ISA Dialog Manager

Programming Techniques

A.06.03.b

This manual provides basic techniques for the development of user interfaces with the ISA Dialog Manager. Modularization, use of Models, object-oriented programming and the Datamodel are among the topics covered.

ISA Informationssysteme GmbH

Meisenweg 33

70771 Leinfelden-Echterdingen

Germany

Notation Conventions

DM will be used as a synonym for Dialog Manager.

The notion of UNIX in general comprises all supported UNIX derivates, otherwise it will be explicitly stated.

< >	to be substituted by the corresponding value
color	keyword
.bgc	attribute
{ }	optional (0 or once)
[ ]	optional (0 or n-times)
<A> | <B>	either <A> or <B>

Description Mode

All keywords are bold and underlined, e.g.

variable **integer** **function**

Indexing of Attributes

Syntax for indexed attributes:

[I]

[I,J] meaning [row, column]

Identifiers

Identifiers have to begin with an uppercase letter or an underline ('_'). The following characters may be uppercase or lowercase letters, digits, or underlines.

Hyphens ('-') are ***not*** permitted as characters for specifying identifiers.

The maximal length of an identifier is *31* characters.

Description of the permitted identifiers in the Backus-Naur form (BNF)

<identifier>	::=	<first character>{<character>}
<first character>	::=	_ | <uppercase>
<character>	::=	_ | <lowercase> | <uppercase> | <digit>

```
<digit>          ::=    1 | 2 | 3 | … 9 | 0

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

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

Table of Contents

1 Conventions for Names

As is also usual in other programming languages, the use of name conventions on establishing a dialog can substantially increase the legibility of a dialog. By giving unambiguous names, the developers are able to read foreign dialog parts easily without having to search for all used items in the dialog script. This is why such a naming should be defined before starting a larger project.

The following naming can only be considered as a basic structure or a suggestion. It must be adapted to the individual projects according to the relevant conditions.

1.1 Purpose of the Naming

With help of the naming it is achieved that the reader receives most information possible with the smallest amount of characters possible. The name should convey the type of the object, in which module the corresponding object is defined, and what function it has. In addition, the number of the names should be kept as small as possible and only those objects should be named which are referred to in the rules and which cannot inherit names from their model. Usually it is not reasonable to overwrite the name which was inherited by a model. This only increases the number of the names and is by no means a contribution to the legibility of the dialog.

The class and the type (model or instance) always turn out to be very important in the use of objects. The first character should thus contain this kind of information. Since the instance usually is the object type most used, it should not be labeled by a special letter, only the models. After this, the object type should be attached to the label which indicates whether it is a module or an instance. If modules are used, the module where the corresponding object is defined should then be indicated - separated by an underline. Then the actual name of the object should follow - again separated by an underline for better legibility. This name should convey the task of the corresponding object.

The following scheme is the result for the naming:

Definition

```
{M}<abbreviation object class>{_<module name>}_name of object
```

The module name should only be used if you really work with modules. Otherwise this part of the name is ignored. When giving names, please note that the sign limit for the name is 31.

1.2 Abbreviations for Individual Object Classes

In this chapter the abbreviations for the individual object classes are defined.

object type	abbreviation for name convention
image	Im

object type	abbreviation for name convention
canvas	Cn
checkbox, tristatebutton	Cb
edittext	Et
window	Wn
groupbox	Gb
listbox	Lb
menubox	Mb
menuitem	Mi
menuseparator	Ms
messagebox	Mx
notebook	Nb
notepage	Np
poptext	Pt
pushbutton	Pb
radiobutton	Rb
rectangle	Re
scrollbar	Sc
statusbar	Sb
statictext	St
tablefield	Tf
application	Ap
record	Rc
timer	Ti

object type	abbreviation for name convention
accelerator	Ac
cursor	Cu
color	Cl
format	Fm
tile	Ti
text	Tx
font	Fn
function	Fc
global variable	V
rule	Rl
user-defined attribute	A

The user-defined attributes and the global variables can be followed by a data type name, so that the corresponding data type can be read directly from the name.

1.3 Abbreviations for Data Types

The data types can be abbreviated as follows:

data type	abbreviation for name convention
anyvalue	Av
attribute	At
boolean	Bo
class	Cl
datatype	Dt
enum	En
event	Ev

data type	abbreviation for name convention
index	Ix
integer	In
method	Mt
object	Ob
pointer	Pt
string	St

1.4 Examples for Naming

In this chapter some examples are listed to illustrate how object names can be created from the conventions above.

description of object	name of object
model of tablefield	MTf_Overview
model of OK pushbuttons	MPb_OK
global object variable	VOb_ActualWindowr
object attribute	AOb_InitWindow
tristate pushbutton	Cb_Switch

2 Use of Models

This chapter describes how the inheritance of attributes functions internally. You get suggestions for the structure of a modular construction from which you can use objects whenever you want.

2.1 Object Types

Attributes which are visible to the user on the screen can be defined on different object definition levels.

The different object types are:

- DM internal default values for the individual attributes
- default
- model (different layers, any number)
- instance

The different object types are explained in more detail in the following chapter.

2.1.1 Default Objects

Each object class used in the dialog should have a defined default object. Using these default objects, the individual object attributes can once be globally defined. That means that all the following declared objects of the same object class contain implicitly the attributes which are defined in the object model, if these are not locally defined otherwise. With help of this you only have to declare additional attributes or attributes which differ from the default.

The declaration of an object model starts with the key word **default**, then follows the object class and finally the object definition in braces.

Definition

```
{ export | reexport } default <object class> { <Identifier> }
{
  <attributes>
}
```

Within the default definitions all attributes belonging to the object class can be used. Of course you can also add user-defined attributes to the default object.

Example

```
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;
}
```

If a pushbutton is declared later on, only the position of the top left corner and the labeling must be indicated.

In this case:

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

Unlike all other objects, default objects cannot have children. If the default object is not named, as usual, it is addressed to via the name of its class in capital letters; e.g. PUSHBUTTON represents the default object of the pushbuttons, WINDOW is the default object of all windows.

The following table shows all the names of default objects.

Object Class	Name of Default Object
canvas	CANVAS
checkbox, tristate button	CHECKBOX
edittext	EDITTEXT
groupbox	GROUPBOX
image	IMAGE
menubox	MENUBOX

Object Class	Name of Default Object
menuitem	MENUITEM
menuseparator	MENUSEP
messagebox	MESSAGEBOX
notebook	NOTEBOOK
notepage	NOTEPAGE
poptext	POPTEXT
pushbutton	PUSHBUTTON
radiobutton	RADIOBUTTON
rectangle	RECTANGLE
scrollbar	SCROLLBAR
statictext	STATICTEXT
statusbar	STATUSBAR
tablefield	TABLEFIELD
timer	TIMER
window	WINDOW

2.1.2 Models

Models are other auxiliary means for the object generation. With the models, it is possible to define a named model object which is used for the definition of a real object.

This is very helpful when a bigger amount of similar objects is to be generated (e.g. several OK buttons or several edittexts, into which the user can set one item number each). The object referring to a model inherits all the attributes which it does **not redefine locally**.

Definition

```
{ export | reexport } model <object class> <Identifier>
{
  <attributes>
}
```

Within the model definition all attributes belonging to the object class can be used. Such models can then serve as children of other objects or models. In addition, it is possible to give children to such a model which are automatically inherited to all instances derived from it. Complex hierarchical models which are to be used again and again in the current dialog can thus be constructed.

The definition of such hierarchical models is as follows:

```
{ export | reexport } model <object class> <Identifier>
{
  { export | reexport } { child } <object class> { <Identifier> }

{
    <attributes>
  }
}
```

In order to structure the use of models more efficiently, you should make the effort to create complex models if certain structures do always reappear in the dialog. Thus, the maintenance and the effort for changes can be substantially reduced.

If a model is to be used for the definition of an object, the object must be defined as follows:

```
<model identifier> { <Identifier> }
{
  <attributes>
}
```

Example: Edittext for Item Number

In different windows of an application edittexts are to be defined into which an item number can be entered. Usually the scheme for an item number is the same for all applications. This is why you should create a model in order to use it in the corresponding windows. If the scheme for the item number changes later on, only one position in the dialog, i.e. the model, must be altered.

```
model edittext MEtitemnumber
{
  .posraster true;
  .sizeraster true;
  .format "AA.NN.NN/NN";
  .width 12;
}
```

This scheme can then be used in different windows and their windows can be changed.

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

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

In the first window WnFenster1 the item number is used without giving it a new name. Thus, the instance derived from the model can be addressed to via window WnFenster1.MEtArtikelNummer. In the second window WnFenster2 the model is used even on two positions. Here, the objects should be renamed so that these can unambiguously be addressed.

Example: Window with Cancel and OK Pushbutton

In a dialog several windows, into which data can be entered, are to be created. Therefore those windows should be equipped with two pushbuttons. The first pushbutton "OK" takes over the alterations made by the user into the program, the "cancel" pushbutton rejects all actions of the user. In order to get a comfortable structure of the corresponding systems, a pushbutton model "MPbButton" is created first. Only the pushbutton fixing at the lower border of the parent is defined. The actual pushbuttons "MPbOK" and "MPbCancel" are derived also as model. Then a window model is defined which contains derivations of the pushbuttons "MPbOK" and "MPbCancel". Finally an instance is created for this window. The instance is structured as follows:

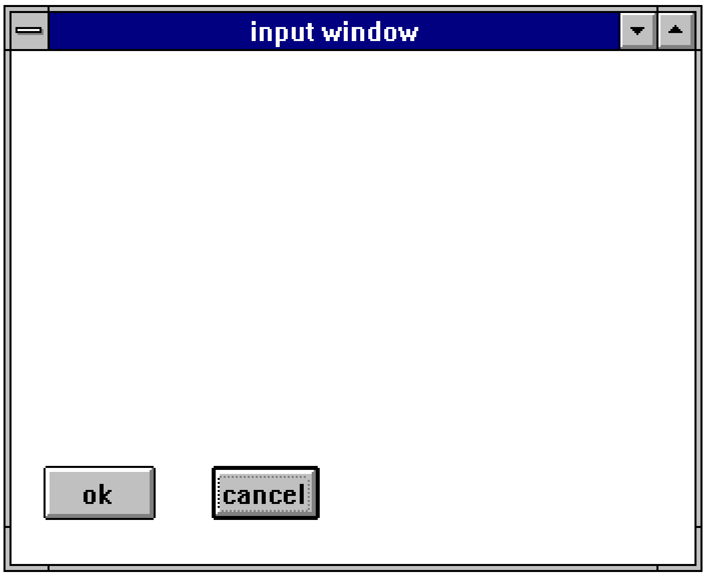


Figure 1: Window with two pushbuttons

The corresponding dialog script is structured as follows:

```
model pushbutton MPbButton
{
  .yauto -1;
}
model MPbButton MPbAbbrechen
{
  .xleft 15;
  .text "cancel";
}
model pushbutton MPbOK
{
  .yauto -1;
  .text "ok";
}
model window MWnEingabeFenster
{
  .title "input window";
  child MPbAbbrechen
  {
  }
  child MPbOK
  {
  }
}
MWnEingabeFenster WnAddresse
{
}
```

In the window "WnAddress" you can redefine as many children as you want of course.

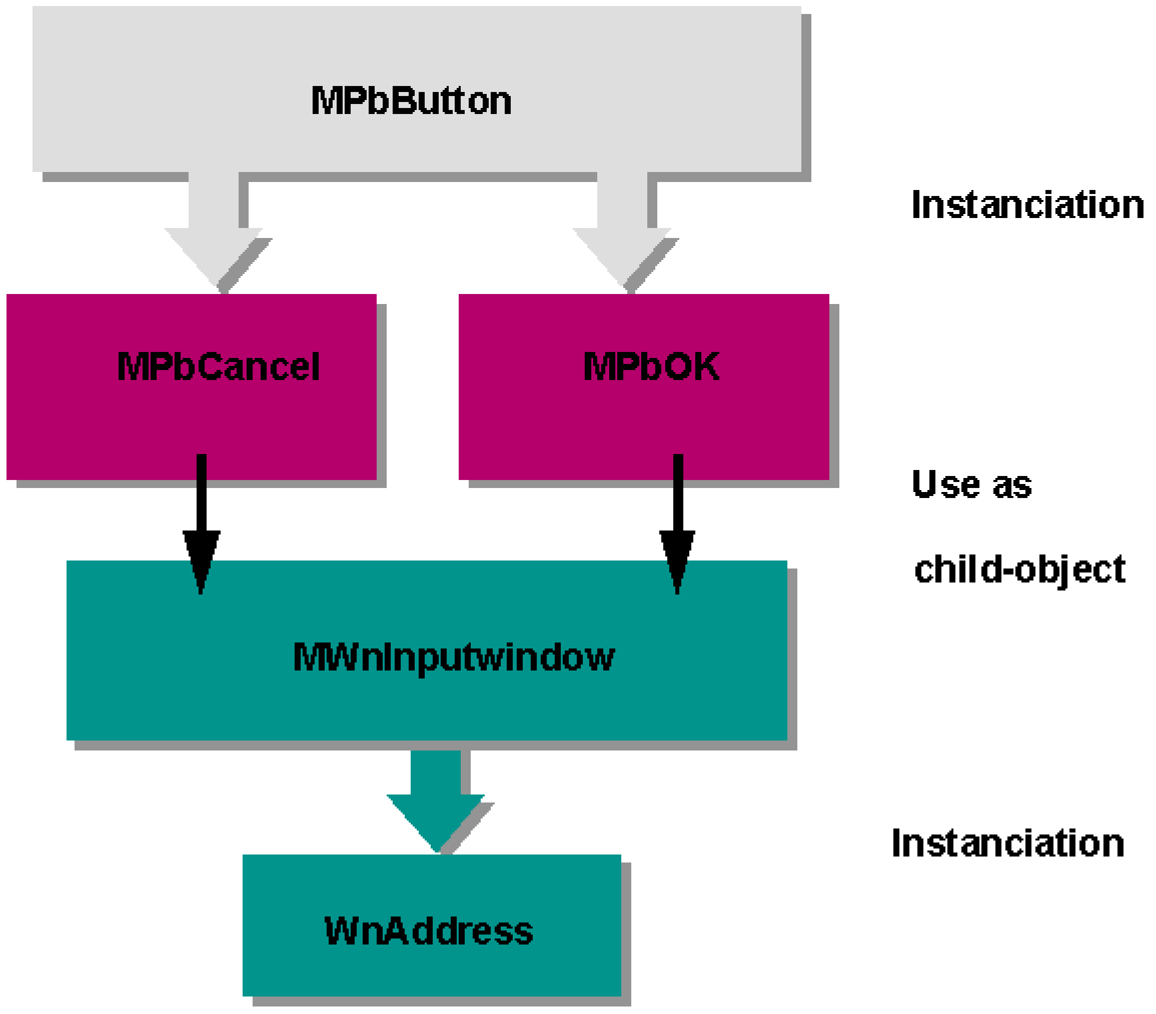


***Figure 2:** Representation of the model hierarchy*

2.1.3 Instance

Instances are those objects which can be made visible on the screen. They can be derived from a model and can have any number of children apart from the children inherited from the model. At an instance all attribute values can be overwritten and even the attributes of inherited children can be changed. Children which are defined in the model must be taken over, they cannot be deleted in the instance.

2.2 Inheritance of Attributes

Attribute values at objects are always inherited from the underlying model. These attribute values can then be overwritten by new values. Doing so, the importance of attributes increases gradually from default values to model values to instance values. For example, when an instance is to have a certain color, the color is defined at the corresponding instance. Thus the instance gets the corresponding color, regardless of what has been defined in the underlying models.

inheritance

Above the defaults which are defined in the dialog, there are Dialog Manager default values representing the values for individual attributes. The Dialog Manager can internally work with these attributes. You should never rely on these values when programming the dialog, since the values can always be re-adapted. In addition, there are also default values in the window system for certain attributes. Those default values become important when no value for the attribute was set neither at the instance nor at the model or the default. This is useful especially with layout attributes such as the background color or the font, as the user can change the corresponding values by system settings. The following diagram illustrates the hierarchy of these attributes.

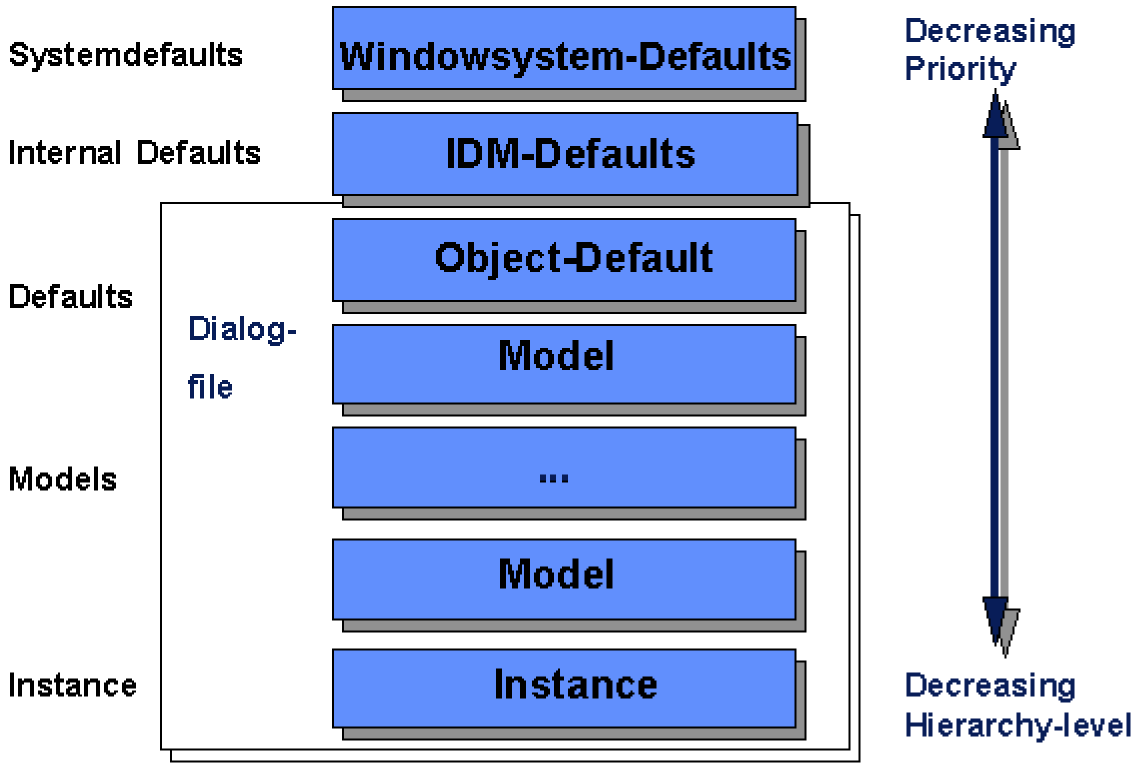


Figure 3: Inheritance priority

If an instance is now made visible on the screen, an object is created which contains the attributes of the instance and the underlying model. The hierarchy is always searched through in the minimum scale necessary. As soon as there is a value defined for an attribute, the search for this attribute is cancelled and the found value is taken over. In the following table the columns represent the attributes, and the lines indicate the values of the used defaults, models, and instances The last line represents the attribute values for the shown object.

	attribute 1	attribute 2	attribute 3	attribute 4	attribute
window system	FS	FS			FS

	attribute 1	attribute 2	attribute 3	attribute 4	attribute
internal default			ID	ID	ID
model 1	M1				
model 2			M2		
model 3					M3
instance				I	
indicated value	M1	FS	M2	I	M3

2.3 Specific Features Regarding the Inheritance of Attributes

Apart from the above mentioned inheritance of attributes from the model to the instance, there is a special kind of inheritance from parent to children for the attributes *.visible* and *.sensitive*. If one of these attributes has the value *false* at the parent, the values at the children have automatically the same value. The reason for this is: e.g. a pushbutton itself can only be visible when the window where it is defined is also visible. If it is not visible, the pushbutton automatically cannot be seen either. The same applies to the behavior regarding the selection.

In order to ascertain whether the pushbutton is visible, it would be necessary to ask the pushbutton and all its parents if they are visible. Only when all of them are visible, the pushbutton as well is really visible. To make it easier, it is possible to ask the object for the attributes *.real_visible* and *.real_sensitive*. By making a query, the program then recognizes whether the object is really visible or really can be selected.

The following table illustrates the behavior of the attribute *.visible*. The same applies to the attribute *.sensitive*.

Table 1: *Inheritance of the attributes .visible and .sensitive*

Parent.visible	Child.visible	Parent.real_visible	Child.real_visible
true	true	true	true
false	true	false	false
true	false	true	false
false	false	false	false

2.4 Exceptions when Inheriting Attributes

Internally in Dialog Manager almost all attributes which can be set are inherited from the model to the instance. The exceptions are briefly listed below:

Attribute	Cause
.content of a listbox	Usually the content of a listbox is dynamically filled by the application and thus not inherited in the hierarchy.
export at all objects	The labeling of an object for exportation must be repeated whenever an object is to become an instance, if the instance is to be exported as well.

2.5 Effects on the Rule Processing

If an event occurs, the DM starts to search for appropriate rules for the event. The DM begins to search for rules with "before" at the corresponding default object. From there, it is searched for these rules at the model or models belonging to the object. Finally DM checks if a rule with "before" has been defined. All found rules are executed.

After that, the DM starts to search for "normal" rules at the object and - in case there are no rules - the DM continues the search at the corresponding model. If there is a rule defined at the object, however, the DM does not go to the model. At the model it is checked if there is an appropiate rule defined. If not, the DM continues at the corresponding model of the model or of the default. In case there is a rule at the model, however, this rule is executed and the search is interrupted.

Finally the DM starts again to search at the object for appropriate rules for the event. This time, rules are searched which are labelled with the key word "after". From the object the DM goes to the model to start the search again. Then the DM continues the search at the model or the default of the model and searches for appropriate rules as well. All found rules, however, are executed.

There is an exception to this scheme for rules depending on keyboard events ("key") and on the help event ("help"). The scheme mentioned above is also valid for these events. In addition to that, the DM goes to the parent of the object, too, in case no appropriate rule has been found all the way through. Applying this, it is quite easy to link e. g. a help system in form of a rule, if the corresponding rule is pre-served in the dialog.

The following diagram illustrates the search for rules. It is assumed that the object has a cor-responding model which is directly derived from the default.

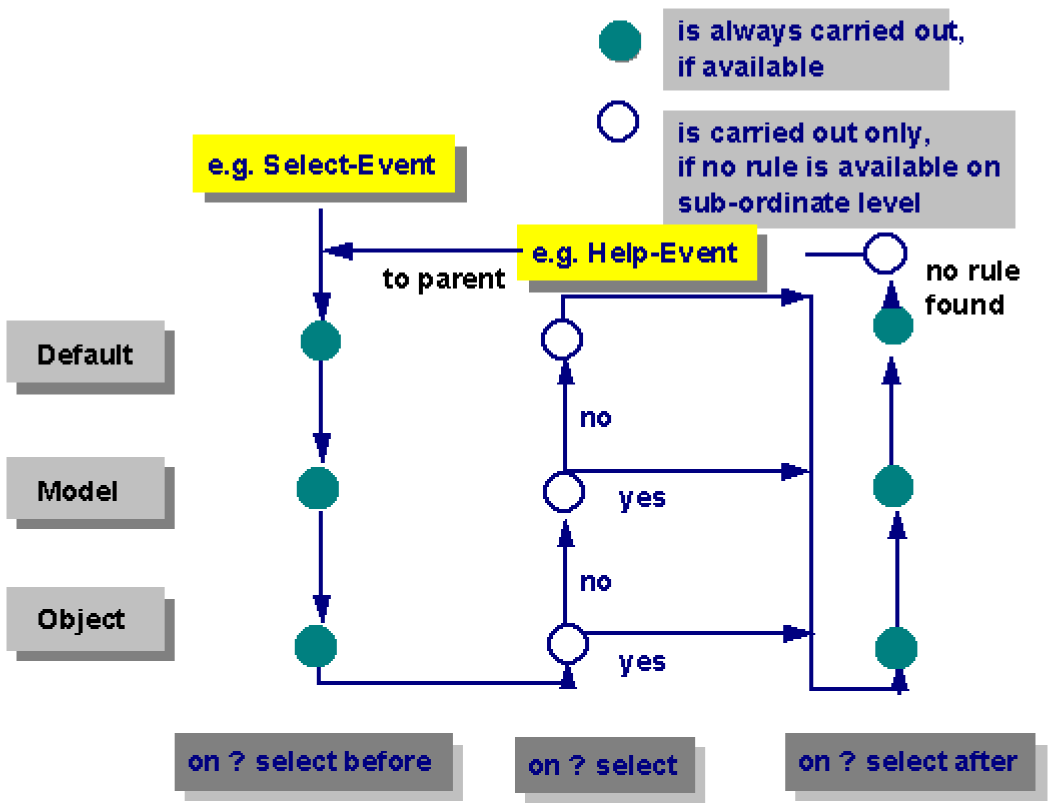


Figure 4: Order of rule processing

Example

The following 4 rules are defined for a window object:

```
(1) on WINDOW close before
(2) on WnFenster1 close
(3) on WINDOW close
(4) on WnFenster1 close after
```

The result is the following order: 1-2-4.

Rule 3 is never executed due to the existence of rule 1.

The following 5 rules are defined for a pushbutton:

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

From this the following order results: 4-2-5-3.

Rule 1 is never executed due to the existence of rule 2.

Here, a system for data input can be named as application example. In this system many windows are defined which process the contents via the OK-button and reject the contents by the cancel-button. The window is closed with both buttons.

The rules can be defined so that there is a rule which closes the corresponding window and which is dependent on the model of the cancel-button. The OK-button is more difficult to handle, since there must be carried out a window-specific processing beforehand. This is why there is a more useful possibility:

At each instance of the OK-button there is a rule which calls the current function, and at the model of the OK-button there is an "after" rule which always closes the window. Thus, it is possible to close the window centrally and need not be programmed in each instance rule.

3 User Data and User-defined Attributes

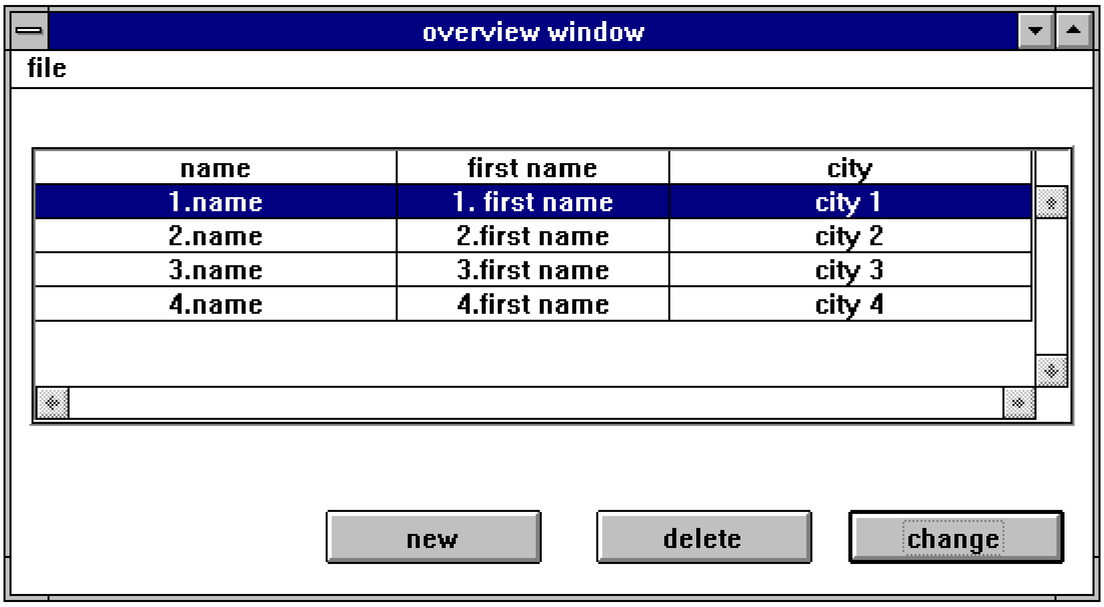
In this chapter it is shown how the attribute .userdata and user-defined attributes can be reasonably used for storing.

In order to illustrate the reasonable use of these attributes, an example representing a simple address administration is to be structured.

In the starting window, a table containing the name, surname and residence shall be represented. Using pushbuttons, the user will be able to delete and to change the entries or to add new ones. The changing and adding is done in another window, where further data such as the street and the post-code are recorded. This requires that this window can be open exactly once in each line of the table, so that the user will not get confused by double windows with identical data on the screen.

3.1 Realization of Windows

First of all the two windows which are required in the example are created with the editor. One window is created as instance as usual, since it can be open only once. The detail window, however, should be opened once each line. This is why it is defined as instance first and then it is transformed into a model.



name	first name	city
1.name	1. first name	city 1
2.name	2.first name	city 2
3.name	3.first name	city 3
4.name	4.first name	city 4

***Figure 5:** Starting window of the address administration*

In addition to that, a menu is assigned to this window, so that the program can be ended regularly by selecting the menuitem. The attributes of the table are set in the way that it has 3 columns and no shadows. Additionally all columns are aligned centered.

At the objects, rules for the behavior of the object have been partly deposited in the object already. On selecting the "new" pushbutton a new instance of the detail window is generated and displayed. The user can then enter his data into it. By selecting the menuitem "cancel", the whole system is left by calling the function "exit". By selecting the pushbutton "change", a named rule is called, since the same functionality can be triggered also by a double click on the table element.

The window in the dialog script has the following structure:

