialogs bzw. Moduls werden die Daten der Model-Objekte an die gekoppelten View-Komponenten weitergegeben.
dopt_propagate_on_changed	***true***	Model	Änderungen an einem Model-Attribut werden an die gekoppelten View-Komponenten weitergereicht.
dopt_cache_data	***true***	Model	Dieser Indexwert ist nur für den ***doccursor*** verfügbar. ***true*** Die vom ***doccursor*** selektierten Daten werden für weitere Zugriffe zwischengespeichert („Caching“). *false* Der ***doccursor*** selektiert die Daten bei jedem Zugriff neu aus dem XML-Dokument.

Ein ganz wichtiges Merkmal des Datenmodells ist, dass Datenänderungen immer „asynchron“ über die Ereignisverarbeitung gemeldet und abgearbeitet werden. Ist z.B. das Datenmodell (Model-Objekt) ein ***Record*** bzw. ein anderes Objekt mit benutzerdefinierten oder vordefinierten Attributen, so wird eine Attributänderung, die über ein *datachanged*-Ereignis signalisiert wird, an die verbundenen View-Objekte gemeldet damit diese sich aktualisieren können. Der Dialogprogrammierer kann zwar über die Operation „::=“ ein *changed*-Ereignis unterbinden, niemals aber ein *datachanged*-Ereignis.

Eine globale Variable kann ebenso als Model-Komponente verwendet werden, erlaubt aber keine Steuerung der Synchronisation. Es erfolgt bei Variablen immer eine Wertepropagierung bei Start des Dialogs bzw. Moduls und bei Änderung des Variablenwertes.

Wird `.dataoptions[dopt_apply_on_unmap] := true;` an einer **Dialogbox**, ***Messagebox*** oder ***Filereq*** über einen **querybox**-Aufruf genutzt, so erfolgt die Übertragung der Datenwerte von der View zum Model nur bei einer positiven Bestätigung (*button_ok* oder *button_yes*).

Ein manueller Eingriff ist in den meisten Fällen nicht notwendig. Falls doch erforderlich, kann eine manuelle Synchronisation über die folgenden Methoden passieren:

Tabelle 9: *Manuell aufrufbare Datenmodellmethoden*

Methode	Komponente	Funktion
:represent()	View	Die Daten aller gekoppelten Model-Attribute (*.dataget*) werden geholt und dem View-Objekt zugewiesen
:apply()	View	Für alle gekoppelten View-Attribute (*.dataset*) werden die Daten geholt und den Models zugewiesen
:propagate()	Model	Das Model-Objekt propagiert eine Änderung aller seiner Model-Attribute an die View-Objekte, die eine Kopplung aufweisen
:collect()	Model	Das Model-Objekt holt sich alle Daten aus den gekoppelten View-Objekten um sie seinen Model-Attributen zuzuweisen

Eine Synchronisierung findet nur zwischen Instanzen statt, niemals mit Default-Objekten oder Modellen.

Für eine optimierte Synchronisation von Teilwerten gibt es noch eine weitere Besonderheit. Wird eine Wertänderung signalisiert so wird der Index dieser Wertänderung weitergeleitet sodass nur dieser Einzelwert von der View-Komponente zu aktualisieren ist.

Beispiel

Über ein ***Timer***-Objekt werden zufällige Farben ausgewählt und zufälligen Zellen einer Tabelle zugeordnet.

```
dialog D
color CoRed "red";
color CoYellow "yellow";
color CoGreen "green";
color CoBlue "blue";

timer TiRandomColors {
  .active true;
  .starttime "+00:00:00";
  .incrtime "+00:00:00'100";
  object Color[index] := null;

  on select
  {
    variable index RandIndex := [1 + random(5),1 + random(5)];
    variable object RandColor := D.color[1 + random(D.count[.color])];
    /* change the color at a specific index */
```

```
    this.Color[RandIndex] := RandColor;
  }
}

window Wi
 {
  .title "Datamodel - random colors";
  .width 400;  .height 400;

  tablefield Tf
  {
    .xauto 0;  .yauto 0;
    .rowcount 5;  .colcount 5;
    .colwidth[0] 60;  .rowheight[0] 30;
    .dataoptions[dopt_represent_on_map] false;  /* avoid initial :represent()
*/
    .datamodel TiRandomColors;
    .dataget[.bgc] .Color;
  }
}
```

Bildschirmfoto dazu:

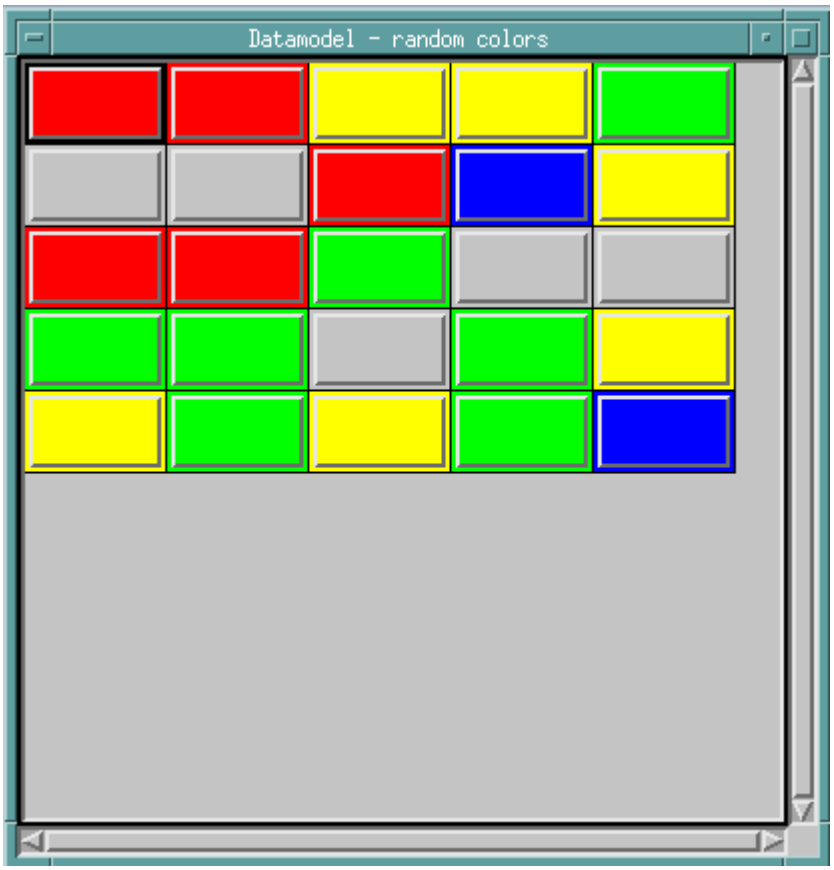

Abbildung 31: *Zufällig gefärbte Tablefield-Zellen mit einem Timer als Datenmodell*

6.5 Konvertierung und Konvertierungsmethoden

Bei der Synchronisation zwischen Model- und View-Komponente werden Datenwerte ausgetauscht, was meist einem Setzen oder Erfragen von Attributen (analog zu **setvalue()**, **getvalue()**, **setvector()**, **getvector()**) entspricht. Ein Eingriff des Anwendungsprogrammierers durch Überdefinition der Methoden **:set()** oder **:get()** ist dabei **nicht** möglich.

Allerdings gibt es für die View-Komponente die Möglichkeit, über die überdefinierbaren Methoden **:represent(<Value>, <Attribute>, <Index>)** und **:retrieve(<Attribute>, <Index>)** Einfluss zu nehmen.

Beim Setzen von Attributen an der View-Komponente wird normalerweise, in der Standardbehandlung durch den IDM, eine Typkonvertierung in den Zieldatentyp durchgeführt. Im nachfolgenden Beispiel wird aufgezeigt, wie durch Überdefinition eine Sonderbehandlung für das View-Attribut *.activeitem* durchgeführt wird.

Beispiel

Eine Flughafenliste wird in der ***Listbox*** „Lb“ angezeigt. Die überdefinierten Methoden **:represent()** und **:retrieve()** sorgen dabei für eine korrekte Umsetzung von und zu einem IATA-Kürzel.

```
dialog D
record RecAirport
{
  string Name[4];
  string IATA[4];
  .Name[1] := "Hartsfield Jackson Atlanta International";
  .IATA[1] := "ATL";
  .Name[2] := "Dubai International";
  .IATA[2] := "DXB";
  .Name[3] := "Stuttgart";
  .IATA[3] := "STR";
  .Name[4] := "Orkney Islands";
  .IATA[4] := "KOI";
  string NearestIATA := "STR";
  string Selected := "";
}

window Wi
{
  .title "Airports";
  .width 200;  .height 200;

  listbox Lb
  {
    .xauto 0;  .yauto 0;
    .ybottom 30;
    .dataoptions[dopt_apply_on_event] true;
    .datamodel RecAirport;
    .dataget .Name;
```

```
    .dataget[.userdata] .IATA;
    .dataget[.activeitem] .NearestIATA;
    .dataset[.activeitem] .Selected;

    :represent()
    {
      if Attribute = .activeitem then
        Value := this:find(.userdata, Value);
      endif
      pass this:super();
    }

    :retrieve()
    {
      if Attribute = .activeitem then
        return this.userdata[this.activeitem];
      endif
      pass this:super();
    }
  }

  edittext Et
  {
    .yauto -1;  .xauto 0;
    .editable false;
    .datamodel RecAirport;
    .dataget .Selected;
  }

  on close { exit(); }
}
```

Bildschirmfoto dazu:

Abbildung 32: *Fenster des Beispieldialogs zum Überschreiben von* ***:represent()*** *und* ***:retrieve()***

6.6 Verwendung von XML mit dem Datenmodell

Verfügbarkeit

Ab IDM-Version A.06.01.b

XML lässt sich als Datamodel nutzen, wobei Knotentexte und Knotenattribute die Daten – in der Regel Strings – enthalten können. Ein XML-Dokument wird mit einem ***Doccursor*** an eine View gekoppelt. Dafür werden am ***Doccursor*** Datamodel-Attribute und Selektionsmuster für Knoten des XML-Dokuments definiert. Weiterhin lässt sich festlegen, ob das Datamodel-Attribut an den Inhalt oder einen Attributwert des Knotens gekoppelt ist. Bei Operationen, die das XML-Dokument ändern, werden die Datenänderungen an alle Datamodel-Attribute weitergereicht.

