Mjolner Integrated Development Tool − Tutorial

Mjølner Informatics Report
MIA 91−40
February 2002

Table of Contents

Table of Contents

List of Figures

1 Introduction

This tutorial starts with a guided tour in using the Mjolner tool on an assignment. The guided tour demonstrates how the tool can be used for modeling, GUI design and implementation.

After the guided tour, more detailed tutorials are given to the components of the Mjolner tool: the source browser, the editor, the CASE tool, the interface builder and the debugger.

2 Guided Tour

2.1 Introduction

This online tutorial is based on a fictitious assignment, which will be described below. The tutorial will demonstrate the Mjolner tool, by showing how to build an application that solves this assignment.
The application is structured into three parts.

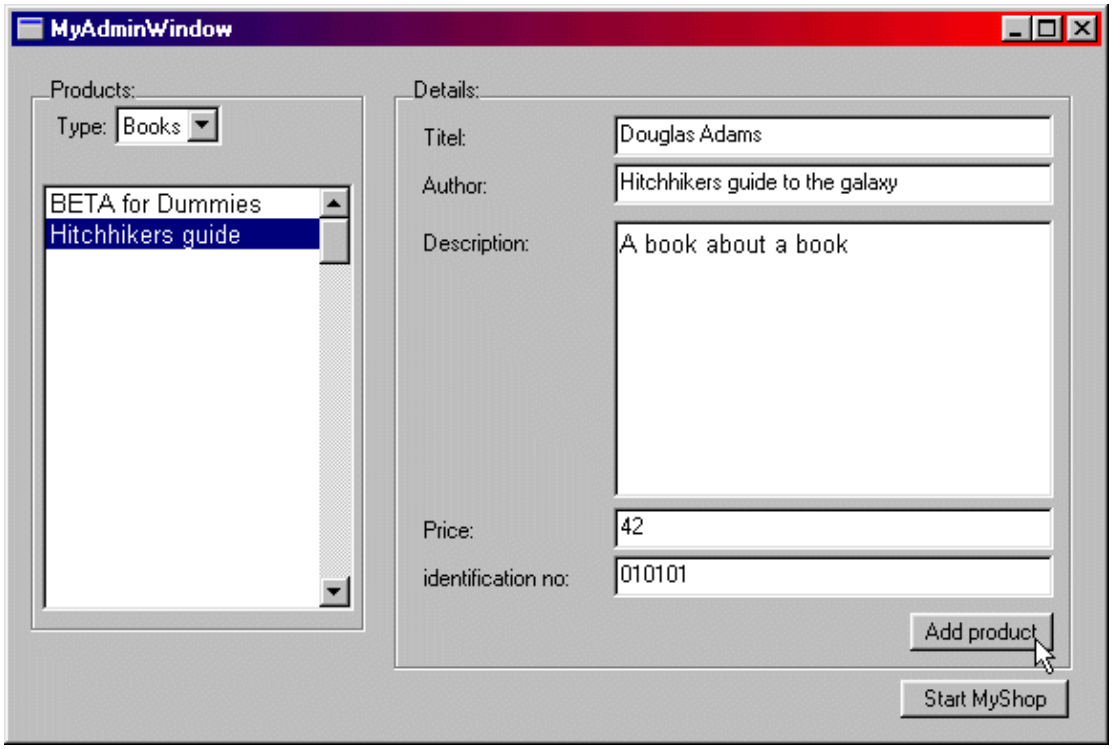
MyShop	The classes that define the objects in the model
MyGUI	The graphical user interface.
MyProgram	The main application which ties the model with the graphical user interface.

To get an impression of the code structure of the application you can look at the files MyShop.bet, MyGUI.bet and MyProgram.bet respectively.

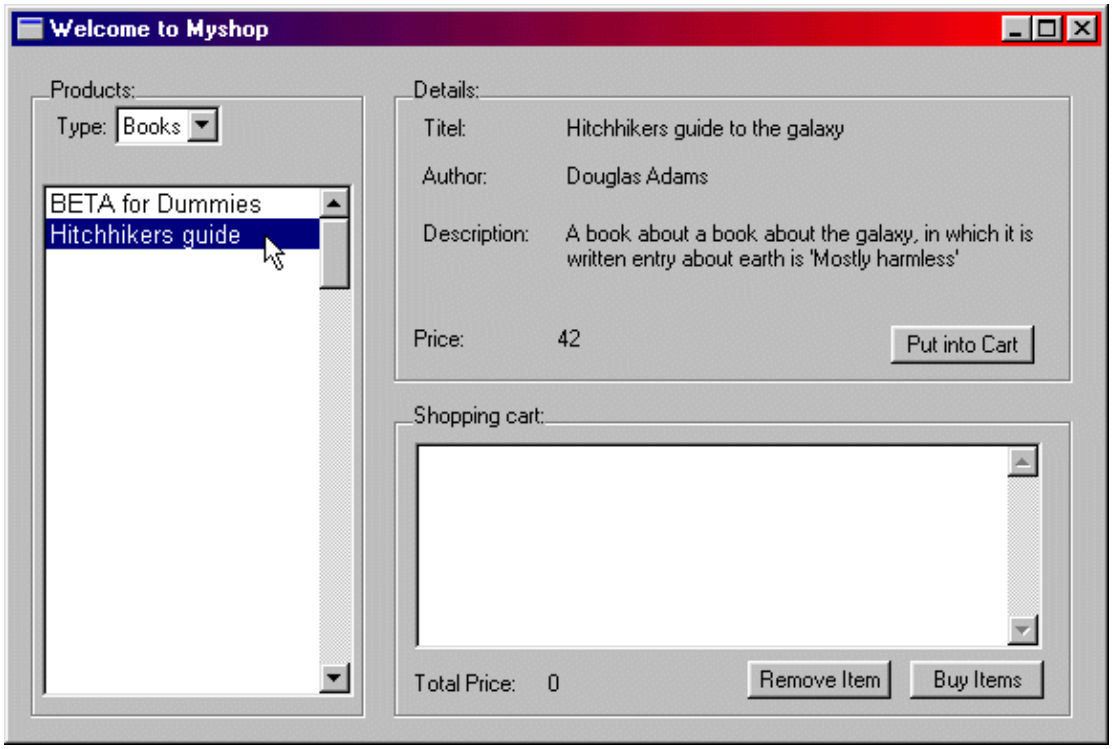
2.2 The Assignment

Picture you got an assignment from a bookstore, to engineer a system, which customers can use online in the store, to browse through books and CDs. The customer can then choose from the commodities, and put them into a virtual shopping cart. When the customer decides to buy, the system will then send a message to the clerk. The clerk then finds the selected products.
One solution to this assignment, is to make two separate windows (shown below), one window for administration of the system, in other words adding new books and CDs, and one window which is the interface to the customer. To make it simple a perfect world is pictured, in which people never make mistakes while inserting a new product into the system.

The first screen is the administration part of the program, where new products can be inserted. On the left is the list of products already in the system, the right is for inserting new products, and finally the start button is for starting the customer interface of the program.

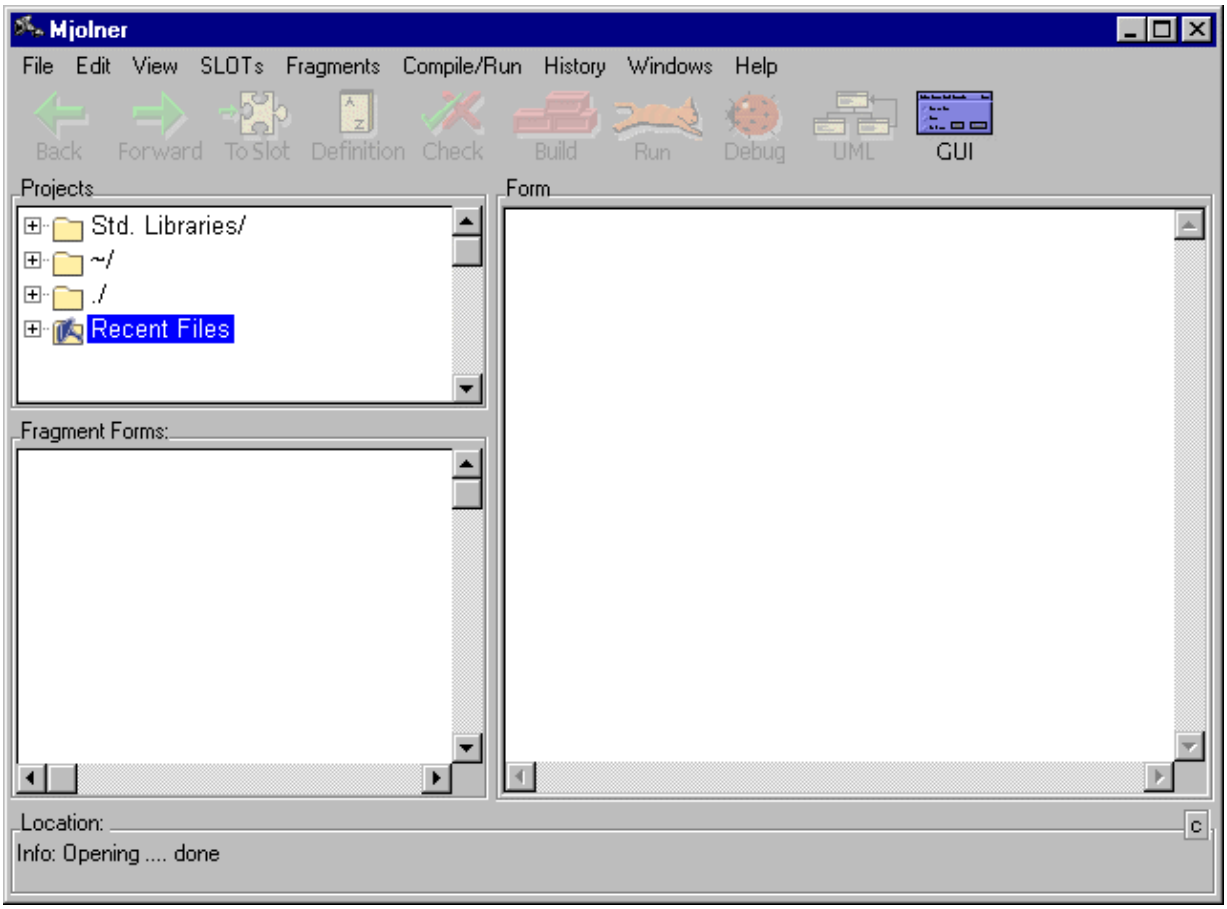


The next window is the sales window, where the customer can browse through the products, and enter them into the virtual shopping cart, which ends up with a print out at the clerk's desk. The clerk then finds the products, and make them ready for the customer.
Again the left box is a list of the products from which the customer can choose. If a product is selected from the list, the box named 'Details' will give the customer the details. In the bottom right is where the shopping cart is presented, with a list of items currently in the cart, the total price, and a button 'Buy Items' which sends the printout to the clerk.

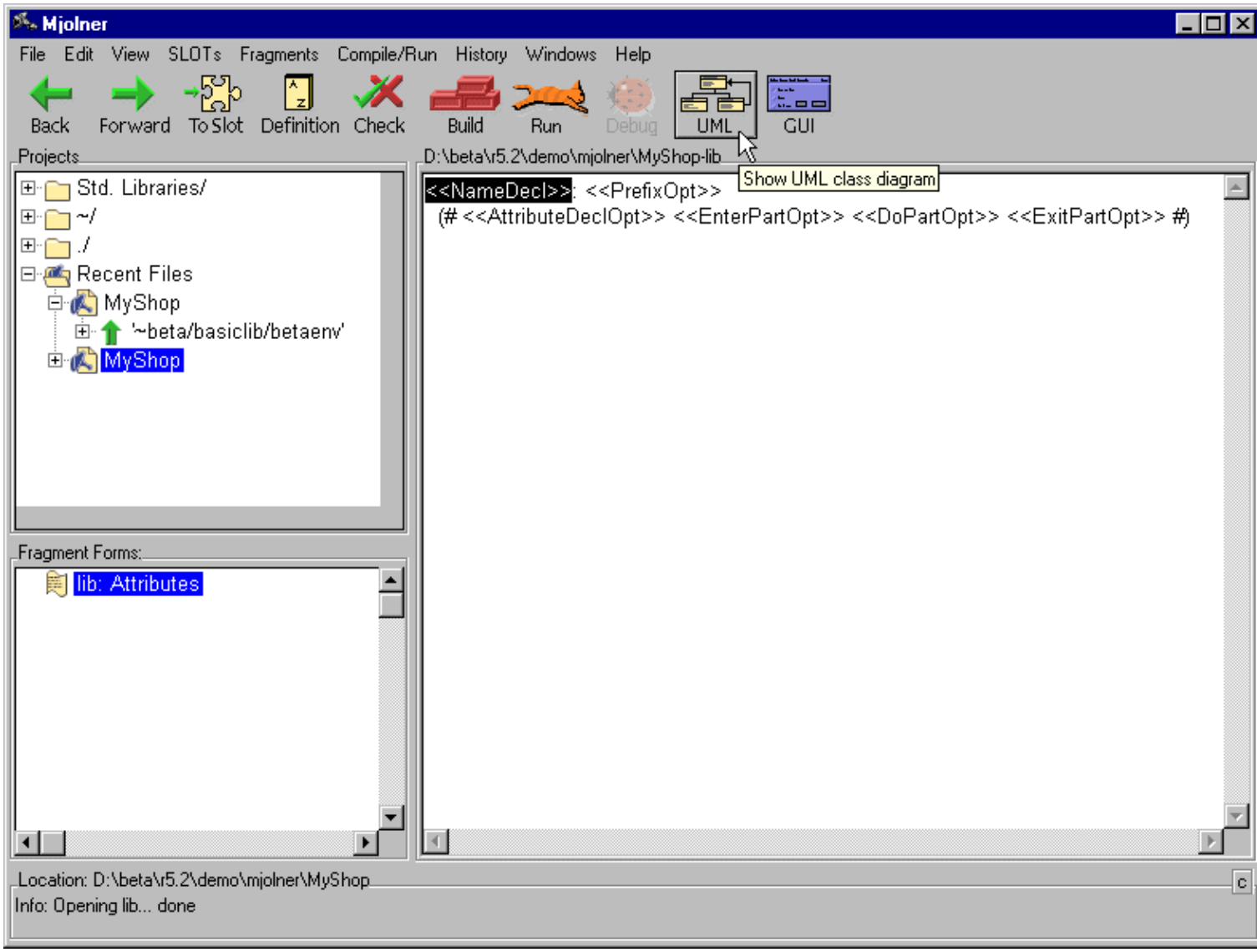


2.3 Getting Started

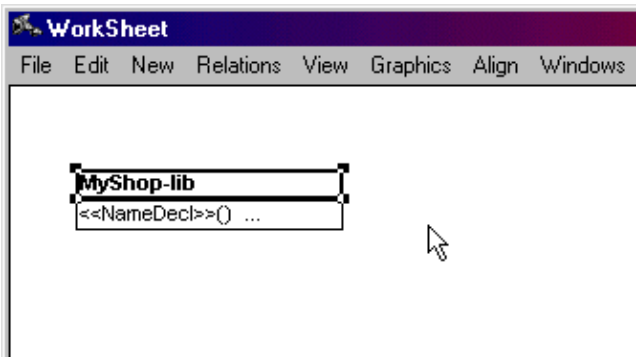
To start engineering the application, the part of the real world (referent system), which need modeling, is to be analyzed. This analysis will be the basis for the model(Model system). For the purpose of making this model, the diagram editor *Freja* is used. When you start the Mjølner tool, you will get a window as shown below. This is the source browser and editor (Sif). When making a diagram using the diagram editor (Freja) or graphical user interface (GUI) using the GUI−editor, the source code generated will appear in this window.



You have to create a new library file using the **New BETA Library...** command in the **File** menu. The library file used here is called MyShop, which results in:



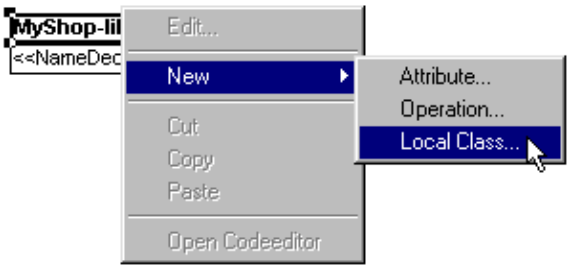
To start up the diagram editor you click on the **UML** command in the toolbar (as indicated in the picture above) and a digram work sheet will appear. With this work sheet it is now possible to make the model, while the source code is generated in the code editor, as you will see.