```
window WnUebersicht
{
  .width 61;
  .height 10;
  .title "overview window";
  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] "first name";
    .content[1,3] "city";
    .content[2,1] "1.name";
    .content[2,2] "1. first name ";
    .content[2,3] " city 1";
    .content[3,1] "2.name";
    .content[3,2] "2. first name ";
    .content[3,3] " city 2";
    .content[4,1] "3.name";
    .content[4,2] "3. first name ";
    .content[4,3] " city 3";
    .content[5,1] "4.name";
    .content[5,2] "4. first name ";
    .content[5,3] " city 4";
    on dbselect
    {
      !! If something in the table is really selected,
      !! display of the change dialog
      if (first(this.activeitem) > 0) then
          RlChange();
      endif
    }
  }
  child pushbutton PbNew
  {
    .xleft 17;
    .ytop 7;
    .text "new";
    on select
    {
      !! Generating the new object, first
      !! invisible
       variable object New window :=
          create(MWnDetail, this.dialog, true);

       !! Display of new object
       New window.visible := true;
    }

  }
  child pushbutton PbDelete
  {
    .xleft 32;
    .ytop 7;
    .text "delete";
  }
  child pushbutton PbChange
  {
    .xleft 46;
    .ytop 7;
    .text "change";
    on select
    {
       !! If Change is selected, display of
       !! Change dialog
       RlChange();
    }
  }
  child menubox MbFile
```

```
    {
      .title "file";
      child menuitem MiSave
      {
        .text "save";
      }
      child menuitem MiEnd
      {
        .text "cancel";
        on select
        {
             exit();
         }
      }
    }
  }
```

The detail window is as follows:

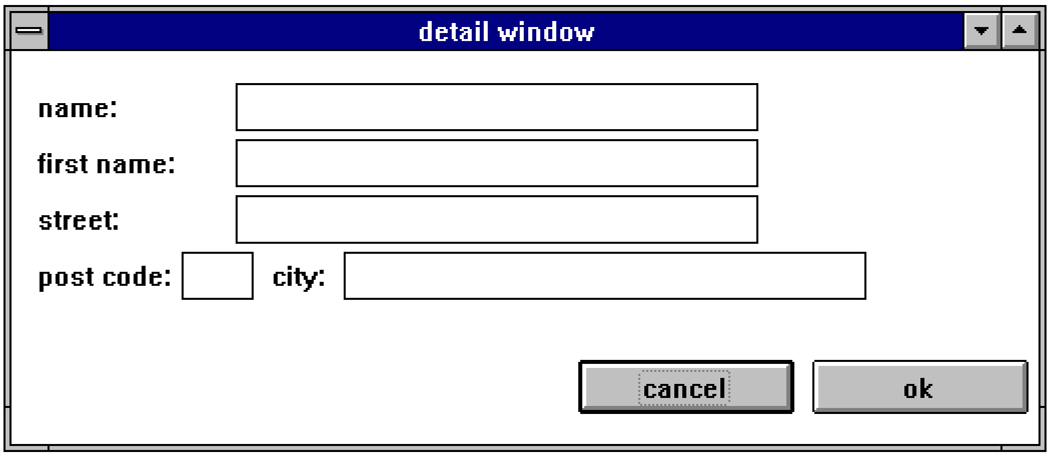


Figure 6: Window for address input

This window, as it is needed parallel during the runtime several times, it is saved as a model. Both pushbuttons are derived from a joint model. The model serves exclusively to simplify the rules.

The window is defined as follows in the dialog script:

```
model window MWnDetail
{
  .xleft 16;
  .width 58;
  .ytop 303;
  .height 8;
  .title "detail window";
  integer ZugehoerigeZeile := 0;
  child MPbOkAbbrechen PbOK
  {
    .xleft 44;
    .ytop 5;
    .text "ok";
  }
  child MPbOkAbbrechen PbAbbrechen
  {
    .xleft 31;
    .ytop 5;
    .text "cancel";
  }
  child statictext
  {
    .xleft 1;
    .ytop 0;
    .text "name:";
  }
  child statictext
  {
    .xleft 1;
    .ytop 1;
    .text "first name:";
  }
  child statictext
  {
    .xleft 1;
    .ytop 2;
    .text "street:";
  }
  child statictext
  {
    .xleft 1;
    .ytop 3;
    .text "post code:";
  }
  child statictext
  {
    .xleft 14;
    .ytop 3;
    .text "city:";
  }
  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 The MPbOkCancel Model

The model "MPbOkCancel" is defined as follows:

```
model pushbutton MPbOkAbbrechen
{
    !! General rule for the
    !! selection of the pushbuttons
    !! in the detail window
    on select after
    {
      !! On closing, the window must
      !! be destroyed
      destroy(this.window, true);
    }
}
```

By introducing this model it is possible to simplify the implementation of rules at the pushbuttons "cancel" and "ok". As both pushbuttons are derived from this model and the rules at this model are defined with the key word "after", this rule will be triggered whenever one of the both pushbuttons is selected. It is thus not necessary to define another rule for the cancel-pushbutton; but the acceptance of changes must still be implemented for the OK-pushbutton.

3.3 Assigning Detail Windows to a Line

Assigning detail windows to a line in the table is made by the following mechanism:

In the userdata of the first column of each line, the ID of the corresponding window is deposited. If a window belonging to a line is now to be opened, first the userdata is checked, whether there has already been saved a window. In this case, the ID which is stored there is not the ID *null*. Then the window is only activated setting it again on visible. If the ID, which is stored in the userdata, is the ID *null*, there is no window belonging to this line. Then a new window is generated, filled with the corresponding data and displayed. In this case, the use of userdata is very reasonable, since each cell of the table disposes of userdata and Dialog Manager takes over the management of this input. If a line is deleted or added, the userdata is corrected respectively. This would not apply to user-defined attributes and would have to be managed in the application.

When assigning a window to a line it is different. It would be possible to use even here the userdata of the window in order to memorize the necessary information. But normally this is made by the user-defined attribute, since one single attribute at the object is sufficient for assigning. The legibility of the dialog code is so substantially increased.

The detail window in the dialog script has thus the following structure:

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

In the attribute "ZugehoerigeZeile" the number of the line, the window of which has been opened, is memorized. If the window is reopened by the "new" pushbutton, no new value is deposited, but the defined value *0* remains. On selecting the "ok" pushbutton it can now be recognized whether the window has been opened by "new" or "change".

The rule to change a data record has the following structure:

```
rule void RlChange
```

```
{
    !! Setting a variable to save
    !! the window
    variable object New window;

    !! Taking over userdata from column 1 of the
    !! active line of the tablefields
    NeuesFenster := TfAdressen.userdata
        [first(TfAdressen.activeitem),1];
    !! If there is no window yet,
    !! a window must be generated
    !! and entered in the userdata
    if (NeuesFenster = null) then
      NeuesFenster := create(MWnDetail, this.dialog, true);
      !! Memorizing the new windowin the userdata
      TfAdressen.userdata[first(TfAdressen.activeitem),1]
        := NeuesFenster;
      !! Taking over the values from the active line
      !! into the new window
      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];
      !! Now the window must memorize
      !! the line from which it was opened.
      !! This is achieved by the attribute RelevantLine.
      NeuesFenster.ZugehoerigeZeile :=
        first(TfAdressen.activeitem);
    endif
    !! Finally the window is made visible.
    !! If it was already visible, it is just shifted into the
    !! foreground
    NeuesFenster.visible := true;
}
```

By selecting the "ok" pushbutton, the data from the detail window is transferred to the overview window. At the value of the attribute "ZugehoerigeZeile", it is possible to recognize if the window has been opened by "new" or "change". If it has been opened to change, the data is recorded in the original line. If it has been opened for a new data record, the table is extended by one line and the new data record is added at the bottom of the table.

```
!! If OK is selected,
!! the data must be taken over into the
!! overview window.
on PbOK select
```

```
{
    !! In the attribute RelevantLine of the window
    !! the information is stored,
    !! where the data must be written again.
    !! There is one exception: 0.
    !! This value says that the window has been opened
    !! via NEU.
    if (this.window.ZugehoerigeZeile = 0) then
      !! In this case a line must be inserted at the
      !! end.
      TfAdressen.rowcount := (TfAdressen.rowcount + 1);
      this.window.ZugehoerigeZeile := TfAdressen.rowcount;
    endif
    !! Now the contents can be taken over.
    !! Attention: The accesses in the detail window must
    !! all be relative, since it is a model.
    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;
}
```

This realization has an additional advantage: all references to an object are removed in the dialog, if this object is destroyed. On destroying a detail window, the programmer thus does not need to set the value to *null* in the corresponding userdata. This is automatically carried out by Dialog Manager.

3.4 Deleting a Line

When deleting a line, the method ":delete" can be used. This method deletes one or more lines/-columns from a table. All information about the concerned line of the column is then deleted, also the corresponding userdata.

If a line is now deleted, all windows which have been opened from a succeeding line must be informed that their "ZugehoerigeZeile" has changed. To avoid the examination of all table elements, this loop should start with the line number of the just deleted line and run until the number of lines.

```
!! Deleting the active line in the table
!! If, additionally, there is a window open,
!! destroy this window first, then delete the line
on PbLoeschen select
{
    !! loop variable
    variable integer I;
    !! Check if there is a corresponding window
    if (TfAdressen.userdata[first(TfAdressen.activeitem),1]
    <> null) then
     destroy(TfAdressen.userdata[first(TfAdressen.activeitem),
       1], true);
    endif
    !! Memorizing of the active line. After deleting,
    !! this information can no longer be queried.
    I := first(TfAdressen.activeitem);
    !! Deleting the line
    TfAdressen:delete(first(TfAdressen.activeitem), 1, false);
    !! Shifting the line information
    !! in all following windows
    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 Additional Rules

The dialog starting rule makes sure that nothing is selected in the table and that all userdata of the table is initialized with the value *null*.

```
on dialog start
{
    !! Secure that nothing is saved in the
    !! userdata
    TfAdressen.userdata[0,0] := null;
    !! Deleting all selections
    TfAdressen.activeitem := [0,0];
}
```

In order not to implement the selectivity of both pushbuttons "change" and "delete" several times, a rule is defined which reacts to the changes in the state of selection in the table. Only if there is a line selected after such a change, both pushbuttons may be selectable.

```
!! Whenever the selection is changed in the table
!! it must be checked, if the pushbuttons
!! may still remain sensitive or if they have to become sensitive
on TfAdressen select, .activeitem changed
{
    !! If there is no line selected,
    !! set both pushbuttons on insensitive;
    !! otherwise set both on sensitive
    if (first(TfAdressen.activeitem) = 0) then
      PbLoeschen.sensitive := false;
      PbAendern.sensitive := false;
    else
      PbLoeschen.sensitive := true;
      PbAendern.sensitive := true;
    endif
}
```

4 Object-oriented Programming

This example is intended for programmers who are already experienced in the use of the Dialog Manager. Here, methods about object-oriented programming in the dialog description language are shown. The possibility to define separate attributes which is available in the system is used intensively. In these user-defined attributes all necessary information which is normally found in the class structure of an object is deposited.

4.1 Description of the Systems to be Developed

Using the Dialog Manager a prototype for the administration for customers and orders is to be developed. This is to be carried out object-oriented, so that the system can be easily extended and maintained. The following structure is valid for both of these application objects:

Customer

- List of all customers
  - Information about the customer
    - Invoices as detail information
    - Open orders as detail information

Order

- List of all orders
  - Information about the order
    - Production orders belonging to the order
    - Machines covered by the order

The following diagram represents the structure of windows in the system

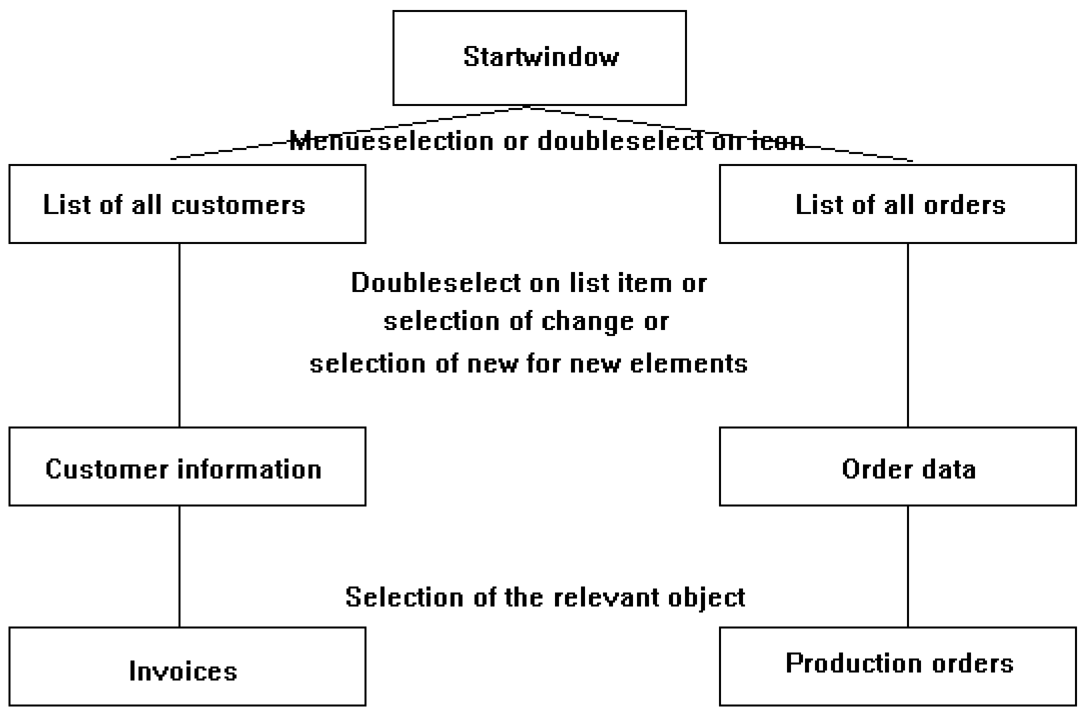

Startwindow
Menueselection or doubleselect on icon
List of all customers
List of all orders
Doubleselect on list item or selection of change or selection of new for new elements
Customer information
Order data
Selection of the relevant object
Invoices
Production orders

Figure 7: *Structure of a customer system*

4.2 Approach of Realization

The procedure of the implementation was in the way that the processing parts (rules) do not actually know with which kind of objects they currently work. The information which is necessary for the rules are all transmitted in the form of parameters. In order to be able to form them as universal as possible, first the real objects were examined and the functions which can be applied to these objects were defined. In doing so, it was recorded in which parts the individual methods at the objects differ and in what way. These differences were then translated into appropriate DM structures. With this translation, all parts which are used several times were deposited in the models. Then the corresponding individual structures were structured from these models. The individual structures thus contain all the information which is necessary for the processing of the object.

To be able to administrate correctly this information in the window, each open window has a reference to the basic object structure and a reference to the object which is really represented in the window. The reference to the basic object is necessary to be able to carry out the usual reaction of the object to the user events. The reference to the real object is necessary to find out if the object is already represented in a window. Apart from the windows, all objects which must work directly with the object structures, contain information about how these objects must be treated. At the edittexts the element of the internal data structure the contents of which they shall display is deposited. With images the object type to which they belong and how they are to react to events is deposited. At the pushbuttons, the method which is to be triggered on selecting them is deposited.

This deposit of information at individual object types is only valid for this example, of course. It only illustrates how such information is to be deposited and processed in a reasonable way.

4.3 Implementation in the Dialog Script

4.3.1 Naming

In the dialog script a certain naming for the objects was applied in order to be able to recognize the individual object types by the name. This scheme has the following structure:

```
[M]<object identification>object name
```

M

If "M" is the first letter it is a model. Then follow the letters identifying the object type.

object identification

The following abbreviations were used for the object identification:

Pb	specifies Pushbutton
Wn	specifies Window
Rc	specifies Records
Et	specifies Edittexts
Lb	specifies Listboxes
Im	specifies Images

object name

Describing name of the object

4.3.2 Data Structures

The class structure is reflected in a dialog structure. For this several substructures are made in order to be able to administrate all elements in the simplest way possible later on. This structure contains the following individual structures: a structure for orders, a structure for customers, a structure to store objects, a structure for methods.

4.3.2.1 Structure for Orders

In the order structure all information belonging to an order is stored. This includes the number of the order, the number of the item, the number, the dates for starting, ending and delivery. This structure is stored as a model, since any number of orders can be entered into the system.

```
!!data structure for the order
model record MRcAuftrag
{
  string AuftragsNr := "";
  string Artikel := "";
  integer Anzahl := 0;
  string FStart := "";
  string FEnde := "";
  string Liefer := "";
}
```

4.3.2.2 Customer Structure

The customer structure contains all the information concerning the customer. This includes the customer number, the company, the address, and the contact.

```
!!data structure for the customer
model record MRcKunde
{
  string KundenNr := "";
  string Firma := "";
  string Strasse := "";
  integer Plz := 0;
  string Ort := "";
  string Partner := "";
  string Telefon := "";
  string Fax := "";
}
```

4.3.2.3 Structure for Saving Elements

In order to be able to administrate also internally any number of objects (customers or orders), there is a structure which can be adapted in its size to the necessity during runtime. Any number of objects of any kind can be stored in the field which is realized accordingly.

```
!!data structure for saving the values. The structure
!!is extended if necessary.
model record MRcSpeicher
{
  object Elemente[5];
  .Elemente[1] := null;
  .Elemente[2] := null;
  .Elemente[3] := null;
  .Elemente[4] := null;
  .Elemente[5] := null;
}
```

In order to record values in this structure, the following rule, which can take over any number of elements into this structure, was defined.

```
!!Taking up a new element into the memory table
rule object Rl_NeuesElement (object ObjektInfo input)
{
  variable integer I;
  variable integer FreierPlatz := 0;

  !! Looking for free space in the list
  for I:= 1 to ObjektInfo.MRcSpeicher.count[.Elemente] do
    if ((ObjektInfo.MRcSpeicher.Elemente[I] = null)
    and (FreierPlatz = 0)) then
      FreierPlatz := I;
    endif
  endfor
  !! No space was found: memory area must
  !! be extended
  if (FreierPlatz = 0) then
    ObjektInfo.MRcSpeicher.count[.Elemente] :=
    (ObjektInfo.MRcSpeicher.count[.Elemente] + 10);
  !! Initialization of new elements with 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

  !! Creating of the object to be saved newly
  ObjektInfo.MRcSpeicher.Elemente[FreierPlatz] :=
    create(ObjektInfo.ObjektArt, this.dialog);
  !! Return of the newly created object
  return ObjektInfo.MRcSpeicher.Elemente[FreierPlatz];
}
```

4.3.2.4 Structure for Depositing Methods

This object represents the kernel of the example. In the method structure all methods which can be applied to the different objects in the system are deposited. First, in all objects it is examined, which methods must be realized. In this example, these are the following methods:

Create

In this item it is deposited which window is to be used for creating and modifying the main information about an object.

Init

Here, the initialization function is deposited. That is the function which is to load all necessary data from the database. Since no functions are implemented in this example, there is only deposited a dummy example each.

Detail1-3

Here, the windows which can deliver different detail information to the objects are deposited.

List

In this element the window in which the object list is to be shown is deposited.

Rlist

Here the rule which can fill the list in the overview window is deposited.

Tlist

This item contains the text which is to serve as title for the list window

```
!!This object describes the methods which are
!!available for an object.
!!Create: window to create an object
!! (Acquisition of basic data)
!!Init: rule/function to initialize the data structure
!!Detail[1]: window for 1 subinformation
!!Detail[2]: window for 2 subinformation
!!Detail[3]: not used
!!List: window, in which the listof the objects is
!!to be displayed
!!RList: rule which is responsible for the editingof data
!!for the list
!!TList: text which is to be displayed as title
!!in the list window
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 Object Structure

The actual object structure consists of other structures. This object structure contains an element where the object type (MRcorder or Mrcustomer) is stored. In addition, this structure contains both substructures: MRcSpeicher to store the objects and MRcMethoden for the definition of methods belonging to the object.

```
!!Structure representing the basic object of an object class
```

```
model record MRObjekt
{
  object ObjektArt := null;
  child MRcSpeicher
  {
  }
  child MRcMethoden
  {
  }
}
```

4.3.2.6 Order Object

The order object is an instance of the object structure. Here, the corresponding items are set.

```
!!Realized object for the orders
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 := "order list";
}
```

4.3.2.7 Customer Object

The customer object is an instance of the object structure. The corresponding items for the methods are set here.

```
!!Realized object for the customers
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 := "customer list";
}
```

4.3.2.8 Superordinate Structure

To be able to use automatically all data structure on starting and ending the program, a superordinate structure was created which contains only references to subordinate structures. This enables you to create a loop over all data structures existing internally on starting the program in order to make them initialize according to their initialization method.

```
!!The following data structure only helps to initialize
!!all structures on
!!starting the program. These structures must be read
!!from the database, of course; in this example, however,
!!it has not been implemented. In order to maintain the data,
!!the dialog saves itself again.
!!Thus, even the changed values are preserved.
record RcObjekttypen
{
  object Objekte[2];
  .Objekte[1] := RcAuftraege;
  .Objekte[2] := RcKunden;
}
```

4.3.3 Extension of Existing Objects

The objects which are contained in DM were partly extended by corresponding attributes in order to be able to store all the necessary information there.

4.3.3.1 Extensions at window Object

This object was extended by four attributes in order to be able to process all the windows with the same rule. The attribute "dynamic" indicates if the window in the window can only be opened exactly once or if the window may be opened at any time by the user with other data. The attributes "MethodenObjekt" and "DargestelltesObjekt" are references which are used to identify the contents represented in the window. The "Methoden Objekt" is a reference to the basic object of orders or customers. The "Dargestelltes Objekt" is a reference to the internal data structure belonging to this window. The attribute "ZuAktivierendesObjekt" recognizes which object is to be activated on opening the window.

```
!!Attribute MethodObject, DisplayedObject, DynamicObject
!!and Object to be activated are inserted
!!Meaning of attributes
!!MethodObject contains all information about the object type
!!DisplayedObject is the currently processed object
!!in the window
!!Dynamic is a label indicating how the window is
!!to be treated on making it invisible.
!!Object to be activated contains the object which is to be
```

```
!!activated on opening the window.
default window
{
  ...
  object MethodenObjekt := null;
  object DargestelltesObjekt := null;
  boolean Dynamisch := true;
  object ZuAktivierendesObjekt := null;
}
```

Apart from these attribute extensions a global rule for the default window was defined which can be used to make the window visible or invisible. On making it invisible it must be checked if the window was dynamically generated. In this case the window again must be internally destroyed. On making a new window visible the right object in the window is activated by this rule. The user can thus start the process of the window at once.

```
!!General rule for the treatment of windows which have been
!!made visible / invisible.
on Dialog.WINDOW close, .visible changed
{
  variable integer I;

  if ( not this.visible) then
    !!Window was generated dynamically, i.e. it must
   !! be deleted now
    if this.Dynamisch then
      !! destroying the window
      destroy(this, true);
    endif
  else
    !! Window has been made visible
    !! Looking for the object to be activated
    !! or focussed
    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
}
```

If a window for a certain data record is to be opened, it is checked first whether this window has been already opened. In this case, all windows are searched and checked if the currently regarded window belongs to the basic object type, if it has the type searched for and if it represents the data object. If such a window is found, it is returned by the rule, otherwise *null* is returned.

```
!!This rule searches in all open windows for a window
!!with given data. Once this window is found,
!!it is returned as result,
!!otherwise null is returned
rule object Rl_SucheNachFenster (object ObjektInfo input, object FensterArt
input, object DargObj input)
{
  variable integer I;

  !! Checking all children of the dialog
  for I := 1 to this.dialog.childcount do
    !! Looking if the object is a window
    if (this.dialog.child[I].class = window) then
      !! Looking if the window belongs to the searched for
      !! Methodobject
      if(this.dialog.child[I].MethodenObjekt=ObjektInfo)
      then
        !! Looking if the window belongs to the window type
        !! searched for
        if (this.dialog.child[I].model = FensterArt) then
          !! Looking if the same information is
          !! displayed
          if (this.dialog.child[I].DargestelltesObjekt =
          DargObj) then
            return this.dialog.child[I];
          endif
        endif
      endif
    endif
  endfor
  !! Window was not found, null is returned.
  return null;
}
```

The next rule creates a window of a defined type for the indicated object. First the window is generated invisibly in order to be able to fill data into the window even in the invisible state.

```
!!Creating a window for a defined data record.
!!First it is checked whether the window is already open
!!and can be re-used. If the window is marked as
!!dynamic, then the found
!!window is re-used
```

```
rule object FensterNeuAnlegen (object ObjektInfo input, object FensterArt
input, object DargObj input)
{
  variable object Obj := null;

  !! checking if a window type is given
  if (FensterArt <> null) then
    !! checking if the window type is a window or a
    !! messagebox
    if (FensterArt.class = window) then
      !! checking if the window already exists
      Obj := Rl_SucheNachFenster(ObjektInfo, FensterArt,
      DargObj);
      !! checking if the window is to be generated
      !! dynamically
      if ((FensterArt.Dynamisch = true) and (Obj = null))
      then
        !! invisible generating of the window
        Obj := create(FensterArt, this.dialog, true);
      else
        !! activating the window
        if (Obj <> null) then
          Obj.visible := true;
        else
          !! Resetting data into the static window
          FensterArt.visible := true;
          Obj := FensterArt;
        endif
      endif
      !! Entering values into the found / newly generated
      !! window
      Obj.MethodenObjekt := ObjektInfo;
      Obj.DargestelltesObjekt := DargObj;
      return Obj;
    else
      !! Opening message window
      querybox(FensterArt);
    endif
  endif
  return null;
}
```

4.3.3.2 Extensions at the Object edittext

The edittext was extended by an attribute describing the element of the internal structure out of which the contents are to be indicated in the object. Therefore an attribute of the type "attribute" was added to be able to take the attribute of the data structure.