Der ***Doccursor*** besitzt folgende Attribute um ihn als Datamodel zu verwenden:

Tabelle 10: *Datenmodellattribute des Doccursors*

Attribut	Bedeutung
.dataselect	Definiert ein Datamodel-Attribut mit zugehörigem Selektionsmuster.
.dataselectattr	Ordnet ein Datamodel-Attribut einem Attribut der selektierten Knoten zu.
.dataselecttype	Datentyp-Konvertierung eines Datamodel-Attributs.
.dataselectcount	Definiert die Kardinalität eines Datamodel-Attributs.
.dataoptions	Steuert das Caching über den Index *dopt_cache_data*.

Dabei ist das Attribut *.dataselect* von zentraler Bedeutung. Es definiert die Datamodel-Attribute des ***Doccursors*** sowie deren Verknüpfung mit Knoten des XML-Dokuments mithilfe von Selektionsmustern. Die Syntax der Selektionsmuster entspricht den Musterdefinitionen der **:select**-Methode. Das Selektionsmuster eines *.dataselect*-Attributs ohne Index – also ohne Datamodel-Attribut – wird **vor** den Selektionsmustern aller Datamodel-Attribute angewendet. Dies ermöglicht relative Selektionsmuster für die Datamodel-Attribute, ausgehend von den Knoten, die vom *.dataselect*-Attribut ohne Index vorselektiert worden sind. Gleichzeitig lässt sich dadurch der Aufwand für das Sammeln der Daten reduzieren.

Beim Sammeln der Daten für ein Datamodel-Attribut wird das Dokument anhand des Selektionsmusters in einer Schleife durchlaufen. Dabei wird entweder der Knoteninhalt über das *.text*-Attribut des ***Doccursors*** oder der Wert des über *.dataselectattr* gesetzten Knotenattributs geholt.

Für die mit *.dataselect* definierten Datamodel-Attribute können mit den Attributen *.dataselecttype* und *.dataselectcount* der Datentyp und die Kardinalität geändert werden. Standardmäßig enthalten die Datamodel-Attribute Vektoren mit String-Werten (Datentyp *vector[string]*). Mit dem Attribut *.dataselecttype* kann ein Datentyp (z. B. *integer*, *boolean*) definiert werden, in den die Werte konvertiert werden. Die Kardinalität (Vektor oder Skalar) von Datamodel-Attributen kann über das Attribut *.dataselectcount* gesteuert werden. Wenn der Datentyp des Datamodel-Attributs eine Sammlung oder

seine Kardinalität der Datentyp *integer* ist, dann enthält das Datamodel-Attribut einen *vector*, ansonsten nur einen Skalar mit dem ersten Wert.

Einmal geholte Werte eines Datamodel-Attributs werden zwischengespeichert bis sich entweder das XML-Dokument oder die Datenmodellattribute des ***Doccursors*** ändern. Dieses Zwischenspeichern („Caching“) kann durch Setzen von *.dataoptions[dopt_cache_data] = false* am ***Doccursor*** unterbunden werden.

Das Speichern von Daten in einem XML-Dokument erfolgt analog mit sinngemäß umgedrehten Operationen. Allerdings können im XML-Dokument keine Knoten automatisch angelegt oder gelöscht werden.

Hinweise

» Das Sammeln der Daten kann eine „teure“ Operation sein, da im gesamten XML-Dokument nach dem Selektionsmuster gesucht werden muss. Der Aufwand lässt sich möglicherweise reduzieren, indem die Suche mithilfe eines *.dataselect*-Attributs ohne Index auf vorselektierte Teilbäume beschränkt wird.

» Ein ***Doccursor***, der als Datenmodell verwendet wird, sollte nicht für andere Zwecke genutzt werden, da er typischerweise ständig seinen Selektionspfad ändert.

6.6.1 Beispiel

Eine Liste von Nobelpreisträgern soll aus folgender XML-Datei ausgelesen und in einer Tabelle dargestellt werden:

```
<?xml version="1.0"?>
<nobelprizes>
  <category id="p">Physics</category>
  <category id="c">Chemistry</category>
  <winner year="1" category="p">Wilhelm Conrad Röntgen</winner>
  <winner year="11" category="c">Marie Curie</winner>
  <winner year="18" category="p">Max Planck</winner>
  <winner year="70" category="c">Luis Leloir</winner>
</nobelprizes>
```

Die Namen der Preisträger werden aus dem Inhalt der XML-Knoten „winner“ geholt und im Datamodel-Attribut „.Winner“ abgelegt. Die Jahreszahlen der Auszeichnung stehen im Attribut „year“ der XML-Knoten und werden in der Datei ab 1900 gezählt. Sie werden zu *integer* gewandelt und als *vector* in das Datenmodell-Attribut „.Year“ eingelesen. In der überschriebenen **:represent**-Methode werden die Jahreszahlen für die Anzeige zu vierstelligen Zahlen ergänzt.

```
dialog D

document Doc
{
  doccursor DocCur
```

```
  {
    .dataselect[.Winner]    "..winner";
    .dataselect[.Year]      "..winner";
    .dataselectattr[.Year]  "year";
    .dataselecttype[.Year]  integer;
    .dataselectcount[.Year] integer;
  }
}

window Wi
{
  .title "Nobel prize winners";
  .width 300;  .height 220;

  tablefield Tf
  {
    .xauto 0;  .yauto 0;
    .datamodel DocCur;
    .colcount 2;  .rowcount 1;
    .rowheader 1;  .colheader 1;
    .rowheight[0] 25;
    .colwidth[0] 180;  .colwidth[1] 60;
    .content[1,1] "Year";
    .content[1,2] "Winner";
    .dataget[.field]     .Winner;
    .dataget[.userdata]  .Year;
    .dataindex[.userdata] [0,1];

    :represent()
    {
      variable integer I;
      if Attribute=.userdata then
        for I := 1 to itemcount(Value) do
          Value[I] := 1900 + (Value[I] % 100);
        endfor
        setvector(this, .content, Value,[2,1],
                  [1 + itemcount(Value),1]);
        return;
      endif
      pass this:super();
    }
  }

  on close { exit(); }
}

on start
```

```
{
  Doc:load("nobelprizes.xml");
}
```

Dieser Dialog erzeugt folgendes Fenster:

Abbildung 33: *Tabelle mit XML-Daten als Datenmodel*

6.6.2 Indexwert *dopt_cache_data* des Attributs *dataoptions*

Attribut-Index	Default	Komponente	Bedeutung
dopt_cache_data	***true***	Model	Dieser Indexwert ist nur für den ***doccursor*** verfügbar. **true** Die vom ***doccursor*** selektierten Daten werden für weitere Zugriffe zwischengespeichert („Caching“). **false** Der ***doccursor*** selektiert die Daten bei jedem Zugriff neu aus dem XML-Dokument.

6.7 Aktionen

Um Manipulationsfunktionen der Model-Komponente allgemein für die Presenter-Logik zugänglich zu machen, dient das Konzept der Aktionen. Dies sind Methoden mit Parametern, welche von den Datenmodellen zur Verfügung gestellt werden können. Aufrufbar sind diese über die **:calldata()**-Methode.

6.8 Ablaufverfolgung

Folgende Trace-Codes wurden zusätzlich eingeführt um die Synchronisationsabläufe zwischen Datenmodellen (Models) und Präsentationsobjekten (Views) sichtbar zu machen und um mögliche Anwendungsfehler aufzudecken.

Trace-Code	Bedeutung
[CD]	„Call Data“ um den Aufruf der Synchronisation zwischen View und Model zu verfolgen. Tracing erfolgt typischerweise für **:get()**und **:set()** am Model-Objekt sowie **:represent()** und **:apply()** am View-Objekt.
[CE]	„Data Changed Event“ – Ereignis das eine Änderung am Datenmodell signalisiert, typischerweise durch das Setzen eines Attributs oder durch den Aufruf der Applikationsfunktion **DM_DataChanged()** ausgelöst.

6.9 Restriktionen

Von IDM abgelehnt werden Änderungen an der Datenkopplung oder der Steuerung der Synchronisation (Attribute *.datamodel*, *.dataget*, *.dataset*, *.datamap*, *.dataindex*, *.dataoptions*) während man innerhalb einer überdefinierbare Methode **:represent()** oder **:retrieve()** bzw. eines Aufrufs einer Datenfunktion ist.