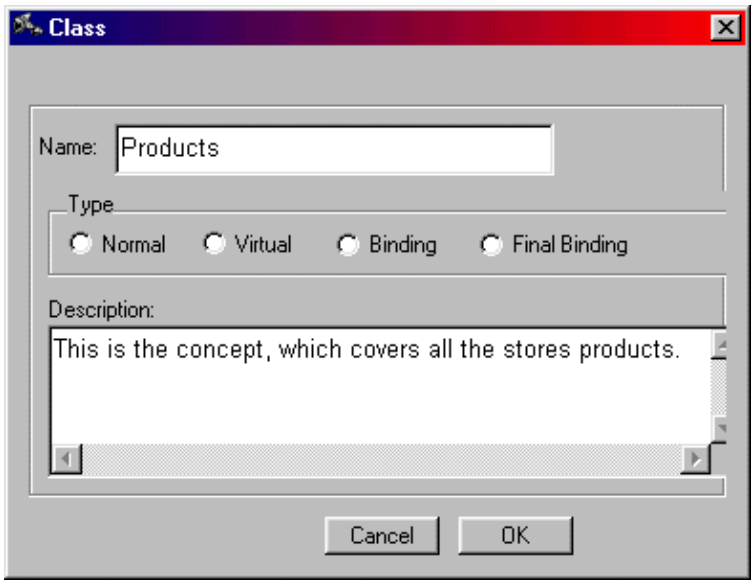
2.4 Modeling

Now look at the assignment. Which concepts of the bookstore, should be considered as classes in the model.
Of course the products need to be modeled. Also a shopping cart, should be modeled. It will now be illustrated how this is done.

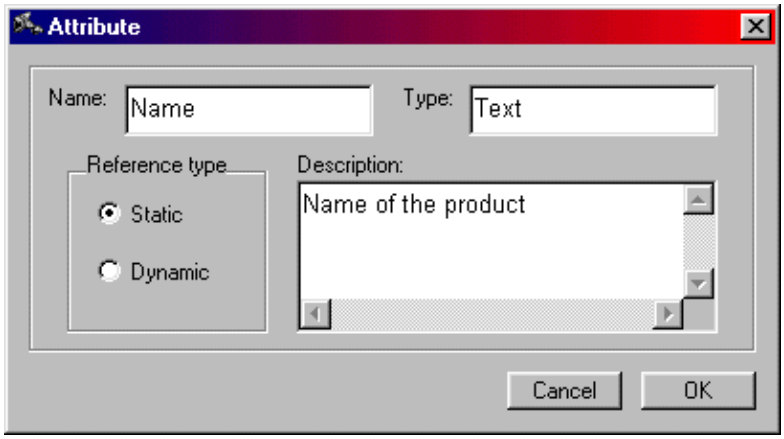
To make a class corresponding to the concept product, you first right click on the **MyShop−lib** box, Then select **New** and then **Local class**.



A popup window, will now allow you to make your new class. First, I make the class 'Products' and next make the class 'Shopping Cart'.

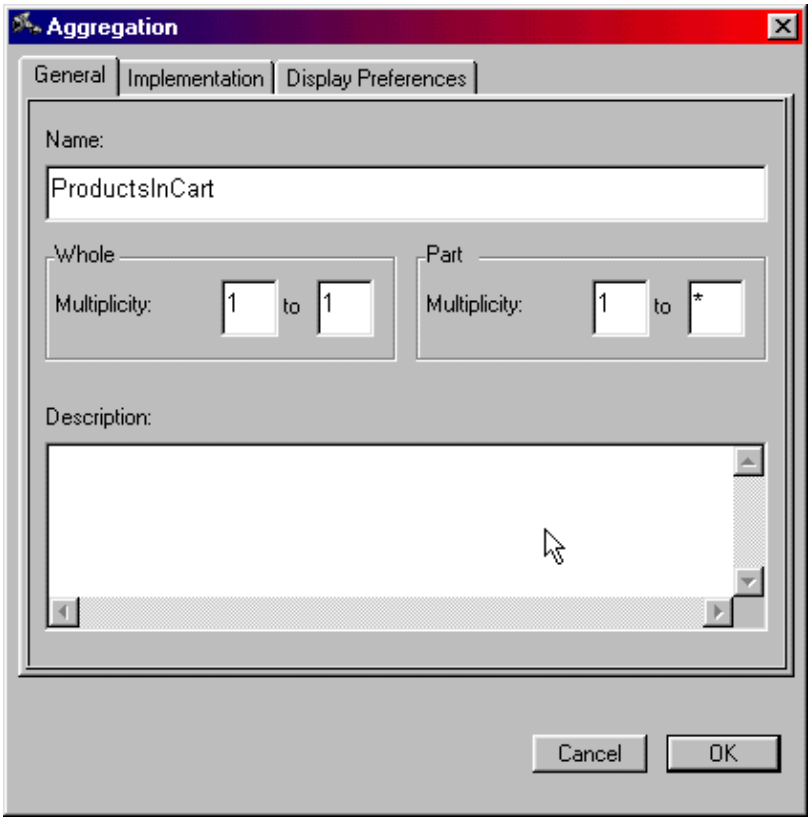


Now lets look at what classifies these concepts, First 'Product': a product has a name, a price and an unambiguous identification. A virtual shopping cart, should have a way of keeping track of the products currently in the cart, and for this shopping cart a total price.
To insert these into the model, you right click on the 'Product' box select 'New' and then 'Attribute', In this popup window, you can now write the name, type of the new attribute, The picture below shows how the product attribute name.

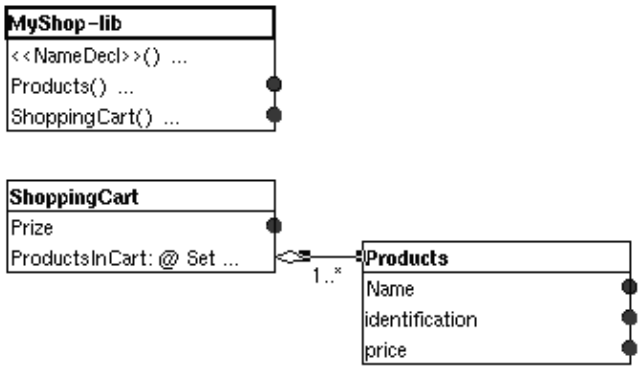


The rest of the attributes are inserted in exactly the same way. except for 'productInCart' in the

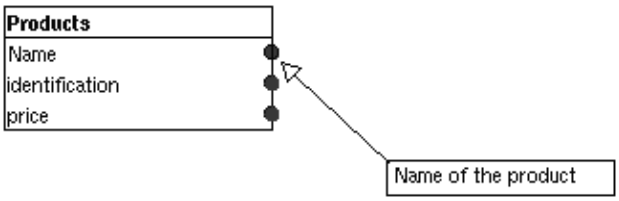
'Shopping Cart', which is an one−to−many aggregation between the 'Shopping Cart' and the 'Product'. The way you make an aggregation, is to select **Relations** in the menu bar, then **Aggregations.**



In a popup menu like shown above, it will then ask you to give the aggregation a name. For lack of a better name, here its called 'ProductsInCart'. Then select the multiplicity of the whole and the part. The multiplicity of the whole is of cause only one, and the part is one−to−many, which is written as 1 to *.
If we now take a look of our model, it should look like

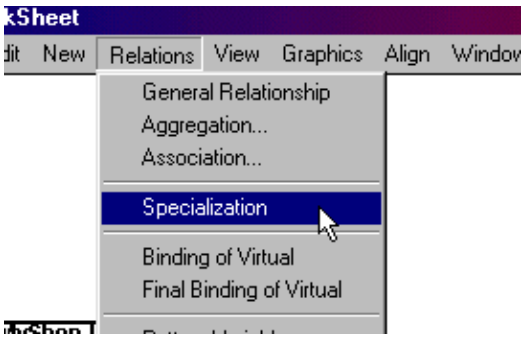


The line from the 'Shopping Cart' attribute ProductsInCart to the 'Product' class illustrates the one−to−many aggregation between them.
The black bullets on the right side of the boxes indicates a description connected to the attribute or class. by double clicking on one of the black dots, a new box appears like shown below. Which is the comment connected to the attribute 'name'.



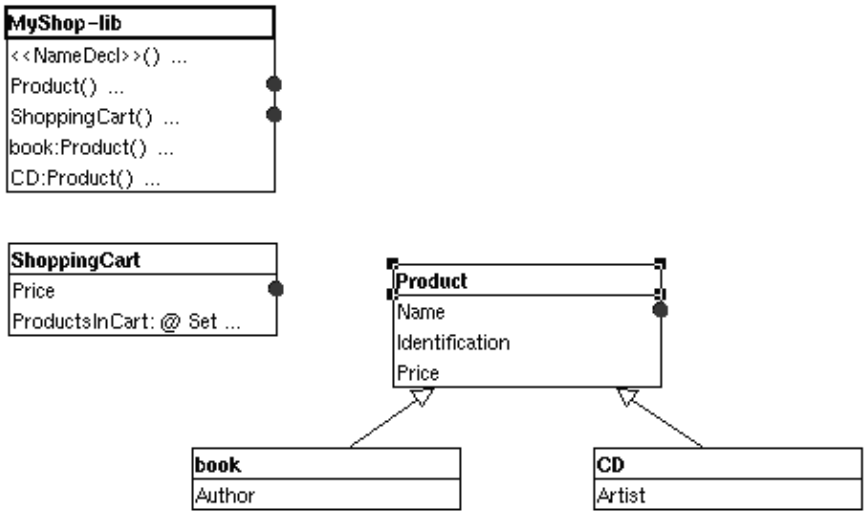
Now look at the products, which the bookstore wants to sell through this program: books and CDs. An important information about a book is who has written it, and for CDs who is the artist, so we need two sub−classes, one called book and another called CD.

To make a subclass of a superclass. First make the class which is desires to be the subclasses. This is done like when the 'Products' class was made. Now make a 'Book' class and a 'CD' class. the defining property about a book is the author, and artist for a CD, add these attributes to the new classes, like it was done with the attributes in the 'products' class. Then use the menu **Relations** and the **specialization**,



now left click on the superclass and right click on the subclass. In this case left click on 'product' and right click on 'book'. Then repeat the procedure to make 'CD' an subclass of product to. The 'book' and 'CD' classes are now specialization's of the concept 'product.

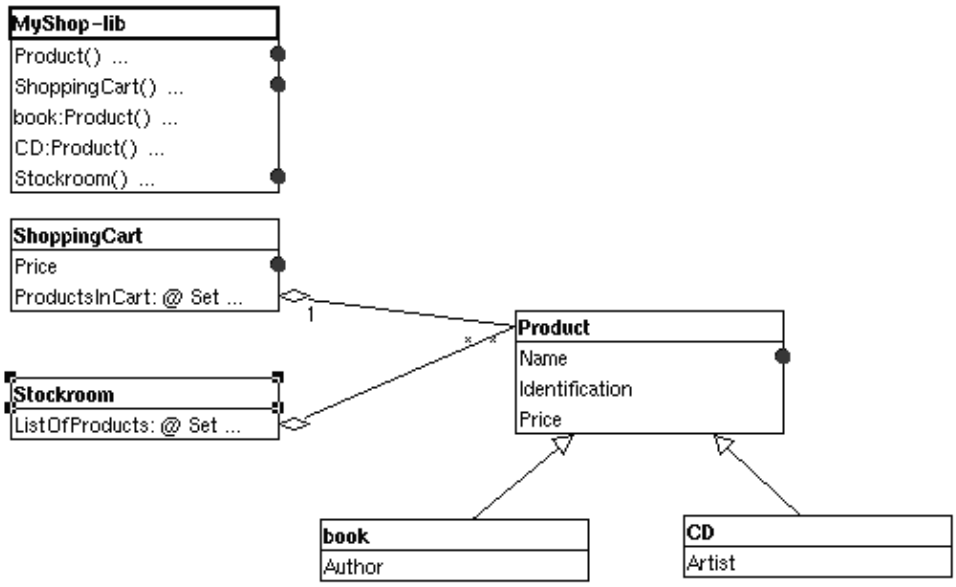
The model should now look like:



A common book store usually have more than one book or CD to sell so the application needs to keep track of a list of products.
Now look at the model, it is clear that this list should hold both books and CDs. Here the object oriented language (OOL) comes in handy, because the concept book is a subpattern of product, and will inherit all its properties, as well as the concept of CD is a subpattern of product, which

means it is enough to make af list of 'product' objects, then the list will both hold books and CDs.
In the real world all the products will be in a stock room. So what we need is a Stock room class. To make it more simple, the stockroom will always have a copy of the books and CDs, so there is no need to register to number of copies.
By using the same procedure as with the 'ShoppingCart' class, it's easy to make this class. Just make a new class, called 'Stockroom', and make an aggregation, like the one between 'ShoppingCart' and 'product' only here its between 'Stockroom' and 'product'.

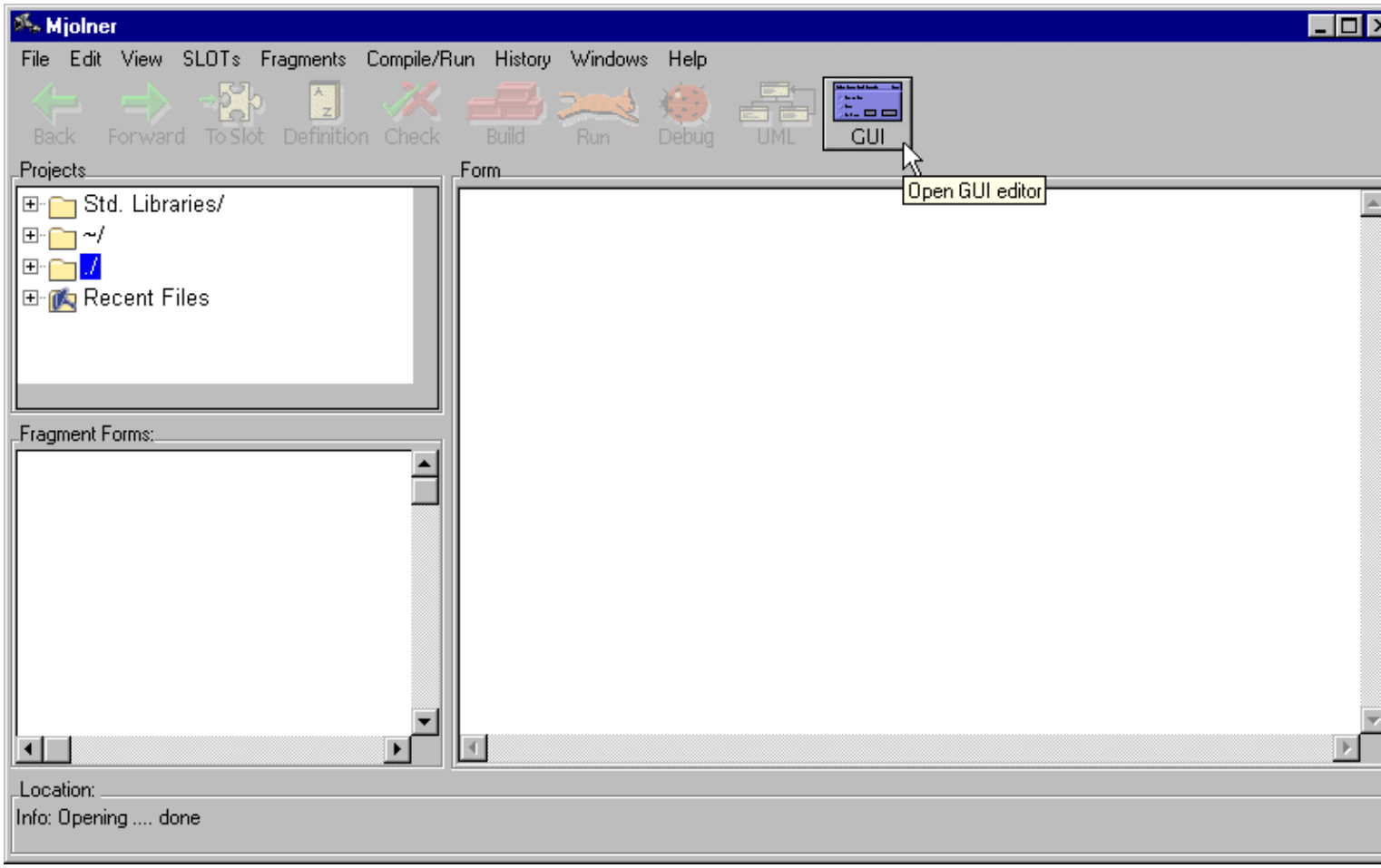
The final model should now look like:



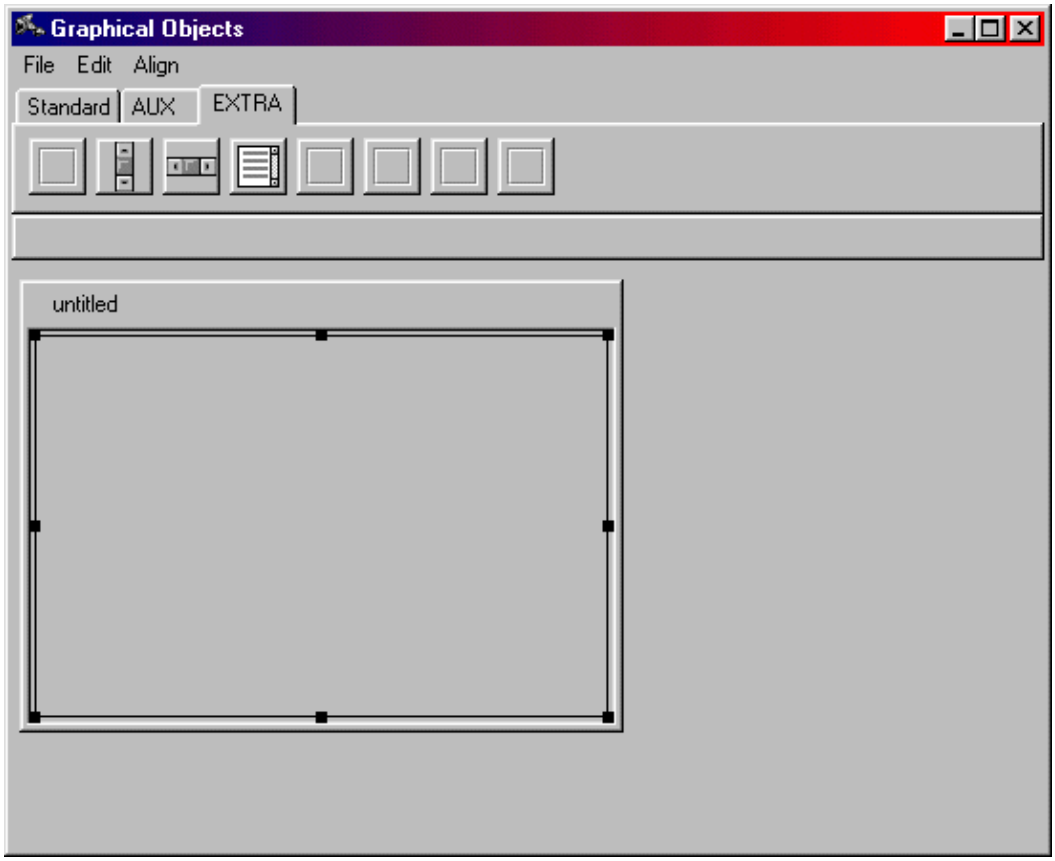
This analysis/modeling is of cause far from complete, but it suffices for the purpose of showing, how the most general functions of the CASE−tool Freja works. Now to the design of the graphical user interface (GUI)

2.5 GUI Design

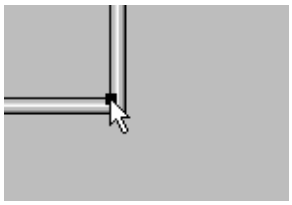
From the code editor, you can start up the graphical user interface (GUI) editor. This can be done by using **GUI** command in the toolbar. Make sure the content of the **Form** pane is empty before you select the **GUI** command, e.g. by clicking on a folder in the treeview. In this way a new library file will be created for the GUI elements.



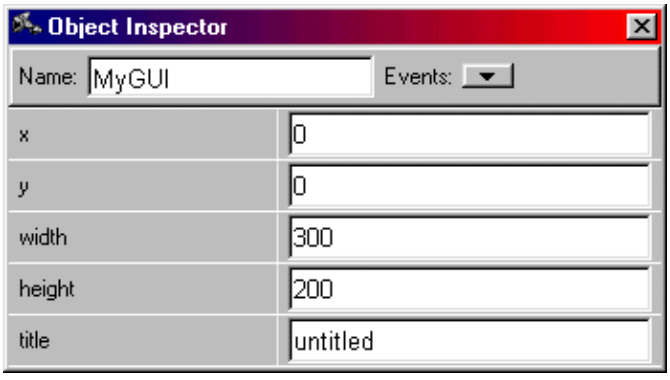
Now the skeleton source code of a window is shown and the graphical editor activated. The start window should look like:



The source code generated in code editor, is a subpattern of a general window pattern, and it thereby inherits the properties of the window pattern.
To start making your GUI first resize the window to the desirable size (it is of course possible to adjust the size of the window later), This is done by clicking inside the window, this will activate the window object, and you can now resize the window by dragging one of the small black boxes.



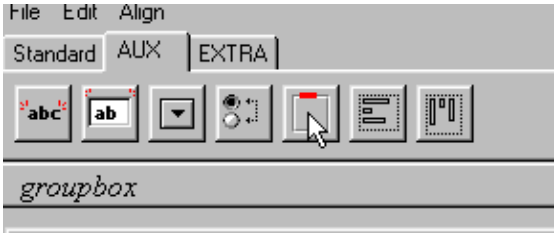
Another way of doing it is by double clicking on the window, and thereby evoking the object inspector. which looks like



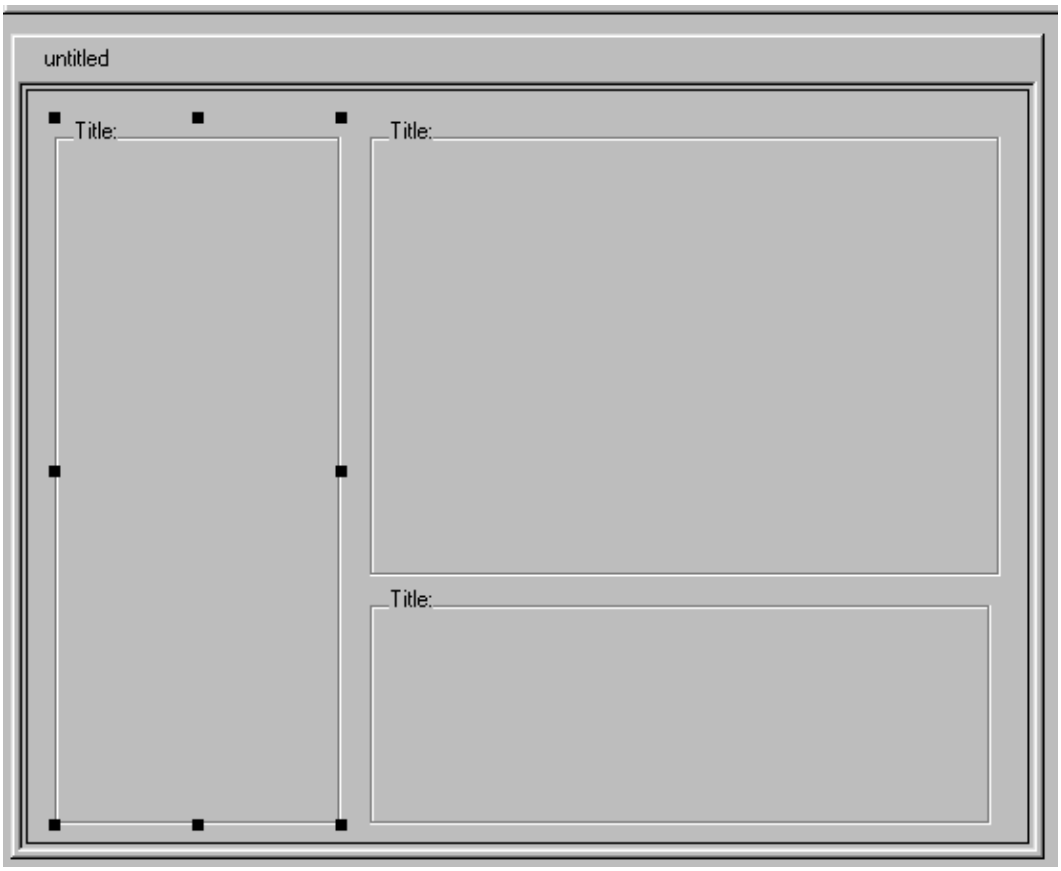
With the Object inspector, it is possible to insert the exact height and with of the window. The text field in the top of the inspector, is used to edit the name of the sub pattern which generates the

window. I will get back to the 'Events' option button later.
the x and y rows, is used to position the object relative to the upper left corner, which is not relevant for the actual window.
The title is used to give the window a name, which in the picture above is 'untitled'.
Now to design the window, again take a quick look at our model To make it easy for the common user, use one of the basic OOD criteria and try and make the concepts of the referent system (the real world) visible in the graphical user interface (GUI).

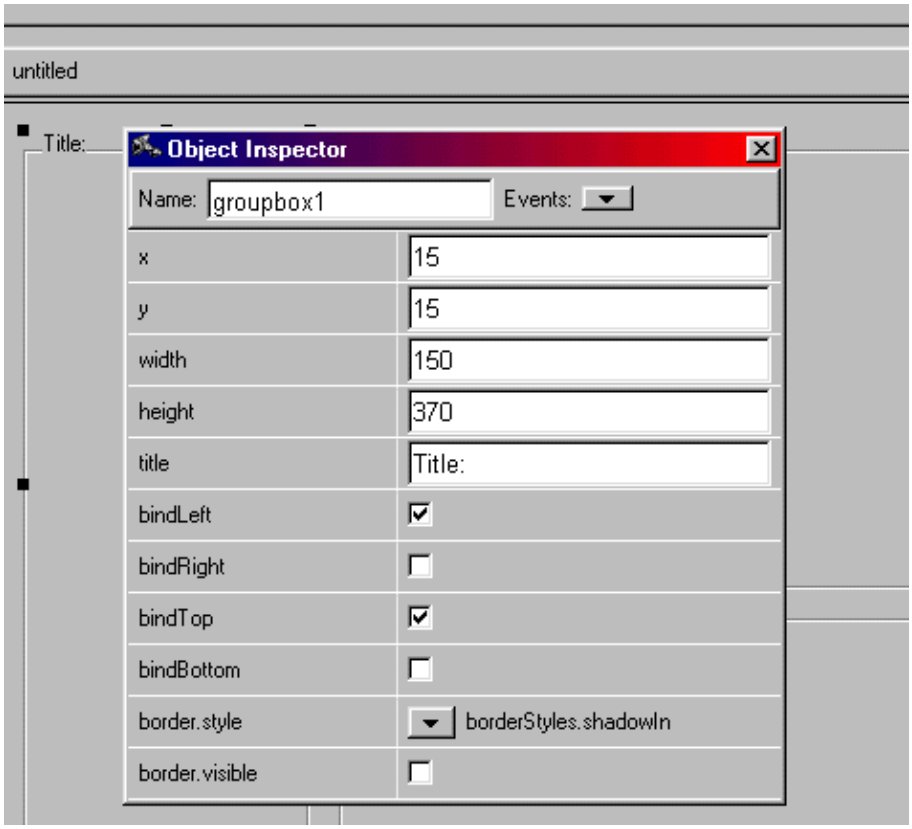
The main concepts from the model are 'stockroom', 'product' and' shoppingCart'. For each of these concepts, make a closed area, this can either be done by a canvas, or a groupbox, which is a subpattern of a canvas. basically they are the same, except the groupbox, have a title in the upper left corner, and the canvas have not.
By placing the mouse arrow above one of the icons placed, the name of the object the icon represents will be posted. Like shown bellow, where the pointer i placed on the 'groupbox' icon.



The groupbox icon is placed under the **AUX** tab, by pressing the left mouse button, it's possible to drag a groupbox onto to window.
After resizing the window and dragging three groupboxes into the window, the window should now look like the one illustrated:

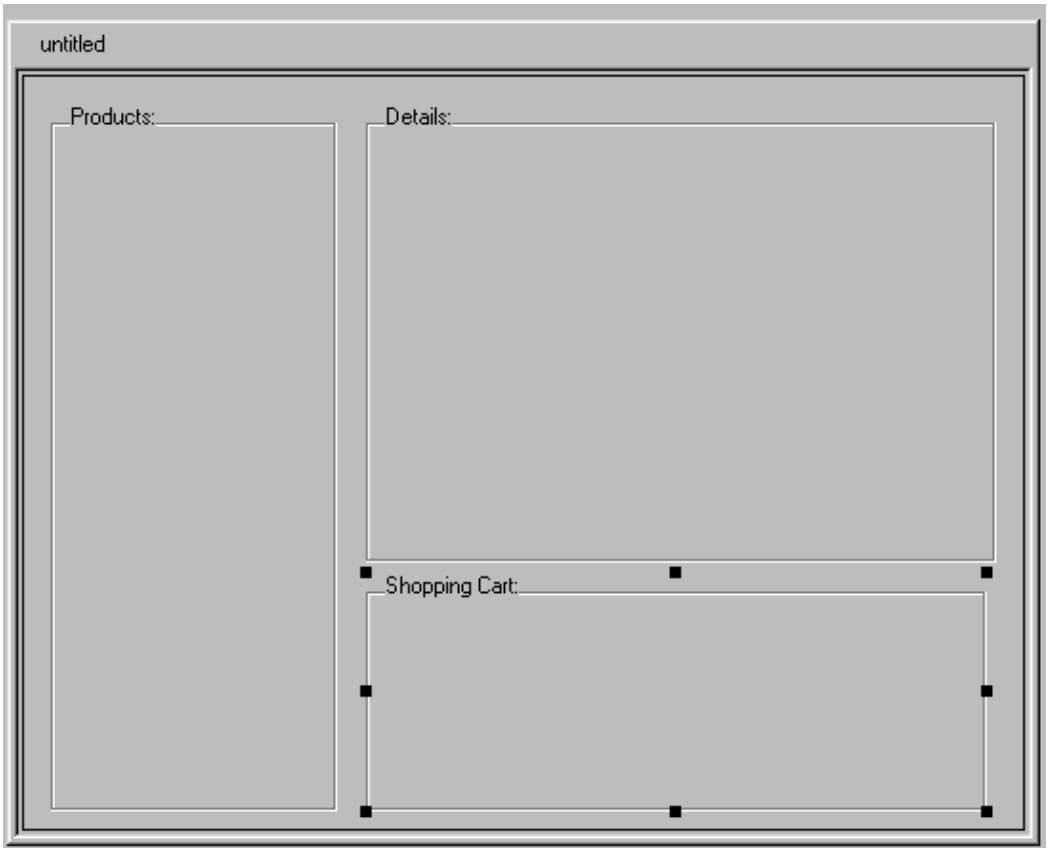


Now it is time to use the object inspector again, this time launch it on the leftmost groupbox, by double clicking inside the it.