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

Due to this extension a general rule can be defined now. This rule copies the data from the display object to the internal structures and vice versa. All children from a given window are looked at and checked if they have an equivalent in the internal structure. If this is the case, the values are copied according to their data types.

```
!!rule for the handling of information display and taking
!!over the changed data into the internal structure
rule void Rl_AnzeigeHandhaben (object ObjektInfo input, object Fenster input,
boolean Display input)
{
  variable integer I;

  !! data to be taken over into the internal
  !! structure
  if ( not Display) then
    !! Checking if an internal object already exists
    if (Fenster.DargestelltesObjekt = null) then
      !! Creating an internal object
      Fenster.DargestelltesObjekt :=
      NeuesElement(ObjektInfo);
    endif
    if (Fenster.DargestelltesObjekt <> null) then
    !! Passing through the window´s children and taking over
    !! the values into the internal strukcture
    for I := 1 to Fenster.childcount do
    !! Checking if the object is an Edittext
    if (Fenster.child[I].class = edittext) then
      !! Checking if the object belongs to a meaningful
      !! attribute
      if (Fenster.child[I].Datenelement <> .visible) then
      !! Checking the data type of the internal structure
      if (Fenster.DargestelltesObjekt.type
        [Fenster.child[I].Datenelement] = integer) then
        !! Checking if there has really been input
        !! a number
        if fail(atoi(Fenster.child[I].content)) then
        !! Taking over the value into the internal structure
        setvalue(Fenster.DargestelltesObjekt,
          Fenster.child[I].Datenelement, 0);
        else
        !! Taking over the value into the internal structure
        setvalue(Fenster.DargestelltesObjekt,
          Fenster.child[I].Datenelement,
          atoi(Fenster.child[I].content));
        endif
      else
        !! Taking over the value into the internal structure
        setvalue(Fenster.DargestelltesObjekt,
          Fenster.child[I].Datenelement,
          Fenster.child[I].content);
      endif
      endif
    endif
    endfor
    !! Re-structuring the selection list
    ListeFuellen(Fenster.MethodenObjekt,
    InstanzVon(Fenster.MethodenObjekt.MRcMethoden.List,
      Fenster.MethodenObjekt));
  endif
  else
    !! Passing through the window´s children and taking over
    !! the values into the internal structure
    for I := 1 to Fenster.childcount do
      !! Checking if the object is an Edittext
      if (Fenster.child[I].class = edittext) then
      !! Checking if the object belongs to a meaningful
      !! attribute
      if (Fenster.child[I].Datenelement <> .visible) then
        if (Fenster.DargestelltesObjekt <> null) then
          if (Fenster.DargestelltesObjekt.type
            [Fenster.child[I].Datenelement] = integer)
          then
            !! Taking over the value into the display
            Fenster.child[I].content :=
              itoa(getvalue(Fenster.DargestelltesObjekt,
              Fenster.child[I].Datenelement));
          else
            !! Taking over the value into the display
            Fenster.child[I].content :=
              getvalue(Fenster.DargestelltesObjekt,
              Fenster.child[I].Datenelement);
          endif
        else
          !! Taking over the value into the display
          Fenster.child[I].content := "";
        endif
      endif
      endif
    endfor
  endif
}
```

To work efficiently the rule must be able to process the list window which is responsible for this object type. The following rule is available.

```
!!Searching for a window which is the instance of
!!a given model
rule object InstanzVon (object Modell input, object Methode input)
{
  variable integer I;

  !! Looking at all children of the dialog
  for I := 1 to this.dialog.childcount do
    !! Checking if the object is a window
    if (this.dialog.child[I].class = window) then
      !! Checking if the window is an instance of the
      !! model
      if ((this.dialog.child[I].model = Modell)
      and (this.dialog.child[I].MethodenObjekt = Methode))
      then
        return this.dialog.child[I];
      endif
    endif
  endfor
  !! no instance was found, return null.
  return null;
}
```

4.3.3.3 Extensions at the Object image

The image was extended by an attribute representing a reference to the method object. With this attribute the corresponding methods can be called when there are events at images.

```
!!attribute MethodObject inserted.
!!This object contains all information which has to be
!!memorized concerning an "object"
default image
{
   ..
  object MethodenObjekt := null;
}
```

4.3.3.4 Extensions at the Object pushbutton

The pushbutton was extended by an attribute which indicates the method to be called.

```
!!attribute AttributIndex inserted:
!!In this attribute the index of the method which is to be
!!called on selecting the pushbuttons is memorized
default pushbutton
{
  ...
  integer AttributIndex := 0;
}
```

4.3.4 Definitions for Individual Windows

4.3.4.1 Actions on Starting and Ending a Program

On starting the program, the internal data structures are initialized . Usually these values should be read out of the database. In this example, however, this is ignored, but in order not to lose the values, the entire dialog is saved with the input values when ending the program.

```
!!This rule provides the initialization of all
!!basic object types
rule void InitialisierenObjekte
{
  variable integer I;

  !! Passing all RcObjecttypes
  for I := 1 to RcObjekttypen.count[.Objekte] do
    !! Checking if an object is really saved.
    if (RcObjekttypen.Objekte[I] <> null) then
      !! Calling the initialization method if it is
      !! available.
      if (RcObjekttypen.Objekte[I].MRcMethoden.Init
      <> null) then
        RcObjekttypen.Objekte[I].MRcMethoden.Init(
          RcObjekttypen.Objekte[I].MRcSpeicher,
          RcObjekttypen.Objekte[I].ObjektArt);
      endif
    endif
  endfor
}

!!Saving the current dialog state on ending the dialog
on dialog finish
{
  save(this, "kunden.sav");
}

!!Starting rule for the system
on dialog start
{
  InitialisierenObjekte();
}

!!This rule is to initialize the objects of one type,
!!i.e to load them out of the database
rule void InitObjekt (object Anker input, object Typus input)
{
print "The loading out of the datatbase should be realized here";
}
```

4.3.4.2 The Starting Window

The starting window consists of two icons representing the different object types and of a menu to trigger the action and to end the dialog.

```
!!The following window appears when starting the program.
!!From this window the windows to customers and orders
!!can be opened.
window WnStart
{
  .userdata null;
  .active false;
  .xleft 400;
  .width 38;
  .ytop 396;
  .height 8;
  .title "Selection";
  .MethodenObjekt := null;
  .Dynamisch := false;
  child image IKunden
  {
    .xleft 5;
    .ytop 2;
    .text "Customers";
    .picture TiKunde;
    .MethodenObjekt := RcKunden;
  }
  child image IAuftraege
  {
    .xleft 15;
    .ytop 2;
    .text "Orders";
    .picture TiAuftrag;
    .MethodenObjekt := RcAuftraege;
  }
  child menubox MbAuswahl
  {
    .title "Selection";
    child menuitem MiListe
    {
      .text "show list";
    }
    child menuitem MiBeenden
    {
      .text "exit";
    }
  }
}

!!Menuitem for the display of the object list
on MiListe select
{
  variable object Obj;
  variable object NeuesObjekt;

  Obj := this.window.focus;
  !! Looking which windows to which object type are
  !! to be displayed
  if (Obj.MethodenObjekt <> null) then
    NeuesObjekt := FensterNeuAnlegen(Obj.MethodenObjekt,
      Obj.MethodenObjekt.MRcMethoden.List, null);
    ListeFuellen(Obj.MethodenObjekt, NeuesObjekt);
    NeuesObjekt.visible := true;
  endif
}

!!Ending the system
on MiBeenden select
{
  exit();
}
```

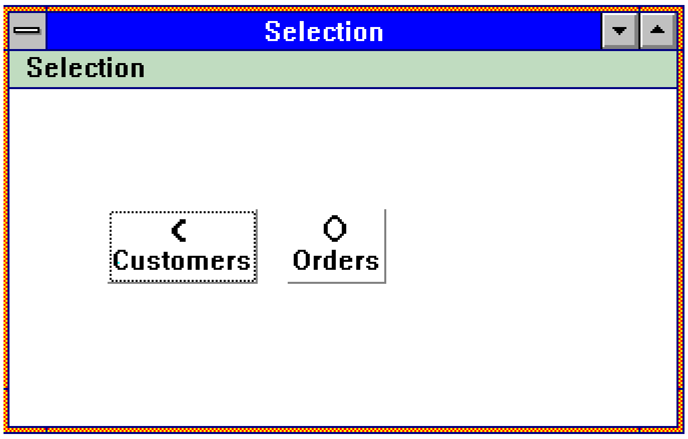


Figure 8: Starting window

4.3.4.3 The Overview Window

The overview window represents the lists of all customers and orders. Actions can be triggered with the pushbuttons at the bottom of the window. The following functionality has been implemented for this: A doubleclick on the item in the list or selecting the "information" pushbutton indicates the main data for the corresponding data record. This is either the address or the data for production. With the pushbutton "delete" the items in the list and also those in the internal structures can be deleted. By selecting the pushbutton "new", new data records can be created. This window is deposited as model, since it can be open several times with different data at the same time.

```
!!Model for listing all objects
model window MWnListe
{
  .userdata null;
  .active false;
  .xleft 78;
  .ytop 9;
  .title "list";
  .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 "new";
    }
    child pushbutton PInfos
    {
      .sensitive false;
      .xleft 16;
      .yauto -1;
      .text "information";
    }
    child pushbutton PLoeschen
    {
      .sensitive false;
      .xauto -1;
      .xright 13;
      .yauto -1;
      .text "delete";
    }
    child MPOK
    {
    }
}
```

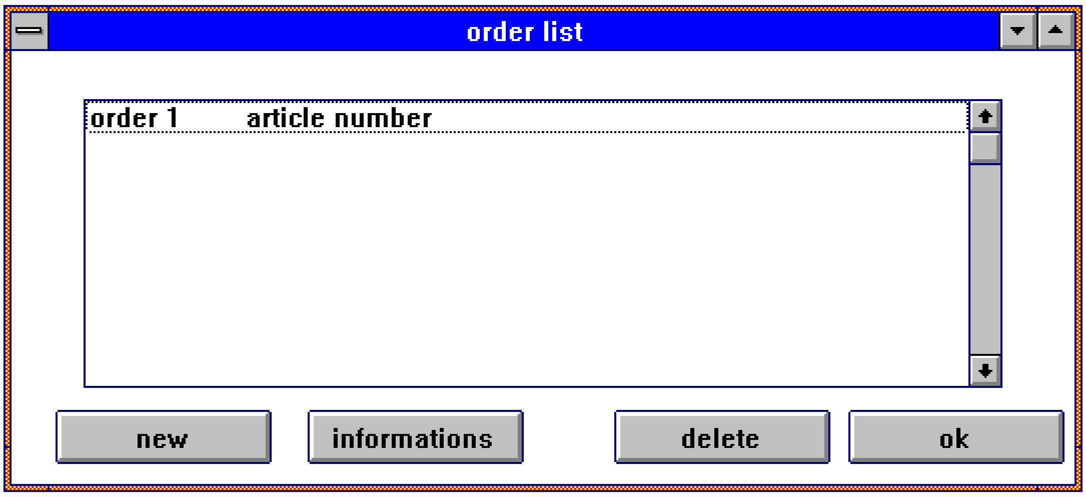


Figure 9: Order list

Rules which can fill the list belong to this window. Each object structure is searched through and the wanted text information is extracted from the object available and is displayed. Additionally the corresponding method "Rlist" of the objects is called. This method generates the external representation of the object from the internal data structures.

```
!!Structure of display string for the order list
rule string RListAuftrag (object Auftrag input)
```

```
{
    return ((Auftrag.AuftragsNr + "    ") +
      Auftrag.Artikel);
}

!!Structure of display string for customer list
rule string RListKunde (object Kunde input)
{
    return ((((Kunde.KundenNr + "  ") + Kunde.Firma) +
      "   ") + Kunde.Ort);
}

!! rules to fill the list of the objects
rule void ListeFuellen (object ObjektInfo input,
object Target input)
{
  variable string String;
  variable integer I;
!! setting the window title
  Target.title := ObjektInfo.MRcMethoden.TList;
  !! deleting all available items
  Target.LListe.itemcount := 0;
  !! Switching the pushbuttons to insensitive for deleting
  !! and detail information
  Target.PLoeschen.sensitive := false;
  Target.PInfos.sensitive := false;
  !! Passing the list of objects and items
  !! in the selection list
  for I := 1 to ObjektInfo.MRcSpeicher.count[.Elemente] do
    !! Checking if there is an object saved
    if (ObjektInfo.MRcSpeicher.Elemente[I] <> null) then
      !! Having calculated the string
      String := ObjektInfo.MRcMethoden.RList
        (ObjektInfo.MRcSpeicher.Elemente[I]);
      !! Taking over the string
      if (String <> "") then
        Target.LListe.content[(Target.LListe.itemcount +
           1)] := String;
      endif
    endif
  endfor
}
```

In addition to the listing functionality there are also rules belonging to this window. These rules control the selectivity of pushbuttons. Pushbuttons shall only be selectable when there is a selected item in the list.

```
!!Something is selected in the list, therefore the
!!pushbuttons "delete" and "information" are activated
on LListe select
{
  this.window.PLoeschen.sensitive := true;
  this.window.PInfos.sensitive := true;
}
```

If objects are newly created, the corresponding window is displayed on the screen without its contents. This action is triggered by selecting the pushbutton "new".

```
!!Rules for re-creating any kind of object
rule void ObjektNeuAnlegen (object ObjektInfo input, object Info input)
{
  variable object Obj;

  !! Generating the window / looking for an available
  !! window
  Obj := FensterNeuAnlegen(ObjektInfo,
    ObjektInfo.MRcMethoden.Create, Info);
  !! Setting data in the window
  Rl_AnzeigeHandhaben(Obj.MethodenObjekt, Obj, true);
  !! Making the object visible
  Obj.visible := true;
}

!!Rules for re-creating any kind of object
on PNeu select
{
  ObjektNeuAnlegen(this.window.MethodenObjekt, null);
}
```

To be able to change existing values you can react to the selection in the listbox by a doubleclick or by simply selecting the information pushbutton indicating the corresponding data.

```
!!Information is to be displayed.
on LListe dbselect
{
  InfosZeigen(this.window);
}

!! Information is to be displayed.
on PInfos select
{
  InfosZeigen(this.window);
}

!!Rule for the display of information about an object
```

```
rule void InfosZeigen (object Fenster input)
{
  variable object Obj;

  !! Checking if an item in the list is selected.
if (Fenster.LListe.activeitem <> 0) then
    !! Creating and displaying a window
    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
}
```

On deleting with the pushbutton "delete", the internal structures are also changed accordingly.

```
!!An object from the list is to be deleted
on PLoeschen select
{
  variable integer I;

  !! Examining all windows if the object to be deleted
  !! is still displayed somewhere.
  for I := 1 to this.dialog.childcount do
    !! Checking if the child is a window
    if this.dialog.child[I].class = window then
      !! Comparing the MethodObject and the
      !! DisplayedObject
      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
            !! Fading out the window
            this.dialog.child[I].visible := false;
          endif
      endif
    endif
  endfor
  if (this.window.LListe.activeitem > 0) then
  !! Destroying the internal structure of the window
  destroy(this.window.MethodenObjekt.MRcSpeicher.Elemente
    [this.window.LListe.activeitem], true);
  !! Closing the relevant window
  this.window.MethodenObjekt.MRcSpeicher.Elemente
```

```
    [this.window.LListe.activeitem] := null;
  !! Neuaufbau der Liste
  ListeFuellen(this.window.MethodenObjekt, this.window);
  endif
}
```

4.3.4.4 The Customer Window

The customer window illustrates the information which directly concerns the customer such as company, address, contact, and telephone. It is possible to input new data into the window or to change existing data. Before you define this window, it is necessary to define models for the pushbuttons "detail information", "ok" and "cancel" which also occur with the orders.

```
!!Model for the pushbutton asking for detail information
model pushbutton MPDetail
{
  .yauto -1;
}

!!Model for OK button
model pushbutton MPOK
{
  .xauto -1;
  .yauto -1;
  .text "ok";
}

!!Model for Cancel pushbutton
model pushbutton MPAbbruch
{
  .xauto -1;
  .xright 13;
  .yauto -1;
  .text "cancel";
}

!!Message window for parts in the dialog which have not been
!! realized yet.
messagebox Meldung
{
  .text "Not implemented yet!";
  .title "message window";
  .button[2] nobutton;
}

!!Model for the customer information window
```

```
model window MWnKunden
{
  .userdata null;
  .active false;
  .xleft 467;
  .ytop 113;
  .height 13;
  .title "customer information";
  .MethodenObjekt := null;
  .DargestelltesObjekt := null;
  .Dynamisch := true;
  .ZuAktivierendesObjekt := EtKundenNr;
  child statictext
  {
    .xleft 2;
    .ytop 0;
    .text "customer number:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 1;
    .text "company:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 2;
    .text "street:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 3;
    .text "post code:";
  }
  child statictext
  {
    .xleft 24;
    .width 0;
    .ytop 3;
    .height 0;
    .text "city:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 5;
    .text "contact:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 6;
    .text "telephone:";
  }
  child statictext
  {
    .xleft 2;
    .ytop 7;
    .text "fax number:";
  }
  child MPDetail PAuftraege
  {
    .xleft 2;
    .yauto -1;
    .text "orders";
    .AttributIndex := 2;
  }
  child MPDetail
  {
    .xleft 20;
    .text "invoices";
    .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
  {
  }
}
```

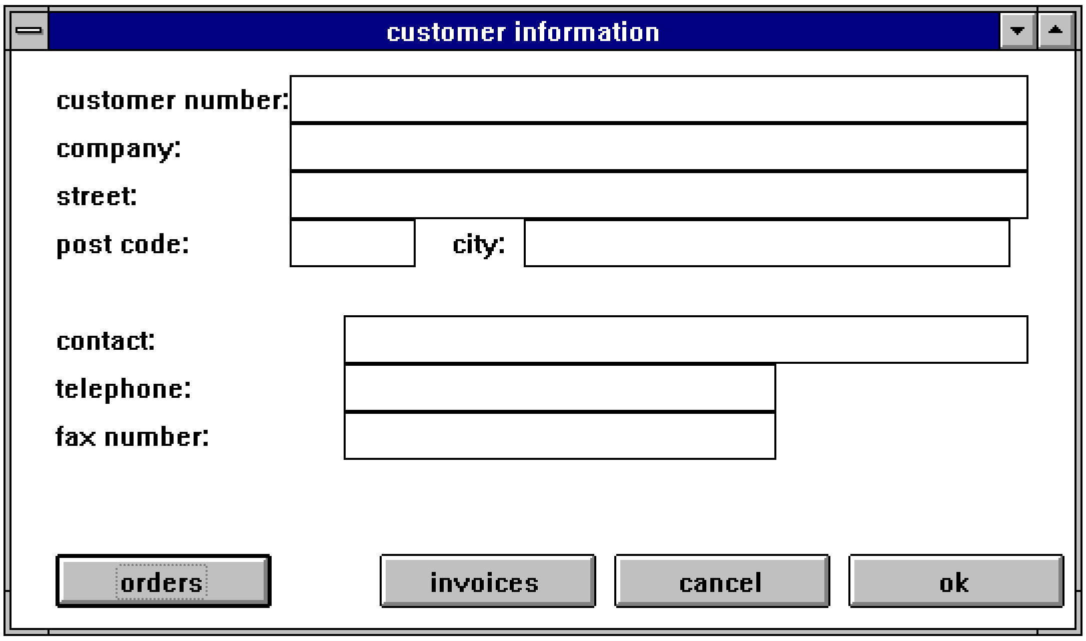


Figure 10: Customer information

By selecting the pushbutton "invoices" the invoices concerning the customer are displayed in a window. The interconnection to the orders is not realized in this prototype. This is why there appears a message on selecting the pushbutton "orders".

```
!!Rule for handling detail information 1
rule void Detail1Handhaben (object ObjektInfo input, object Fenster input,
boolean Display input)
{
  variable integer I;
  variable object TFObj;

  !! Searching for the tablefield
  for I := 1 to Fenster.childcount do
  if (Fenster.child[I].class = tablefield) then
    TFObj := Fenster.child[I];
  endif
  endfor
  !! Now the tablefield is to be filled out of the
  !! database.
  if (TFObj <> null) then
    if ( not Display) then
    endif
  endif
}

!!Displaying the detail information about the objects
on MPDetail select
{
  variable object Obj;
  !! Creating and displaying the window
  Obj := FensterNeuAnlegen(this.window.MethodenObjekt,
    this.window.MethodenObjekt.MRcMethoden.Detail
      [this.AttributIndex],
    this.window.DargestelltesObjekt);
  Detail1Handhaben(Obj.MethodenObjekt, Obj, true);
  Obj.visible := true;
}
```

If the pushbutton "ok" is selected, the data displayed in the window are taken over into the internal structures.

```
!!By selecting the OK buttons the values are to be taken over
on MPOK select
{
  Rl_AnzeigeHandhaben(this.window.MethodenObjekt,
    this.window, false);
  this.window.visible := false;
}
```

Selecting the pushbutton "cancel" the changed data is rejected and the window is closed.

```
!!By selecting Cancel the relevant window is made
!! invisible.
on MPAbbruch select
{
  this.window.visible := false;
}
```

4.3.4.5 The Order Window

The order window illustrates the information which directly concerns the customer such as company, address, contact, and telephone. Here it is possible to input data or to change existing data. This window does not dispose of rules, for the necessary rules already exist in connection with the customer window.

```
!!Model for the order window
model window MWnAuftrag
{
  .userdata null;
  .active false;
```

```
.xleft 305;
.ytop 44;
.title "order information";
.MethodenObjekt := null;
.DargestelltesObjekt := null;
.Dynamisch := true;
.ZuAktivierendesObjekt := EtAuftragsNr;
child statictext
{
  .xleft 2;
  .ytop 0;
  .text "order number:";
}
child statictext
{
  .xleft 2;
  .ytop 1;
  .text "item:";
}
child statictext
{
  .xleft 2;
  .ytop 2;
  .text "number:";
}
child statictext
{
  .xleft 2;
  .ytop 3;
  .text "production start date:";
}
child statictext
{
  .xleft 2;
  .ytop 4;
  .text "production finish date:";
}
child statictext
{
  .xleft 2;
  .ytop 5;
  .text "delivery date:";
}
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 "machines";
      .AttributIndex := 2;
    }
    child MPDetail
    {
      .xleft 17;
      .text "prod. orders";
      .AttributIndex := 1;
    }
  }
```

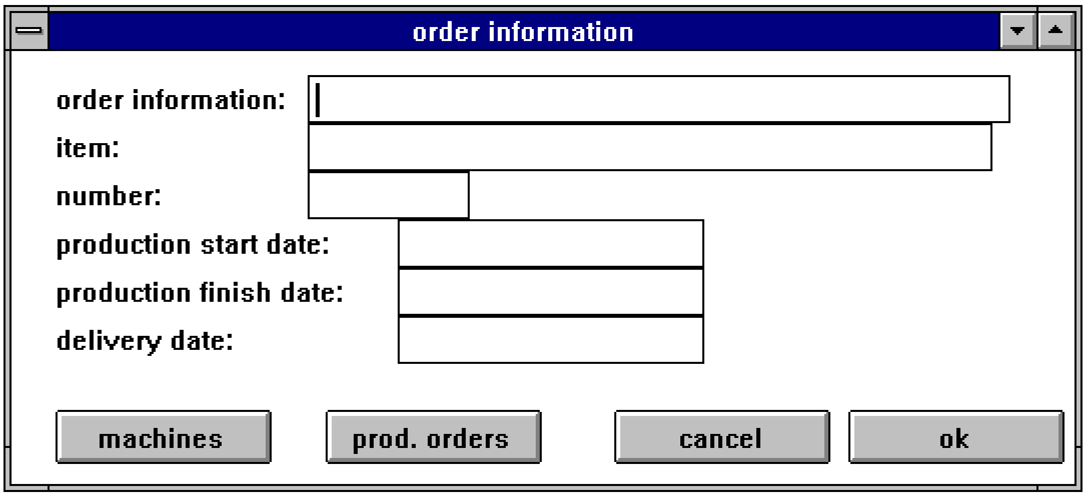


Figure 11: Order information

4.3.4.6 The Invoice Window

The invoices concerning a customer should be illustrated in a tablefield. This was not implemented in this prototype. Due to the use of models, this window does not have its own rules.

```
!!Model for the invoice window
model window MWnRechnung
{
  .userdata null;
  .active false;
  .xleft 414;
  .ytop 156;
  .title "invoices";
  .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] "invoice number ";
    .content[1,2] "date";
    .content[1,3] "price";
    .content[1,4] "state";
  }
  child MPOK
  {
  }
  child MPAbbruch
  {
  }
}
```

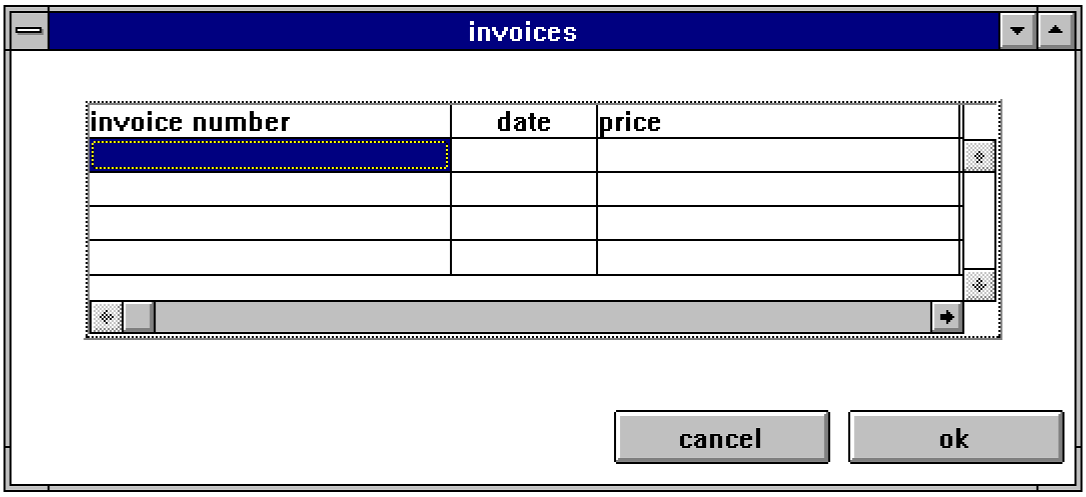


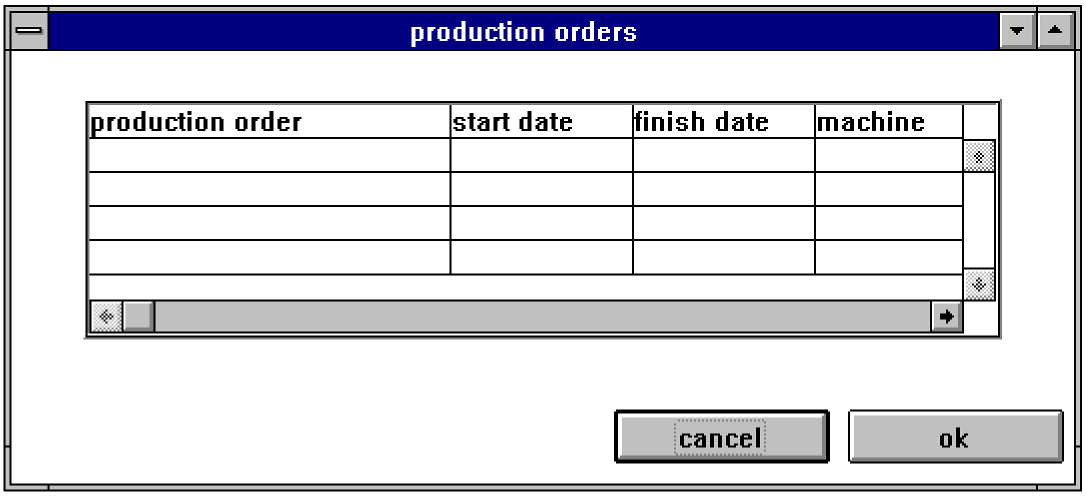
Figure 12: Invoice window

4.3.4.7 The Production Order Window

The window for production order should illustrate the production orders belonging to an order in a tablefield. This was not implemented in this prototype. This window has no rules of its own due to the use of models.

```
!!Model for production order window
model window MWnFertigung
{
  .userdata null;
  .active false;
  .xleft 220;
  .ytop 55;
  .title "production order";
  .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] "production order";
      .content[1,2] "start date";
      .content[1,3] "finish date";
      .content[1,4] "machine";
      .content[1,5] "material";
    }
  }
```



production order	start date	finish date	machine

Figure 13: Production window

5 Modularization

The modularization makes it easy to develop one dialog in teams, provided that the dialog has been split up into useful dialog parts (so-called “modules”). Moreover, resources (e.g. colors and fonts) can be provided by a central position and used by all application developers. Such dialog parts may also contain models which are available to all users. Thus it is no longer necessary to create one's own basic models, but one can use the models generated by the central position. This simplifies considerably e.g. the realization of a company's style-guide. In these dialog parts you may of course also store functionality which usually re-occur in the dialogs. Such partial functionality can so be re-used and maintained centrally.

You will find a detailed description of further application possibilities in the last section.

Apart from the aspects related to the dialog development, the modularization also has an impact on the runtime behavior of dialogs. Single dialog parts can be loaded and unloaded when required. This property makes the loading time seem shorter, since this loading time can be better spread over the entire working time of the dialog. Functions that are only rarely used can be loaded only when required and be unloaded after using to reduce the memory allocation of the dialog process.

If large dialogs are realized by partial dialogs, the maintenance increases since the dialog units are much smaller than usual. But as these dialogs are smaller you can maintain them more easily.

The modularization also bears an advantage for the use of the “Distributed Dialog Manager” (DDM): partial dialogs reduce the number of network links between client and server. Before the availability of the modularization every independent dialog had its own network connection. Now, by using partial dialogs one single connection is sufficient, which is one single server process.