Ebenso vom IDM abgelehnt werden Änderungen an den Attributen *.visible* oder *.mapped*, wenn man diese innerhalb der Synchronisation der View-Komponente ausführt und der IDM sich gerade bei der Umsetzung der Sichtbar- oder Unsichtbarkeitsschaltung befindet.

7 Multiscreen Support unter Motif

Der IDM für Motif verfügt über eine Programmier-Unterstützung für mehrere Screens.

Bedeutung von Multiscreen-Support unter X

Damit ist eine X-Server-Konfiguration mit mehreren Screens (mehrere Bildschirme entweder durch eine Grafikkarte mit mehreren Frame-Buffern oder durch mehrere Grafikkarten im gleichen Display-Host) gemeint. Dabei können sich die Screens in der Auflösung wie auch in den Farbmöglichkeiten unterscheiden. Sie teilen sich hingegen die Eingabegeräte wie Maus und Tastatur. Außerdem lässt sich ihre Anordnung zueinander normalerweise in der X-Server-Konfiguration definieren (z.B. *Screen#1* ist rechts von *Screen#0*).

Auch bisher schon konnte eine IDM-Anwendung in einem spezifischen Screen angezeigt werden, z.B. über die X-Option *-display*`<host>:<display>.<screen>`. **Neu** ist, dass innerhalb einer IDM-Anwendung **Fenster in verschiedenen Screens** erscheinen können.

Unterstützung durch den IDM

Der IDM unterstützt den Anwendungsprogrammierer dahingehend, dass er die verfügbaren Screens und ihre Merkmale erfragen kann (dies geschieht über das setup-Objekt, siehe dazu auch die Attribute .screencount, .screen, .screen[integer] usw.).

Außerdem erlaubt der IDM die Zuordnung, welches Fenster auf welchem Screen darzustellen ist. Dies geschieht über das .display-Attribut am Fenster. Es muss auf eine display-Resource gesetzt werden, die vom Anwender zu definieren ist. Der IDM sorgt dann für die notwendige Resourcen-Verwaltung bei Farben, Cursor und Bilder.

Eine dynamische Umschaltung des Fensters auf einen anderen Screen kann entweder über das .display-Attribut oder die display-Ressource geschehen. Für einen Beispieldialog siehe auch die Dokumentation zur display-Ressource.

Dialogfenster wie Messageboxen oder der Filerequestor werden im gleichen Screen dargestellt wie der beim querybox-Aufruf angegebene Vater. Ist kein Vater angegeben so erfolgt die Sichtbarmachung im Default-Screen.

Anmerkungen

- Das Layout der Screens zueinander ist über X/Motif leider nicht ermittelbar und somit vom IDM aus auch nicht zugänglich.
- Der Speicherbedarf des IDMs ist für den Standardfall der Single-Screen-Anwendung optimiert.
- Multiscreen und Non-Default Visuals: Der IDM erlaubt über die Umgebungsvariable IDM_VISUAL_ID die Angabe einer Visual-ID um ein anderes Visual als das vorgegebene zu benutzen (z.B. Graustufen-Screen auf einem TrueColor-Display). In einer Multiscreen-Konfiguration wirkt sich diese Angabe nur auf den Default-Screen aus, d.h. dass die Fenster in anderen Screens im Default-Visual dargestellt werden.

» Die Unterstützung der neu hinzugekommenen Attribute und Ressourcen ist für den IDM Editor jetzt gegeben.

Siehe auch

setup, .screen, .real_screen, display, .display, .xdpi, .ydpi, .screen_width, .screen_width[integer], .screen_height, .screen_height[integer], .pointer_width, .pointer_height, .color_type, .colorcount, .screencount

Verfügbarkeit

Ab IDM-Version A.05.01.c

Diese Unterstützung ist in der IDM 4-er Version ab A.04.04.j ebenfalls verfügbar.

8 Multi-Monitor Support unter Windows

Unter Windows ist es jetzt möglich, die Anzahl der Monitore (.screencount) und die Koordinaten der Monitore (.screen_x[integer], .screen_y[integer], .screen_width[integer] und .screen_height[integer]) über das setup-Objekt abzufragen. Die erfragten Werte sind dabei dynamisch, da man jederzeit die Skalierung eines Monitors ändern oder auch Monitore hinzufügen oder wegnehmen kann.

Werden mehrere Monitore verwendet, werden diese in einem virtuellen Desktop positioniert. Die Koordinaten dieses virtuellen Desktop sind mit .vscreen_x, .vscreen_y, .vscreen_width, .vscreen_height abfragbar. Es werden immer die Koordinaten des Arbeitsbereichs, also ohne die Taskleiste, geliefert. Zur Berechnung des virtuellen Desktops werden auch die Arbeitsbereiche herangezogen.

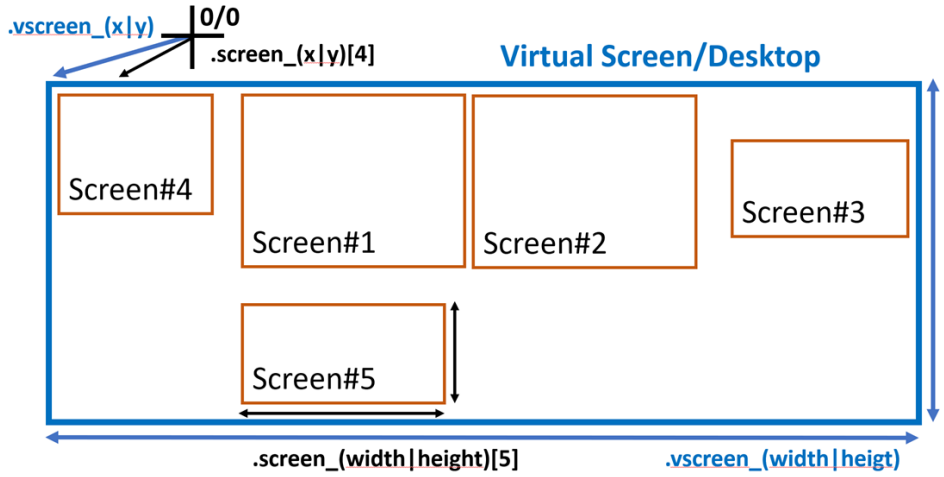

Abbildung 34: *Relation der .screen_*[] und .vscreen_* Attribute*

Achtung:

Bei unterschiedlichen Vergrößerungseinstellungen der Monitore ist zu beachten:

IDM für Windows 11

Die Werte erscheinen nicht konsistent. Ausgehend von der für Fenster empfohlenen Verwendung von *.xauto = 1* und *.yauto = 1* wird die Position mit dem Systemvergrößerungsfaktor umgerechnet, während die Größe mit dem Monitorvergrößerungsfaktor umgerechnet wird. Hierdurch lässt sich ein Fenster auf eine Position und Größe setzen, die den gesamten Monitor ausfüllt.
Die Position und die Größe des virtuellen Desktops werden mit dem Systemvergrößerungsfaktor umgerechnet. Zu welchem Monitor ein Fenster, das sich über mehrere Monitore erstreckt, zugeordnet wird, kann von dem Monitor abhängen, auf dem sich das Fenster gerade befindet. Für ein definiertes Verhalten sollten deshalb Koordinaten nur im unsichtbaren Zustand geändert werden.

IDM für Windows 10

Microsoft Windows skaliert Anwendungen, die DPI-Unaware sind, intern. Hierbei führen die unterschiedlichen Vergrößerungsfaktoren zu Falschdarstellungen. Um dies zu vermeiden sollten

entweder alle Fenster nur auf dem primären Monitor geöffnet werden, oder alle Monitore denselben Vergrößerungsfaktor besitzen.

Anmerkung zu Windows 10 und 11 mit mehreren Monitoren:

Unter WINDOWS kann es nach dem Hinzufügen/Entfernen eines Monitors oder dem Ändern der Anzeigeeinstellungen zu folgenden Problemen kommen:

- Es erscheint kein Verschieberahmen beim Verschieben eines **toolbar** Objektes. Es kann auch vorkommen, dass ein Rechteck in der Größe des **toolbar** Verschieberahmens mit einem falschen Hintergrund auf einem anderen Monitor dargestellt wird.
- Ausgedockte **toolbar** Objekte passen sich nicht an Änderungen der DPI Wertes an. Sowohl beim Verschieben auf einen anderen Monitor oder auch beim Ändern der Vergrößerung.

Das Problem liegt an WINDOWS, das zum einen die Koordinaten beim Zeichnen auf den Desktop falsch umsetzt und zum anderen wichtige Nachrichten (*WM_DPICHANGED*) nicht an sogenannte Tool-Fenster versendet und den Window DPI Wert nicht ändert.

Nachdem der Bildschirmschoner aktiv war oder man sich neu angemeldet hat (und nach einem Neustart), tritt das Problem nicht mehr auf.

Siehe auch

Attribute .screencount, .screen_x[integer], .screen_y[integer], .screen_width[integer], .screen_height [integer], .vscreen_x, .vscreen_y, .vscreen_width, .vscreen_height

Verfügbarkeit

Ab IDM-Version A.06.03.a

9 HighDPI Unterstützung

Die Auflösung moderner Bildschirme wächst stetig. Damit ändern sich auch die Anforderungen an Design und Qualität der Bedienelemente und Oberflächen. Probleme sind dabei häufig, dass Anwendungen viel zu klein erscheinen und Text und Grafiken unscharf oder pixelig dargestellt werden. Die im System eingestellte Auflösung, DPI und Skalierung sowie Fonts und Größe und Qualität der Bilder beeinflussen das Darstellungsbild einer Anwendung. Damit bestehende Anwendungen auch auf hochauflösenden Bildschirmen detailreich und scharf dargestellt werden, müssen diese auf HighDPI vorbereitet werden.