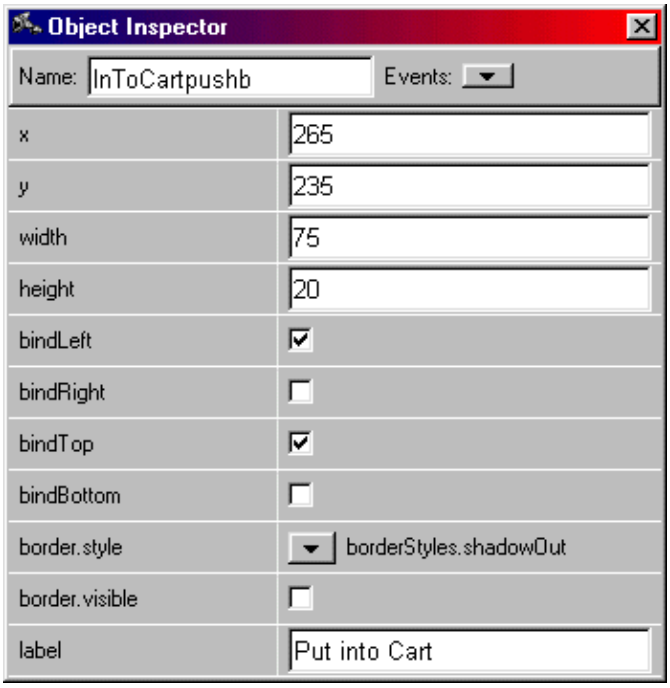
Compared to the object inspector shown earlier this has more tupels. Again the 'name' field in the top is the name of the object in the source code, the x and y are coordinates with (0,0) in the upper left corner, x gets bigger downward, and y to the right. The width and height the dimensions of the object. The title, is the title of the groupbox.
When a window is resized, the objects in the window can be bound to one or more of the sides of the window. This is controlled with the next four tupels. Border.style and border.visible help you define the looks of the border of the groupbox.
To make the GUI, and the source code more understandable, both the name and the title should be changed, to something more meaningful, and usually its a good idea to make the names close to each other, this will make it easier to make the connection between the code and the actual window, to the programmer.

The illustration below, shows how it looks, after changing the titles on all three groupbox (to Products, Details and Shopping Cart) and the title of the window has been changed to 'My BookStore'.

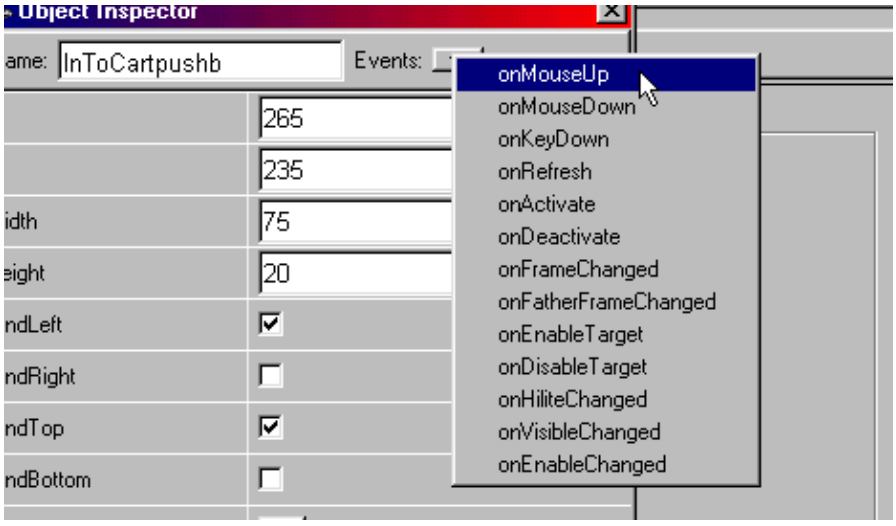


It is now time to insert the edittext fields, buttons etc....
Inserting new objects are done like shown above, as an example it will now be shown how to insert a pushbutton, and giving it an event.
First drag and drop a pushbutton into the window:

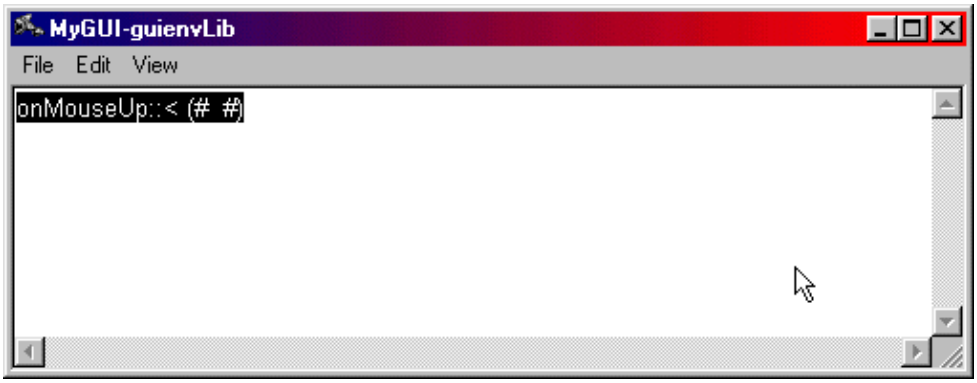
Again double click on the object to invoke the object inspector (se bellow), The name of the object is changed to 'InToCartpb' (pb short for pushbutton), and the label on the button to 'Put into Cart'



It is now possible to associate an event to this button, by using the **Event** menu in the object inspector,



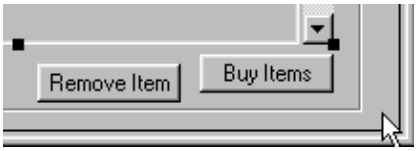
When an event is chosen (here the event chosen is onMouseUp) a sub−editor will appear.



In this window, it is now possible to connect the event of pushing the button, to some BETA code.

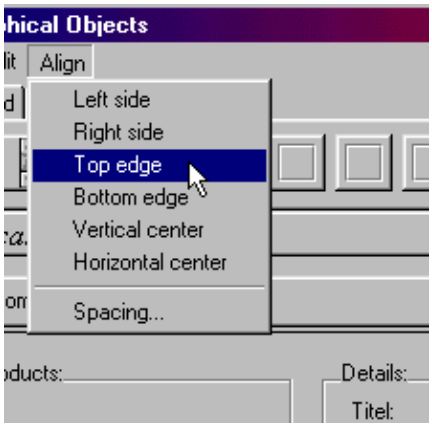
To make the rest of the window like the one in the beginning of this tutorial, is by drag and drop the items as just shown with the push button.

A feature in the interface builder, to help the engineer, is the align function. Look at the two buttons in the image bellow, they are not aligned.

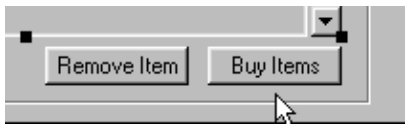


To align the two buttons, first select, the button which the other shall be aligned with. then press <SHIFT> and the select the other button.

Now use the **top edge** in the **Align** menu



The buttons will now be aligned with the object first selected. The alignment is made with the top edge of the button



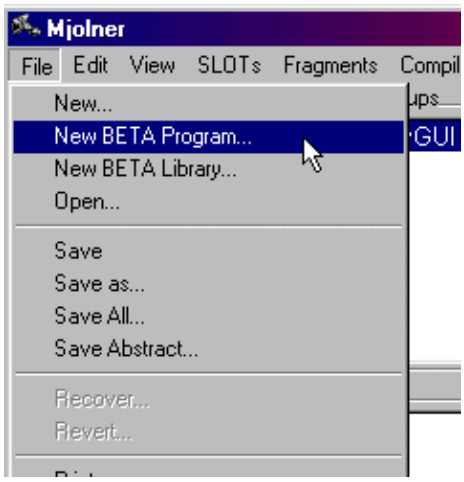
When the sales window is finished, close down the GUI−editor. To start the administration window it is not necessary to make more window library files, simply use the **Create window** in the **GUI menu,** and a new empty window, is ready.

The administration window, also shown in the getting started section is made the same way.

2.6 Implementation

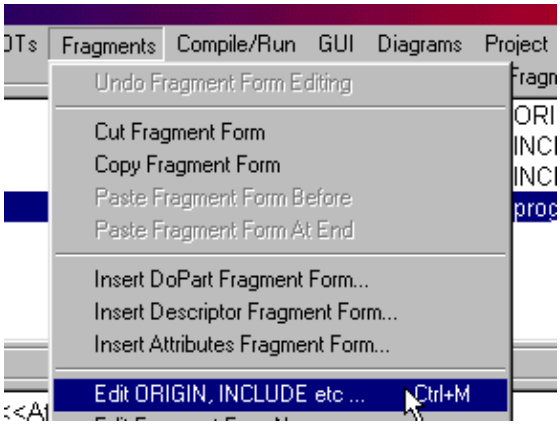
Now the simple model, and the graphical user interface is finished. It is now time to put the fragments together and build the final application.
The final element needed is an executable BETA program. The files made so far are what is called lib files. These files can be seen as personal additions to the standard patterns and objects in the basic environment (betaenv), and can be used in the same way in the BETA program, if included.

To make the Beta program, use the source code editor and select **New BETA program** in the **File** menu. This will generate a new BETA program. Again it will ask for a name, We called it MyProgram.

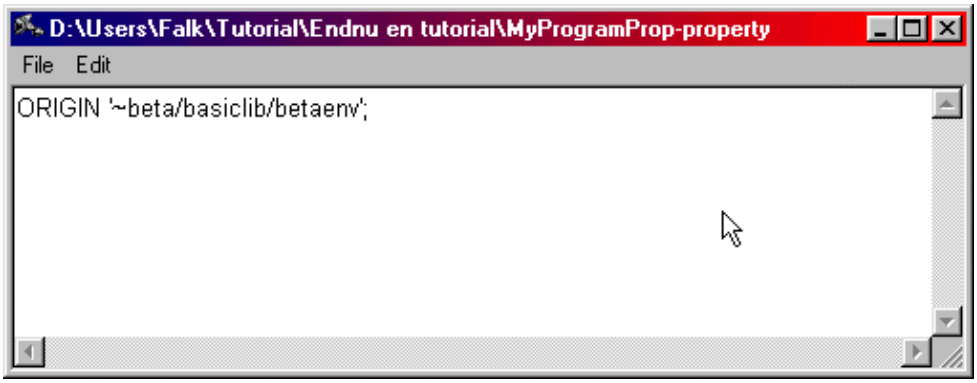


If you look at the fragments window in the upper right corner of the code browser, you will notice that, instead as in the MyShop file (the model file) and MyGUI file (The GUI file), where it said either *lib: Attributes* and *Guienvlib: Attributes*, it now says *Program: Descriptor*.

To include the lib files in MyProgram's environment, they have to be inserted into the fragment list, which is shown in the upper right corner of the code editor. This is done by using either **Edit ORIGIN, INCLUDE etc.** in the **Fragments** menu.

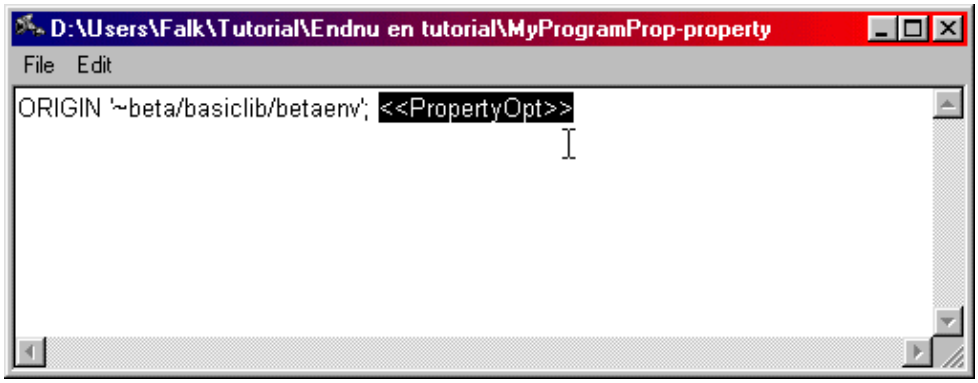


A sub editor will appear

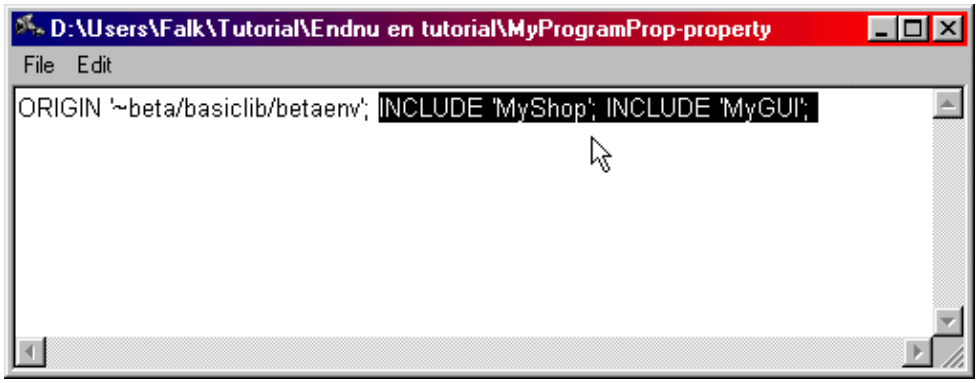


Inserting new text into the fragment list, is now done by first using the **Show Optional Non terminals** in the **Edit** menu or pressing <CTLR> and <I> this will prompt an non terminal (Non

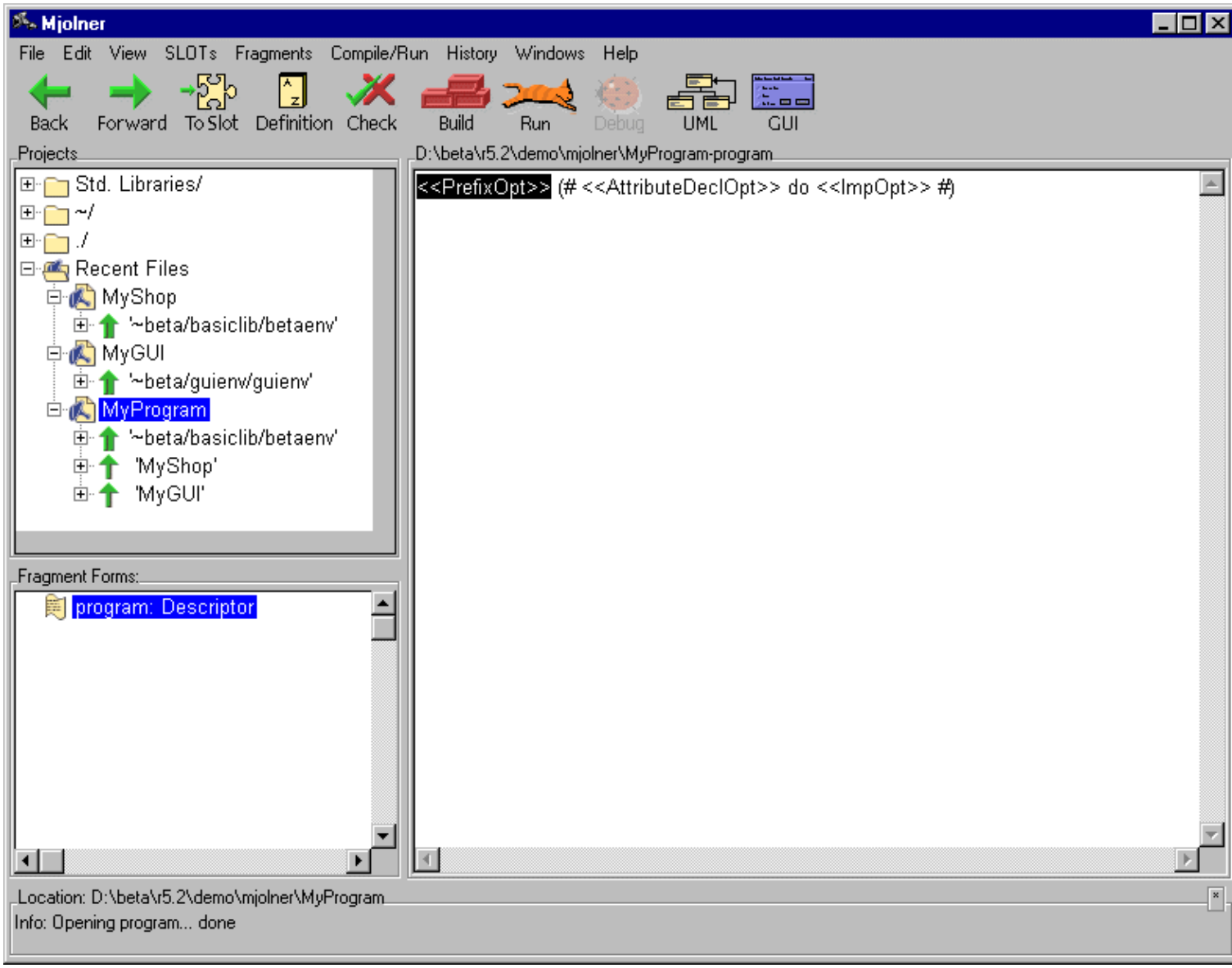
terminals are always shown as << NonTerminal>>),



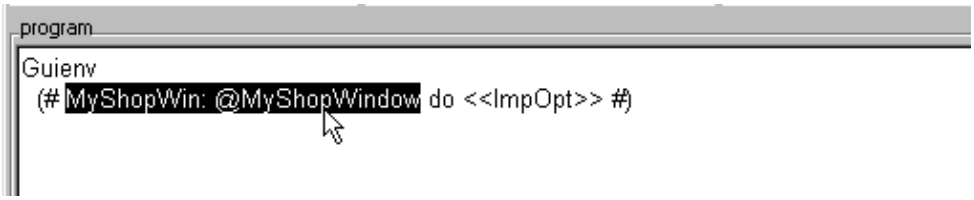
in which it is possible to write the the additional text, Highlight the optional non terminal as shown above, and start writing, the code editor will then automatically go into **Edit text** mode, this mode can also be entered by using the shortcut <CTLR> and <SPACE>. The text which should be inserted is:
INCLUDE 'MyShop';
INCLUDE 'MyGUI';
To get out of **Edit Text** mode simply press <CTLR> and <SPACE> and the fragment list should then look like



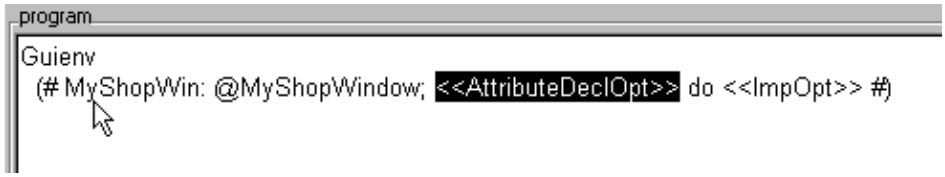
When the files are included, this window can now be closed down. And you should now turn to the code editor.