5.1 Conversion of Modularization

Without modularization a dialog can be divided up into several parts. It roughly has the following structure:

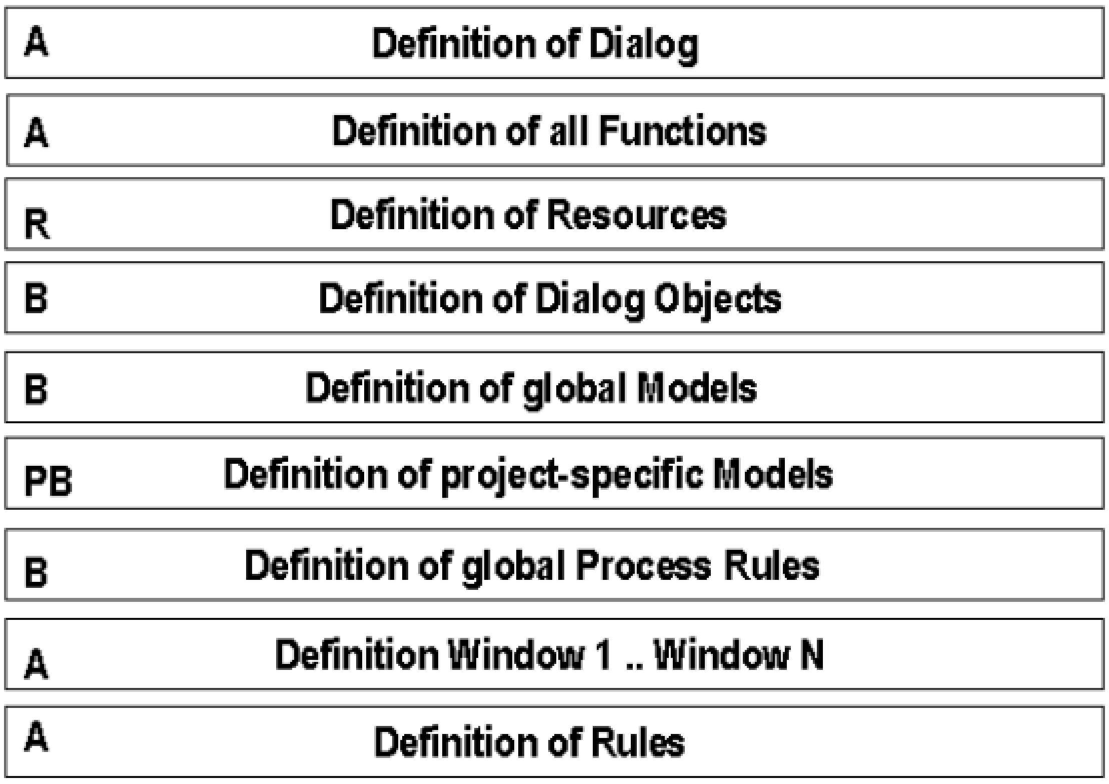

A Definition of Dialog
A Definition of all Functions
R Definition of Resources
B Definition of Dialog Objects
B Definition of global Models
PB Definition of project-specific Models
B Definition of global Process Rules
A Definition Window 1 .. Window N
A Definition of Rules

Figure 14: *Dialog structure*

Meaning

- A: application-specific dialog parts
- B: repeatedly occurring model parts
- PB: project-specific dialog parts occurring in several dialogs
- R: resources used always on the platform

The parts marked with B, PB, and R can be considered as those parts which will normally be re-used again and which should be made globally available.

This is exactly what is supplied by the modularization: these marked parts are turned into independent partial dialogs (modules) which can be used by several developers.

The dialog has the following internal structure now:

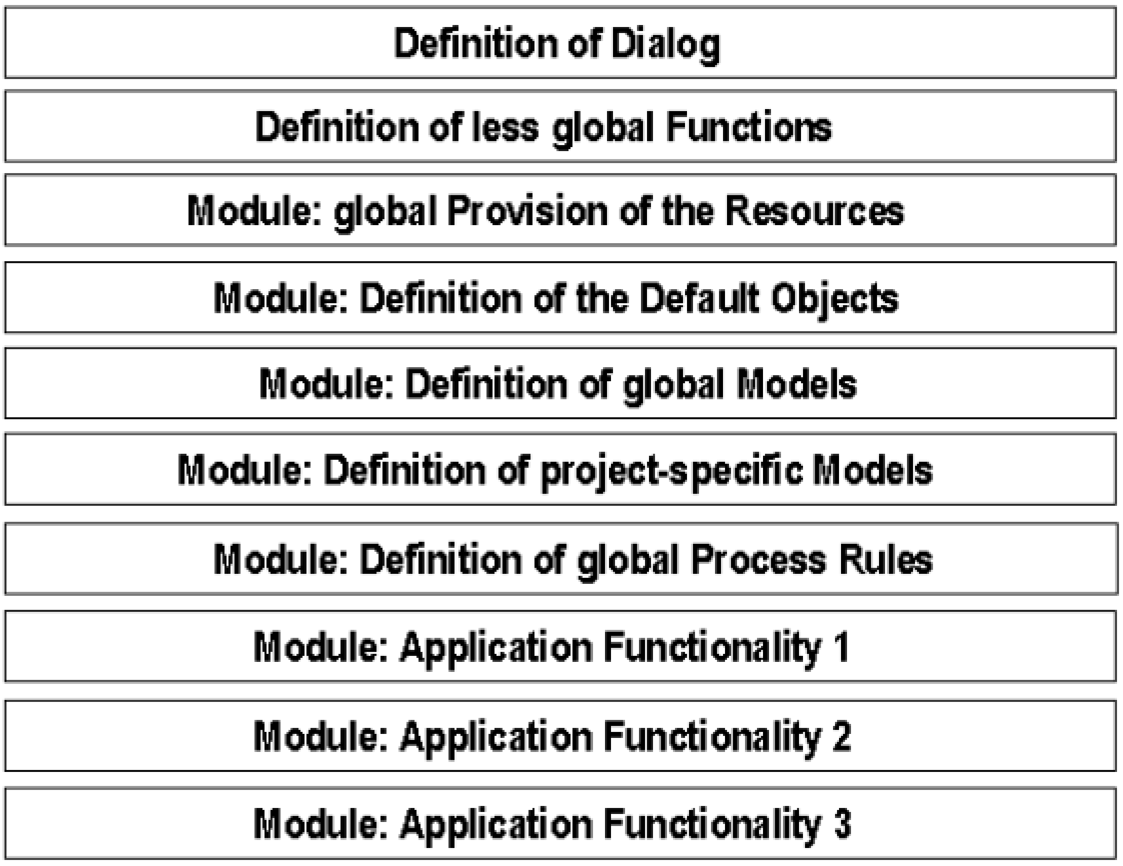

Definition of Dialog
Definition of less global Functions
Module: global Provision of the Resources
Module: Definition of the Default Objects
Module: Definition of global Models
Module: Definition of project-specific Models
Module: Definition of global Process Rules
Module: Application Functionality 1
Module: Application Functionality 2
Module: Application Functionality 3

***Figure 15:** Structure of a dialog with modules*

Each of these modules is saved in a file (binary or as script). This reduces the single file sizes, and thus can be easier maintained by the developers.

Developing without modules results in an object hierarchy which starts with the dialog as root and which ends with the objects in the individual windows. This structure is then directly saved in a file.

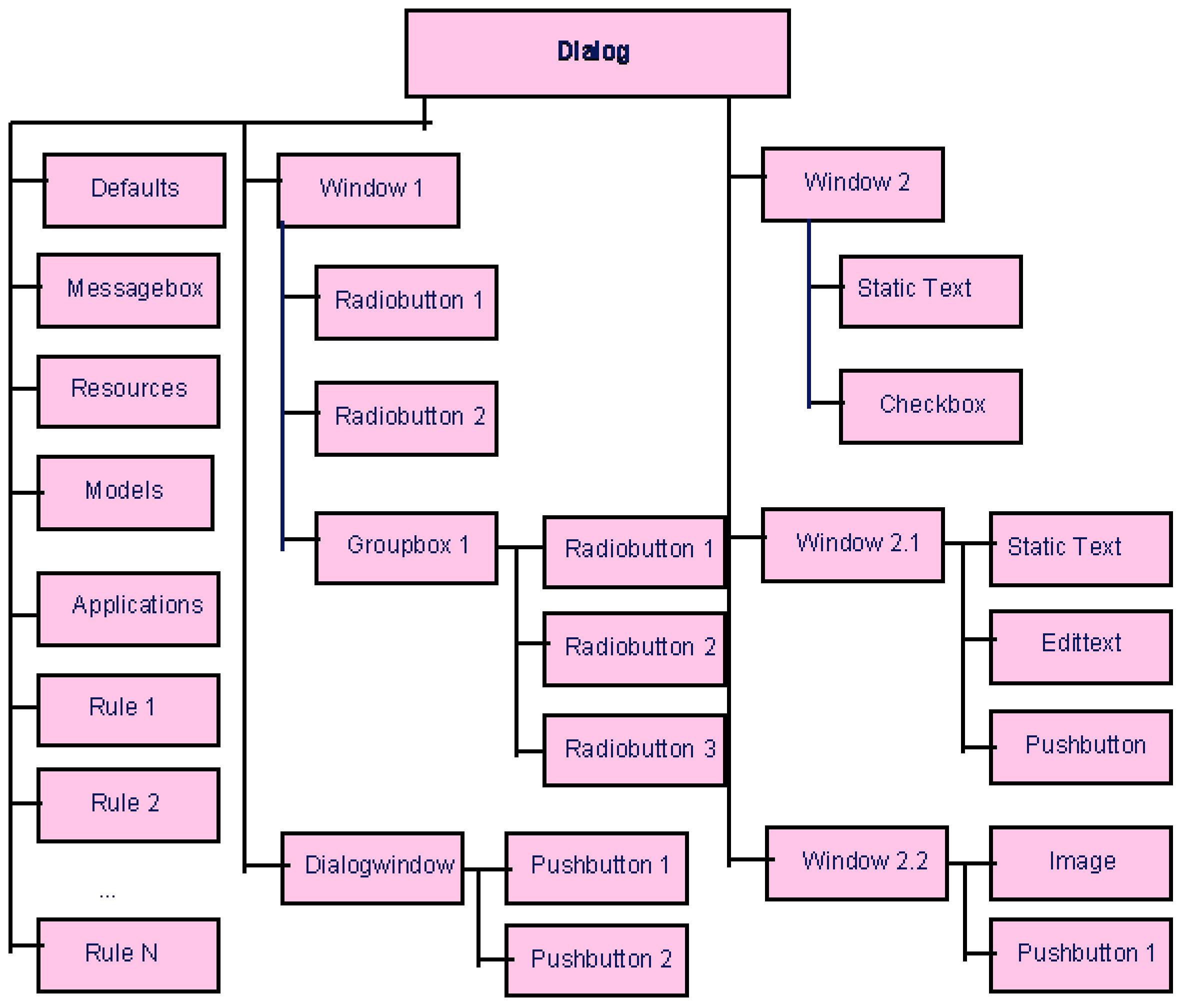


Figure 16: *Hierarchy of a dialog without modules*

Since this dialog is split up into different modules, the levels of hierarchy are extended, for there is also the hierarchy level of the modules. The modules, however, are stored in separate files in order to make the entire structure clearer.

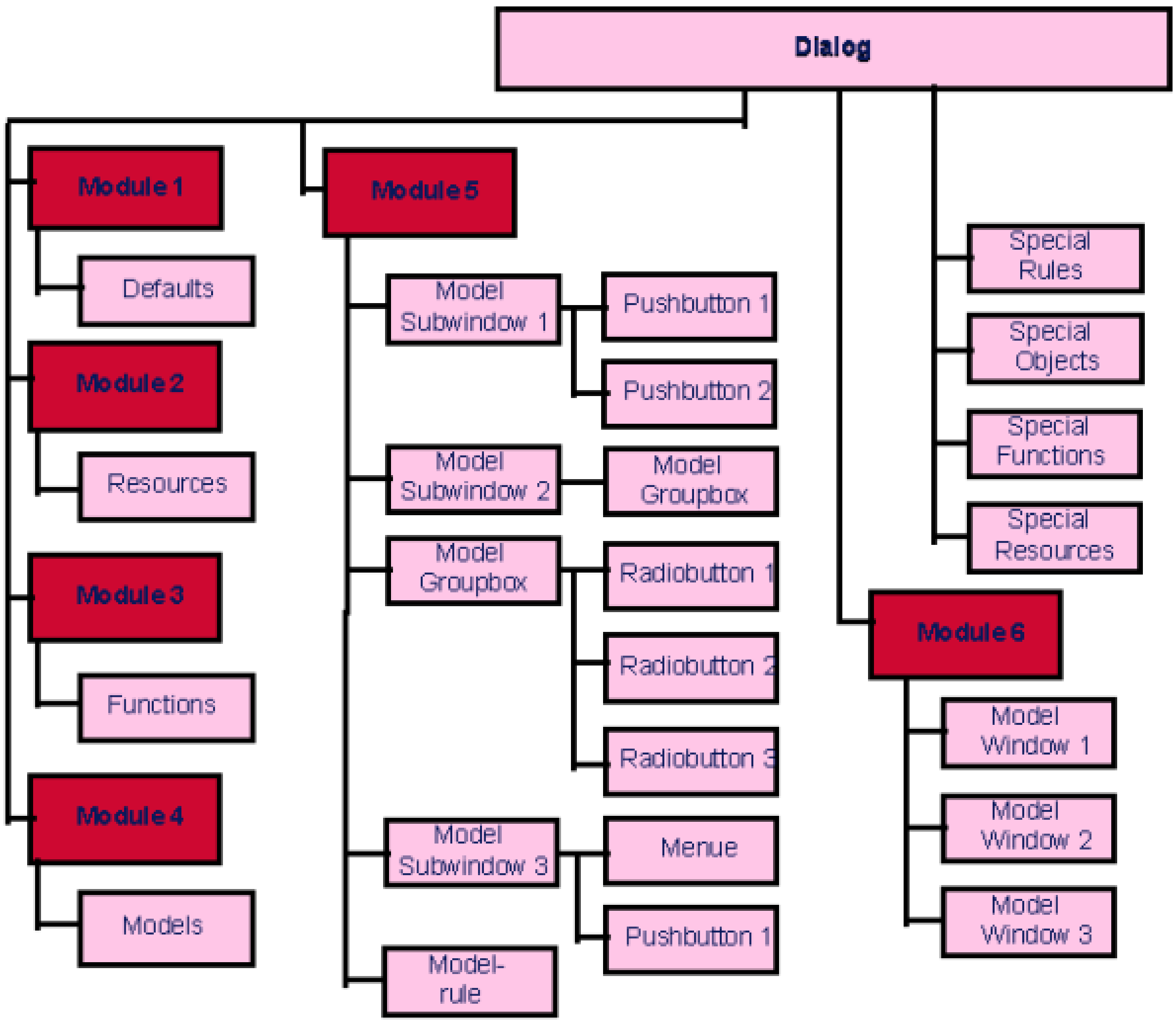


Figure 17: *Hierarchy of a dialog with modules*

5.2 Language Description

5.2.1 Keywords

The existing dialog description language has been expanded by new elements to make the modularization available. These elements are briefly listed below.

The following new objects and attributes have been integrated into the Rule Language:

Objects

- module
- import
- use (since IDM version A.06.02.g)

Attributes

- **export**
- **use**

These keywords show the module which objects are transparent outwardly and which of them are defined for module-internal use only. A detailed description of these attributes follows in the following chapters.

5.2.2 The Module

The module is an independent entity and can be compared to the DM-object dialog. It constitutes the brackets for all objects contained in it; these objects can be saved in one file. The file containing a module definition is started with the keyword **module** - in the same way a dialog file is started with **dialog**. The syntax is the same as the dialog's definition.

The keyword **module** is followed by the unambiguous name of the module.

Any resources, defaults, models or instances which build the module's functionality can be defined in a module. These are defined the same way as it is done with the object ***dialog***.

In the following example attributes which do not concern the actual understanding of the example have not been used. Missing attributes are indicated by 3 points (...).

Example

```
module TestExample

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

This example defines a module that defines a model only available in the module.

5.2.2.1 Events of the Object module

As with a dialog, a start rule and an end rule can be defined for the module. They are introduced with the keywords **start** and **finish**.

In contrast to the dialog, rules for the help-event (help) and the key-event (key)are not available for this object.

Example

```
module SimpleModule
```

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

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

This example shows that events can be defined for modules.

The same events as for the dialog are valid for the module - except for the above-mentioned events help and key.

The keyword **module** can also be used instead of the logical name in rules.

Example

```
module SimpleModule

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

Both examples have identical actions as results.

5.2.2.2 Children of the Object module

In contrast to the dialog, a module can have any objects as children. Whether the syntax is correct will become clear by the use of the children in the calling module. It is thus also possible to define non-top-level-windows as children of the module, if these are correctly integrated in the object tree.

Example

```
module Objectcollection

pushbutton MyPush1 {}

window MyWindow1 {}

listbox Myist1 {}
```

5.2.2.3 Attributes of the Object module

The object ***module*** does not have any attributes which can be requested in the Rule Language.

5.2.3 Export of Objects

Objects have to be marked with a special keyword if they are to be addressed outside the module in which they were defined. Only objects marked with the keyword **export** are outwardly transparent and form the interface to this module. Using this keyword, all objects in the sense of Dialog Manager, like resources, models, instances, and named rules, can be made visually available. Thus, it is possible to export out of the module all objects having a name, making them available for the user of the module. **export** cannot be applied to single attributes, but always to complete objects. The keyword **export** must not occur in a dialog itself, since dialogs cannot be used in other modules. Exported objects have to have unambiguous names except for defaults, which can also be exported without names.

Example

```
!! Color definition
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 );
```

In this example, the programmer of the module Color wants only four colors to be visible for the module user. There is also a fifth color in the module itself and it would be usable there, but outwardly only the resources and objects marked with **export** are known.

Example 2

```
module WindowWithButton

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

Only the window itself is known to the user in this example. The children (here: PushMe, Button) are unknown. Thus, the children cannot be used outside this modules. If a child is also be used outside this module **export** has to be typed in front of the child:

```
module WindowWithButton

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

In the modified version of the module WindowWithButton (the name makes sense now: it is a window with a pushbutton) the objects window and button are visible to the user, meaning that he can use them in his own dialog (or module). But he still does not have access to the pushbutton PushMe.

To export a child, you also have to export its parent. If this is not the case, the child will not be exported either and the keyword **export** will be ignored.

export is ignored in the following example:

```
!! export is ignored
module WindowWithButton

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

Rules which are directly linked to an object, i.e. beginning with the keyword **on** cannot be exported.

5.3 Import with use

Availability

Since IDM version A.06.02.g

5.3.1 The Alternative Import Mechanism

In addition to the way of accessing a module through an ***import*** object, the keyword **use** can be utilized to facilitate the handling of imports, modules, interface and binary files.

In contrast to an import, the module is not selected by means of a file path to the interface file, but rather by an identifier path – more precisely the **Use Path**.

The last identifier in a Use Path is, similar to the identifier of an import, the parent identifier that is used to make the exported objects known in the importing module. The preceding identifiers in the Use Path define a “package hierarchy” and can therefore be used to organize the modules. These parts will be ignored when object paths are evaluated.

Example

```
!! 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";
```

The directories in which the actual module and interface file is searched can be set as a command line option or a specific environment variable, with a call of **DM_ControlEx()** or in the Rule Language.

The existence of an interface file is actually not mandatory, but useful to facilitate loading of modules without direct usage and to support loading of the implementation as needed.

5.3.1.1 Special Features

Modules that are accessed via **use** are only loaded once in the dialog. This facilitates the use of base modules, since a module is not accidentally loaded several times due to a faulty import chain.

The decoupling from an interface or module file name has several advantages:

1. Using dedicated environment variables for the locations of interface and binary modules is no longer necessary.
2. The generation of interface and binary files is simplified.
3. It is now possible to load modules without an interface file, since a universal search path for interfaces, modules and binary modules is available.

Another special feature is the presetting of file extensions (file suffixes) for source code files (*.dlg* and *.mod*), interface files (*.if*) and binary files (*.bin*).

The conversion from a Use Path to a file path is done like this: The preceding package identifiers correspond to a directory path, the last identifier to the base name of the module.

In the example above, this means that for the Use Path `Base.Fonts`, the base file name “Base\Fonts” results. The source module has the name *Base\Fonts.mod*, the associated interface file is *Base\Fonts.if* and the binary module is *Base\Fonts.bin*.

If a module is now accessed by `use Base.Fonts`, it will be searched for these three file names in the search path to load the interface or implementation. In the development version of the IDM, the sequence is interface file → source code file → binary module. The runtime version of the IDM can only load binary files.

Some more differences to the ***import*** object exist: There are no control options through the attributes *.application*, *.static* and *.load*. It is always attempted to use the interface first and then the implementation (source code, binary). The implementation is only loaded when needed. Direct control of the loading is therefore not available.

For dynamic loading and unloading the new methods **:use()** and **:unuse()** are provided.

Basically, binary modules and interface files generated for usage with an ***import*** object also work unchanged with **use**. Conversely, this is not necessarily the case since here the reference to the module path is usually missing in the interface.

5.3.1.2 Upper and Lower Case in File Names

It is important for identifiers in the IDM and therefore also for the Use Path as well as for the file names!

5.3.1.3 Recommendations

Although a mixture of **import** and **use** is possible, it is **not** recommended, since the **import** object provokes multiple loading of modules. It is therefore recommended to switch completely from **import** to **use**.

It is also recommended to use the same spelling for file and directory names as in the Use Path. At best modules should have a unique name which does not appear in some varied form with differing upper and lower case.

Instead of loading a module with **import**, one should use Models that can be instantiated more than once without problems.

5.3.2 Language Specification and Use Path

The following language specification applies to the **use** keyword:

```
{ export | reexport } use { <Use Path> }

<Use Path>     ::= [ <package path> ] <Identifier>
<package path> ::= [ <package path> ] <Identifier> .
```

An **identifier** should begin with an uppercase letter ('A' – 'Z') and may then also contain lowercase letters, digits and underscores ('_'). An identifier may have up to 31 characters at most.

Examples for Valid Usages of use

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

The **Use Path** consists of an identifier at the end and represents the base name of the module as well as the parent identifier for further access to the exported objects (if these cannot be resolved uniquely). This identifier does not necessarily have to be the same as the module identifier.

The **package path**, in turn, is used to logically structure several modules and is prepended as a directory chain when searching for the module.

5.3.3 Use Path, File Names and Name Restrictions

The IDM internally converts a Use Path into a file path to be used for searching the interface file, source code file or binary module.

Thus the Use Path `Modul78.Aa.MasterA__1z` becomes the file path *Modul78\Aa\MasterA__1z*.

In addition, there is a predefined set of file extensions that are relevant for search and access and are internally appended to the base file path:

File Extension	Meaning
.dlg	source code of a dialog
.mod	source code of a module
.if	interface file
.bin	binary file of a dialog or module

Since the Use Path is built from IDM identifiers, this of course means file and directory names may begin with uppercase letters as well as upper and lower case has a meaning and is important! This has to be observed in case of working on file systems that do not distinguish these.

In the IDM this means: `use Default;` is something else than `use DEFAULT;`. On Windows, however, the module may be found if a module file “DEFault.Mod” exists.

In principle, the dialog can also be accessed (loaded) via a Use Path.

If there are unresolvable problems with the conversion between Use Path and file path, the internal conversion behavior can be controlled with the **-IDMusepathmodifier** option.

5.3.4 Search Path for Interface, Module, Dialog, and Binary Files

The IDM possesses a universal search path for finding IDM files. This search path may contain one or more absolute or relative directory specifications separated by semicolons (“;”).

By default, the search path is set to “~;”, thus to the special path “~” and the empty path “”. These have the following meaning:

~ or ~:	Search beneath the directory in which the application is located.
"" (empty path)	Search in the current working directory (same behavior as in previous versions).

<ENVNAME>:	Search in the paths that are defined in the environment variable `<ENVNAME>`.

When searching for an IDM file, e.g. via **DM_LoadDialog()** or the built-in function **load()**, proceeding is like this:

1. If the given path is absolute, the file will be searched for at the exactly given path without appending an extension (compatible behavior as before).
2. In case of a relative path with file extension, this will be searched for in the search path with exactly that extension. The default entry *""*(empty path) shows the compatible behavior as before.
3. If the file name does not have an extension, the possible extensions are tried for all search paths, depending on the action. For example, **DM_LoadDialog()** tries the extensions *.dlg*, *.mod*, *.bin*. For a “use” it is attempted with priority to get the interface, so the sequence *.if*, *.bin*, *.mod* is tried.

These are the available possibilities for setting the search path:

- With the option **-IDMsearchpath <search path>** when starting an IDM application.
- Through the environment variable `IDM_SEARCHPATH`.
- By calling **DM_ControlEx()** with *DMF_SetSearchPath* and the search path as *data* argument.
- Via the *.searchpath* attribute on the ***setup*** object.

For a self-built IDM application it is recommended to use **DM_ControlEx()** in the **AppMain()** routine with an application-relative path using “~” and without other relative specifications like the empty path or *"."*. So the invocation of *DM_ControlEx((DM_ID)0, DMF_SetSearchPath, "~:dlg")* would ensure that the dialog files are only searched in the subdirectory *dlg* parallel to the application. Once the search path is set, initial loading of the main dialog may also take place via **DM_LoadDialog()** using a relative file path without file extension.

Examples of Correct and Permitted Search Paths

"" "."	Searches in the current working directory.
"~"	Searches in the directory where the application is located.
"if;bin;mods;customer/modules"	
"~:dlg;~:../mods;."	
".;MODPATH:if;MODPATH:bin"	Searches in the working directory as well as in the *bin* and *if* directories within the directories defined in the environment variable `MODPATH`.

When opening and loading IDM files, the IDM attempts to determine the file type to recognize whether the file is an interface file, dialog, module, or binary file. For this purpose, the first 1,024 bytes of the file are examined , whether they contain the signature of an IDM binary file or the appropriate keywords identifying an interface, dialog, or module. Otherwise, the file extension is used for determination.

5.4 Comparison Between import and use

Function, Concept	**use**	**import**
Multiple loading of modules.	**Not** possible. Module is only loaded once in a dialog (applies only to access with “use”).	Multiple loading of the same module possible through a different import identifier or through gaps in the import hierarchy of the importing modules up to the dialog.
Linkage of functions to an application object.	Possible with the *.masterapplication[enum]* attribute.	Possible with the *.application* attribute.
Creation of interface and binary files in one step.	Possible with **-compile** or **-recompile**. The dialog should access all modules by “use” however.	**Not** possible.
Source code, interface and binary module in the same directory.	**Yes**, possible due to the different file extensions.	**Not** possible.
Distribution of the modules into subdirectories.	**Yes**, via package path (part of the Use Path) or via the search path.	**Yes**, through search symbol, path lists in search variables and as listing in interface and module file.

5.5 Interface and Binary Files when Using import

5.5.1 From the Module to the Interface

If a programmer wants to use the functionality of a module in his dialog (or module), he has to know the names of the exported objects included in it.

To simplify this, the Dialog Manager simulator is able to generate a so-called "interface file" from a module´s description. Thus, the names of the objects exported in the module are given to the user of the module. The user has access to these objects by using the names. The actual realization of these objects is not revealed to the user. Therefore the module designer is able to arbitrarily change the real implementation of his objects, as long as the interface file - the interface outwardly known!- is not changed.

The actual objects are not linked with Dialog Manager until runtime or generating the binary file(s), so that the shaping of the objects is known only from this time on.

With the starting option

```
+writeexport
```

the developer can automatically generate the interface file from his module. Comments with "!!" in front of the actual objects are also transferred into the interface file. Thus, comments of the dialog sources can also be made available for the user of a module.

The syntax of this call is as follows:

```
idm +writeexport <export file name> <module name>
```

Example (file "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 );
```

In order to generate the interface file, the simulator is called by the **+writeexport**:

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

After having executed the command **+writeexport**, the generated file has the following structure:

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

The user can get the necessary information of the module "color.mod" from the file "color.if" without knowing how the colors were being defined. This method e.g. allows company-wide standardized color libraries.

The children hierarchy is also maintained, as can be seen in the following module extract.

Extract of a module with a window model:

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

Extract of its interface file:

```
...
model window W
```

```
{
  child pushbutton P;
}
...
```

5.5.2 From the Interface to the Module

The file names of the interface files have to be unambiguous, otherwise subsequently generated files will overwrite previous interface files.

The files are searched for in the actual directory. If they are not found, Dialog Manager interrupts with an error message. To manage the modules themselves independently of the actual directory, there are search symbols that can be given when generating interface files. Maybe you are familiar with these search symbols from the functions for loading dialogs or profiles and from the localization of images. The symbols are - separated by a colon - put in front of the the actual file name. Dialog Manager interprets these symbols as environment variables and their contents as search paths for the given file. The environment variables have to consist of at least two characters when using operating systems which distinguish single drives with characters followed by a colon (e.g. MICROSOFT WINDOWS).

Then the module is searched in the path that is defined by such environment variables. If the relevant file is found in a directory, it will be loaded and the search will be stopped.

Example

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

To make sure that the environment variable is considered on loading, indicate the following option when generating the interface file:

```
+searchsymbol IDMLIB
```

In doing so, all file names are marked with the symbol IDMLIB.

The general syntax is as follows:

```
idm +writeexport <export file name> +searchsymbol
  <environment variable> <module name>
```

Example

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

The result (interface file "color.if") for the above module example Color is as follows:

```
interface of module Color "IDMLIB:color.mod"
color Red;
color Green;
color Blue;
color Yellow;
```

The Dialog Manager will look in the directories indicated in the search path of the environment variable after the module "color.mod".