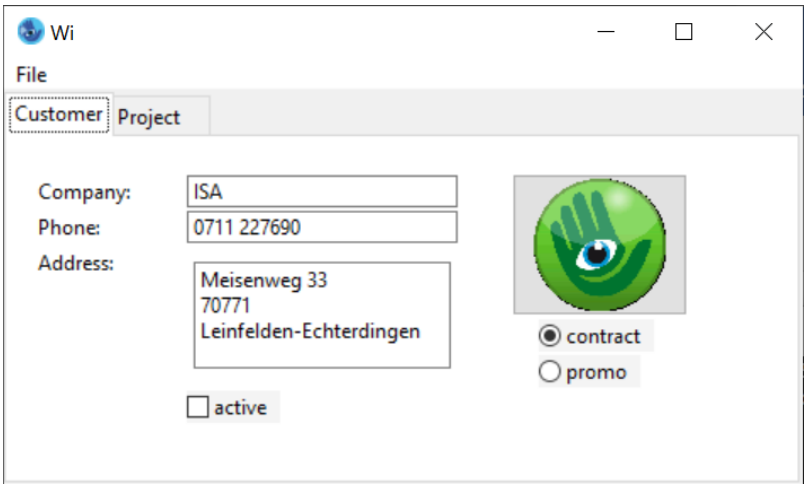

***Abbildung 35:** ohne HighDPI Unterstützung*

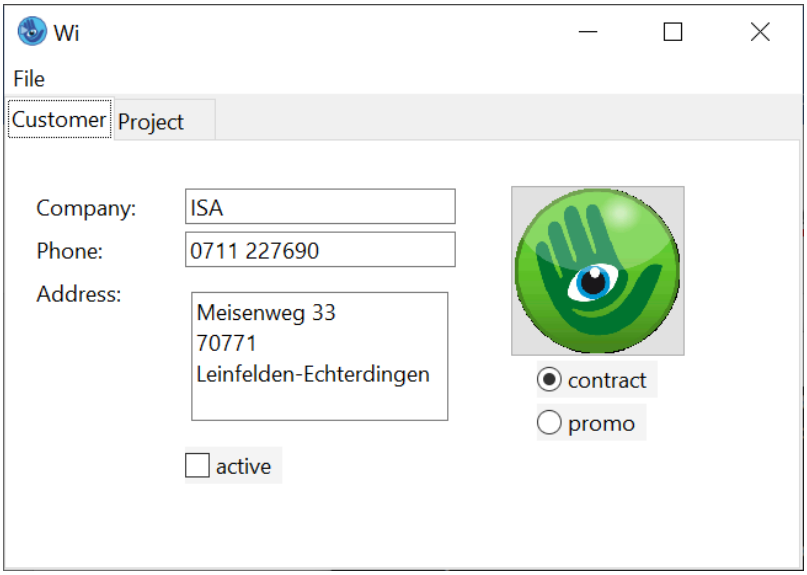

***Abbildung 36:** mit HighDPI Unterstützung*

Der IDM für MOTIF, QT und MICROSOFT WINDOWS ab Version A.06.03.a (unter MICROSOFT WINDOWS nur der IDM FÜR WINDOWS 11) bietet nun die geeigneten Werkzeuge um hochauflösende Bildschirme

zu unterstützen und Anwendungen zukunftssicher zu gestalten. Außerdem wurde das Arbeiten mit mehreren Bildschirmen verbessert.
IDM-Anwendungen beachten nun die DPI-Einstellungen und Skalierungsfaktoren des Systems. Bisherige Probleme wie pixelige, matschige oder zu klein angezeigte Anwendungen oder verringerte Schriftgrößen bei hohen Auflösungen und geringer Bildschirmgröße (die Effekte können je nach System variieren) treten somit nicht mehr auf. Der IDM passt Geometrie, Layout, Schriftgrößen, Grafiken sowie Cursor nun automatisch an die entsprechende Auflösung und vom System bestimmte DPI an. Der Skalierungsfaktor basiert dabei auf den Systemeinstellungen der jeweiligen Desktop-Umgebung. Es steht dem Anwender aber frei durch verschiedene neue Attribute, Optionen und Mechanismen andere Vorgaben zu machen bzw. gestalterisch einzugreifen.

9.1 Startoptionen

-IDMscale <integer>

Über die Startoption **-IDMscale** kann die Skalierung durch den IDM aktiviert oder deaktiviert werden. Unter QT und MOTIF gibt der Wert die Skalierung in % an.

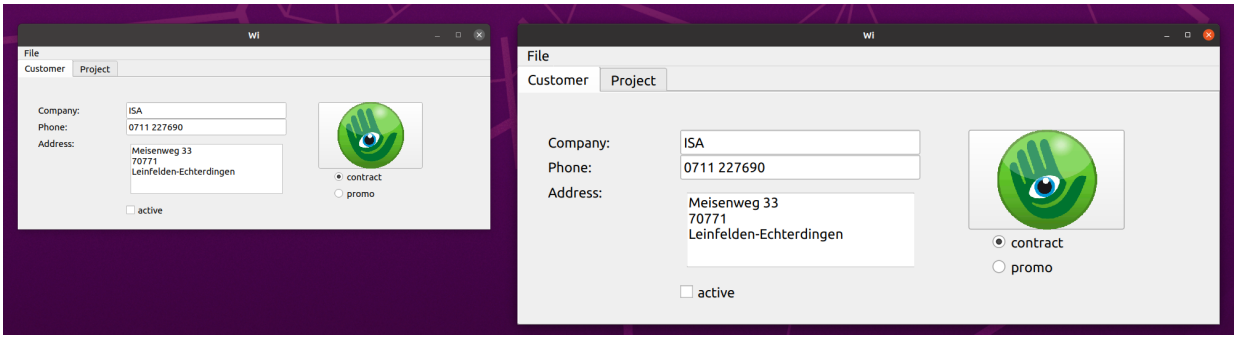

Abbildung 37: *links normal gestartet (Systemskalierung 100%), rechts mit 150% Skalierung gestartet*

Hinweis

Diese Option sollte unter MICROSOFT WINDOWS nicht verwendet werden. Die DPI-Awareness ist eine Eigenschaft der Anwendung und sollte in einer Manifest-Datei spezifiziert werden. Für Testzwecke kann die DPI-Awareness eingeschaltet (Wert: *1*) oder ausgeschaltet (Wert: *0*) werden.

Achtung:

Es wird empfohlen eine Skalierung > 0 und < 100% nicht zu verwenden, da es hier zu Einschränkung in der Darstellung und Bedienung von Objekten kommen kann.

-IDMtiledpi <integer>

Grafiken werden automatisch entsprechend des eingestellten Vergrößerungsfaktor skaliert. Es wird hierbei davon ausgegangen, dass die Bilder für einen DPI-Wert von *96* entworfen wurden. Sind die Bilder der Anwendung für eine höhere Auflösung entworfen worden, kann diese über die Startoption -IDMtiledpi eingestellt werden. Die Größe eines Bildes wird dann ausgehend von diesem Wert auf den aktuell geltenden DPI-Wert umgerechnet und dementsprechend skaliert.

Ebenso kann diese Einstellung direkt am setup-Objekt mit dem Attribut .tiledpi (siehe Erweiterung am Setup-Objekt) vorgenommen werden.

9.2 Layout-Ressourcen

Bisher waren die verfügbaren, vordefinierten Ressourcen beim IDM immer WSI- bzw. systemabhängig und bedurften bei Übertragung auf andere Systeme immer aufwändiger Anpassungen. Der IDM A.06.03.a bietet nun erstmals WSI-unabhängige und HighDPI-konforme, vordefinierte Layout-Ressourcen an. Dazu gehören folgende Ressourcen, die auf allen Systemen in ähnlicher Ausprägung bzw. Bedeutung verfügbar sind: font (Zeichensatz), color (Farbe), cursor.
Das bietet den enormen Mehrwert, dass Dialoge, die komplett auf die neuen UI-Ressourcen umgestellt wurden, nun einfach und ohne Aufwand auf andere Systeme übertragen werden können.
Erkennbar sind diese Ressourcen über das Namensschema ***UI*_FONT***, ***UI*_COLOR*** bzw. ***UI*_CURSOR***.

Beispiel:

```
module Resc

font FontNormal "UI_FONT";
font FontBig "UI_FONT", 16;
font FontFixed "UI_FIXED_FONT";

cursor CurStop "UI_STOP_CURSOR";

!! Background color for group objects
color ColGrp "UI_BG_COLOR";
!! Background color for input objects
color ColInp "UI_INPUTBG_COLOR";
```

Die UI_-Ressourcennamen sind ebenfalls – wie alle anderen vordefinierten Ressourcennamen - am setup-Objekt über die Attribute .colorname[integer], .fontname[integer], .cursorname[integer] abfragbar.