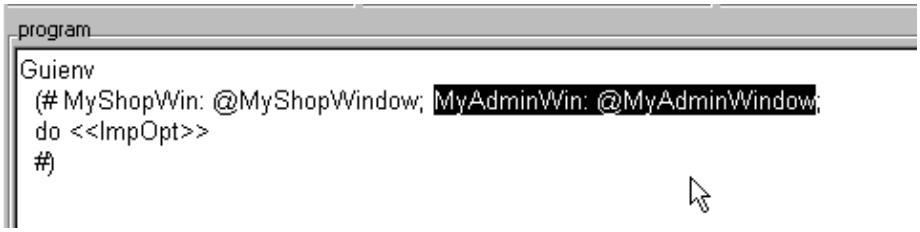
First make the program af subclass of GUIenv, by highlighting the <<Prefixopt>> like shown above, and writing 'guienv'.
then to make use of the windows designed earlier in this tutorial, we want to make instances of the first an instance of the sales window. Now highlight the <<AttributeDeclOpt>> either by using the arrow buttons or with the mouse. Then write 'MyShopWin: @MyShopWindow;', the program should now look like this



To insert an instance of the Administration window, highlight like above and simply press <ENTER>, this will make a new <<AttributeDeclOpt>>, like shown here



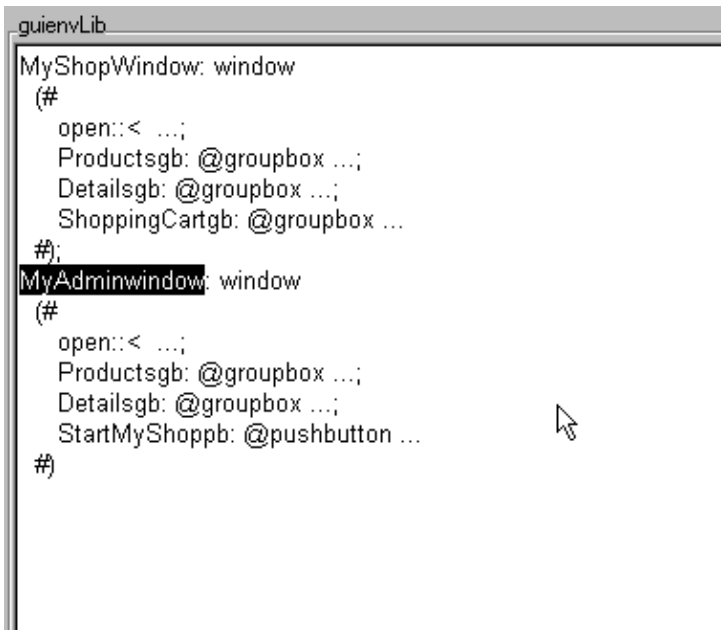
To make the instance of the Administration window, simply write as shown bellow.



Now there is an instance of the MyShopWindow and an instance of the MyAdminWin.These instances of windows will not be opened without a call to the *open* method in the Window pattern.

2.6.1 Browse Code

The Code editor, can also be used to browse the code, which will comes in handy, as soon as the programs gets bigger. In the example above with the windows, if you are curious about what the *open* method does. The normal thing to do is to track the actual window pattern by opening the files, in which it is inherited from, this can be a time consuming affair. In this editor simply, double click on either the MyShopWindow or the MyAdminWindow, and the browser will go to the file in which the window pattern is declared. Before this can be done the code has to be checked by Mjølner, to do that all the Optional Terminals has to be removed, this is done by using the shortcut <CTRL> and <n> to **Remove Optional Terminals**. When the Terminals have been removed simply double click on MyAdminWindow. You will be asked if you want to check the code. If the code is checked the browser will now find the declaration of the MyAdminWindow, in the 'MyGUI' file.



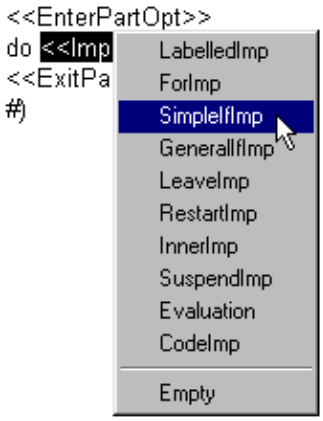
Now the *open* method can be shown by double clicking on the three dots, this will show the virtual pattern open. Again if you want to see the virtual method declaration just double click on open

and Mjolner will follow the virtual binding, up the hierarchy.
Now to get back, use the **Back** shortcut <CTRL> and <b> or right click and use the menu.
Among many other features, the menu **View** gives the programmer the opportunity to make the view of the code as an **overview**, **abstract**, **Detail** and **Detail Recursively**, these features makes it easier to comprehend and overview large programs.

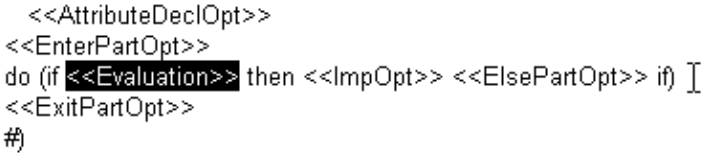
2.6.2 Edit Code

In the startup phase of the implementation of the BETA program, the Optional Terminals, as used above, is a great help, to make the the program structured right.
Now look at a do part

by right clicking on the optional



A list of imperatives will appear, ex a simple If Imperative, which will make an the structure for a simple If imperative. Like shown below



It is now easy to write the if imperative.
These optionals, as written earlier, is a big help to the programmer in the startup phase. When the actual programming is finished and only editing on the code is made, the shortcut <CTLR> and

<SPACE> or using the **Edit Text** in the **Edit** menu, will make it possible to edit the highlighted code.

Good luck with your use of the Mjølner tools.

3 Source Browser

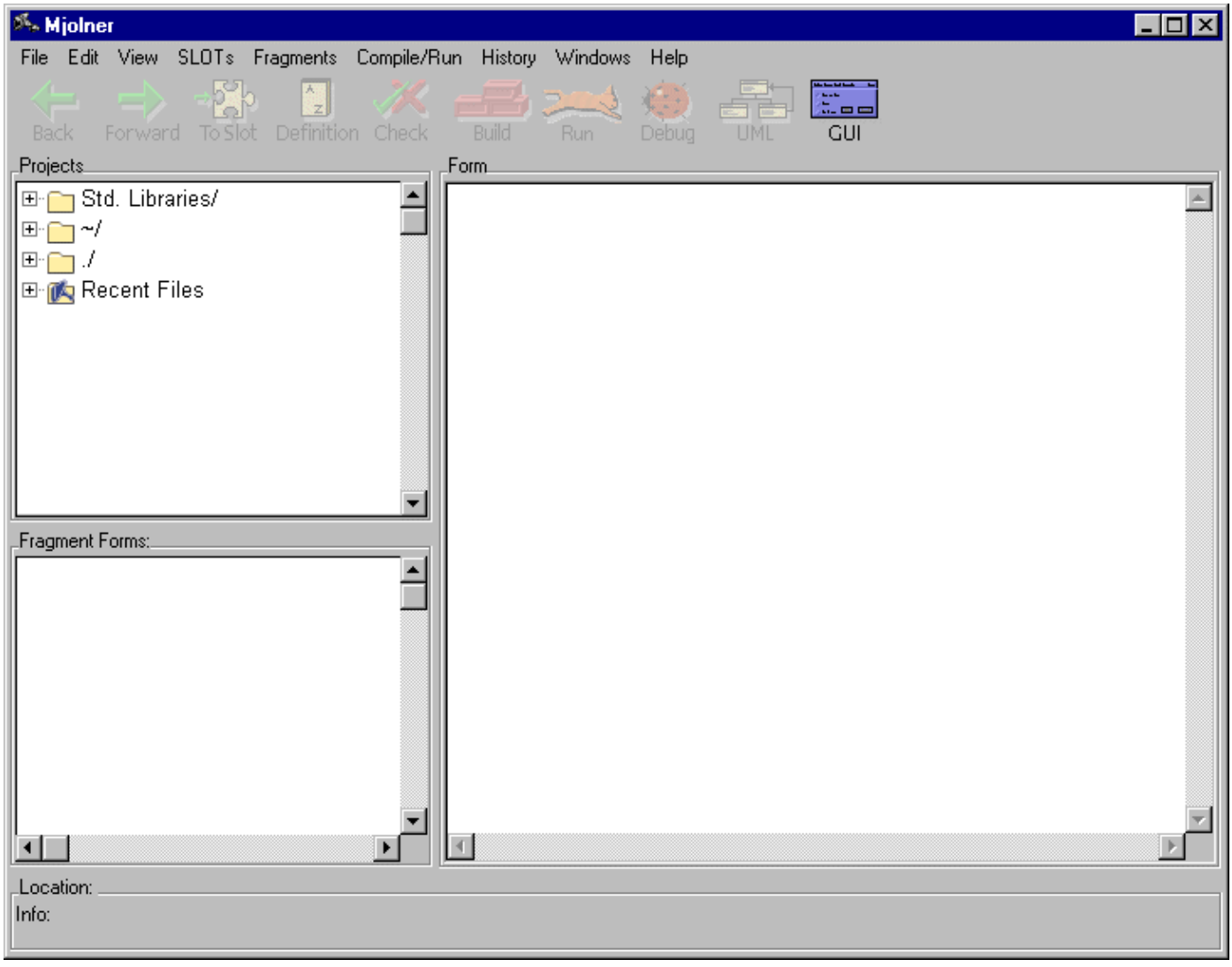
3.1 How to Get Started

The Mjolner tool is activated by writing

mjolner

on the command line (UNIX or Windows) or by double−clicking on the mjolner icon (Windows or Macintosh):

Figure 1



This window is called a source browser window or just browser window and has 3 important panes. Followed counter−clock−wise from the upper left pane, the panes are described below.

Pane 1 is called the Projects pane, it contains a list of projects presented in a treeview. A project can be a folder and/or a list of files

Pane 2 is called the Fragment Forms pane, it contains a list of fragment forms of the fragment group selected in the Projects pane. A selected fragment group is called a fragment group viewer/editor or just group viewer/editor.

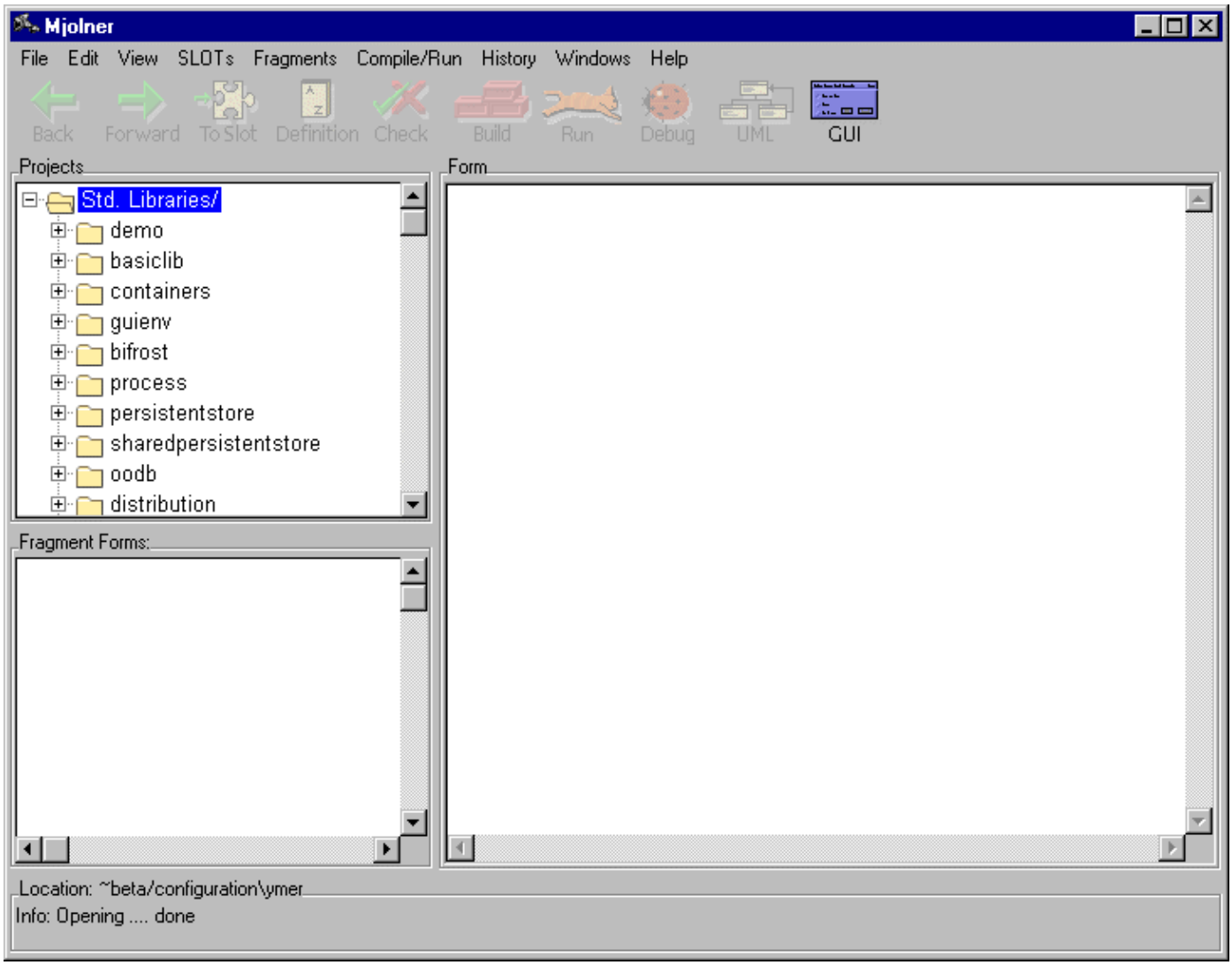
Pane 3 displays the BETA code of the fragment form selected in the Fragment Forms pane. It is called the code viewer/editor. This code viewer will be seen in many different windows and its functionalities are described later.

Browsing can be done at basically 3 levels: the project level, the group level or at the code level.