5.5.3 Import Modules in Modules

The keyword

import

signals the use of a module in another module or in a dialog. The keyword is followed by the module´s logical name under which it is to be known. In general, the logical name is not the name of the module itself. You may also use one and the same module under different logical names, although this can be better described by models. The imported module´s logical name is followed by a string with the name of the interface file.

Example From a Module

```
module Models

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

export model pushbutton MPB1
{
  .fgc Green;   // color from StandardColors
  .bgc MyGreen; // color from Colors
}
export model pushbutton MPB2
{
  // Double names can be distinguished by putting the
  // module name in front of it
  .bgc StandardColors.Red;
  .fgc Colors.Red;
}
```

Modules importing modules can be re-imported from other modules, too. Thus, you will get a kind of tree with the dialog itself as its root.

The imports of a module can be exported. Thus, modules importing this module with the exported imports do also have these imports.

```
!! Interfacefile: module.if
module Module
export import StandardColor "color.if";
...
module Main
/*
  import StandardColor "color.if";
  This module does not have to be imported here, since it
  is indirectly loaded via the use of the color module
*/
import ImportModule "module.if";
...
```

If the interface files are not to be in the same directory like the module, the file names can be marked with a search symbol on the import. This search symbol represents an environment variable containing a search path (see also section above).

Example of a module which is to search its interface files in a different directory. The environment variables are set on the following values.

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

and

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

```
module Models

import StandardColor "IDMLIB:resrc/color.if";
  // The file color.if is searched for in the following directories:
  // /usr/idm/lib/resrc
  // /usr/idm/ISAstandard/resrc
  // /usr/idm/lib/dia/resrc
import Colors "PRIVLIB:mycolor.if";
  // The file mycolor.if is only searched for in the directory
  // /home/project/lib

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

5.5.4 Use of the Object – use

Objects in modules cannot only be accessed or assigned to attributes, but can also be used as children of objects from other modules. This is signaled by the keyword

use

The attributes of this child cannot be directly changed any more when using the object. But the change of attributes is still possible in the rules.

Example

```
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
}
...
```

A child can be used only **once**. This means that objects can be inserted by the keyword **use** in the existing object tree at one position at the most. If objects are to be used several times, models have to be used as it was usually done before the modularization was available.

Note

Models are generally the better choice, but there are exceptions where objects that are defined with **use** are an advantage: e.g. window as such and a tablefield object.

5.5.5 Binary Files

The developer has dialogs and modules available as ASCII files. The end-user receives only the binary-coded form of modules. The runtime version can only read these binary-coded modules, i.e. the binary files.

The modules are individually translated into binary files. The translation is made with the Dialog Manager option **+writebin**:

```
idm +writebin<binary file name> <dialog file name>
```

The binary file is an identical translation of the module into a machine-readable form. Therefore, the dialog loading process is considerably accelerated.

Binary files and modules in ASCII format may absolutely be mixed. If an interface file, however, changes, the dependent modules have to be re-translated into the binary form. If the models or defaults in modules change, the dependent modules also have to be re-translated into the binary form. Therefore such module files can be compared with C-header-files: if a C-header-file changes, all files dependent on it have to be re-translated to get a uniform version again. The same applies to the binary files of the modules.

Recommendation

The above mentioned requirements can best be described and automatically included by so-called "makefiles" or similar instruments of software development.

5.6 Compiling Interface and Binary Files for Imports with use

During the development of an IDM application and the usage with source code files, the interface files serve as a representation of the externally available (exported) objects of a module, i.e. the external interface. By separating interface definition and implementation, the IDM is able to load the implementation (that is, the module) only when needed.

With “use”, this interface definition can also originate directly from the module, i.e. the implementation. For the reason mentioned above it still makes sense to work with interface files though.

For the distribution of an IDM application to the end customer, a binary version of the dialogs and modules is required.

Basically, interface and binary files can be generated the usual way via the commands **-writeexport** and **-writebin**. As long as these files with the correct file extension are in the right place within the search path, there is no problem with the “use” statement.

However, for modularized dialogs that employ “use”, there are much simpler commands to create all interface and binary files in one step.

-compile	Updates all interface and binary files for modified modules and dialogs.
-recompile	Recreates all interface and binary files.
-cleancompile	Deletes all interface and binary files.

For this the dialog needs to be loaded without errors. Interface and binary files are only created for the specified dialog and all modules loaded per “use”. **No** interface and binary files are generated for modules that are accessed via “import”!

The module and dialog files to be loaded must be available as source code. In case of a loading error, no files will be created.

Example

```
!! 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";
```

With the following command all interface and binary files can be created in one step:

```
pidm -compile Main
```

Now the following files should lie parallel to the source files:

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

Now if the line *export font FnSmall…* is added to the file *Base/Fonts.mod*, simply the following command is used to update:

```
pidm -compile Main
```

This causes the following files to be refreshed:

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

The other interface and binary files are not rewritten because the file date of these files is newer than that of the corresponding source file.

If all interface and binary files should be recreated in any case, this can be achieved with:

```
pidm -recompile Main
```

If all interface and binary files should be deleted, this can be done with:

```
pidm -cleancompile Main
```

To create the interface and binary files in a different directory, the options **-ifdir <directory path>** and **-bindir <directory path>** may be used.

5.7 Dynamic Module Administration

5.7.1 When Using import

5.7.1.1 Loading Process

A module can be loaded into another module in different ways. Loading means that not only the objects´ names are known to the module, but also their definitions. Thus, they are displayable and changeable. There are three kinds of loading which partly depend on the objects defined and exported in the module.

- load on use
  The module is automatically loaded. The programmer cannot determine the time the module is loaded, since some objects of the module are immediately needed (modules and defaults) when reading the superordinate module. This is the case when the module includes exported necessary modules or resources.
- implicit load
  The module is loaded only when an exported object of the module is really needed, e.g. if the object is accessed in a rule by assignment or request.
- explicit load
  Here, the dialog designer himself decides that he now wants to have the module loaded. He sets the attribute *.load* on true at the logical module name (import name). Then the Dialog Manager loads the module and makes it available.

Example

```
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 has priority to the two other kinds of loading. **implicit load** has priority to **explicit load**. In the example above, **load on use** is carried out, the other two loads do not have any effect since the module is already in the memory and the rules can be executed directly.

Explicit load can be given as an attribute already at the import.

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

To avoid that the module has to be re-loaded in the memory on every single import, a module should be usually shared with other imports. In exceptional cases, which should be rare and well justified, the design can be revised. As already mentioned above, the import has a logical name. If a subordinate module (that means a module nearer to the dialog in the import hierarchy) has an import with the same logical name as a subordinate module, the module is loaded only once.

```
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";
...
```

In the example above the module “color” is loaded only once with its logical name “Colors”. In the example below, however, is unnecessarily loaded twice: by its logical name “Colors” and by “Colros”. Note the undesirable effects of this (intentional) typing error!

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

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

It may also happen that the module is unintentionally loaded twice due to the sequence of the import processes. If - like in the following example - the module M1 is loaded immediately (**load on use**), the import of the colors of M1 may be read before (!) the colors of M2. Since M2 is also loading M1, the color import at that time is known only to M1, and thus the module Color is loaded twice.

```
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";
```

```
...
```

Note

Do not use **start()** on modules and their logical names!

5.7.1.2 Unloading Process

A module not used any more can be removed again from the memory by its import (logical name) only by using the function *.load* in the import. The attribute value is set on *false*.

Note that not every module can be unloaded again. Generally this is possible only when the loaded module contains only exported objects, but no exported resources, models or defaults. The module is not physically removed from the memory until each import with the relevant name of this module is unloaded.

A module cannot be unloaded any more, if the attribute

```
.static true;
```

is set at the object ***import***. This attribute also has effects on the generation of binary files. The file drawn by *.static true* is copied into the calling file during binary writing.

The unloading process takes a lot of computing time and should therefore only be used when really needed, e.g. in case of memory problems.

Note

The function **stop()** cannot be used on a module.

5.7.2 When Using use

There are two methods for the dynamic use of modules:

- **:use()**
  To make a module accessible and load it if necessary. This is equivalent to the `use` statement in a dialog or module.
- **:unuse()**
  Serves to remove the `use` relation, that is to “unload” the module, with the module not being removed completely until there are no further `use` relations (e.g. from other importing modules).

Example

```
!! file DynLoad.dlg
dialog Dlg

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

    on select
    {
      this.dialog:unuse("DynMod");  // Unload module DynMod
      this.sensitive := false;
    }
  }

  on close { exit(); }
}

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

  Module := this:use("DynMod");  // Load module 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 Object Application

The object ***application*** is used for binding clients to a server. This can be done with modules, too. The problem, however, is that one module can be used in different dialogs. It may happen that these dialogs are assigned to different server processes. Therefore, when using a module, you can define where its function is to be searched. The object ***import*** passes this information on to the module. There you can define which application object is responsible for the functions in the module.

Example

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

Each function of the module Modul_X is now called on the server. In doing so, the module which is described in the interface file "modulX.if" can be programmed regardless of the corresponding server.

Furthermore, functions can be exported out of modules without having to export the attached applic-ation. This is the only case that an object’s parent does not have to be exported with. The advantage is that the designer of the module always knows on which computer his functions are running. On the other hand, however, it is not possible any more to make changes from outside.

Example

```
module Modul

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

The function "Calculation" can be called from the importing module without the module knowing of the application "MyServer". These two mechanisms enable the programmer to implement independently usable modules.

5.8.1 Application assignment of module functions

A special case for the usage of functions in modularized dialogs is the use of the .application attribute at the ***import*** object to couple all functions of a module from "outside" to a special application. This procedure is described in the chapter “Programming Techniques” / “Modularization” / „Object Application“.

However, this procedure is not available when importing modules via **use**. As an alternative or supplement, however, the linking of functions via the .masterapplication attribute to the module or dialog can be used. This means that functions defined directly under a module/ dialog can be assigned to an application. If there is no assignment on the module, the assignment from the dialog is used. The attribute can also be indexed with an enum value. This index value should be from the range *lang_default...lang_java* or *func_normal..func_data*. This allows functions to be assigned to different applications according to their language or function type. An explicit assignment (even a null setting) at the module always overwrites/overlaps the assignment of the dialog.

Example

Modularized dialog with the functions in the module “Functions.mod”, where COBOL functions are attached to the server application “ApplServer” and all other functions to the “dynlib” application “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;
}
```

See also

Attribute .masterapplication[enum]

5.9 Application Examples for the Modularization

As described in the introduction, there is the possibility to develop uniform standards and to store those in central modules (see following chapters on Resource and model basis). Moreover there is the possibility to construct fundamental dialogs and to extend or modify them via modules (see chapter on Differenciated dialogs). Applications which can be subdivided into several parts can be distributed by modules and may be loaded dynamically only on runtime (see chapter on Dividable dialogs). By using modularized dialogs prototypes can be easily built and be introduced to the customer. Testing procedures can also be realized more easily by modules (see chapter on Prototyping & Testing).

5.9.1 Resource Basis

Uniform standards support the usability, userfriendliness but also the recognizability of software products. Uniform colors, characters, texts, formats and other resources can be combined to one or several modules and can be made centrally available for the individual development departments. Standardized resources facilitate the administration in so far as changes or extensions can be realized more easily.

Example for a module containing text resources:

```
module TextModul

!! Name of company in shortened form
export text FirmaKurz  "ISA GmbH";
!! Name of company
export text FirmaLang  "ISA Informationssysteme GmbH";
!! Text for pushbuttons
export text EndeText   "Exit"
{
  1:  "Exit";
  // other languages
}
```

Other applications for standard functions are possible.

Example

```
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);
```

Other, even more detailed examples are possible here.

5.9.2 Model Basis

Similar to the resource basis, defaults and models can be defined, administered and stored centrally in modules. With these modules complete libraries can be built which guarantee a uniform design of the entire product range. In model libraries, the design and the accompanying rules or whole object groups (e.g. windows with certain menues) can be standardized and reused for various projects. Especially for the graphical user interface the model basis provides an excellent possibility to reuse object groups.

Example for a default basis ("default.mod"):

```
!! Central defaults for all models and instances
module Defaults
```

```
export default window  // No name is necessary for defaults
{
  .visible false
  ...  // other attributes
  object CloseRule := null;// The user-defined attribute
            // should contain a rule which is
            // called on closing
}
on WINDOW close
{
  if ( CloseRule <> null )
  then
    this.CloseRule( this );
  endif
}
...
```

An application ("model.mod") of the default basis could look like that:

```
module Models

import DefaultBase "MODLIB:default.if";
export model window MMainWin
{
  .CloseRule := MainWinClose;
  child menubox System
  {
    .title "System";
    ...
  }
}
!! This rule need not be exported!
rule void MainWinClose( object ThisObj input )
{
  exit;
}
...
```

With the example above, we have established the initial stage of an expandable model basis. The application programmer now can be provided with a wide range of extended standard objects along with their corresponding functionality. The functionality, however, have to be completed by further product-specific functionality.

Example

```
dialog Main

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

```
MMainWin MainWindow
{
  .System.title "File";
    // Changing the presetting of the model
  child groupbox
  {
    // product-specific objects ...
  }
}
on dialog start
{
  // product-specific rules
}
```

5.9.3 Exchangeable Parts of an Application

In order to adapt an existing specific branch application for a different, but similar branch, often only parts of a dialog have to be exchanged. If these exchangeable parts are known from the start (see also chapter on Prototyping), the dialog can be designed by means of the modularization in such a way that only single modules have to be exchanged to receive a "new" application. Customer-specific desires can be considered with regard to design and functionality in the same way.

Example of an Administration Program for a Gas Station

Even for different companies, the administration of gas stations will almost be similar. Stock administration, cashbook and buying will all be identical. As for matters of invoice and orders, however, there certainly are differences which would make a global solution hard to handle and which would require many awkward adjustments. The developer can write a standard program covering the general tasks of an owner of a gas station, whereas for company-specific differences he should develop exchangeable modules (meaning the same interface). These exchangeable modules, in turn, are normally based on a standardized model basis.

If exchangeable modules are used, the time needed for the module administration should be in a reasonable relation to the saving of time which is needed for the development of these modules.

5.9.4 Dividable Applications

Many applications can be subdivided into various part applications. Some of these applications are always used, others are rarely or never used by certain users. Those rarely or never used parts of an application do not have to occupy the resources of a computer. The modularization makes it possible to describe these part applications in modules by not holding them permanently in the memory. In doing so, resources can be saved because parts which are not used will not be stored in the memory. Moreover the period needed for the entire dialog to be loaded into the memory will be reduced since only the main part or the permanently used parts of the dialog are loaded. The parts used only rarely can be loaded during runtime of the dialog. Thus, the complete loading time can be divided up into several time periods and the user can save unnecessary waiting time at the beginning of the program.

Some dialog parts are needed only for a short time and afterward are not needed any more for a long time. Thus, it is recommended to define these dialog parts in modules. These modules can be loaded into the memory when required, can be carried out, and - if wanted - can be unloaded from the memory again.

(Note: The released memory stays with the program in today´s operating systems, and can be used only by this program, but also by other re-loaded modules.)

Examples for this can be found in almost every application.

5.9.5 Prototyping & Testing

At the beginning of many software projects, there is the task to present the customer a prototype. This prototype does not include the complete functionality of the later product; however, it contains enough for customers and manufacturers to be able to remove misunderstandings and uncertainties before the actual development work. Very often, a prototype allows at the beginning to evaluate the possibilities and usability of a program without losing precious time and work which will again be missing at the project’s end.

Since, for a prototype developed with the Dialog Manager, the user interface and its functionality is central, the topic of prototyping will be especially focussed in the following. From the Dialog Manager´s point of view, the developing program can be divided up roughly in two parts: the user interface with its functionality and the actual application. With the Dialog Manager and its feature modularization, a re-usable prototype can be generated which does work without the actual application. Thus, a prototype can be rapidly presented to the customer.

The interface between Dialog Manager and application is established via functions. These functions can often be substituted by named rules. The developer can substitute these functions by rules, however, he has to make modifications in the actual dialog each time. Having different modules which include the actual functions' definitions and the identically named rules, a simulation of the prototype can be easily generated. However, here a modification of the relevant dialog is not necessary.

The following example goes into the objects and attributes which shall help your understanding.

Examples

```
dialog Main

import Functions "PROTOLIB:func.if";
variable boolean OnLine := false;
on dialog start
{
  OnLine := CheckOnline();
  if ( not OnLine )
  then
    // a message that the program only runs online
    exit;
  endif
  // Program may run
}
...
```

The module func.mod is defined as follows:

```
module FunctionModule

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

The dialog in this example would have to be modified in the way that the dialog's start rule is changed and the program does not abort immediately. This change is not necessary any more if this module is substituted by a module with identically named rules.

The module "rulefunc.mod" then is defined as follows:

```
module FunctionRuleModule

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

In order not to change anything at all at the dialog, an interface file has to be generated out of the module "rulefunc.mod" which has the same name as the one of the module "func.mod" ("func.if"). There is an environment variable (here: PROTPLIB) in order not to newly generate the interface files each time when switching between prototype and progressing developing work. You can file the interface files in different directories with this environment variable.

This scheme can be extended even more using Dialog Manager, in the way that the user can give information about the reactions of the actual application during runtime.

Extension of the module "rulefunc.mod" (request):

```
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 );
 }
 ...
```

This messagebox is not included in the later module (here: "func.mod"). It does only serve the direct navigation of the dialog run, since this is navigated by the actual application which - in this case - is not yet available.

A further advantage of this method is that you can also test dialogs specifically: the developer or quality assurance can also simulate unlikely test cases without having to manipulate the application. When testing, the above example could provide problems when the application is really switched offline. Thus, the part of the dialog which deals with the offline operation, could be tested only awkwardly.

Note: There are also optional interfaces for Dialog Manager (to different databases, transaction monitors and the Shell Interface for Unix).

5.10 Example

You will find the following example in the installation directory under "modular". There you can call "make" or "nmake" according to the environment. In doing so, you will get an executable program.

In this example the modules are intentionally kept small in order to illustrate the use of the modularization.

5.10.1 The Default Module

All default objects are defined in this module. It is then imported into all other modules in order to have access to the same default values. In this module, all objects are exported.

```
!! 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 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 The Module for Pushbutton Models

In this module, a model for a pushbutton is defined. Then a rule is deposited for this model. The module itself needs the default module.

```
!! 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 model above, 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 Further Modules

Further modules were defined for this example. Please have a look at the example directory.

5.10.4 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;
  }
}
```

The starting window of this application has the following structure:

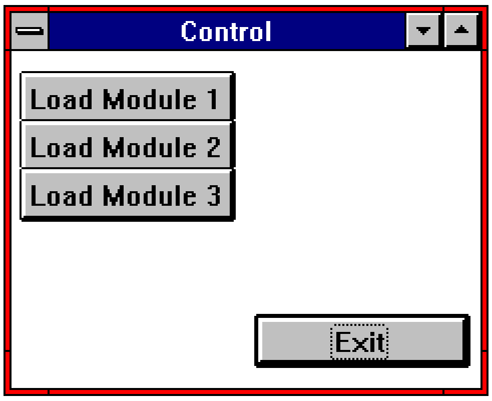


***Figure 18:** Starting window of a modularized application*

By selecting the pushbuttons the individual modules are loaded. You will get the following structure:

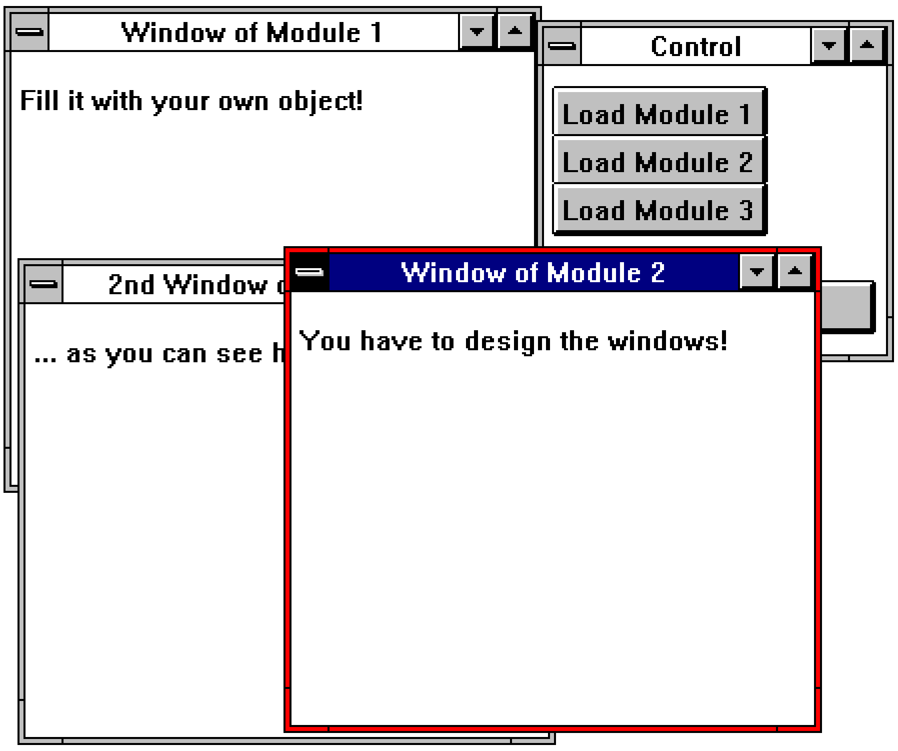


Figure 19: Application with loaded modules

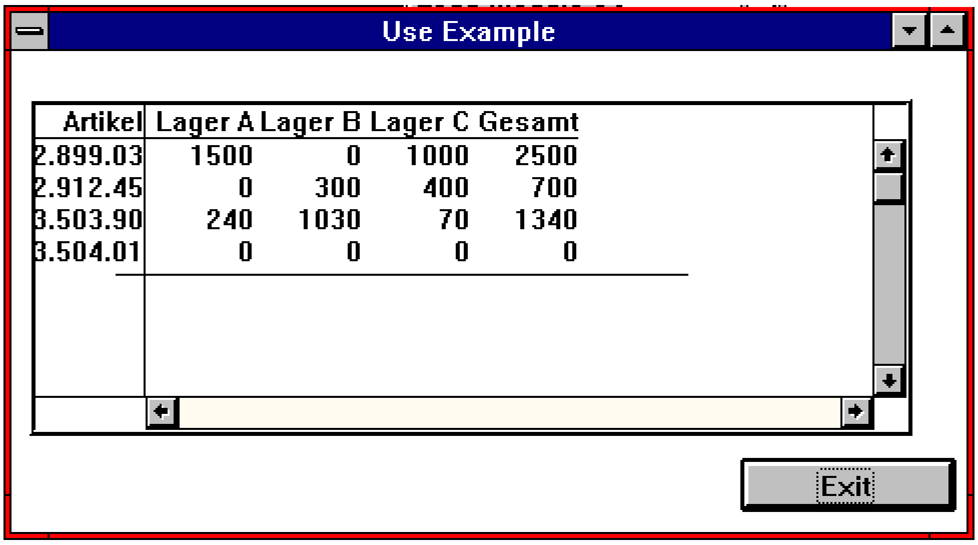
5.10.5 Example for USE Operator

The example "use.dlg" can be used for the USE operator. Here, a tablefield is passed on to a window using the USE operators.