Cursor

Die einheitlichen Cursor-Ressourcen haben das Namensschema ***UI*_CURSOR*** und sind wie folgt definiert:

UI_CURSOR	Standardcursor meist Pfeil
UI_IBEAM_CURSOR	Edittext-Einfügemarke
UI_WAIT_CURSOR	Anzeige, dass die Anwendung beschäftig ist, „Eieruhr“
UI_CROSS_CURSOR	Kreuz

UI_UP_CURSOR	Pfeil nach oben
UI_SIZEDIAGF_CURSOR	Sizing-Pfeil von links oben nach rechts unten
UI_SIZEDIAGB_CURSOR	Sizing-Pfeil von links unten nach rechts oben
UI_SIZEVER_CURSOR	Sizing-Pfeil von oben nach unten
UI_SIZEHOR_CURSOR	Sizing-Pfeil von links nach rechts
UI_MOVE_CURSOR	Verschiebe-Pfeil in alle Richtungen
UI_STOP_CURSOR	Verbotszeichen (durchgestrichener Kreis)
UI_HAND_CURSOR	Handsymbol (Zeigefinger dient zum Zeigen)
UI_HELP_CURSOR	Pfeil mit Fragezeichen (für Kontexthilfe-Modus)

Fonts

Die einheitlichen Font-Ressourcen haben das Namensschema ***UI*_FONT*** und sind wie folgt definiert:

UI_FONT	Schriftart, die standardmäßig für Oberflächenelemente verwendet wird (Default oder Dialog Font).
UI_FIXED_ FONT	Schriftart mit einer festen Buchstabenbreite.
UI_MENU_ FONT	Schriftart, die für Menüs verwendet wird.
UI_TITLE_ FONT	Schriftart, die in der Fensterüberschrift verwendet wird.
UI_ SMALLTITLE_ FONT	Schriftart, die in einer kleinen/niedrigen Fensterüberschrift verwendet wird.
UI_ STATUSBAR_ FONT	Schriftart, die in der Statusbar verwendet wird.
UI_NULL_FONT	Verhalten als wenn keine Font-Vorgabe gemacht würde (null). Das WSI benutzt den für das Oberflächenelement vorgesehenen Standard-Systemfont oder einen Systemfallback-Font.

Color

Die Farb-Ressourcen haben das Namensschema ***UI*_COLOR*** und sind wie folgt definiert:

UI_BG_ COLOR	Farbe, die standardmäßig für allgemeine Hintergründe von (meist Gruppierungs-) Objekten verwendet wird. Verwendung bei Objektklassen mit Attribut .bgc/.bgc[], z.B. Fenster, Groupbox, Notepage, Statictext, Pushbutton...
UI_FG_COLOR	Farbe, die standardmäßig für allgemeine Vordergründe/Textfarbe von (meist Gruppierungs-) Objekten verwendet wird. Verwendung bei Objektklassen mit Attribut .fgc/.fgc [], z.B. Fenster, Groupbox, Notepage, Statictext, Pushbutton...
UI_INPUTBG_ COLOR	Farbe, die standardmäßig bei Hintergründen von Eingabe-/Auswahl-Objekten verwendet wird. Verwendung z.B. bei Edittext, Listen, Tabellen, Poptext, Treeview…(z.B. Objektklassen mit Attributen .bgc/.textbgc)
UI_INPUTFG_ COLOR	Farbe, die standardmäßig bei Vordergründen von Eingabe-/Auswahl-Objekten verwendet wird, wie z.B. Edittext, Listen, Tabellen, Poptext, Treeview (z.B. Objektklassen mit Attributen .fgc/.textfgc)
UI_ BUTTONBG_ COLOR	Farbe, die standardmäßig beim Hintergrund von Button-artigen Objekten, z.B. Pushbutton oder Image, verwendet wird (z.B. Objektklassen mit Attribut .bgc/.imagebgc).
UI_ BUTTONFG_ COLOR	Farbe, die standardmäßig beim Vordergrund von Button-artigen Objekten, z.B. Pushbutton oder Image, verwendet wird (z.B. Objektklassen mit Attribut. fgc/.imagefgc).
UI_BORDER_ COLOR	Farbe, die standardmäßig als Rahmen, Rand oder Begrenzung verwendet wird (z.B. an .bordercolor).
UI_ ACTIVEBG_ COLOR	Farbe, die standardmäßig als Hintergrundfarbe des aktiven Elements z.B. bei Listen, Poptext, Tabellen, ggf. Progressbar, Markierungen etc. verwendet wird. Meist Invertierung der normalen Vorder- und Hintergrundfarben des Widgets (z.B. Objektklassen mit Attribut .bgc).
UI_ ACTIVEFG_ COLOR	Farbe, die standardmäßig als Vordergrund-/Textfarbe des aktiven Elements z.B. bei Listen, Poptext, Tabellen, ggf. Progressbar, Markierungen etc. verwendet wird. Meist Invertierung der normalen Vorder- und Hintergrundfarben des Objekts (z.B. Objektklassen mit Attribut .fgc).
UI_TITLEBG_ COLOR	Farbe, die standardmäßig als Hintergrundfarbe bei Titelleisten oder Umrandungen von Fenstern/Dialogen verwendet wird, sofern unterstützt (z.B. an Attribut .titlebgc).

UI_TITLEFG_ COLOR	Farbe, die standardmäßig als Vordergrund-/Textfarbe bei Titelleisten oder Umrandungen von Fenstern/Dialogen verwendet wird, sofern unterstützt (z.B. an Attribut .titlefgc).
UI_MENUBG_ COLOR	Farbe, die standardmäßig als Hintergrundfarbe bei Menüs verwendet wird, sofern unterstützt (z.B. an Attribut .titlebgc).
UI_MENUFG_ COLOR	Farbe, die standardmäßig als Vordergrund-/Textfarbe bei Menüs verwendet wird, sofern unterstützt (z.B. an Attribut .titlefgc).
UI_ HEADERBG_ COLOR	Farbe, die standardmäßig als Hintergrundfarbe bei Tabellenheadern – sofern unterstützt – verwendet wird.
UI_ HEADERFG_ COLOR	Farbe, die standardmäßig als Vordergrund-/Textfarbe bei Tabellenheadern – sofern unterstützt – verwendet wird. Verwendung bei Tablefield (z.B. an Attribut .rowheadfgc, .cloheadfgc).
UI_NULL_ COLOR	Verhalten, als wenn keine Farbe (null) gesetzt wäre. WSI zeichnet nach seiner Standardpalette.

Es kann vorkommen, dass sich nicht alle UI*COLORS voneinander unterscheiden. Das liegt daran, dass die meisten Desktopfarben aus wenigen Basisfarben "berechnet" werden.

Hinweise Motif:

Neben den ***UI*_COLOR*** Farben wurden die Systemnamen für Farb-Ressourcen vervollständigt. Neu hinzugekommen sind:

- INACTIVE_BACKGROUND
- INACTIVE_FOREGROUND
- INACTIVE_TOP_SHADOW
- INACTIVE_BOTTOM_SHADOW
- INACTIVE_SELECT
- SECONDARY_BACKGROUND
- SECONDARY_FOREGROUND
- SECONDARY_TOP_SHADOW
- SECONDARY_BOTTOM_SHADOW
- SECONDARY_SELECT

Hinweise Windows:

Die ***UI*_COLOR*** Farben greifen auf die Windows Systemfarben zu. Hierbei ist zum einen zu beachten, dass mit Einführung der Visual Styles von den einzelnen Objekten nicht mehr die Systemfarben, sondern die durch das Theme definierten Farben verwendet werden. Je nach

ausgewähltem Theme kann es somit starke Diskrepanzen geben. Deshalb sollten unter Windows nach Möglichkeit keine Farben (oder besser die ***UI_NULL_COLOR***) gesetzt werden.
Zum anderen ist zu beachten, dass mit Windows 10 die Farbvielfalt verringert wurde. Die Farben ***UI_HEADERBG_COLOR***, ***UI_HEADERFG_COLOR***, ***UI_MENUBG_COLOR, UI_MENUFG_COLOR***, ***UI_TITLEBG_COLOR*** und ***UI_TITLEFG_COLOR*** werden zum Beispiel nicht mehr unterstützt. Das heißt, dass diese Farben für den Anwender zwar noch verfügbar und nutzbar sind, sie aber aktuell von keinem Objekt defaultmäßig verwendet werden.

Hinweise Qt:

Die ***UI*_COLOR*** Farben greifen auf die von Qt angebotenen Farben der Standard-Palette zu. Für die Farbausprägung wird dabei die ColorGroup Normal bzw. Active zugrunde gelegt. Die Farben der Standard-Palette wiederum werden durch den aktuell eingestellten Systemstyle vorgegeben.