3.2 Browsing at Project Level

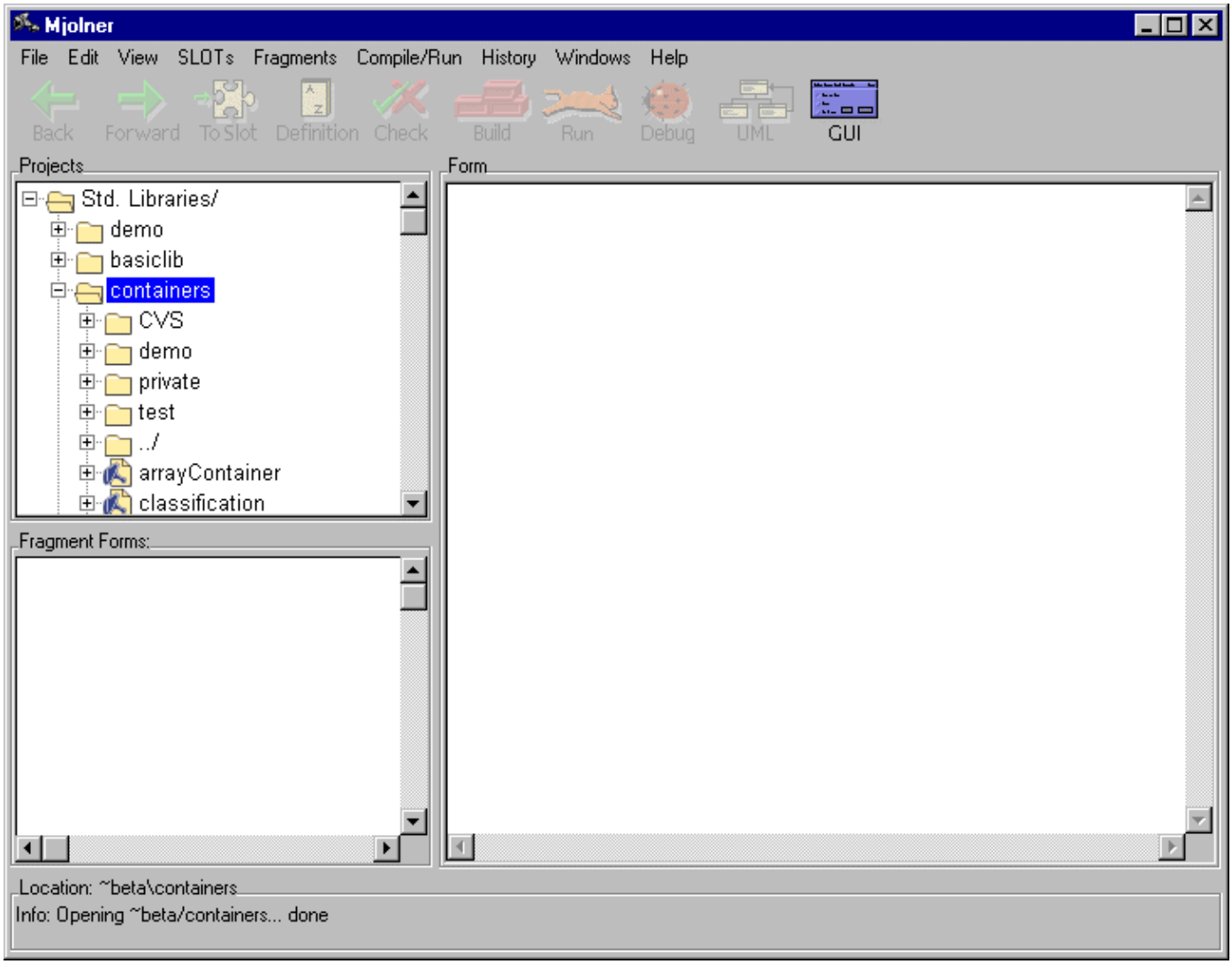
Double−clicking on the Std. Libraries project in the project list pane will give the following result:

Figure 2



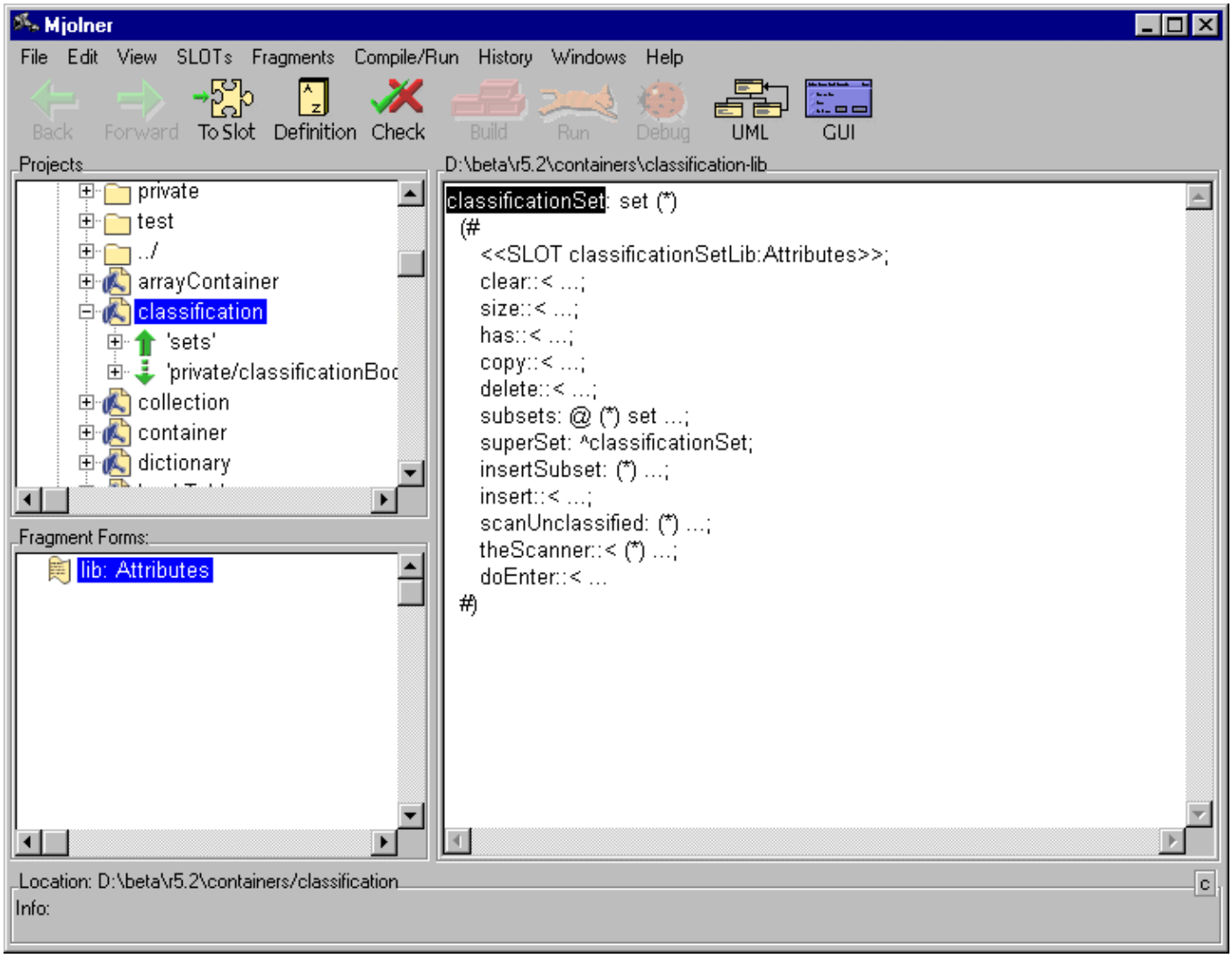
Std. Libraries is the standard project that contains the basic libraries of the Mjølner BETA System. The contents of the Std. Libraries project is a list of file directories. By selecting the container directory its files (fragment groups) will be shown in a subtree:

Figure 3



Now by selecting the + sign of the classification fragment group its properties will be presented in a subtree and the fragment forms will be presented in the Fragment Forms pane. Since classification only contains 1 fragment form it is automatically presented in the code viewer:

Figure 4



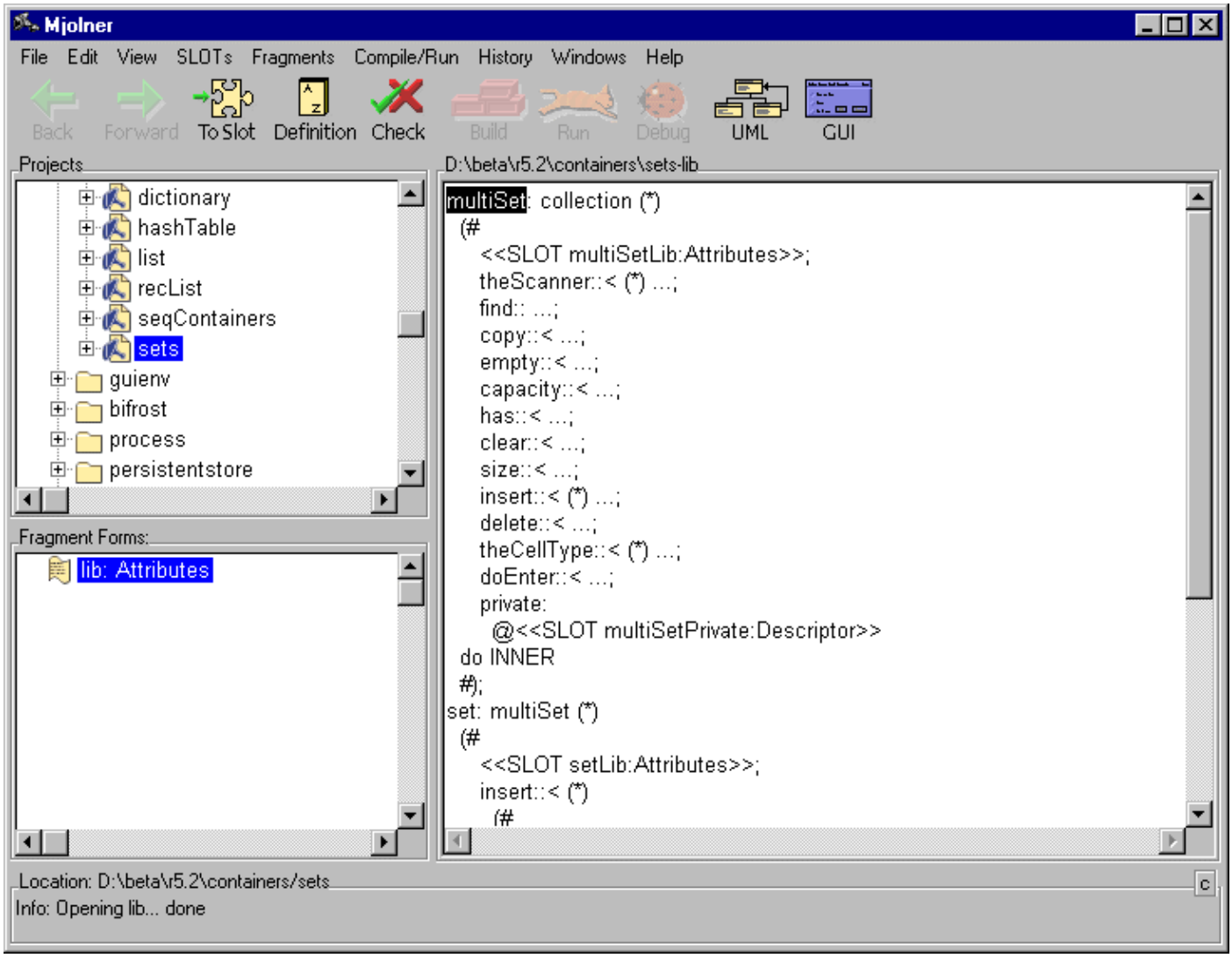
3.3 Browsing at Group Level

Browsing at the group level is provided in the following way:

1. If an ORIGIN (double up−arrow), BODY, MDBODY (dashed down−arrow) or INCLUDE (single up−arrow) entry is selected in the treeview, the link to the specified group is followed. The contents of the panes is replaced with the specified fragment group (if it exists).
2. If a fragment group only contains one fragment form it is automatically presented in the code viewer. When a fragment form is selected it will be presented in the code viewer.
3. The way links between SLOTs and fragment forms can be followed (is described separately under Fragment and SLOT Links)

The screen dump below is the result of clicking on the ORIGIN property in Fig. 4.

Figure 5



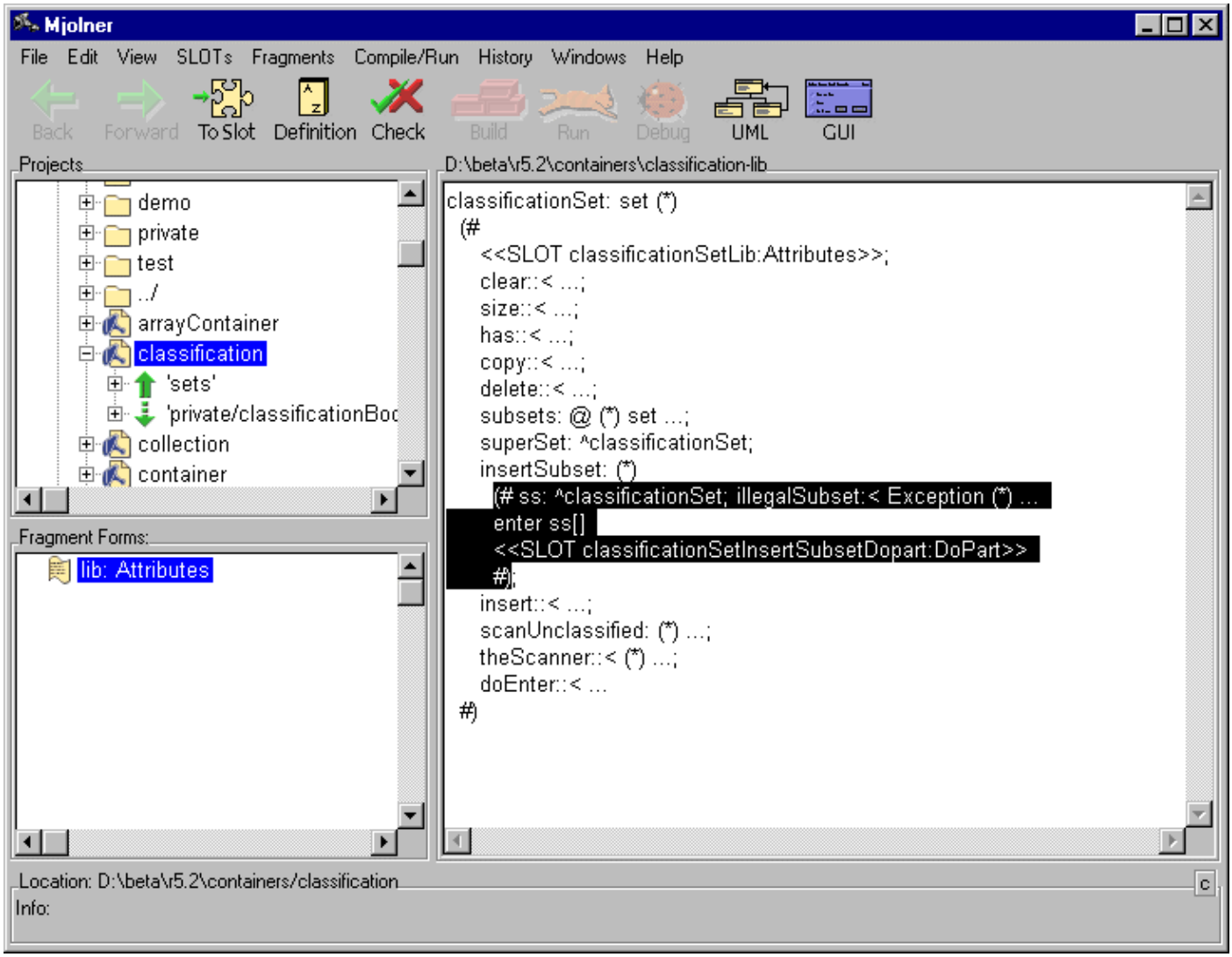
3.4 Browsing at Code Level

3.4.1 Abstract Presentation

When a fragment is shown in the code viewer, i.abstract presentation; is used. Instead of showing the program in full detail, 3 dots (...) are shown for certain syntactic categories. In the presentation of BETA programs these syntactic categories are Descriptor, Attributes and Imperatives.