```
!! 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;
}
```

After having started the application the window has the following structure:



Artikel	Lager A	Lager B	Lager C	Gesamt
2.899.03	1500	0	1000	2500
2.912.45	0	300	400	700
3.503.90	240	1030	70	1340
3.504.01	0	0	0	0

Figure 20: Use of USE operator

5.11 Structure of a Development Environment

If you want to establish an development environment using the modularization, the following aspects must be considered:

» The interface files always have to be updated in accordance with the module files, otherwise the system will be stopped on loading the relevant module.

» The binary files of the modules usually have the same ending as the corresponding ASCII files of the modules. This is the case, since the names of the implementation files are deposited in the interface files, and it is not differentiated if the file will contain data in ASCII or in binary form.

» Each developer should first be able to carry out and test the modifications in his private envir-onment, before he makes his module available for his colleagues. Also far-reaching modifications can thus be at first tested locally and are then made available to the whole project team.

» When establishing a module, note that modules must not import themselves recursively (neither directly over several steps!). The recursivity can be recognized very easily by deleting all interface files and having them made again. If a recursivity has been installed by mistake, the interface files cannot be generated. In this case, the modules have to be re-structured.

Considering these aspects, an development environment can have the following structure making use of the modularization:

First of all a project directory is created where all released modules and dialogs with their corresponding interface files are deposited.

In addition a parallel directory is opened into which the binary versions of modules are stored. The binary files are then automatically created whenever there is a change in the underlying module or in the interface files of the used modules.

Each member of the project gets the same environment for himself as defined in the project directory. No data is deposited there at first. If there are changes at the module to be carried out, the file is copied from the project directory into the private directory and is modified there until it is usable. The corresponding interface file is also deposited in the private directory, of course. When the modifications are tested, the file is returned to the project directory and deleted in the private directory. Thus, all members of the project have access to the modified form of this module.

If, due to performance reasons, binary files are created privately, this must be carried out by the same structure as in the project directory.

The selection of the current source on the program run is controlled by the environment variable which was indicated on generating the interface files. Without the variable the behavior cannot be controlled in a reasonable way. A reasonable behavior is achieved when the environment variable for the ASCII version is set on

Private directory;project directory.

In doing so, each module file is searched in the private directory first. Only if no module file is found there, it is removed from the project directory. The same applies to the binary version. Mixing ASCII form and binary form can have fatal consequences, if the dependence between modules in the development environment has been recorded and rendered in the wrong way. Therefore, you should either work on binary files or on ASCII files.

The following figure illustrates a possible module structure. In this example, resources are directly used in the module "application part 1", whereas in the module "application part 2" the resources are not directly used. This is why the module "resources" is imported into the module "application part 1", but not into the module "application part 2".

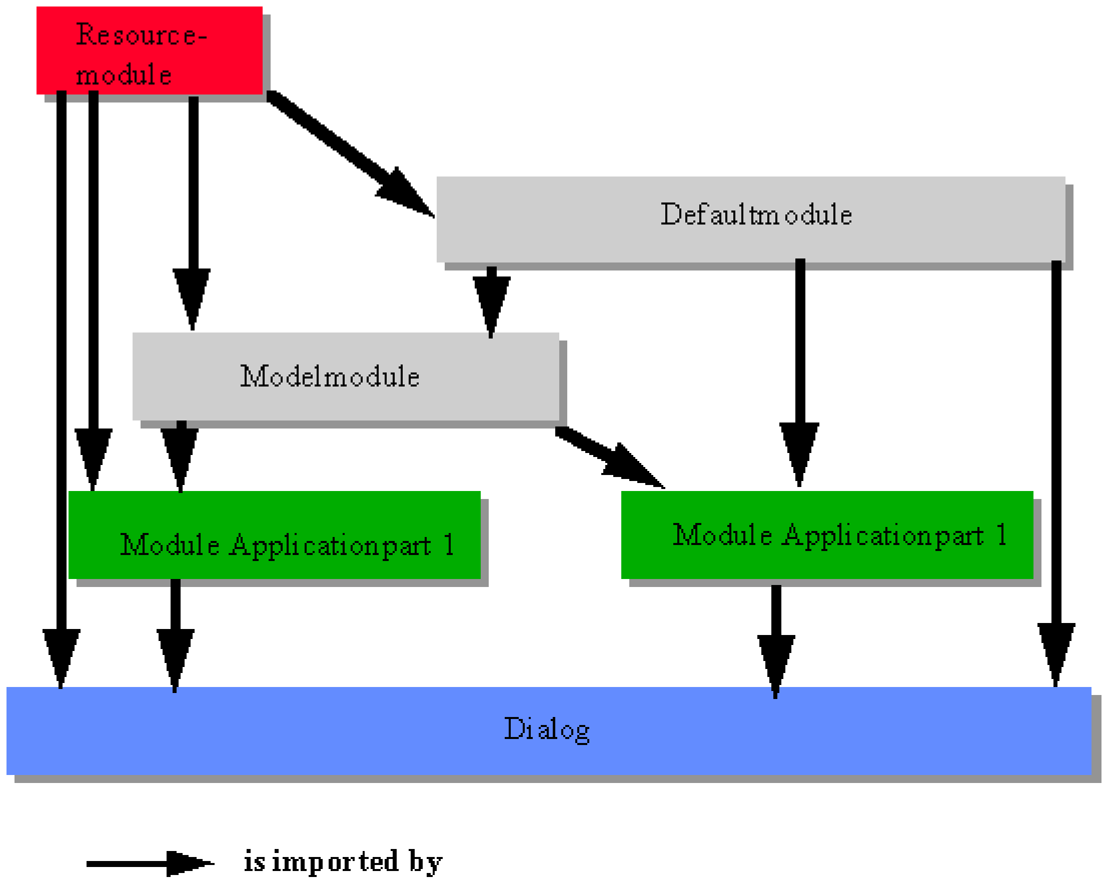


***Figure 21:** Possible module structure*

By using these conditions as basis, a reasonable project structure can be as follows:

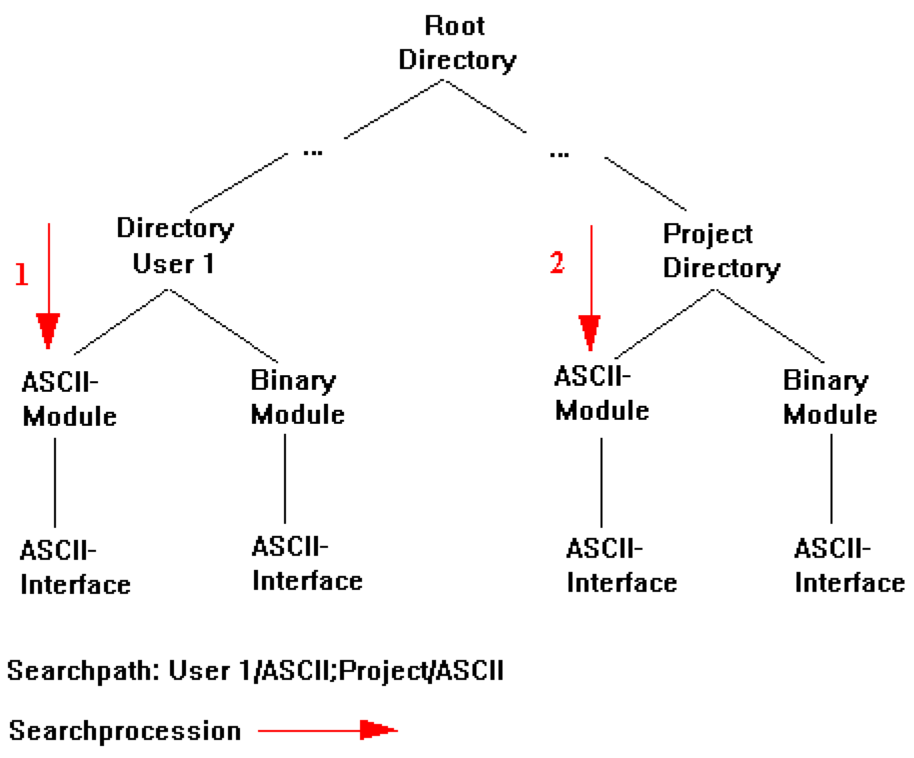


***Figure 22:** Usable project structure*

For this structure, it is necessary that each user has access to the project directories. On MICROSOFT WINDOWS computers, these directories can be on different logical disks, of course. This disk, however, must be available to all users.

In addition, we recommend to use source-code control systems, since they avoid the parallel changing at one and the same source file or reunite the parallel procedures.

6 Datamodel

The **motivation** for the “Datamodel” in the IDM is to achieve an improved separation between user interface objects and the application layer, which provides the data for the representation.

The main characteristics of its design are:

- Separation of the user interface from the application according to the Model-View-Presenter (MVP) design pattern.
- The dialog developer merely defines the linking between user interface objects (View) and the data storage in the application layer (Model).
- As little as possible or even no rule code due to the controllable application of automatic synchronization and automatic type conversions.
- Retention of the familiar data schema and access concept for indexed attributes so that data may be used in scalar or vector form.
- Use of the existing IDM features for data storage and implementation of the Model layer: IDM objects (***record*** objects) and application functions, which as well may be fully transparently located on the DDM server side.

6.1 Introduction

With the Datamodel, the IDM allows a separation into the three components Model[1], View and Presenter according to the MVP design pattern with adaptation to the peculiarities and requirements of the Rule Language.

Model

Manages the data and its underlying logic. Is typically close to or in the application layer. A model can manage several data, which are considered as indexed attributes by the IDM and thus may contain scalars, vectors and matrices.

For instance, a ***record*** object with user-defined attributes and methods.

View

Visual representation of the data for display, input or modification. The data schema is the same as for the model: indexed attributes thus containing scalars, vectors or matrices.

For instance, an ***edittext*** object using a format for input and output.

Presenter

The logic to control the connection between Model and View or to link multiple Views.

[1]Not to be confused with the IDM models in the sense of classes for object instances.

For instance in the form of event rules to assign the new data value at the View to the Model when editing is finished.

Brief notice on terms: In this documentation “Datamodel” (in one word) is used for the concept derived from MVP whereas “Data Model” (in two words) stands for an actual Model.

In contrast to existing MVP concepts in JAVA or QT, the IDM approach allows for transparent handling without restricting the linking and without the need to differentiate between content or selection. Whether a Model controls the content of an edittext, the color of a certain table cell, or the active item in a list is solely left up to the application or dialog programmer.

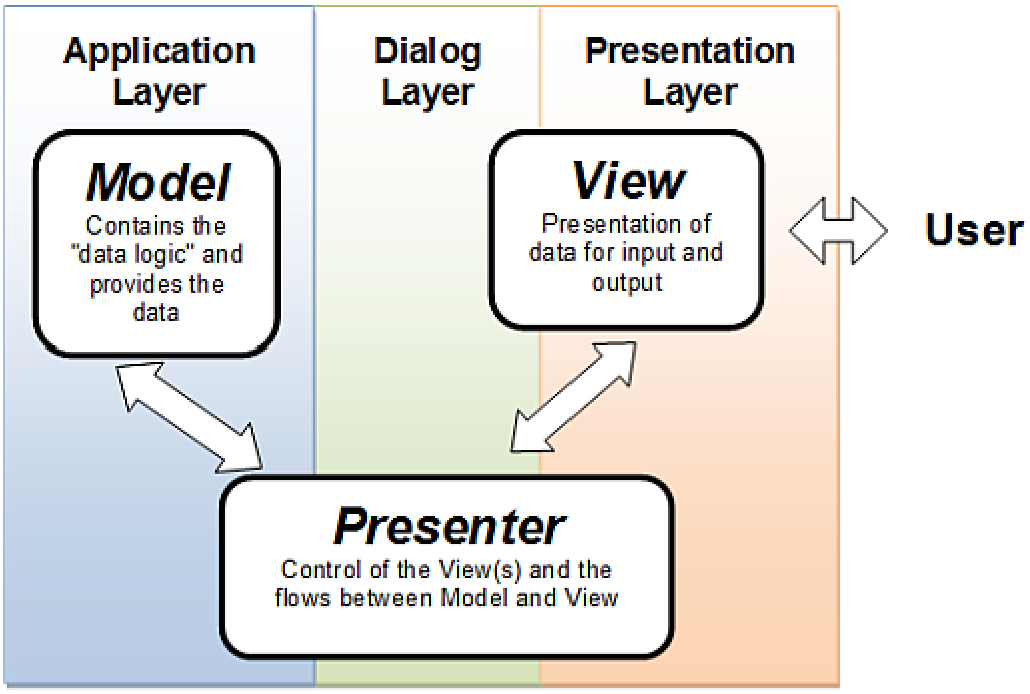

Application Layer
Dialog Layer
Presentation Layer
Model
Contains the "data logic" and provides the data
View
Presentation of data for input and output
User
Presenter
Control of the View(s) and the flows between Model and View

Figure 23: Architectural pattern Model-View-Presenter

In the IDM, the linking between Model and View as well as the most important control functions of the Presenter are defined by a few attributes:

Table 2: Datamodel attributes

Attribute	Meaning
.datamodel	Reference to the Model at the View
.dataget	Defines the linking of a View attribute with a Model attribute for display
.dataset	Defines the linking of a View attribute with a Model attribute for assignment
.dataindex	Provides control over the aggregation of View and Model attributes

Attribute	Meaning
.datamap	Enables the mapping of attributes to resolve “ambiguities”
.dataoptions	Control options for the Presenter logic

An **example** for linking a dialog interface (View) with a Data Model for the display of personal information with name, gender and child names might look like this:

```
dialog D
record RecPerson {
  string  Name   := "Jane Q. Public";
  boolean Female := true;
  string Children[integer];
  .Children[1] := "Amy";
  .Children[2] := "Peter";
}
window Wi {
  .title "Datamodel Example RecPerson";
  .width 200;
  .height 200;
  /* Linking of all View objects to the Model RecPerson */
  .datamodel RecPerson;

  edittext EtName
  {
    .xauto 0;
    /* .content is linked to RecPerson.name */
    .dataget .Name;
  }

  checkbox CbGender
  {
    .ytop 30;
    .text "Female";
    /* .active is filled from RecPerson.Female */
    .dataget .Female;
  }

  listbox LbChildren
  {
    .ytop 60;
    .xauto 0;  .yauto 0;
    /* .content[] is filled from RecPerson.Children[] */
    .dataget .Children;
  }

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

The Datamodel approach in IDM takes some of the Presenter logic off the shoulders of the application programmer, hence the above example works without any further rules. The filling of the interface objects “EtName”, “CbGender” and “LbChildren” is done by the IDM through the linking set by the attribute definitions of *.datamodel* and *.dataget*. Furthermore, the above example takes advantage of the additional value passing feature from parent to child, which applies to the attributes *.datamodel* and *.dataoptions*.

The following diagram is intended to illustrate these automatisms and to show what controlling and influencing options the dialog programmer has for the components View and Presenter as well as the application programmer for the Model component.

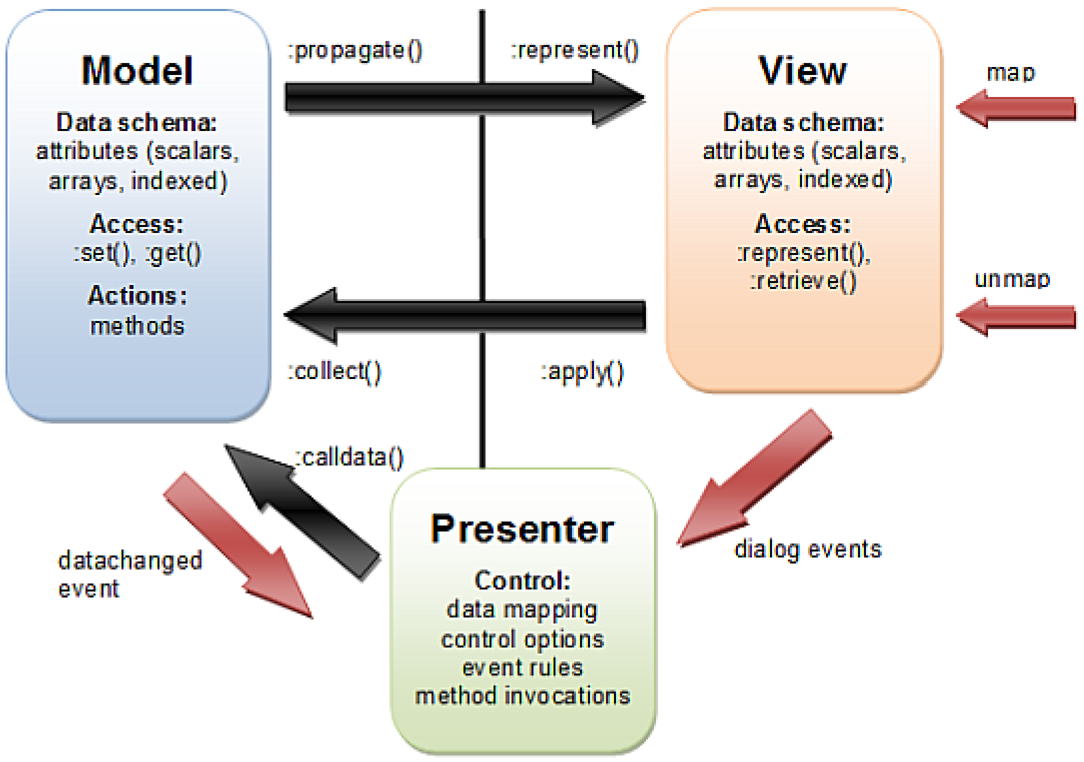

Model
Data schema: attributes (scalars, arrays, indexed)
Access: :set(), :get()
Actions: methods
:propagate()
:represent()
View
Data schema: attributes (scalars, arrays, indexed)
Access: :represent(), :retrieve()
map
unmap
:collect()
:apply()
:calldata()
datachanged event
Presenter
Control: data mapping control options event rules method invocations
dialog events

Figure 24: *Linking between Model, View and Presenter*

Table 3: *Propagation of changes between Model, View and Presenter*

Method, Function	Component	
:represent()	View	The View object is populated with the data from the Model and, if necessary, data values are pre-pared for display
:apply()	View	The View object returns the data to the Model; if necessary, data values are prepared for assign-ment

Method, Function	Component	
Dialog event	View, Presenter	Changes of the View are signaled
:propagate()	Model	The data from the Model component is relayed to the linked View objects for display
:collect()	Model	The Model component retrieves all values from the linked View objects
DM_DataChanged() or attribute assignment	Model	The Model component signals a modification of the data
.dataoptions[]	Presenter, Model	Control options for the automatic synchronization between Model and View

The Presenter logic in the IDM currently comprises four automatisms:

1. If a View object with a linkage to a Model becomes visible ("map"), it fetches the data shortly before it gets visible (enabled by default).
2. If a View object with a linkage to a Model becomes invisible ("unmap"), it assigns the data to the linked model (disabled by default).
3. A dialog event on a View object with a link to a Model, which indicates a modification of a coupled attribute, assigns the data to the linked Model (disabled by default).
4. Modifications of the data on a Model, e.g. signaled by a *datachanged* event, lead to a propagation of the changes to the linked View objects if these are visible (enabled by default).

6.2 Linkage Between Model and View

In principle, the Datamodel is designed in a way that the Model is not aware which View objects are using it. The data schema for Model and View is the same: indexed attributes to enable scalars, vectors and matrices.

The *.datamodel* attribute is used to link a view with a model:

```
object .datamodel[<View attribute>] <Model ID>
```

For data transfer from the Model to the View, it also has to be defined which attribute of the Model should be retrieved.

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

To define the data transfer back from the View to the Model, the following attribute is used:

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

This way, it can be exactly determined what in the View object is populated through a Model and what is reassigned to the Model. By indexing the *.datamodel* attribute, it is also possible to use multiple Data Models for different attributes.

View attributes are typically, but not necessarily, actual attributes of the View object, e.g. *.content* for an ***edittext*** used as a View object. In order to achieve a better distinction between View and Model attributes, it is recommended to use user-defined attributes as Model attributes.

To “activate” the linkage respectively make it effective, the *.datamodel* attribute must always be set without index, as well as a linking with *.dataget* or *.dataset* (with or without index).

This is an **example** for the use of multiple Models (Data Models: “RecUsers”, “RecManager”, “VarValid”, “LbUsers”) by multiple Views (“listbox LbUsers”, “edittext EtName”, “pushbutton PbKill”).

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

record RecManager
{
  string CurrUser := "moreno";
  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(); }
}
```

To make the definition of common links between View and Model attributes as straightforward as possible, the IDM **default linkages** may be utilized (as can be seen in the example for “EtName”, where *.datamodel* and *.dataget* are used without index). These defaults are described in the table below:

Table 4: *Default linkages of View attributes*

Object Class	Default View Attribute for *.dataget*	Default View Attribute for *.dataset*
pushbutton ***statictext*** ***messagebox***	*.text*	–
checkbox ***radiobutton*** ***timer***	*.active*	*.active*
image ***menuitem***	*.text*	*.active*
filereq	*.value*	*.value*
listbox ***treeview*** ***tablefield***	*.content*	*.activeitem*
notepage	*.title*	–
edittext	*.content*	*.content*
poptext	*.text*	*.activeitem*
progressbar	*.curvalue*	–
scrollbar ***spinbox***	*.curvalue*	*.curvalue*
menubox ***toolbar*** ***window***	*.title*	–

The above example shows another particular feature. Once a visible object is used as Data Model (the View component “EtName” is linked to the Model “LbUsers”).

Basically, the IDM supports all object classes that allow user-defined attributes as Data Model, as well as global variables and functions with the function type **datafunc**. Linking multiple Models is also possible. For visible (visual) objects, however, one thing should be kept in mind: the user interaction does not trigger *changed* events for attributes and therefore also no automatic propagation of changes to the Model. This needs to be done explicitly or through the appropriate synchronization setting.

As View objects, as well any object classes that allow user-defined attributes are supported. However, the synchronization automatisms between model and view are designed for view objects to be instances with a visual representation.

6.3 Sequence and Value Aggregation

To populate a View object, a correct sequence is required when setting the View attributes. They are set by the IDM in a predefined, class-specific sequence (as in the attribute list of the object) to take the dependencies between the attributes into account. For a ***poptext***, as an example, the sequence

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

applies to observe the dependency of *.text[]*, *.activeitem* and *.userdata[]* upon the *.itemcount* attribute. However, this sequence can only be observed for the representation (that is, a synchronization triggered at the View component, e.g. via **:represent()**). The IDM has no influence on the sequence when the synchronization is triggered by the Model component, e.g. when using the **DM_DataChanged()** function.

If a Data Model is linked to all four attributes, first the number of elements is set, then the texts, then the active element is altered and finally the *.userdata[]* field is set.

In principle, the IDM does not prohibit using user-defined attributes as View attributes, but cannot ensure a consistent sequence for them. User-defined attributes and predefined attributes that do not belong to the object class are always handled after the predefined attributes for which the object class determines the sequence.

There are different kinds of value aggregation, i.e. the relation of data values between Model and View:

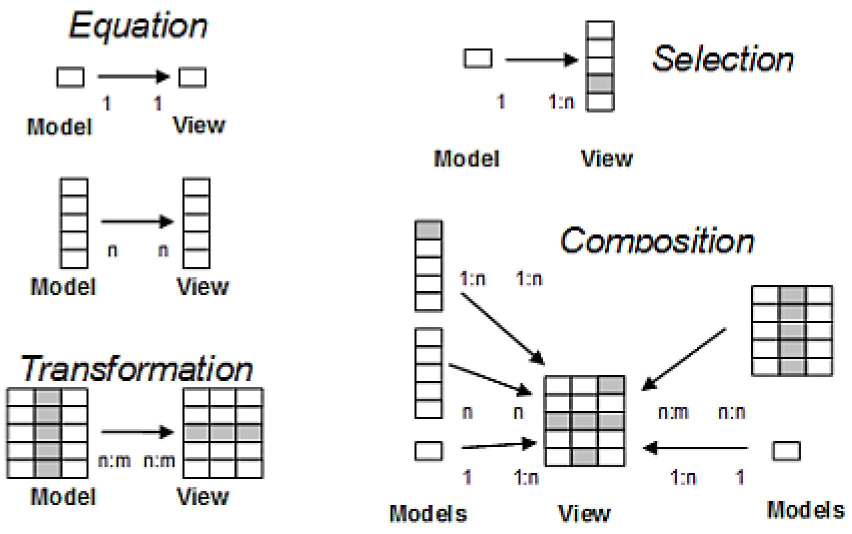


Figure 25: Relations between Model and View

There are two attributes to enable these kinds of relations:

```
anyvalue .dataindex[<View or Model attribute>] <index>
attribute .datamap[<View attribute>] <View attribute>
```

The *.dataindex[]* attribute is used to select a value from a value list and to transform indexed values. It should be noted that the attributes may occur as scalar, array, associative array or matrix (two-dimensional array), but access always only allows a vector (i.e. a one-dimensional array).

When assigning collections (value count <*n*>) to a vector attribute (one-dimensional or two-dimensional), the following transformations are carried out depending on the index:

Table 5: Index transformations when assigning collections

Index	Assignment
[0,<col>]	*[1,<col>] … [<n>,<col>]*
[<row>,0]	*[<row>,1] … [<row>,<n>]*
[<row>,<col>]	*[<row>,<col>]*
0	*1 … <n>*
<row>	*<row>*
void	*void*
others	*void*

The *.datamap[]* attribute allows merging data values (from one or more Data Models), which shall be mapped to exactly one View attribute. The main field of application will be two-dimensional attributes as found on the ***tablefield*** object. For this purpose, a “virtual” View attribute is used, which is then mapped to an “actual” View attribute by means of the *.datamap[]* attribute. Any predefined or user-defined attribute may serve as a “virtual” attribute, but it may be preferable to use an existing attribute on the View object to influence the sequence for a complete representation of all model values.

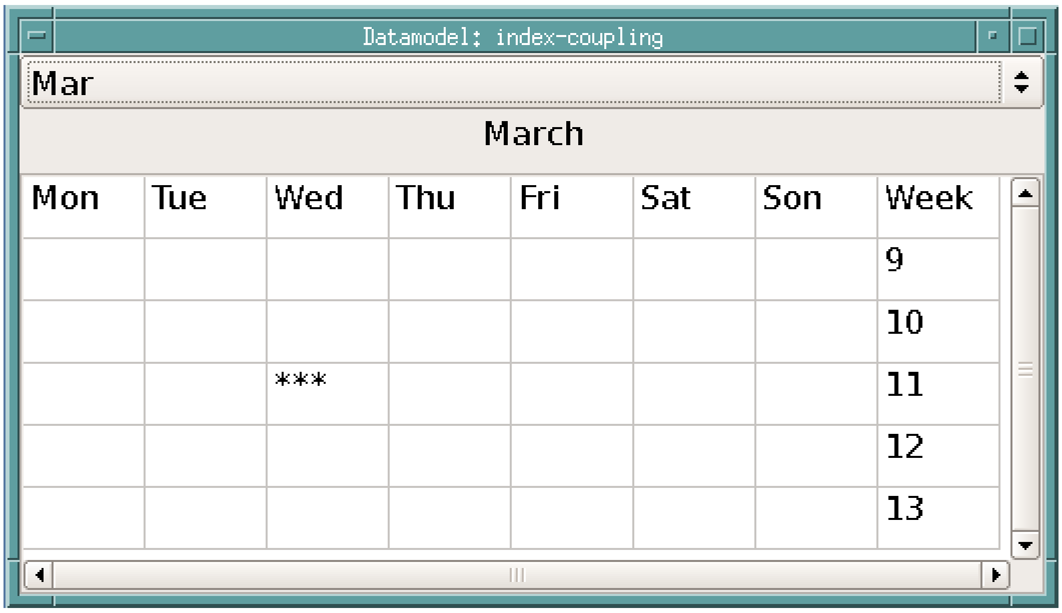
Example

The following example illustrates these different kinds of relations. For instance, the View object “PtMonth” is linked to the entire content of the attribute “.MonthName[]” from the Model object “RecDate”. In contrast, the View object “StCurMonth” displays only the 3rd item from the “RecDate.FullMonthName[]” array.

For the View object “TfCurWeek”, the data linkage is quite more complex. The weekday names from “RecDate.DayName[7]” are distributed to the header row (elements *[1,1] … [1,7]*) by the linkage `.dataindex[.content] [1,0];`. To link the week numbers into the table, the *.field* attribute is used by means of the definitions `.dataget[.field] .WeekNr;` and `.dataindex[.field] [0,8];`. Since there is a header row set through `.rowheader 1;`, the week numbers are written to the elements *[2,8] … [6.8]*. For marking the current day, the “virtual” attribute “.text” is used, for what a mapping `.datamap[.text] .content;` has been established to direct the marker text into the field “.content[4,3”.