Neben den ***UI*_COLOR*** Farben wurden die vordefinierten Farben um weitere vom Qt-Toolkit angebotenen ColorRoles vervollständigt. Neu hinzugekommen (in allen Ausprägungen der ColorGroups) sind:

- BRIGHTTEXT
- ALTERNATEBASE
- TOOLTIPBASE
- TOOLTIPTEXT
- LINK
- LINKVISITED
- PLACEHOLDERTEXT

9.3 Erweiterung an der Tile-Ressource

Die bestehenden Skalierungsoptionen an der Tile-Ressource wurden erweitert. Es kann nun ein .scalestyle angegeben werden, der bestimmt wie ein Tile skaliert werden soll. So können Muster (tiles) nun nach folgenden Merkmalen skaliert werden:

- automatisch
- um einen bestimmten Faktor
- proportional
- in alle Richtungen frei
- an den DPI-Wert anpassen und keine Skalierung

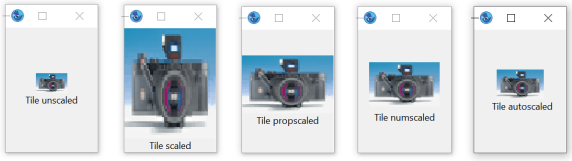

Abbildung 38: *das gleiche Bild mit verschiedenen Ausprägungen des Attributs .scalestyle bei einer Systemskalierung von 150%*

In der Regelsprache stehen dabei folgende Werte zur Auswahl:

Definition am Tile	Dynamische Setzung	Bedeutung
noscale	scalestyle_ none	Das Muster bzw. Bild wird nicht skaliert., setup.tiledpi hat keine Auswirkungen.
scale	scalestyle_any	Höhe und Breite des Musters bzw. Bildes werden voll auf die verfügbare Fläche vergrößert.
propscale	scalestyle_ prop	Höhe und Breite des Musters bzw. Bildes werden auf die verfügbare Fläche vergrößert, wobei Höhen- und Breitenproportionen des Musters bzw. Bildes auf jeden Fall erhalten bleiben. D.h. es können oben und unten bzw. links und rechts Freiflächen entstehen.
numscale	scalestyle_ num	Die Skalierung des Musters bzw. Bildes wird durch einen numerischen Skalierungsteiler vorgenommen. Dabei wird die Skalierung in Viertel-Schritten vorgenommen, also 1.25-fach, 1.5-fach, 1.75-fach, 2-fach, 2.25-fach, 2.5-fach, usw. Eine Verkleinerung finded bis maximal 0.25-fach statt.
dpi	scalestyle_dpi	Das Muster bzw. Bild wird immer entsprechend der eingestellten Bildschirmskalierung skaliert.
Keine Angabe	scalestyle_ auto	Das Muster bzw. Bild wird entsprechend der eingestellten Bildschirmskalierung skaliert. Eine zur Vorgängerversion kompatible Skalierung findet statt. **Defaultwert**

Der ***scalestyle*** kann dabei entweder direkt bei der **Definition des Tiles** oder aber als **dynamisches Attribut** angegeben werden.

Tile-Definition:

```
tileSpec ::= <tileBitmap> | "<Dateipfad>" | "<Grafikressource>" { scale |
noscale | numscale | propscale | dpiscale} ;
```

Dynamisches Attribut:

	Bezeichner	Datentyp
Regelsprache	**.scalestyle**	enum
C	AT_scalestyle	DT_enum
COBOL	AT-scalestyle	DT-enum
Klassifizierung	Objektspezifisches Attribut	

	Bezeichner		Datentyp
Objekte	tile		
Zugriff	get, set	Standardwert	scalestyle_auto
changed-Ereignis	nein	Vererbung	–

Beispiel:

```
dialog D
tile Ti_Rect_Pattern 5,5,
  "#####",
  "# #",
  "# #",
  "# #",
  "#####" noscale;          // entspricht scalestyle_none
tile Ti_BG „bg.gif“ scale;  // entspricht scalestyle_any
tile Ti_Logo „logo.gif“;    // Default: scalestyle_auto
...
on dialog start {
  if setup.scale > 100 then
  Ti_Logo.scalestyle := scalestyle_prop;   // dynamisch
endif
}
```

Siehe auch Kapitel „tile (Muster)“ im „Ressourcenreferenz“-Handbuch.

9.4 Erweiterung der Font-Ressource

Die Font-Ressource hat eine neue Eigenschaft .propscale. Dies steuert, ob das horizontale und vertikale Raster proportional auf den maximalen Wert gesetzt werden soll, welcher durch die Werteberechnung von xraster und yraster des Font-Rasters ermittelt wird.

Die ***propscale***-Eigenschaft kann dabei entweder direkt bei der **Definition des Fonts** oder aber als **dynamisches Attribut** angegeben werden.

Der Divisor dient der Verfeinerung des Rasters.

Font-Definition:

```
fontspec ::=
...
...
{ x:= * <xscale> % + <xoffset> | / <xdivider>} { y:= * <yscale> %
+ <yoffset> | / <ydivider>}
{ propscale }
{ r:= "<RefString >" } ;
```

Dynamisc hes Attribut:

	Bezeichner		Datentyp
Regelsprache	**.propscale**		boolean
C	AT_propscale		DT_boolean
COBOL	AT-propscale		DT-boolean
Klassifizierung	Objektspezifisches Attribut		
Objekte	font		
Zugriff	get, set	Standardwert	false
changed-Ereignis	nein	Vererbung	–

Siehe auch Kapitel „font (Zeichensatz)“ im „Ressourcenreferenz“-Handbuch.

9.5 Erweiterung am Setup-Objekt

	Bezeichner		Datentyp
Regelsprache	**.tiledpi**		integer
C	AT_tiledpi		DT_integer
COBOL	AT-tiledpi		DT-integer
Klassifizierung	Objektspezifisches Attribut		
Objekte	setup		
Zugriff	get, set	Standardwert	–
changed-Ereignis	nein	Vererbung	–

	Bezeichner		Datentyp
Regelsprache	**.tiledpi**		integer
C	AT_tiledpi		DT_integer
COBOL	AT-tiledpi		DT-integer
Klassifizierung	Objektspezifisches Attribut		
Objekte	setup		

	Bezeichner		Datentyp
Zugriff	get, set	Standardwert	96
changed-Ereignis	nein	Vererbung	–

Dieses Attribut bestimmt für welche Auflösung die Bilder/Muster entworfen wurden. Die Größe eines Bildes/Musters wird ausgehend von diesem Wert auf den aktuell geltenden DPI-Wert umgerechnet.

Siehe auch Kapitel „setup“ im „Objektreferenz“-Handbuch.

9.6 Erweiterungen am Image-Objekt

Für eine bessere Ausrichtung und erweiterte Layoutmöglichkeiten können nun über die folgenden Attribute am Image-Objekt die Ränder bzw. Abstände gesteuert werden:

	Bezeichner		Datentyp
Regelsprache	**.xmargin**		integer
C	AT_xmargin		DT_integer
COBOL	AT-ymargin		DT-integer
Klassifizierung	Objektspezifisches Attribut		
Objekte	image		
Zugriff	get, set	Standardwert	0
changed-Ereignis	nein	Vererbung	ja

Dieses Attribut bestimmt den horizontalen Abstand zwischen Rand und Darstellungsbereich.

	Bezeichner		Datentyp
Regelsprache	**.ymargin**		integer
C	AT_ymargin		DT_integer
COBOL	AT-ymargin		DT-integer
Klassifizierung	Objektspezifisches Attribut		
Objekte	image		
Zugriff	get, set	Standardwert	0
changed-Ereignis	nein	Vererbung	ja

Dieses Attribut bestimmt den vertikalen Abstand zwischen Rand und Darstellungsbereich.

Siehe auch Kapitel „image (Bild)“ im „Objektreferenz“-Handbuch.

9.7 Unterstützung von HiDPI Bild-Varianten

Für eine optimale Gestaltung der Oberfläche bzgl. Bilder und Applikations-Icons durch Ausnutzung der höheren Auflösung sind für die Skalierungsstufe angepasste Bilder sinnvoll.
Bisher gab es hier nur für die Fenstersysteme WINDOWS und QT eingeschränkte Möglichkeiten. Windows bietet hier bislang zur Unterstützung den ICON/BITMAP-Mechanismus und Qt den Icon-Ressource-Mechanismus. Beide Mechanismen ermöglichen das Vorhalten mehrerer Auflösungsstufen, sind jedoch begrenzt was Bildtyp oder Verwendung angeht. Einen solchen Mechanismus gibt es für MOTIF nicht.
Daher wurde mit dem IDM A.06.03.A eine Möglichkeit zum Laden mehrerer Auflösungsstufen geschaffen, die gleichermaßen für WINDOWS, QT und MOTIF angewandt werden kann und somit auch systemunabhängig ist.
Die Tile-Verwaltung vom IDM war bisher nur in der Lage eine Bilddatei zu laden (bspw.: tile Ti „smiley.gif“) und konnte nicht mehrere Auflösungsstufen der Datei vorhalten (bis auf o.g. Ausnahmen).
Liegt nun eine Skalierung der Anwendung vor, versucht der neue Lademechanismus für Tile-Dateien nun zuerst die Bilddaten für die entsprechende Skalierung zu laden. Dazu wird in Anlehnung an den QT-Mechanismus die Endung ***@nx*** an den Dateinamen und vor dem Suffix angehängt. Dabei steht ***n*** für die Skalierungsstufe zwischen *2* und *9*. Die Zahl *2* entspricht dabei einer Skalierung von *150%-249%*, *3* von *250%-349%*, usw.
Sinnvollerweise sollten die Größenänderungen der Varianten Bilder der Skalierungsstufe entsprechen, wobei eine Überprüfung durch den IDM nicht erfolgt. Also z.B. Orginalbild „smiley.gif“ (32x32 Pixel), smiley@2x.gif (64x64 Pixel), smiley@3x.gif (96x96 Pixel), smiley@4x.gif (128x128 Pixel).