These 3 dots are called a contraction. By clicking the Definition command of the toolbar (or double−clicking) when a contraction is selected the next level of detail is shown. E.g. when double−clicking on the 3 dots belonging to insertSubset in Fig. 4, the result will be:

Figure 6



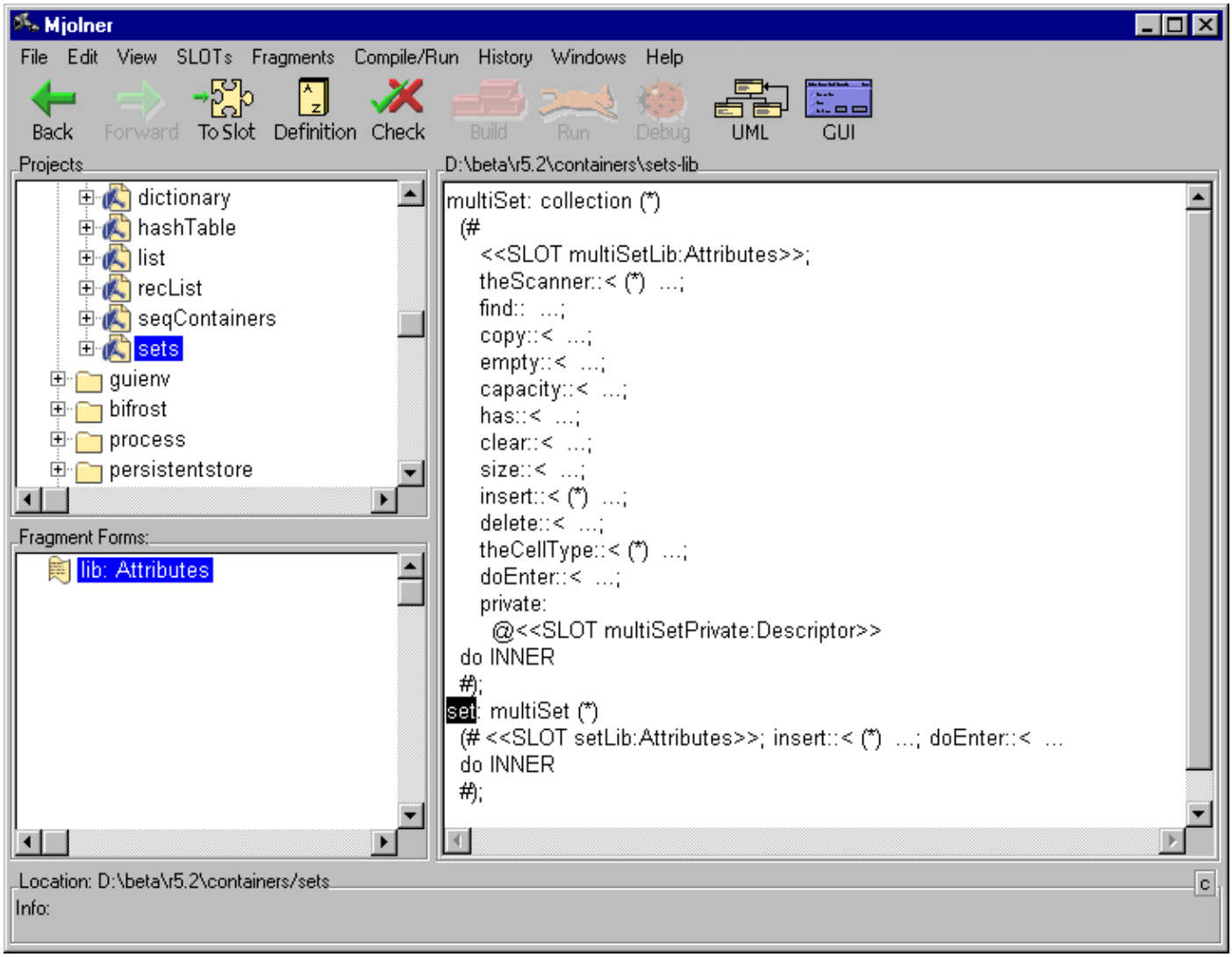
Abstract presentation can be used to browse around in the program or to produce documentation. By means of the commands in the View menu or by double−clicking an appropriate abstraction level can be chosen. This presentation can then be written on a text file by means of the Save Abstract... command in the Group menu.

3.4.2 Semantic Links

This facility is only relevant for BETA programs. If the program has been checked it is possible to browse around in the semantic structure of the program. If, for example, a name application is selected in the code viewer, the corresponding name definition can be found by clicking the Definition command of the toolbar (or double−clicking).

If the definition is in another fragment, a code viewer is opened on that fragment. Fig. 7 is the result of double−clicking on 'set' in the first line of the code viewer in Fig. 6.

Figure 7

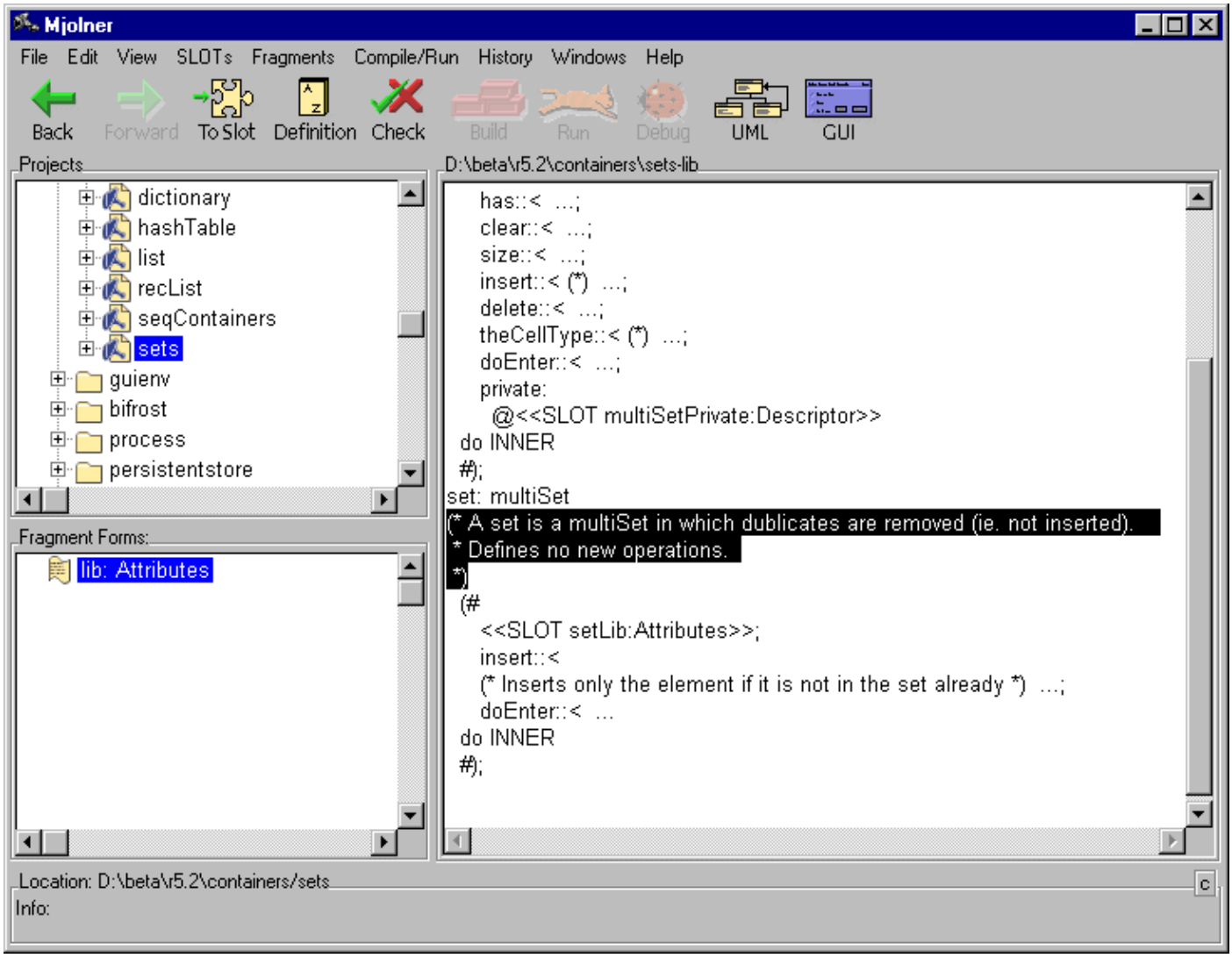


If the program has not been checked, no semantic links are available. Or if the program has been modified, the semantic links may become inconsistent and the program then has to be re−checked. If this command is used and the program needs to be re−checked and a dialog box, that offers re−checking, is popped up.

3.4.3 Comments

In order to compress information in the windows, comments are not shown in the code viewer window, but instead a so−called comment mark (*) is shown. By clicking the Definition command of the toolbar (or double−clicking) on such a comment mark, the comment mark is expanded and the whole comment is shown. Fig. 8 is the result of double−clicking on the comment mark after multiSet in Fig. 7:

Figure 8



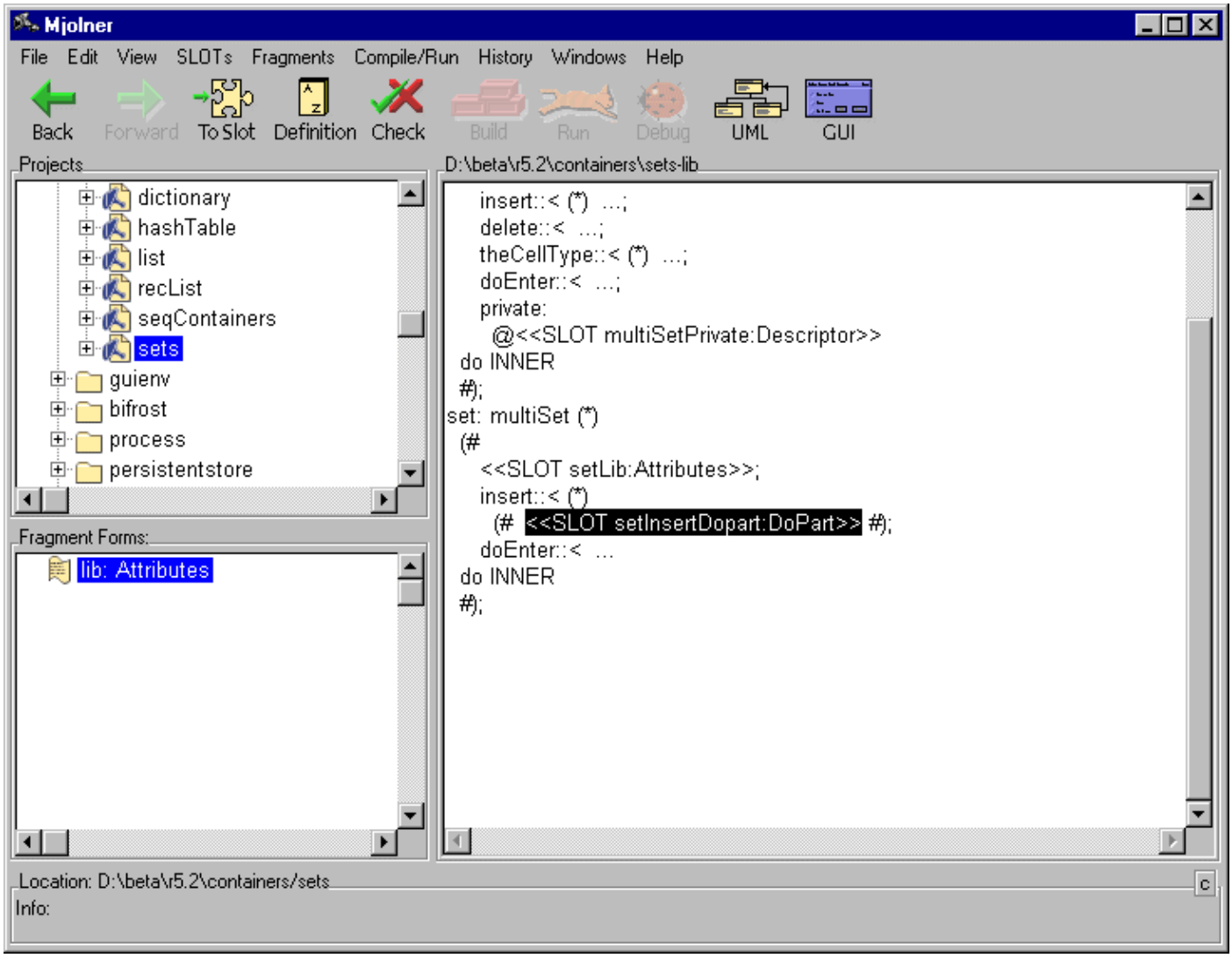
Likewise the expanded comment can be collapsed to a comment mark again by double−clicking on it.

3.4.4 Fragment and SLOT Links

Follow link to fragment form

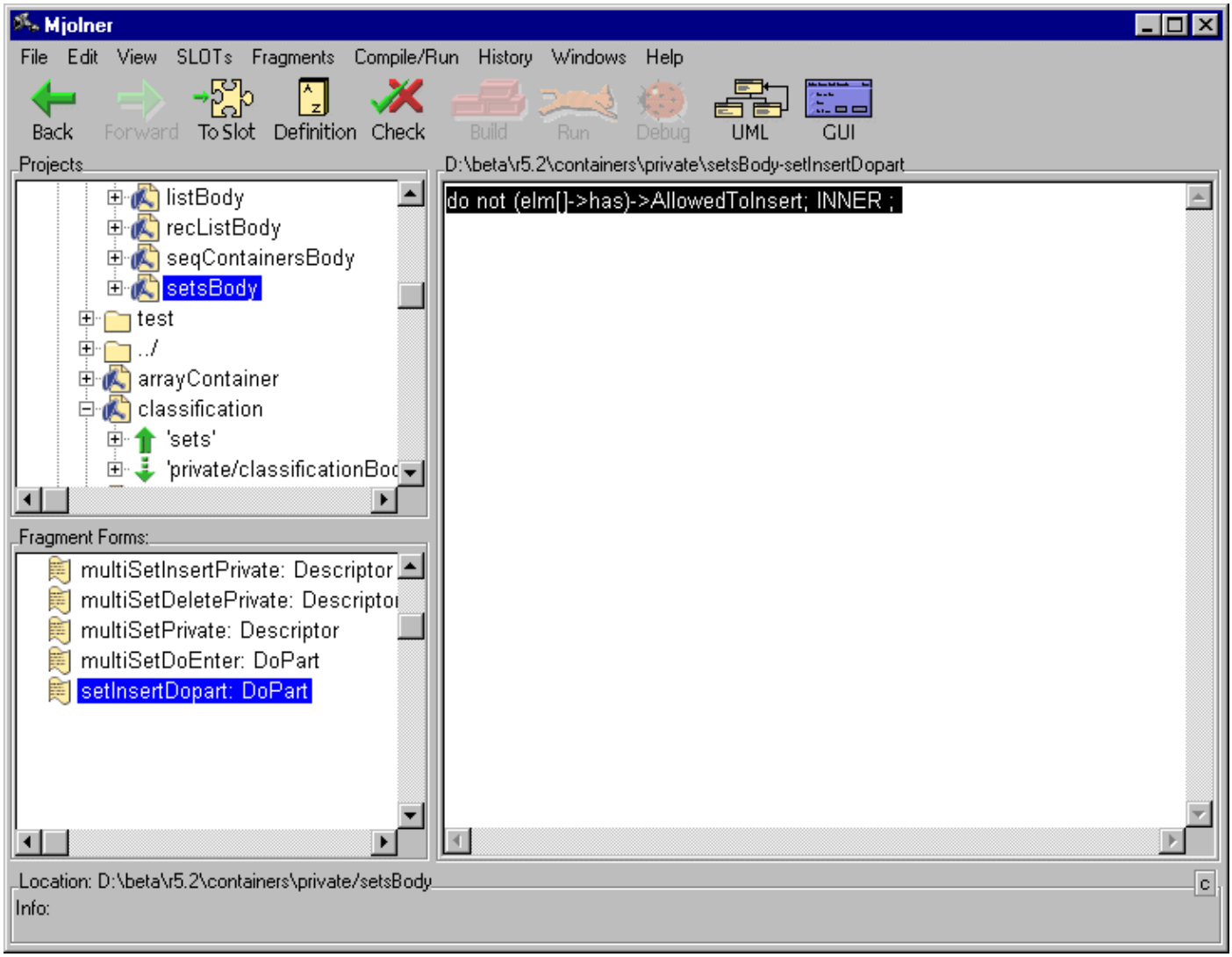
Given a SLOT definition, the link to the binding of the definition (i.e. a fragment form with the same name and type) can be followed.