```
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(); }
}
```

The screenshot for it looks like this:



March

Mon	Tue	Wed	Thu	Fri	Sat	Son	Week
							9
							10
		***					11
							12
							13

***Figure 26:** A tablefield populated from different Data Models*

It is crucial to understand this fact:

Data exchange between Model and View works via the existing mechanisms of the IDM, i.e. **setvalue ()**, **getvalue()**, **setvector()** und **getvector()**. This implies that the exchanged data values are either scalar or vectorial.

If the change of a Model attribute is signaled (e.g. in a custom data function by calling the **DM_ DataChanged()** function), this usually yields a Model index as an indication for the changed detail (typically of type *void*, *integer* or *index*). Since this change signaling is processed by the IDM as a *datachanged* event, multiple changes are combined into the overall change (*void* index) respectively redundant events are omitted.

In conjunction with the ability to influence the value aggregation through the *.dataindex[]* attribute, this has several consequences and particularities to be considered:

1. Complete population of a two-dimensional attribute (e.g. *.content[]* at the ***tablefield***) is possible without problems if the alignment as well as the number of columns and rows match the data length. The table size is adjusted through the **setvector()** functionality.
   However, single value modifications can only be updated at the correct cell in the tablefield if the Data Model issues a change notification with the respective index.
   Transfer of a two-dimensional array with its row and column form to an arbitrary position in a table-field is however not possible, only the signaling of value changes at any position.
2. With the use of *.dataindex[]* at the ***tablefield***, a scalar or vector data value can be moved to an explicitly defined cell, row or column. Automatic cell extension happens according to the rules of **setvalue()** or **setvector()** and therefore also depends on the index transformation. Please note that user-defined attributes (e.g. arrays) do not support automatic extension.
3. The same applies to vector and matrix attributes as they exist on ***listbox***, ***treeview*** and ***poptext***.

4. The index for a Model attribute is used to avoid unnecessary propagation of value changes (e.g. a change of “Data.Attr[5]” is of no relevance for View objects linked to “Data.Attr[2]” only).
5. When using *void* (default value for *.dataindex[]*) for the index transformation, propagation of the Model index – signaled by a *datachanged* event – is performed depending on the attribute type (scalar, vectorial, two-dimensional) up to the presentation by **:represent()**. Among others this is used in the example **randomcolors.dlg** to link color values from an associative array to the *.bgc[]* attribute of a ***tablefield*** object. However, this also means that linkages with different value dimensions (e.g. an array is mapped to a scalar) result in a value update depending on the index of the last change - which is usually not surprising. A possible solution is to use the correct index for the View and Model attributes by means of the *.dataindex[]* attribute.
6. Setting default values (index *[0]*, *[0,0]*, *[0,*]* or *[*,0]*) of the involved Model and View attributes is not possible.

The following examples (located in the *examples/datamodel/* subdirectory of the IDM installation directory) are worth a look at the source code as well as the trace file during execution:

relations.dlg

Demonstrates various value aggregations of scalar or vector data values in a ***listbox*** object with and without adaptation of the *.content[]* field as well as the effect of single changes.

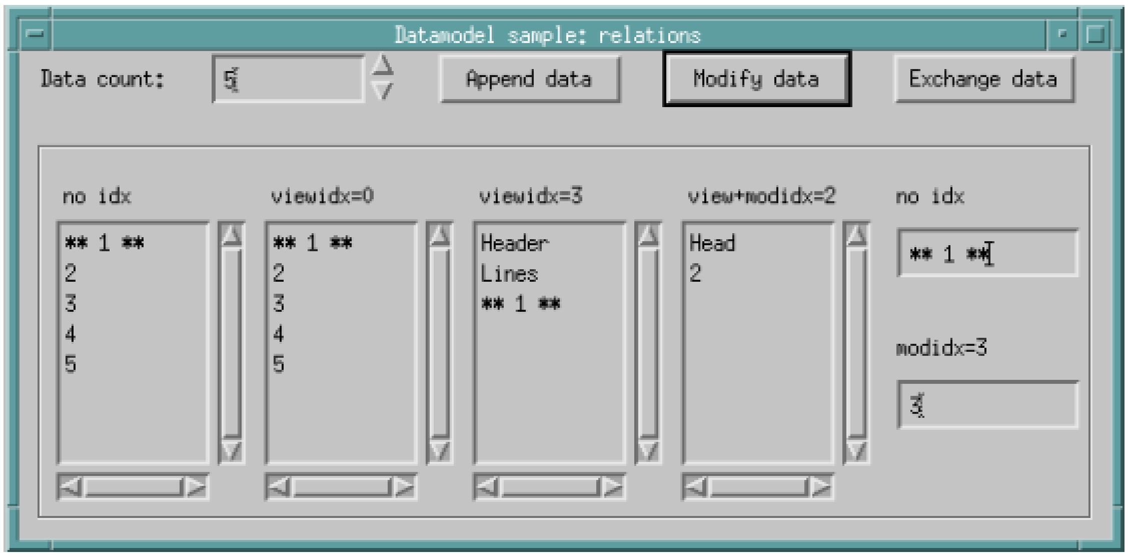
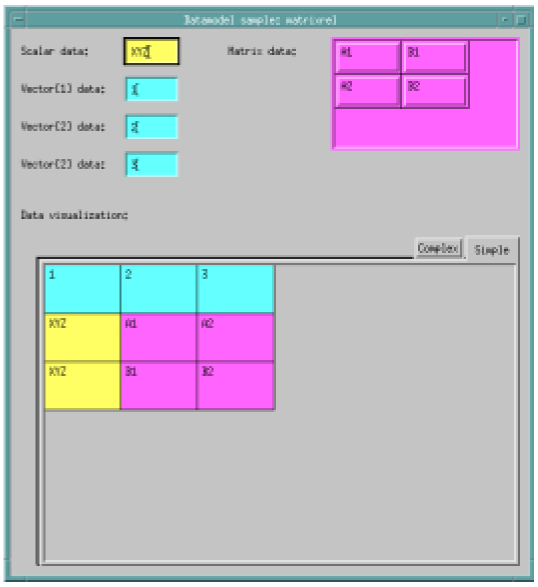


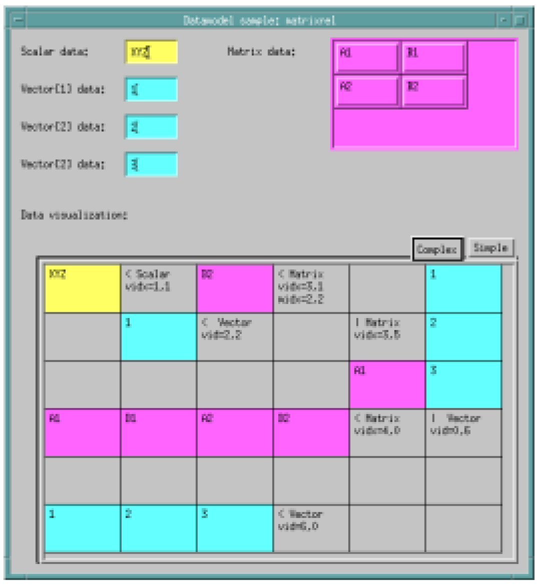
Figure 27: Value aggregations for list objects, example relations.dlg

matrixrel.dlg

Demonstrates several simple and complex value aggregations of different data values into a ***table-field*** object.



1	2	3
XYZ	A1	A2
XYZ	B1	B2

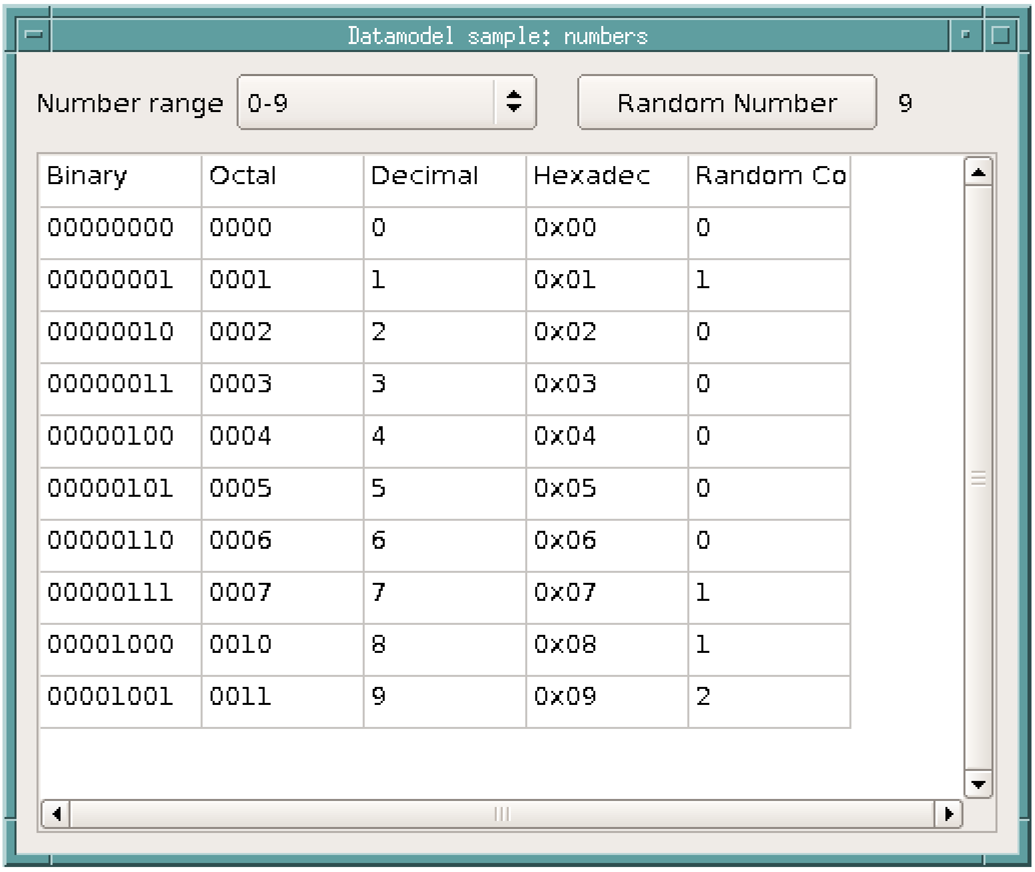


XYZ	< Scalar vid=1.1	B2	< Matrix vid=3.1 mid=2.2		1
	1	< Vector vid=2.2		| Matrix vid=3.5	2
				A1	3
A1	B1	A2	B2	< Matrix vid=4.0	| Vector vid=0.6
1	2	3	< Vector vid=6.0		

Figure 28: *Value aggregations for tablefields, example matrixrel.dlg*

numbers.dlg

Demonstrates the complete population of a ***tablefield*** object with a header row, including automatic row count adjustment.

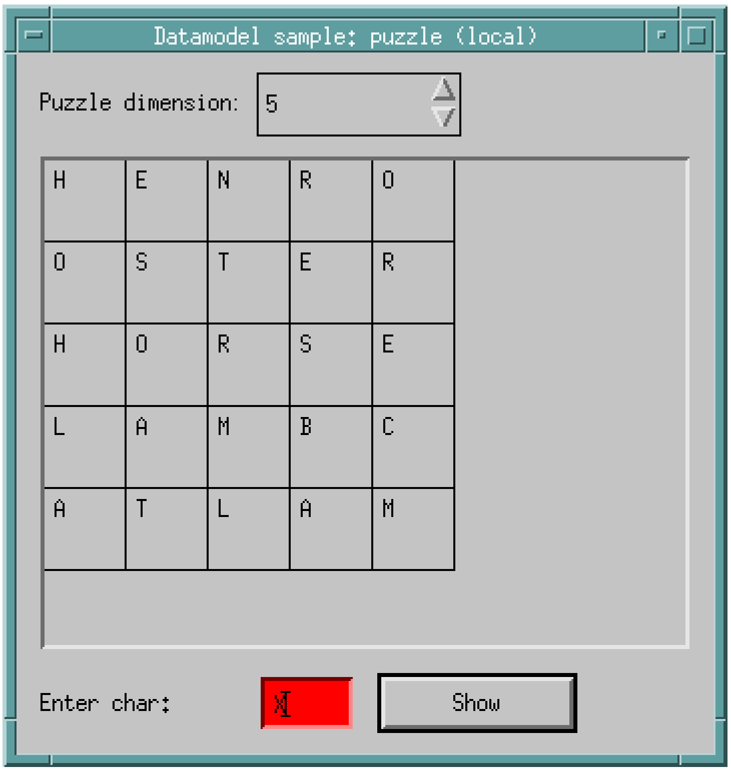


Binary	Octal	Decimal	Hexadec	Random Co
00000000	0000	0	0x00	0
00000001	0001	1	0x01	1
00000010	0002	2	0x02	0
00000011	0003	3	0x03	0
00000100	0004	4	0x04	0
00000101	0005	5	0x05	0
00000110	0006	6	0x06	0
00000111	0007	7	0x07	1
00001000	0010	8	0x08	1
00001001	0011	9	0x09	2

Figure 29: *Populating an entire tablefield, example numbers.dlg*

puzzle.dlg

Demonstrates populating a ***tablefield*** from a ***record*** as a Data Model or with a data function (local or remote).


Datamodel sample: puzzle (local)

Puzzle dimension: 5

H	E	N	R	O
O	S	T	E	R
H	O	R	S	E
L	A	M	B	C
A	T	L	A	M

Enter char: X

Figure 30: *Populating a tablefield from record and data function, example puzzle.dlg*

6.4 Synchronization Between Model and View

Automatic synchronization between Model and View is controlled by the *.dataoptions[]* attribute.

Table 6: *Options for synchronization between Model and View*

Attribute Index	Default	Component	Meaning
dopt_represent_on_map	***true***	View	Immediately before the View is made visible, the data values are fetched from the Model components and set on the View object.
dopt_represent_on_init	***false***	View	During object initialization (**:init** method), the data values are retrieved and set on the View object.

Attribute Index	Default	Component	Meaning
dopt_apply_on_unmap	***false***	View	Immediately before the View is made invisible, the data values are fetched from the View object and assigned to the linked Model components.
dopt_apply_on_event	***false***	View	If a user interaction triggers a dialog event which indicates a possible change of a View attribute, this is assigned to the linked Model components.
dopt_propagate_on_start	***true***	Model	When a dialog or module is started, the data from the Model objects is forwarded to the linked View components.
dopt_propagate_on_changed	***true***	Model	Modifications to a Model attribute are forwarded to the linked View components.
dopt_cache_data	***true***	Model	This index value is only available for the ***doccursor***. ***true*** The data selected by the ***doccursor*** is buffered for further accesses ("caching"). *false* With each access, the ***doccursor*** selects the data anew from the XML Document.

A key feature of the Datamodel is that data changes are always signaled and processed "asynchronously" via the event processing. For example, if the Data Model (Model object) is a ***record*** or another object with user-defined or predefined attributes, an attribute change, which is signaled by a *datachanged* event, is transmitted to the linked View objects so that they can update themselves. The dialog programmer can suppress a *changed* event with the operation "::=", but never a *datachanged* event.

A global variable may be used as a Model component as well, but does not allow controlling the synchronization. Variable values s are always propagated when the dialog or module is started and when the variable value is changed.

When `.dataoptions[dopt_apply_on_unmap] := true;` is used on a **dialogbox**, ***messagebox*** or ***filereq*** via a **querybox** call, the data values are only transferred from the View to the Model in case of a positive confirmation (*button_ok* or *button_yes*).

Manual intervention is not necessary in most cases. However if required, manual synchronization can be accomplished using the following methods:

Table 7: *Manually callable Datamodel methods*

Method	Component	Function
:represent()	View	The data of all linked Model attributes (*.dataget*) is retrieved and assigned to the View object
:apply()	View	The data from all linked View attributes (*.dataset*) is fetched and assigned to the Models
:propagate()	Model	The Model object communicates a change of all its Model attributes to the View objects that have a linkage.
:collect()	Model	The Model object fetches all data from the linked View objects to assign it to its Model attributes.

Synchronization only happens between instances, never with Default objects or Models.

For optimized synchronization of partial values, there is another particular feature. If a value change is signaled, the index of this value change is forwarded so that only this single value needs to be updated by the View component.

Example

Through a ***timer*** object, random colors are picked and assigned to random table cells.

```
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;
  }
}
```

This is the screenshot for it:

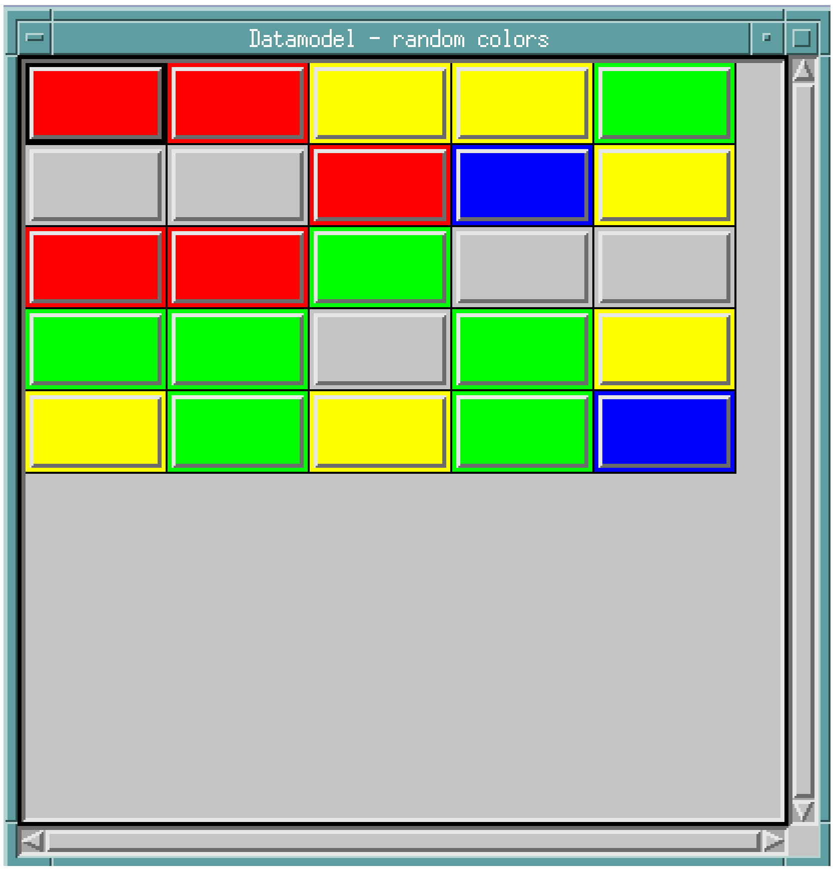


Figure 31: Randomly colored tablefield cells with a timer as Data Model

6.5 Conversion and Conversion Methods

During synchronization between Model and View components, data values are exchanged, which usually means setting or querying attributes (similar to , **getvalue()**, **setvector()**, **getvector()**). Intervention of the application programmer by redefinition of the **:set()** and **:get()** methods is **not** possible.

However, on the View component it is possible to exert influence by means of the redefinable methods **:represent(<Value>, <Attribute>, <Index>)** and **:retrieve(<Attribute>, <Index>)**.

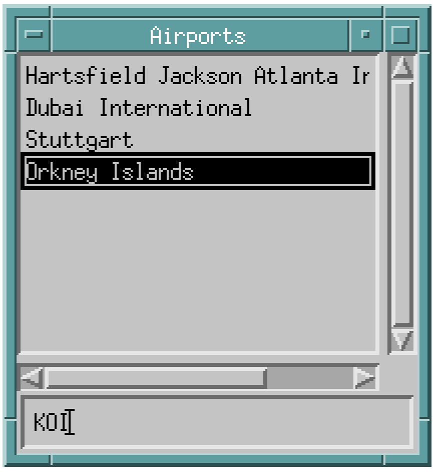
When attributes are set at the View component, the IDM default handling normally performs a type conversion to the target data type. The following example demonstrates how a special handling for the View attribute *.activeitem* is achieved through redefinition.

Example

An airport list is displayed in the ***listbox*** “Lb”. The redefined methods **:represent()** and **:retrieve()** ensure a correct conversion from and to an IATA code.

```
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(); }
}
```

This is the screenshot for it:



```
Hartsfield Jackson Atlanta Ir
Dubai International
Stuttgart
Orkney Islands
```

```
KOI
```

__Figure 32:__ Window of the example dialog for overwriting __:represent()__ and __:retrieve()__

6.6 Use of XML with the Datamodel

Availability

Since IDM version A.06.01.b

XML may be used as a Data Model, where node texts and node attributes can contain the data – usually strings. An XML Document is linked to a View through a ***doccursor***. For this purpose, Data Model attributes and selection patterns for nodes of the XML Document are defined at the ***doccursor***. Further it can be defined whether the Data Model attribute is linked to the content or an attribute value of the node. For operations that change the XML Document, the data changes are forwarded to all Data Model attributes.

The ***doccursor*** has the following attributes to use it as a Data Model:

Table 8: *Datamodel attributes of the doccursor*

Attribute	Meaning
.dataselect	Defines a Data Model attribute with associated selection pattern.
.dataselectattr	Maps a Data Model attribute to an attribute of the selected nodes.
.dataselecttype	Data type conversion of a Data Model attribute.
.dataselectcount	Defines the cardinality of a Data Model attribute.
.dataoptions	Controls caching via the index *dopt_cache_data*.

The *.dataselect* attribute is of crucial importance here. It defines the Data Model attributes of the ***doccursor*** and their linking to nodes of the XML Document using selection patterns. The syntax of the selection patterns is the same as the pattern definitions for the **:select** method. The selection pattern of a *.dataselect* attribute without index – that is, without a Data Model attribute – is applied before the selection patterns of all Data Model attributes. This enables relative selection patterns for the Data Model attributes originating from the nodes preselected by the *.dataselect* attribute without index. At the same time, this reduces the effort required to collect the data.

When collecting the data for a Data Model attribute, it is looped through the document using the selection pattern. Either the node content is fetched using the *.text* attribute of the ***doccursor*** or the value of the node attribute defined by *.dataselectattr*.

For the Data Model attributes defined with *.dataselect*, data type and cardinality may be altered using the attributes *.dataselecttype* and *.dataselectcount*. By default, the Data Model attributes contain vectors with string values (data type *vector[string]*). With the . *dataselecttype* attribute a data type (e.g. *integer*, *boolean*) can be defined to convert the values into. The cardinality (vector or scalar) of Data Model attributes can be controlled via the attribute *.dataselectcount*. If the data type of the Data Model attribute is a collection or its cardinality is the data type *integer*, then the Data Model attribute contains a vector, otherwise only a scalar with the first value.

Once retrieved values of a Data Model attribute are cached until either the XML Document or the Data Model attributes of the ***doccursor*** change. This caching can be prevented by setting *.dataoptions [dopt_cache_data] = false* at the ***doccursor***.

Saving data in an XML Document happens in a similar way with reversed operations. However, nodes cannot be created or deleted automatically in the XML Document.

Notes

- Collecting the data can be an “expensive” operation, since the selection pattern must be searched in the whole XML Document. The effort may be reduced by limiting the search to preselected sub-trees by using a *.dataselect* attribute without index.
- A ***doccursor*** used as a Data Model should not be used for other purposes, as it typically changes its selection path constantly.

6.6.1 Example

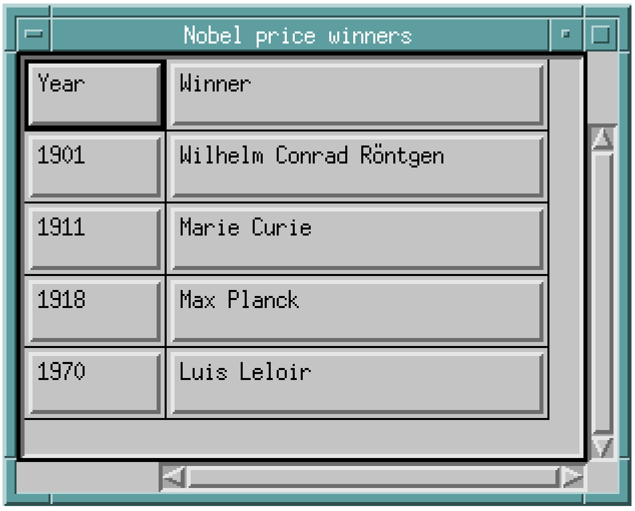
A list of Nobel laureates shall be read from the following XML file and displayed in a table:

```
<?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>
```

The names of the laureates are taken from the content of the XML nodes “winner” and stored in the Data Model attribute “.Winner”. The years of the award can be found in the “year” attribute of the XML nodes and are counted in the file from 1900. They are converted to *integer* and read as *vector* into the Data Model attribute “.Year”. In the overwritten **:represent** method, the years are completed to four-digit numbers for display.

```
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");
}
```

This dialog produces the following window:



Year	Winner
1901	Wilhelm Conrad Röntgen
1911	Marie Curie
1918	Max Planck
1970	Luis Leloir

***Figure 33:** Table with XML data used as Data Model*

6.6.2 Index Value *dopt_cache_data* of the Attribute *dataoptions*

Attribute Index	Default	Component	Meaning
dopt_cache_data	***true***	Model	This index value is only available for the ***doccursor***. **true** The data selected by the ***doccursor*** is buffered for further accesses ("caching"). **false** With each access, the ***doccursor*** selects the data anew from the XML Document.

6.7 Actions

In order to make manipulation functions of the Model component generally available for the Presenter logic, the concept of actions is used. These are methods with parameters which are provided by the Data Models. They can be invoked via the **:calldata()** method.

6.8 Tracing

The following trace codes have been introduced to reveal the synchronization processes between Data Models and presentation objects (Views) and to detect possible application errors.

Trace Code	Meaning
[CD]	"Call Data" to track the call for synchronization between View and Model. Typically **:get()** and **:set()** on the Model object as well as **:represent()** and **:apply()** on the View object are traced.
[CE]	"Data Changed Event" – Event that signals a change on the Data Model, typically triggered by setting an attribute or calling the application function **DM_DataChanged()**.

6.9 Constraints

The IDM rejects modifications of the data linking or of the synchronization control (attributes *.datamodel*, *.dataget*, *.dataset*, *.datamap*, *.dataindex*, *.dataoptions*) while being within a redefinable method **:represent()** or **:retrieve()** or within a call of a data function.

Likewise the IDM rejects changes to the attributes *.visible* or *.mapped* if these are executed during synchronization of the View component and the IDM currently is in the state of converting visibility or invisibility.

7 Multiscreen support under Motif

The IDM for Motif has programming support for multiple screens.

Meaning of multiscreen-support under X

This refers to an X server configuration with multiple screens (multiple screens either by one graphics card with multiple frame buffers or by multiple graphics cards in the same display host). The screens can differ in resolution as well as in color capabilities. On the other hand, they share input devices like mouse and keyboard. In addition, their arrangement to each other can usually be defined in the X server configuration (e.g. *screen#1* is to the right of *screen#0*).

Even before, an IDM application could be displayed in a specific screen, e.g. via the X option *-display* `<host>:<display>.<screen>`. What is **new** is that **windows can appear in different screens** within an IDM application.

Support by the IDM

The IDM supports the application programmer in that he can query the available screens and their characteristics (this is done via the setup object, see also the attributes .screencount, .screen, etc.).

In addition, the IDM allows the assignment of which window is to be displayed on which screen. This is done via the .display attribute on the window. It must be set to a display resource, which is to be defined by the user. The IDM then takes care of the necessary resource management for colors, cursors and images.

Dynamic switching of the window to another screen can be done either via the .display attribute or the display resource. For an example dialog, see also the documentation for the display resource.

Dialog boxes such as message boxes or the file requester are displayed in the same screen as the parent specified in the querybox call. If no parent is specified, the dialog boxes are displayed in the default screen.

Remarks

- Unfortunately, the layout of the screens in relation to each other cannot be determined via X/Motif and is therefore not accessible from the IDM.
- The IDM's memory requirements are optimized for the standard single-screen application case.
- Multiscreen and Non-Default Visuals: The IDM allows the specification of a visual ID via the environment variable IDM_VISUAL_ID in order to use a visual other than the default one (e.g. grayscale screen on a TrueColor display). In a multiscreen configuration, this specification only affects the default screen, i.e. the windows in other screens are displayed in the default visual.
- Support for the newly added attributes and resources is now available for the IDM Editor.

See also

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

Availability

Since IDM version A.05.01.c

This support is also available in the IDM 4 version from A.04.04.j.

8 Multi-monitor support under Windows

With Windows it is now possible to query the setup object for the number of monitors (.screencount) and the coordinates of the monitors (.screen_x[integer], .screen_y[integer], .screen_width[integer] und .screen_height[integer]). The requested values are dynamic, because one can change the scaling of a monitor or add or remove monitors at any time.

If several monitors are used, they are positioned in a virtual desktop. The coordinates of this virtual desktop can be queried with .vscreen_x, .vscreen_y, .vscreen_width, .vscreen_height. The coordinates of the workspace (without the taskbar) are rendered. The workspaces are included in the calculation of the virtual desktop.

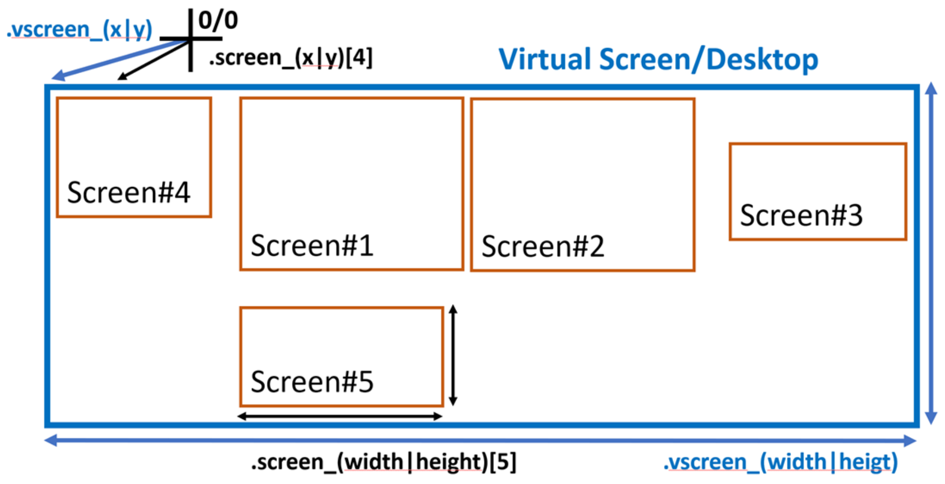


Figure 34: *relation of .screen_*[] and .vscreen_* attributes*

Attention:

If the magnification settings of the monitors differ, please note:

IDM for Windows 11

The values do not appear consistent. Based on the recommended use of *.xauto = 1* and *.yauto = 1* for windows, the position is converted with the system scaling factor, while the size is converted with the monitor scaling factor. This allows a window to be set to a position and size that fills the entire monitor.
The position and size of the virtual desktop are converted using the system scaling factor. The monitor to which a window that spans several monitors is assigned can depend on the monitor on which the window is currently located. For a defined behavior, coordinates should therefore only be changed in the invisible state.

IDM for Windows 10

Microsoft Windows internally scales applications that are DPI-unaware. In this case, the different scaling factors lead to misrepresentations. To avoid this, either all windows should be opened on the primary monitor only, or all monitors should have the same scaling factor.

Remark for Windows 10 and 11 with several monitors:

In WINDOWS, after adding/removing a monitor or changing the display settings, you may encounter the following problems:

- No moving frame appears when moving a toolbar object. It can also happen that a rectangle the size of the **toolbar** frame is displayed with an incorrect background on another monitor.
- Undocked **toolbar** objects do not adjust to changes in DPI value. Both when moving to another monitor or changing the magnification.

The problem is due to Windows, which on the one hand incorrectly converts the coordinates when drawing to the desktop and on the other hand does not send important messages (*WM_DPICHANGED*) to so-called tool windows and does not change the window DPI value.

After the screen saver was active or you logged in again (and after a reboot), the problem no longer occurs.

See also

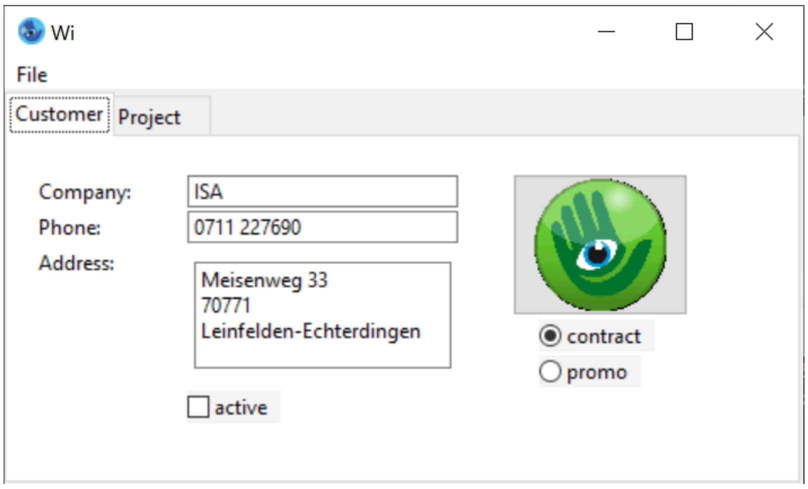
attributes .screencount, .screen_x[integer], .screen_y[integer], .screen_width[integer], .screen_height[integer], .vscreen_x, .vscreen_y, .vscreen_width, .vscreen_height

Availability

Since IDM version A.06.03.a

9 HighDPI Support

The resolution of modern screens is constantly growing. This means that the requirements for the design and quality of the operating elements and interfaces are also changing. Problems are often that applications appear much too small and text and graphics are displayed blurred or pixelated. The resolution, DPI and scaling set in the system, as well as fonts and the size and quality of images, influence the appearance of an application. To ensure that existing applications are also displayed in detail and sharpness on high-resolution screens, they must be prepared for HighDPI.

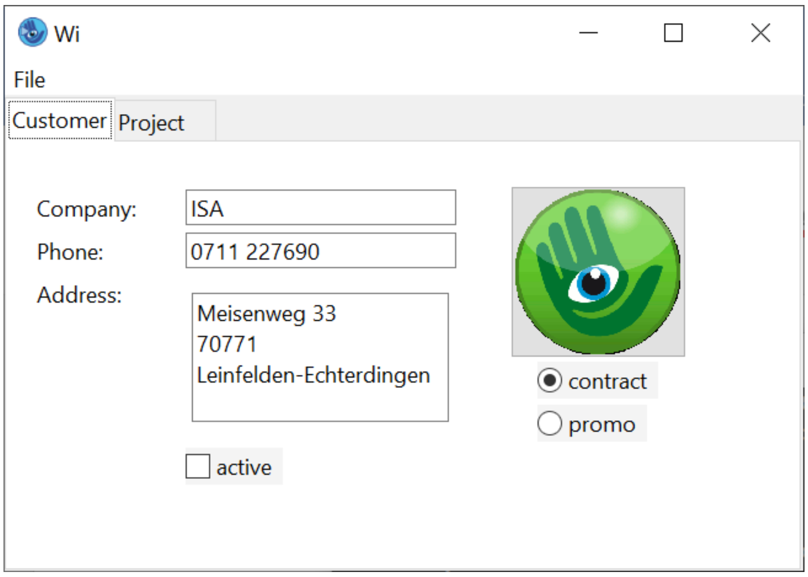


Company: ISA
Phone: 0711 227690
Address: Meisenweg 33
70771
Leinfelden-Echterdingen

[ ] active

[● contract] ○ promo

Figure 35: *without HighDPI support*



Company: ISA
Phone: 0711 227690
Address: Meisenweg 33
70771
Leinfelden-Echterdingen

[ ] active

[● contract] ○ promo

Figure 36: *with HighDPI support*

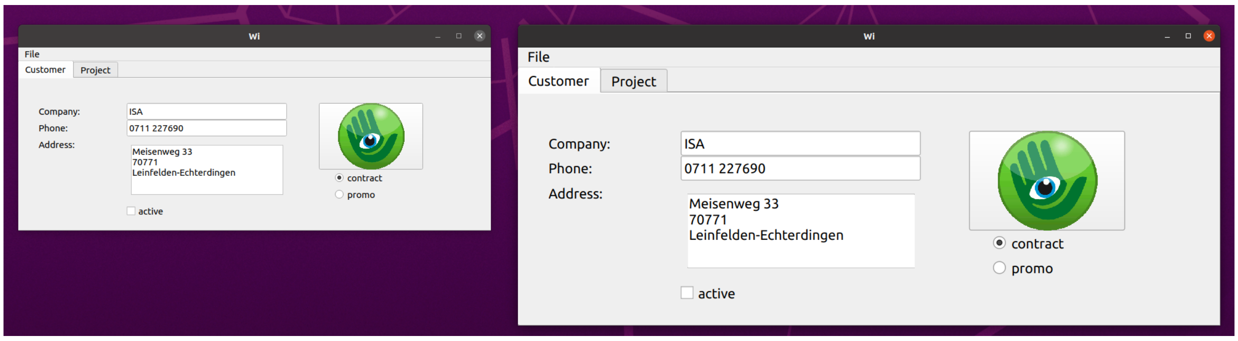
The IDM for MOTIF, QT and MICROSOFT WINDOWS as of version A.06.03.a (only the IDM FOR WINDOWS 11 on MICROSOFT WINDOWS) now provides the appropriate tools to support high-resolution screens and future-proof applications.
In addition, working with multiple screens has been improved. IDM applications now respect the

system's DPI settings and scaling factors. As a result, previous problems such as pixelated, muddy, or too-small applications or reduced font sizes at high resolutions and small screen sizes (the effects can vary by system) no longer occur. The IDM now automatically adjusts geometry, layout, font sizes, graphics and cursors to the appropriate resolution and DPI determined by the system.
The scaling factor is based on the system settings of the respective desktop environment. However, the user is free to make other settings or intervene creatively through various new attributes, options and mechanisms.

9.1 Start options

-IDMscale <integer>

The startup option **-IDMscale** can be used to enable or disable scaling by the IDM. Under QT and MOTIF, the value specifies the scaling in %.



Company: ISA
Phone: 0711 227690
Address: Meisenweg 33
70771
Leinfelden-Echterdingen
[✓ contract] promo
☐ active

Company: ISA
Phone: 0711 227690
Address: Meisenweg 33
70771
Leinfelden-Echterdingen
[✓ contract] promo
☐ active

Figure 37: *left started normally (system scaling 100%), right started with 150% scaling*

Note

This option should not be used under MICROSOFT WINDOWS. DPI awareness is a property of the application and should be specified in a manifest file. For testing purposes, DPI awareness can be turned on (value: *1*) or off (value: *0*).

Attention:

It is recommended to not use a scaling > 0 and < 100%, as this may impact the display and operation of objects.

-IDMtiledpi <integer>

Graphics are automatically scaled according to the set scaling factor. It is assumed here that the images have been designed for a DPI value of *96*. If the images of the application have been designed for a higher resolution, this can be set via the start option -IDMtiledpi. The size of an image is then converted to the currently valid DPI value based on this value and scaled accordingly.

Likewise, this setting can be made directly on the setup object with the .tiledpi attribute (see Enhancement to the setup object).

9.2 Layout resources

Until now, the available predefined resources in IDM were always WSI- or system-dependent and always required complex adaptations when transferred to other systems. IDM A.06.03.a now offers WSI-independent and HighDPI-compliant predefined layout resources for the first time. This includes font (Zeichensatz), color (Farbe) and cursor resources, which are available on all systems in similar form and meaning. This offers the enormous added value that dialogs which have been completely converted to the new UI resources can now be transferred to other systems easily and without effort. These resources can be identified by the naming scheme ***UI*_FONT***, ***UI*_COLOR*** or ***UI*_CURSOR***.

Example:

```
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";
```

The UI_resource names can also be queried - like all other predefined resource names- at the setup object via the attributes .colorname[integer], .fontname[integer], .cursorname[integer].

Cursor

The uniform Cursor resources have the naming scheme ***UI*_CURSOR*** and are defined as follows:

UI_CURSOR	Default cursor usually arrow
UI_IBEAM_CURSOR	Edittext insertion marker
UI_WAIT_CURSOR	Busy indicator, "hourglass"
UI_CROSS_CURSOR	Cross
UI_UP_CURSOR	Up arrow
UI_SIZEDIAGF_CURSOR	Sizing arrow from upper left to lower right
UI_SIZEDIAGB_CURSOR	Sizing arrow from lower left to upper right
UI_SIZEVER_CURSOR	Sizing arrow from top to bottom
UI_SIZEHOR_CURSOR	Sizing arrow from left to right

UI_MOVE_CURSOR	Moving arrow in all directions
UI_STOP_CURSOR	Prohibition sign (crossed out circle)
UI_HAND_CURSOR	Hand symbol (index finger used for pointing)
UI_HELP_CURSOR	Arrow with question mark (for context help mode)

Fonts

The unified font resources have the naming scheme ***UI*_FONT*** and are defined as follows:

UI_FONT	Font used by default for UI elements (Default or Dialog Font).
UI_FIXED_FONT	Font with a fixed character width.
UI_MENU_FONT	Font used for menus.
UI_TITLE_FONT	Font used in the window title bar.
UI_SMALLTITLE_FONT	Font used in a small/low window title bar.
UI_STATUSBAR_FONT	Font used in the status bar.
UI_NULL_FONT	Behavior as if no font setting was made (null). The WSI uses the default system font provided for the UI element or a system fallback font.

Color

The color resources have the naming scheme ***UI*_COLOR*** and are defined as follows:

UI_BG_COLOR	Color used by default for general backgrounds of (mostly group) objects. Used for object classes with attribute .bgc/.bgc[], e.g. window, groupbox, notepage, statictext, pushbutton...
UI_FG_COLOR	Color used by default for general foreground/text color of (mostly group) objects. Used for object classes with attribute .fgc/.fgc [], e.g. window, groupbox, notepage, statictext, pushbutton...
UI_INPUTBG_COLOR	Color used by default for backgrounds of input/selection objects. Use e.g. for edittext, lists, tables, poptext, treeview...(e.g. object classes with attributes .bgc/.textbgc)
UI_INPUTFG_COLOR	Color used by default for foregrounds of input/selection objects, such as edittext, lists, tables, poptext, treeview (e.g. object classes with attributes .fgc/.textfgc).

UI_ BUTTONBG_ COLOR	Color used by default in the background of button-like objects, e.g. pushbutton or image (e.g. object classes with attribute .bgc/.imagebgc).
UI_ BUTTONFG_ COLOR	Color used by default when foregrounding button-type objects, e.g. pushbutton or image (e.g. object classes with attribute .fgc/.imagefgc).
UI_BORDER_ COLOR	Color used by default as border (e.g. at .bordercolor).
UI_ ACTIVEBG_ COLOR	Color that is used by default as background color of the active element, e.g. for lists, pop text, tables, possibly progressbar, markers etc. Mostly inversion of the normal foreground and background colors of the widget (e.g. object classes with attribute .bgc).
UI_ ACTIVEFG_ COLOR	Color that is used by default as foreground/text color of the active element, e.g. for lists, poptext, tables, possibly progressbar, markers etc. Mostly inversion of the normal foreground and background colors of the object (e.g. object classes with attribute .fgc).
UI_TITLEBG_ COLOR	Color used by default as background color for title bars or borders of win-dows/dialogs, if supported (e.g. at attribute .titlebgc).
UI_TITLEFG_ COLOR	Color to be used by default as foreground/ text color for title bars or borders of windows/dialogs, if supported (e.g. at attribute .titlefgc).
UI_MENUBG_ COLOR	Color to be used by default as background color for menus, if supported (e.g. at attribute .titlebgc).
UI_MENUFG_ COLOR	Color to be used by default as foreground/text color for menus, if supported (e.g. at attribute .titlefgc).
UI_ HEADERBG_ COLOR	Color used by default as background color in table headers - if supported.
UI_ HEADERFG_ COLOR	Color used by default as foreground/text color for table headers – if supported. Use with tablefield (e.g. at attribute .rowheadfgc, .cloheadfgc).
UI_NULL_ COLOR	Behavior as if no color (null) was set. WSI draws according to its default palette.

It may happen that not all UI*COLORS differ from each other. This is due to the fact that most colors are "calculated" from a few basic colors from the desktops.

Notes Motif:

In addition to the ***UI*_COLOR*** colors, the system names for color resources have been completed. Newly added are:

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

Notes Windows:

The ***UI*_COLOR*** colors access the Windows system colors. It should be noted that with the introduction of Visual Styles, the individual objects no longer use the system colors, but the colors defined by the theme. Depending on the selected theme there can be strong discrepancies. Therefore, if possible, no colors (or better the ***UI_NULL_COLOR***) should be set under Windows. Furthermore, it should be noted that with Windows 10 the color variety has been reduced. For example, the colors ***UI_HEADERBG_COLOR***, ***UI_HEADERFG_COLOR***, ***UI_MENUBG_COLOR***, ***UI_MENUFG_COLOR***, ***UI_TITLEBG_COLOR*** and ***UI_TITLEFG_COLOR*** are no longer supported. This means that these colors are still available and usable for the user, but they are currently not used by any object by default.

Notes Qt:

The ***UI*_COLOR*** colors access the colors of the standard palette offered by Qt. The colors are based on the ColorGroup Normal or Active. The colors of the standard palette are determined by the currently set system style.

In addition to the ***UI*_COLOR*** colors, the predefined colors have been supplemented by further ColorRoles offered by the Qt Toolkit. New additions (in all forms of the ColorGroups) are:

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

9.3 Enhancement to the tile resource

The existing scaling options at the Tile resource have been extended. It is now possible to specify a .scalestyle that determines how a tile should be scaled. Thus, tiles can now be scaled according to the following characteristics:

- automatic
- according to a specific factor
- proportional
- free in all directions
- based on the DPI value and no scaling

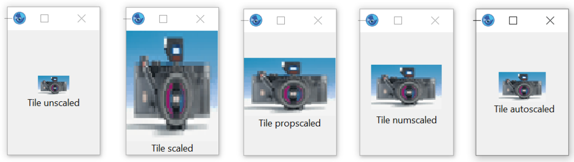


***Figure 38:** the same picture with different values of the attribute .scalestyle at a system scaling of 150%*

The following values are available for selection in the rule language:

Definition on Tile	Dynamic setting	Meaning
noscale	scalestyle_ none	The pattern or image is not scaled. setup.tiledpi has no impact.
scale	scalestyle_ any	The height and width of the pattern or image are fully enlarged to fit the available area.
propscale	scalestyle_ prop	Height and width of the pattern or image are enlarged to the available area, while height and width proportions of the pattern or image are maintained in any case. I.e. free spaces can be created above and below or left and right.
numscale	scalestyle_ num	The scaling of the pattern or image is done by a numerical scaling divider. The scaling is done in quarter steps, i.e. 1.25-fold, 1.5-fold, 1.75-fold, 2-fold, 2.25-fold, 2.5-fold, and so on. A downscaling is done down to a maximum of 0.25-fold.
dpi	scalestyle_ dpi	The pattern or image is always scaled according to the set screen scaling.
Not provided	scalestyle_ auto	The pattern or image is scaled according to the set screen scaling. A scaling compatible to the previous version takes place. **Default value**

The ***scalestyle*** can be specified either directly with the **tile definition** or as a **dynamic attribute**.

Tile definition:

```
tileSpec ::= <tileBitmap> | "<file path>" | "<graphics resource>" { scale |
noscale | numscale | propscale | dpiscale} ;
```

Dynamic attribute:

	Identifier		Data Type
Rule Language	**.scalestyle**		enum
C	AT_scalestyle		DT_enum
COBOL	AT-scalestyle		DT-enum
Classification	object-specific attribute		
Objects	tile		
Access	get, set	Default value	scalestyle_auto
changed event	no	Inheritance	–

Example:

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

See also chapter „tile (Muster)“ in manual „Ressourcenreferenz“.

9.4 Enhancement to the font resource

The font resource has a new property .propscale. This controls whether the horizontal and vertical raster should be set proportionally to the maximum value, which is determined by calculating the value of xraster and yraster of the font raster.

The ***property*** can be specified either directly whith the **font definition** or as a **dynamic attribute**.

The divisor is used to refine the grid.

Font definition:

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

Dynamic attribute:

	Identifier		Data Type
Rule Language	**.propscale**		boolean
C	AT_propscale		DT_boolean
COBOL	AT-propscale		DT-boolean
Classification	object-specific attribute		
Objects	font		
Access	get, set	Default value	false
changed event	no	Inheritance	–

See also chapter „font (Zeichensatz)“ in manual „Ressourcenreferenz“.

9.5 Enhancement to the setup object

	Identifier		Data Type
Rule Language	**.tiledpi**		integer
C	AT_tiledpi		DT_integer
COBOL	AT-tiledpi		DT-integer
Classification	object-specific attribute		
Objects	setup		
Access	get, set	Default value	–
changed event	no	Inheritance	–

	Identifier		Data Type
Rule Language	**.tiledpi**		integer
C	AT_tiledpi		DT_integer
COBOL	AT-tiledpi		DT-integer
Classification	object-specific attribute		
Objects	setup		
Access	get, set	Default value	96
changed event	no	Inheritance	–

This attribute determines for which resolution the images/patterns were designed. The size of an image/pattern is converted to the currently valid DPI value based on this value.

See also chapter „setup“ in manual „Objektreferenz“.

9.6 Enhancement to the image object

For better alignment and extended layout options, the following attributes on the Image object can now be used to control margins or spacing:

	Identifier		Data Type
Rule Language	**.xmargin**		integer
C	AT_xmargin		DT_integer
COBOL	AT-ymargin		DT-integer
Classification	object-specific attribute		
Objects	image		
Access	get, set	Default value	0
changed event	no	Inheritance	ja

This attribute determines the horizontal distance between the border and the display area.

	Identifier	Data Type
Rule Language	**.ymargin**	integer
C	AT_ymargin	DT_integer

	Identifier		Data Type
COBOL	AT-ymargin		DT-integer
Classification	object-specific attribute		
Objects	image		
Access	get, set	Default value	0
changed event	no	Inheritance	ja

This attribute determines the vertical distance between the border and the display area.

See also chapter „image“ in manual „Objektreferenz“.

9.7 Support of HiDPI image variants

To achieve an optimal design of the interface regarding images and application icons when using the higher resolution, it is recommended to use images adapted to the scaling factor.
Until now, there were only limited options for the WINDOWS and QT window systems. Up to now, WINDOWS has offered the ICON/BITMAP mechanism and QT the Icon Resource mechanism for support in this regard. Both mechanisms allow multiple resolution levels to be maintained, but are limited in terms of image type or usage. Such a mechanism does not exist for MOTIF.
The IDM A.06.03.A therefore provides an option for loading multiple resolution levels that can be applied equally to WINDOWS, QT and MOTIF and is thus also system-independent.
The tile management of the IDM was only able to load one image file (e.g.: tile Ti “smiley.gif”) and did not keep several resolution factors of these (except for the exceptions mentioned above). If there is a scaling of the application, the new loading mechanism for tile files now tries to load the image data for the corresponding scaling factor first. For this purpose, the extension ***@nx*** is added to the file name and in front of the suffix, similar to the QT mechanism. Here ***n*** stands for the scaling factor between *2* and *9*. The numbre *2* corresponds to a scaling of *150%-249%*, *3* of *250%-349%*, and so on.
Reasonably the size changes of the variant images should correspond to the scaling factor, whereby a check by the IDM does not take place. So, for example, original image “smiley.gif” (32x32 pixels), smiley@2x.gif (64x64 pixels), smiley@3x.gif (96x96 pixels), smiley@4x.gif (128x128 pixels).

Example:

```
tile Ti „smiley.gif“;  // at a scaling of 200% the IDM
                       // automatically loads the image
                       // smiley@2x.gif, if present (Windows, Motif, Qt)

                       // at a scaling of 300% the IDM
                       // automatically loads the image
                       // smiley@3x.gif, if present (Windows, Motif, Qt)

tile Ti2 „smiley.ico“  // only Windows! ICON mechanism of Windows
                       // takes the best image for the resolution level
```

```
                         // from the .ico file

tile Ti3 „:/smiley"      // only Qt! The resource mechanism of Qt takes
                         // the best image for the resolution level
                         // from the resource file with the given alias
                         //
```

9.8 Installation notes

To prepare an application for different resolutions, it may be necessary to keep the images for the different scaling levels. If you want to use this feature and not use the ICON mechanism of WINDOWS or the resource mechanism of QT, you should take care to deliver the image files according to the @ naming (see chapter "Support of HiDPI image variants") with your application or include them in the installation packages if necessary.

9.9 Geometry and coordinates

Coordinates and dimensions are transformed by the IDM to match the scale factor of the system to ensure a representation consistent with the system settings.

The geometry units are set in **IDM pixel values** or *raster* as before.

The **IDM pixel values** are internally extrapolated and converted into **real pixel values** according to the scaling factor. This scales the application as a whole and maintains ratios and proportions of objects to and among each other.

This conversion is done in both directions. When querying the size and position of objects as well as during event processing, the **real pixel values** from the IDM are also cleaned up and returned in **IDM pixel values**. The scaling factor is then calculated out again. This applies to all pixel-oriented geometry attributes.

Raster values are either linked to the *(HighDPI-capable) character set* used or are also set in **IDM pixel values** and then scaled up. The *raster units* can also be determined to **IDM pixel values** based on the underlying font.

9.10 High resolution support under Motif

The Motif toolkit itself does not contain any special technology to support high resolutions, but if the X display used has the Xrender extension implemented and there is support for XFT fonts with the corresponding fonts, IDM applications for Motif can be run on HighDPI screens respecting the scaling factor of the desktop.

What can be done to check whether support is available on the X-Display/desktop used? For this purpose, the server information of the X-Display can be queried via the **-IDMserver** option:

```
$ 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)
```

The version of the Xrender extension and its usability (active) is listed as well as the presence of XFT and its usability. The "Scaling" line shows the scaling factor set by the user in the desktop/ display settings, or the scaling factor overwritten by the user.

The dpi information provided by X is mostly the default value of 96dpi and thus does not necessarily reflect the real DPI value available on the monitor. The optionally output "tile dpi" and "scaled dpi" are used to control the DPI values used for the images and controls.

Dynamic scaling change is not provided for Motif applications. Likewise, a conversion of the scaling cannot be detected in the desktop.

XFT / Font Handling / Motif Font Support

For a GUI application, the basis for different desktop scaling is provided by the fonts used. In contrast to the fonts that could previously be used under Motif/X, which were mostly specified via an XLFD (X Logical Font Descriptor), XFT fonts are also scaled according to the defined scaling of the desktop.

Until now, the fonts that could be used for IDM FOR MOTIF had not automatically adapted to different desktop scaling.

XFT support is available in the IDM only on Linux platforms. Since most desktops on Linux platforms do not set the default font for Motif widgets properly, the IDM uses ***Liberation Sans*** *as the default font and* ***Liberation Mono*** *as the fixed-width font if no font is set in the dialog. This causes texts in IDM dialogs to scale even without a font set.*
By explicitly disabling scaling (via the -IDMscale=0 startup option), the old default fonts can be preserved.

To set an XFT font, only the font name and any additional modifiers are required, as with previous font definitions in the IDM. If the font is available as an XFT font on the system, it will be loaded. The list of XFT fonts available on the system can be viewed in the IDM EDITOR.

Under CDE-compliant platforms such as AIX and HP-UX, the IDM uses “-dt-interface_system-medium-r-normal-*” as the default font and “-dt-interface_user-medium-r-normal-*” as the fixed-width font if no font has been set. Depending on the scaling factor, different sizes of the font are used. Between *1%...75%* **xs** (extra small) is used, between *75%-150%* **m** (medium) is used, between *150%-250%* **l** (large) is used and from *250%* **xxl** (extra extra large) is used. This allows scalability for such applications even if no XFT fonts are available.

9.11 High resolution IDM support for Windows 11

There is now one ***IDM for Windows 11*** and one for ***Windows 10***. The IDM FOR WINDOWS 11 supports DPI awareness and High DPI.

The IDM FOR WINDOWS 11 supports high resolutions of the “PerMonitor V2” model. Since the support of high resolutions is a feature of the application, it must be marked accordingly in the application manifest. Corresponding settings can be found in the IDM examples. The IDM then recognizes this independently and converts the coordinates accordingly to the configured scaling factor. In other words, the pixel coordinates of the IDM are virtual pixels that are converted according to the defined scaling factor.

The manifest files of the IDM examples mark the built applications as “High DPI-Aware”. To build an application that is not “DPI-Aware”, the entire `<asmv3:application>` block must be removed from the manifest file.

Notes IDM for Windows 11

Note for all functions that process Microsoft Windows messages: If the application supports high resolutions, then all coordinates received via Microsoft Windows messages are the real high-resolution coordinates. The IDM, on the other hand, uses coordinates adjusted for the scaling factor. It must therefore be taken into account that in the case of support for high resolutions, the IDM coordinates are generally no longer identical to the Microsoft Windows coordinates. This affects, for example, input handler, canvas, monitor and subclassing functions, as well as the USW implementations.

Notes IDM for Windows 10 and 11:

The support of high resolutions requires a different processing of Windows messages. As a result, this means that it is no longer possible to adjust windows to underlying constraints, such as grid coordinates, while moving or zooming. Only after releasing the mouse button the window is adjusted to the next grid coordinate.

Internally defined cursors are now enlarged or reduced to the size required by the system in order to automatically adjust them to the defined scaling factor. This ensures that the cursors are displayed correctly if the application supports high resolutions. Previously, the cursors were either filled with transparent areas or simply cut off if the size did not fit.

Images (tile resource) are automatically enlarged according to the defined scaling factor. It is assumed that the images are designed for a DPI value of *96*. If the images of the application are designed for a higher resolution, then this can be set in the setup object with the attribute .tiledpi. Automatic adjustment applies only to the tile resource. Images specified directly via filename at the .picture attribute of the image object are not automatically adjusted.

The dialog or message font defined in the system parameters is now used as the default font, since this adapts to the resolution. This can lead to changes in the display of objects that do not have a font set. The mentioned font can be configured via Windows system settings.

The old system fonts are not DPI-Aware. Only the new ***UI*_FONT*** fonts adapt to the set resolution. The old Windows fonts “ANSI_FIXED_FONT”, “ANSI_VAR_FONT”, “DEVICE_DEFAULT_FONT” and “SYSTEM_FONT” cannot be adapted to different resolutions, so their use is discouraged. The use of fonts without size specification is not recommended, because in such a case a font-specific default size is chosen by Windows font mapper, which does not adapt to different resolutions.

9.12 High resolution support under Qt

To support high resolutions, the IDM enables Qt-side HighDpi scaling as well as the possible rendering of images beyond their actual requested size. I.e. the scaling is done based on the pixel density of the monitor.

Among Linux desktop environments, the support for HighDPI is unfortunately very different and advanced. Since the values for logical and actual pixel density as well as scaling factors are obtained from the Qt framework, less popular desktop environments may provide insufficient values here.

Notes for HighDPI preferences

Qt still offers some environment variables that can be used to control the display at high resolutions. When using such environment variables, however, you should be aware that this overrides the behavior of the IDM, which can lead to unwanted display effects.

Remarks for tile scaling

Objects are displayed by default with the values specified by the desktop style by the WSI/Toolkit. It can therefore happen in certain situations that the desired settings regarding the size and scaling of tiles are not implemented. This applies to the use of tiles for objects with tabs such as the notebook or notepages as well as menus with menuitems. Although the IDM sets the desired display options, the final display is subject to the display policy of the respective DrawingEngine of the set UI style. This means that the display may differ from the set options depending on the style or options may be ignored completely.

9.13 Enhancements/ changes to the IDMED

Arbitrary selection of the font for the user interfaces or rule code is no longer possible. Instead, the font size can be selected between Small - Normal - Large - Extra Large. Of course, this has an influence on the window size.

Under Motif, XFT fonts are also listed in the fonts setting dialog. In addition, the list of selectable fonts can be reduced to UI fonts, X fonts, and XFT fonts for clarity.

The IDMED, TracefileAnalyzer as well as the debugger now use UI resources.

9.14 High-resolution support for USW programming

The DM_GetToolkitDataEx function has been improved to support HighDPI. The Toolkit datena-S strukcture DM_ToolkitDataArgs has been extended for this purpose and now provides detailed information about DPI, scale factor and image. These can be requested via the *AT_DPI* and *AT_XWidget* attributes.

Note Motif

The USW programmer must convert the coordinates between the IDM (this concerns attributes as well as events) and the Motif values/structures/functions (*X.../ Xt.../ Xm...*). The data required for a conversion can be requested via function DM_GetToolkitDataEx using AT_DPI and the DM_ToolkitDataArgs structure (*tile.scale.factor*). Depending on the image type, scaling may also be required.

Note for Windows 11

The IDM for Windows 11 supports high resolutions. If an application is released for high resolutions, then the USW objects it uses must also support high resolutions. This means that the USW objects from Windows will have real coordinates (scaled up by the scaling factor), while the IDM will work with virtual coordinates (not scaled up). Practically, not much will change, since the IDM performs the coordinate calculations for the USW object. But it should be taken into account that the **control** call for “UC_C_PrefSize” expects real coordinates, since these are normally calculated from toolkit values. If fixed numbers are used here as in the *ucarrow* example, then these must be corrected by the current DPI value (see ucarrow example).

If the application was started as DPI-Aware, the task “UC_I_clientarea” of the function **UC_Inquire** now also expects the high-resolution values.

Extensions to the C Interface for DM_GetToolkitDataEx and DM_GetToolkitDataEx()

In order to support high definition, the C Interface functions "DM_GetToolkitData()", "DM_GetToolkitDataEx()" and "DM_ToolkitDataArgs" (for toolkit data structure) have the following:

Attribute AT_DPI

Renders the DPI value of the system or the displayed object. The function "DM_GetToolkitDataEx()" must be selected with a pointer on a "DM_ToolkitDataArgs" toolkitdata structure. Further DPI related information is rendered in this structure. It is extended with the data field "dpi" and the sub data structure "scale".

Attribute "AT_Tile" resp. "AT_XTile"

This attribute already exists for the "tile" ressource. Now the function "DM_GetToolkitDataEx()" can be selected by a pointer on the toolkit data structure "DM_ToolkitDataArgs" that has been extended with the sub data structure "tile". This structure contains, among other values, the DPI value for which the "tile" ressource has been developed.

Attribute "AT_IsNull"

Renders a value unequal "0" if the specified "color" or "font" ressource has previously been defined to "UI_NULL_FONT" resp. "UI_NULL_COLOR".

See also chapter „Toolkit datena-S strukcture DM_ToolkitDataArgs“ in manual „C-Schnittstelle - Grundlagen“.

See also chapter „DM_GetToolkitDataEx“ in manual „C-Schnittstelle - Funktionen“.

See also chapter „DM_GetToolkitData“ in manual „C-Schnittstelle - Funktionen“.

Index