Beispiel:

```
tile Ti „smiley.gif“;  // bei einer Skalierung von 200% wird – sofern
                       // vorhanden – vom IDM automatisch das Bild
                       // smiley@2x.gif geladen (Windows, Motif, Qt)

                       // bei einer Skalierung von 300% wird – sofern
                       // vorhanden – vom IDM automatisch das Bild
                       // smiley@3x.gif geladen (Windows, Motif, Qt)

tile Ti2 „smiley.ico“  // nur Windows! ICON-Mechanismus von Window nimmt
                       // das für die Auflösungsstufe am besten Bild aus
                       // der .ico-Datei

tile Ti3 „:/smiley“    // nur Qt! Der Resource-Mechanismus von Qt nimmt das
                       // für die Auflösungsstufe am besten passende Bild
                       // mit dem hier angegebenen Alias aus der Ressource-
                       // Datei
```

9.8 Installationshinweise

Um eine Anwendung für unterschiedliche Auflösungen vorzubereiten, kann es notwendig sein, die Grafiken für die verschiedenen Skalierungsstufen vorzuhalten. Wer das in Anspruch nehmen möchte und nicht den ICON-Mechanismus von WINDOWS oder den Resource-Mechanismus von QT nutzt, der sollte darauf achten die Bilderdateien entsprechend der @-Namensgebung (siehe Kapitel „Unterstützung von HiDPI Bild-Varianten“) mit seiner Anwendung mit auszuliefern bzw. diese ggf. in die Installationspakete mit aufzunehmen.

9.9 Geometrie und Koordinaten

Koordinaten und Größenangaben werden vom IDM passend zum Skalierungsfaktor des Systems transformiert um eine zu den Systemeinstellungen konsistente Darstellung zu gewährleisten.

Die Geometrieeinheiten werden dabei wie bisher in **IDM-Pixelwerten** oder *Raster* gesetzt.

Die **IDM-Pixelwerte** werden intern entsprechend dem Skalierungsfaktor in **reale Pixelwerte** hochgerechnet und umgesetzt. Dadurch wird die Anwendung als Ganzes skaliert und Verhältnisse und Proportionen der Objekte zu- und untereinander beibehalten.

Diese Konvertierung geschieht in beide Richtungen. Beim Abfragen von Größe und Position von Objekten wie auch bei der Event-Verarbeitung werden die **realen Pixelwerte** vom IDM ebenso wieder bereinigt in **IDM-Pixelwerten** angegeben. Der Vergrößerungsfaktor ist dann wieder herausgerechnet. Dies betrifft alle pixelorientierten Geometrieattribute.

Rasterwerte sind entweder an den verwendeten *(HighDPI-fähigen) Zeichensatz* geknüpft oder werden ebenfalls in **IDM-Pixelwerten** gesetzt und dann hoch skaliert. Die *Rastereinheiten* können basierend auf der zugrundeliegenden Schriftart ebenfalls zu **IDM-Pixelwerten** ermittelt werden.

9.10 Unterstützung hoher Auflösungen unter Motif

Das Motif-Toolkit selbst enthält zwar keine besondere Technologie zur Unterstützung hoher Auflösungen, wenn aber das verwendete X-Display die Xrender-Extension implementiert hat und eine Unterstützung von XFT-Fonts mit den entsprechenden Schriftarten vorliegt, können IDM-Anwendungen für Motif auf HiDPI-Bildschirmen, unter Berücksichtigung des Skalierungsfaktors des Desktops, betrieben werden.

Wie kann geprüft werden, ob auf dem verwendeten X-Display/Desktop eine Unterstützung vorliegt? Dazu kann die Serverinformation des X-Displays über die Option **-IDMserver:** erfragt werden.

```
$ idm -IDMserver
ServerVendor The X.Org Foundation
VendorRelease 12013000
Motif at compiletime @(#)Motif Version 2.3.8
Motif at runtime 2003
XRenderVersion 11 (active)
XFT Support yes (active)
```

```
Scaling 200% (active)
Screen dpi 96 (tile dpi: 96, scaled dpi: 192)
```

Die Version der Xrender-Extension und deren Nutzbarkeit (active) wird darin ebenso aufgelistet wie das Vorhandensein von XFT und seiner Nutzbarkeit. Die „Scaling“-Zeile zeigt den Skalierungsfaktor an, der in den Desktop-/Anzeigeeinstellungen vom Anwender gesetzt wurde.

Die DPI-Information welche X liefert, ist zumeist der Standardwert von 96 und spiegelt damit nicht unbedingt den real am Monitor vorhandene DPI-Wert wider. Die optional ausgegebenen „tile dpi“ und „scaled dpi“ dienen zur Kontrolle der für die Bilder und Bedienelemente zugrunde gelegten DPI-Werte.

Ein dynamisches Wechseln der Skalierung ist für Motif-Anwendungen nicht vorgesehen. Ebenso kann eine Umsetzung der Skalierung im Desktop nicht erkannt werden.

Unterstützung von XFT / Font-Handhabung / Motif-Font

Die Basis für eine GUI-Anwendung für unterschiedliche Desktop-Skalierungen stellen die verwendeten Schriften dar. Im Gegensatz zu den bisher unter Motif/X verwendbaren Schriftarten, die meist über einen XLFD (X Logical Font Descriptor) angeben wurden, werden XFT Fonts entsprechend der gesetzten Skalierung des Desktop mitskaliert.

Die bisher unter dem IDM FÜR MOTIF verwendbaren Schriftarten hatten sich nicht automatisch an unterschiedliche Desktop-Skalierungen angepasst.

Die XFT-Unterstützung ist im IDM nur auf Linux-Plattformen verfügbar. Da die meisten Desktops auf Linux-Plattformen den Standard-Font für Motif-Widgets nur unzureichend setzen, benutzt der IDM ***Liberation Sans*** *als Standard-Schrift und* ***Liberation Mono*** *als Festbreitenschriftart, wenn kein Font im Dialog gesetzt wurde. Dadurch skalieren sich Texte in IDM-Dialogen auch ohne gesetzte Schriftart.*
Durch explizites Ausschalten des Scaling (über die Startoption -IDMscale=0), kann man die alten Default-Schriftarten erhalten.

Zum setzen eines XFT Fonts muss wie bei bisherigen Fontdefinitionen im IDM nur der Fontname und ggf. weitere Modifier gesetzt werden. Ist der Font als XFT Font auf dem System vorhanden, dann wird dieser geladen. Die Liste der auf dem System verfügbaren XFT Fonts kann im IDM EDITOR eingesehen werden.

Unter CDE-konformen Plattformen wie AIX und HP-UX verwendet der IDM als Standard-Schrift „-dt-interface_system-medium-r-normal-*“ und „-dt-interface_user-medium-r-normal-*“ als Festbreitenschrift, sofern keine Schrift gesetzt wurde. Je nach Skalierungsfaktor werden dabei unterschiedliche Größenausprägungen des Fonts herangezogen. Zwischen *1%...75%* wird **xs** (extra small) verwendet, zwischen *75%-150%* wird **m** (medium) verwendet, zwischen *150%-250%* wird **l** (large) verwendet und ab *250%* wird **xxl** (extra extra large) verwendet. Dadurch kann für solche Anwendungen eine Skalierbarkeit erreicht werden auch wenn keine XFT-Fonts verfügbar sind.

9.11 Unterstützung hoher Auflösungen beim IDM für Windows 11

Es gibt jetzt einen ***IDM für Windows 11*** und einen für ***Windows 10***. Der IDM FÜR WINDOWS 11 unterstützt DPI-Awareness und High DPI.

Der IDM FÜR WINDOWS 11 unterstützt hohe Auflösungen des Modells „PerMonitor V2“. Da die Unterstützung hoher Auflösungen eine Eigenschaft der Anwendung ist, muss diese im Anwendungsmanifest entsprechend gekennzeichnet werden. Entsprechende Setzung können in den IDM-Beispielen entnommen werden. Der IDM erkennt dies dann selbständig und rechnet die Koordinaten entsprechend dem eingestellten Vergrößerungsfaktor um. Mit anderen Worten, die Pixelkoordinaten des IDM sind virtuelle Pixel, die entsprechend dem eingestellten Vergrößerungsfaktor umgerechnet werden.

Die Manifest-Dateien der IDM-Beispiele kennzeichnen die gebauten Anwendungen als „High DPI-Aware“. Um eine Anwendung zu bauen, die nicht „DPI-Aware“ ist, muss der gesamte Block `<asmv3:application>` aus der Manifest-Datei entfernt werden.

Hinweise IDM für Windows 11

Alle Funktionen, die Microsoft Windows Nachrichten verarbeiten: Wenn die Anwendung hohe Auflösungen unterstützt, dann sind alle Koordinaten, die über Microsoft Windows Nachrichten erhalten werden, die realen hochauflösenden Koordinaten. Der IDM benutzt dagegen um den Vergrößerungsfaktor bereinigte Koordinaten. Es muss also berücksichtigt werden, dass im Falle der Unterstützung hoher Auflösungen, die IDM-Koordinaten im Allgemeinen nicht mehr identisch zu den Microsoft Windows Koordinaten sind. Betroffen hiervon sind zum Beispiel Inputhandler-, Canvas-, Monitor- und Subclassing-Funktionen, sowie auch die USW-Implementierungen.

Hinweise IDM für Windows 10 und 11:

Die Unterstützung hoher Auflösungen erfordert eine andere Behandlung von Windowsnachrichten. Dadurch ist es leider nicht mehr möglich Fenster während des Verschiebens oder Vergrößerns an Rahmenbedingungen wie zum Beispiel Rasterkoordinaten anzupassen. Erst nach dem Loslassen der Maustaste wird das Fenster auf die nächste Rasterkoordinate angepasst.

Intern definierte Cursor werden jetzt auf die vom System geforderte Größe vergrößert bzw. verkleinert, um diese automatisch auf den eingestellten Vergrößerungsfaktor anzupassen. Hierdurch werden die Cursor richtig dargestellt, wenn die Anwendung hohe Auflösungen unterstützt. Bisher wurden die Cursor entweder mit durchsichtigen Bereichen aufgefüllt oder einfach abgeschnitten, wenn die Größe nicht passte.

Bilder (tile Ressource) werden automatisch entsprechend des eingestellten Vergrößerungsfaktor vergrößert. Es wird hierbei davon ausgegangen, dass die Bilder für einen DPI-Wert von *96* entworfen wurden. Sind die Bilder der Anwendung für eine höhere Auflösung gestaltet, dann kann dies am setup Objekt mit dem Attribute .tiledpi eingestellt werden. Eine automatische Anpassung gilt nur für die tile-Ressource. Bilder die direkt über den Dateinamen am .picture Attribute des image-Objektes angegeben werden, werden nicht automatisch angepasst.

Als Standardschriftart wird nun die in den Systemparameters definierte Dialog- bzw. Message-Schriftart verwendet, da diese sich an die Auflösung anpasst. Hierdurch kann es zu Änderungen der Darstellung von Objekten kommen, die keine Schriftart gesetzt haben. Die erwähnte Schriftart kann über Windows Systemeinstellungen konfiguriert werden.

Die alten Systemschriftarten sind nicht DPI-Aware. Nur die neuen ***UI*_FONT*** Schriftarten passen sich der eingestellten Auflösung an.
Die alten Windows-Schriftarten „ANSI_FIXED_FONT“, „ANSI_VAR_FONT“, „DEVICE_DEFAULT_FONT“ und „SYSTEM_FONT“ lassen sich nicht an verschiedene Auflösungen anpassen, daher wird von der Verwendung abgeraten. Von der Verwendung von Schriftarten ohne Größenangabe wird abgeraten, da in einem solchen Fall vom Windows Font Mapper eine schriftartspezifische Standardgröße gewählt wird, die sich nicht an unterschiedliche Auflösungen anpasst.

9.12 Unterstützung hoher Auflösungen unter Qt

Für die Unterstützung hoher Auflösungen aktiviert der IDM das Qt-seitige HighDpi-Scaling sowie das mögliche Rendern von Bildern über ihre eigentlich angeforderte Größe hinaus. D.h. die Skalierung wird auf Basis der Pixeldichte des Monitors vorgenommen.

Unter Linux Desktop-Umgebungen ist die Unterstützung für HighDPI leider sehr unterschiedlich ausgeprägt und fortgeschritten. Da die Werte für logische und tatsächliche Pixeldichte sowie Skalierungsfaktoren vom Qt-Framework bezogen werden, kann es sein, dass weniger populäre Desktop-Umgebungen hier unzureichende Werte liefern.

Hinweise zu HighDPI Einstellungen

Qt bietet noch einige Umgebungsvariablen an, mit denen die Darstellung bei hohen Auflösungen gesteuert werden kann. Bei Einsatz solcher Umgebungsvariablen sollte jedoch bewusst sein, dass somit das Verhalten des IDM überschrieben wird und es dadurch zu ungewollten Darstellungseffekten kommen kann.

Anmerkungen zur Skalierung von Tiles

Objekte werden standardmäßig mit den vom Desktop-Style vorgegebenen Werten durch das WSI/Toolkit dargestellt. Es kann daher in bestimmten Situationen vorkommen, dass die gewünschten Einstellungen bezüglich Größe und Skalierung von Tiles nicht umgesetzt werden. Dies betrifft die Verwendung von Tiles bei Objekten mit Tabs wie dem Notebook bzw. den Notepages sowie dem Menu mit Menuitems. Der IDM setzt zwar die gewünschten Darstellungsoptionen, jedoch unterliegt die finale Anzeige der Darstellungsrichtlinie der jeweiligen DrawingEngine des gesetzten UI-Styles. Das bedeutet, dass die Anzeige von den gesetzten Optionen Style-abhängig abweichen kann oder Optionen vollständig ignoriert werden.

9.13 Erweiterungen/ Änderungen im IDMED

Eine beliebige Auswahl der Schriftart für die Bedienoberflächen oder Regelcode ist nicht mehr möglich. Dafür kann die Schriftgröße zwischen Klein (Small) – Normal – Groß (Large) – Extra Groß (Extra Large) ausgewählt werden. Diese hat dann natürlich Einfluss auf die Fenstergröße.

Im Schriften- bzw. Font-Einstellungsbereich werden unter Motif auch XFT-Fonts aufgelistet. Außerdem kann die Liste der Auswählbaren Schriften zum Zwecke der Übersichtlichkeit auf UI-Fonts, X-Fonts und XFT-Fonts reduziert werden.

Der IDMED, TracefileAnalyzer sowie der Debugger setzen nun UI-Ressourcen ein.

9.14 Unterstützung hoher Auflösungen bei der USW-Programmierung

Die Funktion DM_GetToolkitDataEx wurde zur Unterstützung von HighDPI verbessert. Die Toolkit datena-S strukcture DM_ToolkitDataArgs wurde zu diesem Zweck erweitert und liefert nun detaillierte Informationen zu DPI, Skalierungsfaktor und Bild. Diese können über die Attribute *AT_DPI* und *AT_XWidget* erfragt werden.

Hinweis Motif

Der USW-Programmierer muss die Koordinaten zwischen dem IDM (dies betrifft Attribute sowie Events) und den Motif-Werten/Strukturen/Funktionen (*X…/Xt…/Xm…*) umrechnen. Die für eine Umrechnung benötigten Daten können über Funktion DM_GetToolkitDataEx mittels AT_DPI und die Toolkit datena-S strukcture DM_ToolkitDataArgs-Struktur (*tile.scale.factor*) erfragt werden. Je nach Bildart muss u.U. auch eine Skalierung vorgenommen werden.

Hinweis für Windows 11

Der IDM FÜR WINDOWS 11 unterstützt hohe Auflösungen. Wenn eine Anwendung für hohe Auflösungen freigegeben wird, dann müssen auch die von ihr verwendeten USW-Objekte hohe Auflösungen unterstützen. Das heißt dann, dass die USW-Objekte von Windows reale Koordinaten erhalten (um den Vergrößerungsfaktor vergrößert), während der IDM mit virtuellen Koordinaten (nicht vergrößert) arbeitet. Praktisch wird sich nicht viel ändern, da der IDM die Koordinatenberechnungen für das USW Objekt ausführt. Aber es ist zu berücksichtigen, dass der **control** Aufruf für „UC_C_PrefSize“ reale Koordinaten erwartet, da diese im Normalfall aus Toolkitwerten berechnet werden. Sind hier fixe Zahlen wie im *ucarrow* Beispiel eingesetzt, dann müssen diese um den aktuellen DPI-Wert korrigiert werden (siehe ucarrow Beispiel).

Wenn die Anwendung als DPI-Aware gestartet wurde, werden nun von der Task „UC_I_clientarea“ der Funktion **UC_Inquire** auch die hochauflösenden Werte erwartet.

Ergänzungen der C-Schnittstelle für DM_GetToolkitData() und DM_GetToolkitDataEx()

Bei den C-Schnittstellen Funktionen "DM_GetToolkitData()", "DM_GetToolkitDataEx()" und Toolkitdaten-Struktur "DM_ToolkitDataArgs" gibt es zur Unterstützung von hohen Auflösungen die

folgenden Erweiterungen:

Attribute AT_DPI

Gibt den DPI-Wert des Systems oder des angegebenen Objektes zurück. Die Funktion "DM_GetToolkitDataEx()" muss mit einem Zeiger auf eine Toolkitdaten-Struktur "DM_ToolkitDataArgs" aufgerufen werden. In dieser Struktur werden weitere DPI-Informationen zurückgeliefert. Sie ist hierzu um das Datenfeld "dpi" und die Unterdatenstruktur "scale" erweitert.

Attribute "AT_Tile" bzw. "AT_XTile"

Dieses Attribut gab es schon für die "tile" Ressource. Jetzt kann die Funktion "DM_GetToolkitDataEx()" mit einem Zeiger auf eine Toolkitdaten-Struktur "DM_ToolkitDataArgs" aufgerufen werden, die um die Unterdatenstruktur "tile" erweitert wurde. Diese enthält unter anderem den DPI-Wert, für den die "tile" Ressource entworfen wurde.

Attribute "AT_IsNull"

Liefert einen Wert ungleich "0" zurück, wenn die angegebene "color" oder "font" Ressource auf "UI_NULL_FONT" bzw. "UI_NULL_COLOR" definiert war.

Siehe auch Kapitel „Toolkit datena-S strukcture DM_ToolkitDataArgs“ im „C-Schnittstelle - Grundlagen“-Handbuch.

Siehe auch Kapitel „DM_GetToolkitDataEx“ im „C-Schnittstelle - Funktionen“-Handbuch.

Siehe auch Kapitel „DM_GetToolkitData“ im „C-Schnittstelle - Funktionen“-Handbuch.

Index

D

K

L

M

N

T

U

V

W

X

Z