Figure 9



This is done by selecting the SLOT definition in a code viewer and clicking the Definition command of the toolbar (or double−clicking). The editor will now search for a fragment form with the same name in the BODY and MDBODY hierarchy of the group that the current fragment is part of. Doing this in the example above will result in the fragment shown in Fig. 11. If Shift−double−click is used a separate code viewer window is shown.

Figure 10



Notice that during an editing session the fragment form may not be found since the program may be incomplete. Another restriction is that binding of Attributes SLOTs only can be found in this way if they are bound in the BODY and MDBODY hierarchy. If not, there is no automatic way of finding the bindings.

Follow link to SLOT definition

Given a fragment form, the link (possibly dangling) to the SLOT definition with the same name can be followed. This can be done by selecting a fragment form and using To SLOT command in the toolbar. The editor searches after a SLOT definition along the ORIGIN chain until it finds it or reaches the top, i.e. betaenv.

3.4.5 Searching

By means of the Find command in the Edit menu it is possible to search for a substring of a lexem, i.e. a name definition, a name application or a string in a codeviewer.

It is also possible to search in

- the current fragment form of the source browser, i.e. the fragment form currently presented in the codeviewer of the source browser
- the current fragment group of the source browser

- ♦ the domain of the current fragment group
- ♦ the extent of the current fragment group

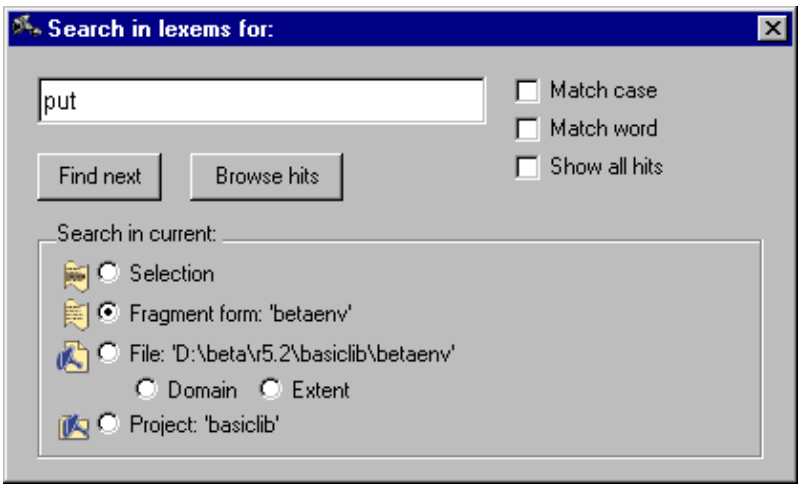
- the current project in the source browser e.g. also in the domain and extent of the current fragment group

An especiallly useful facility is the browser of search hits, that collects all the search hits for easy overview and access.

In the following two examples of advanced search will be demonstrated. First we will search for a text in a fragment group and then we will search for a all applications of a certain name.

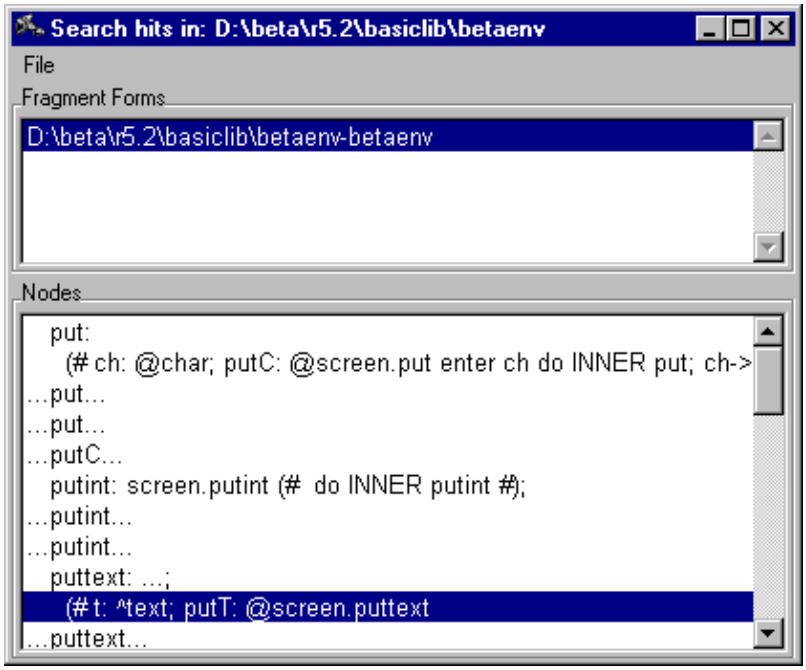
An example of searching in a fragment group will now be shown. We want to find all occurrences of the text 'put' in 'basiclib/betaenv'. First we open 'basiclib/betaenv' in mjolner. When using the Find command the search window pops up:

Figure 11



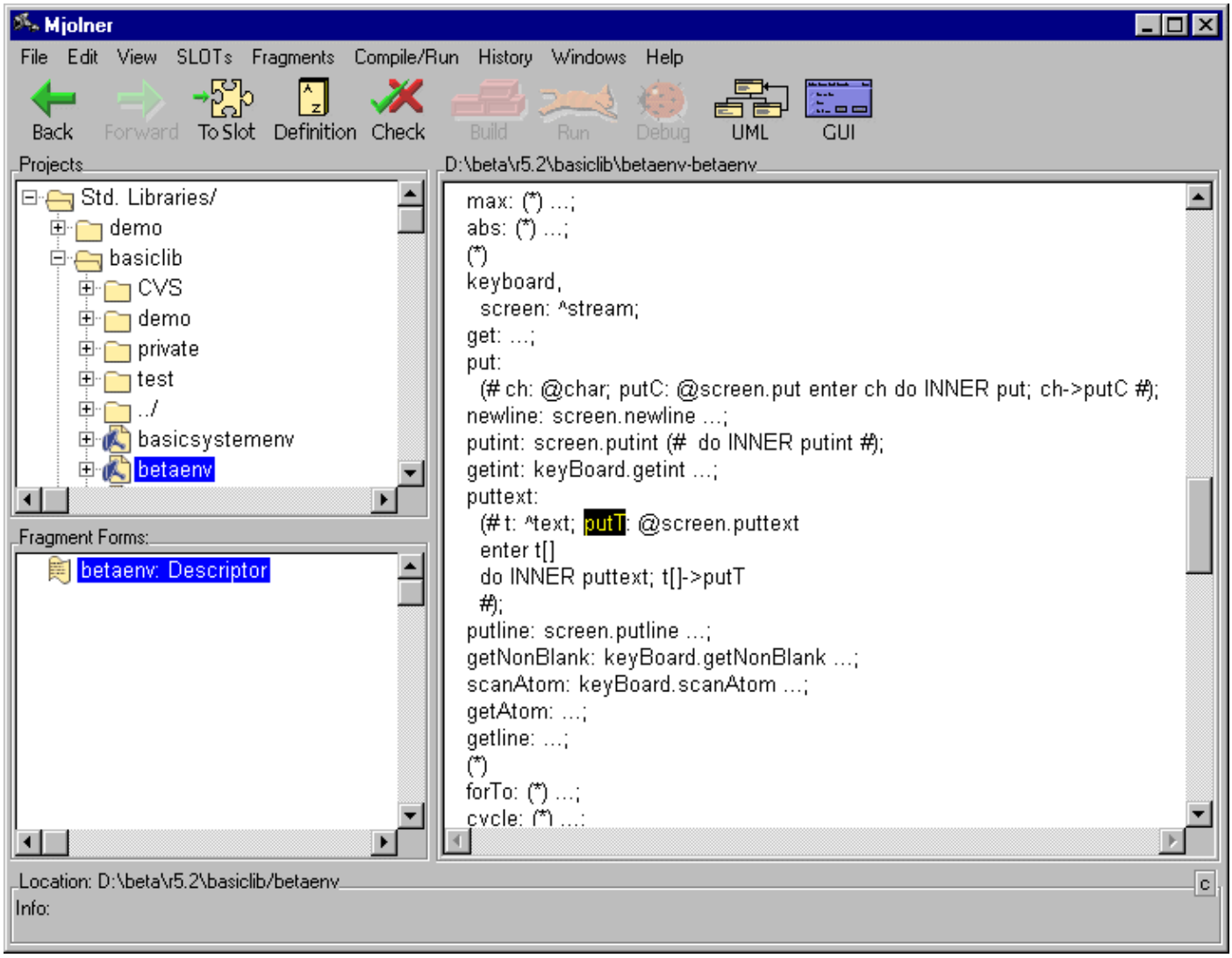
If we enter 'put' and click on the 'Browse hits' button the current fragment group i.e. 'betaenv' will be searched for occurrences of the text 'put'. I.e. all lexems (name declarations, name applications and strings) will be searched. The search hits are then presented in the search hit browser:

Figure 12



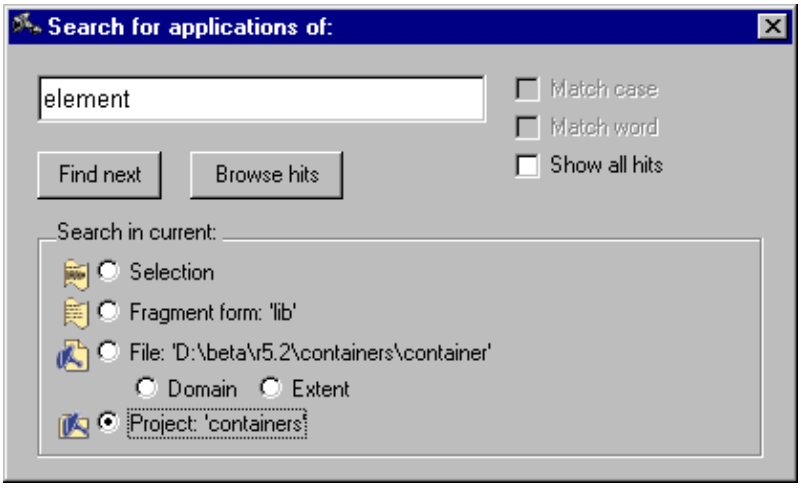
This search hit browser can be used to select some of the search hits. The Return key can be used to browse quickly through the search hits. An example of a selected search hit:

Figure 13



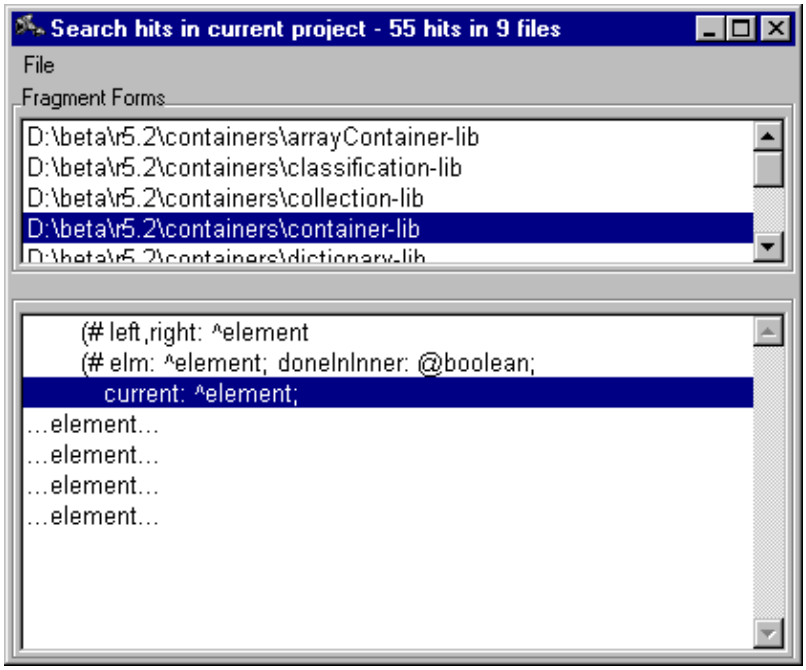
An example of searching for all applications of a certain name in a project will now be shown. If the current selection in the source browser is a name declaration or a name applicatin the Find command will provide the possibility of searching for applications of that name declaration. In this case we want to search not only in a file (fragment group) but in the whole 'containers' directory. We want to find all occurrences of the name 'element' as declared in the 'containers/container' file. If we select the 'element' declaration in 'containers/container' and activate the Find command, the Search window will appear as:

Figure 14



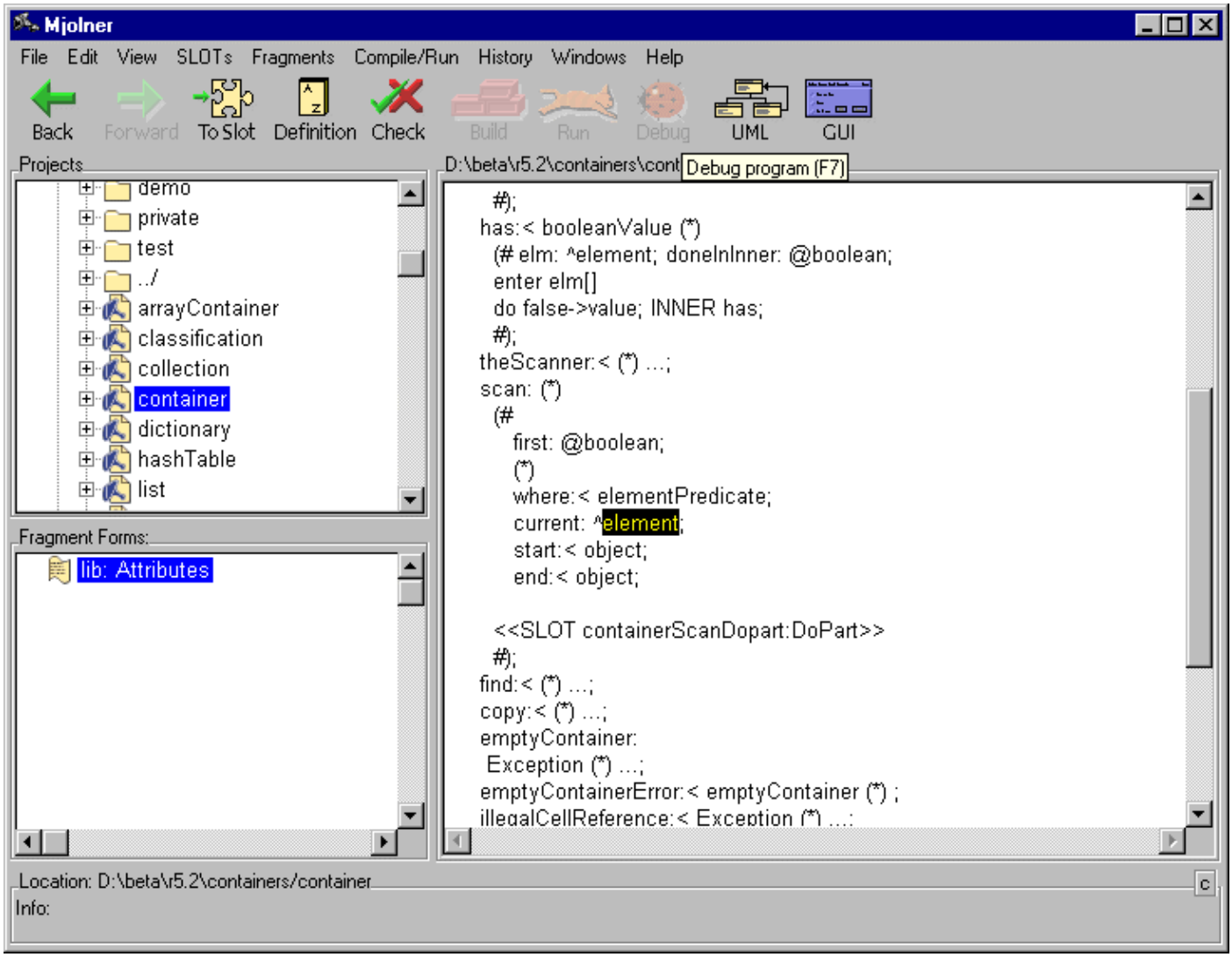
We have changed the search area to be the current project. In this case the current project is the 'containers' directory. Pressing the 'Show all hits' button the 'Search hit browser' will appear as:

Figure 15



An example of a selected search hit:

Figure 16

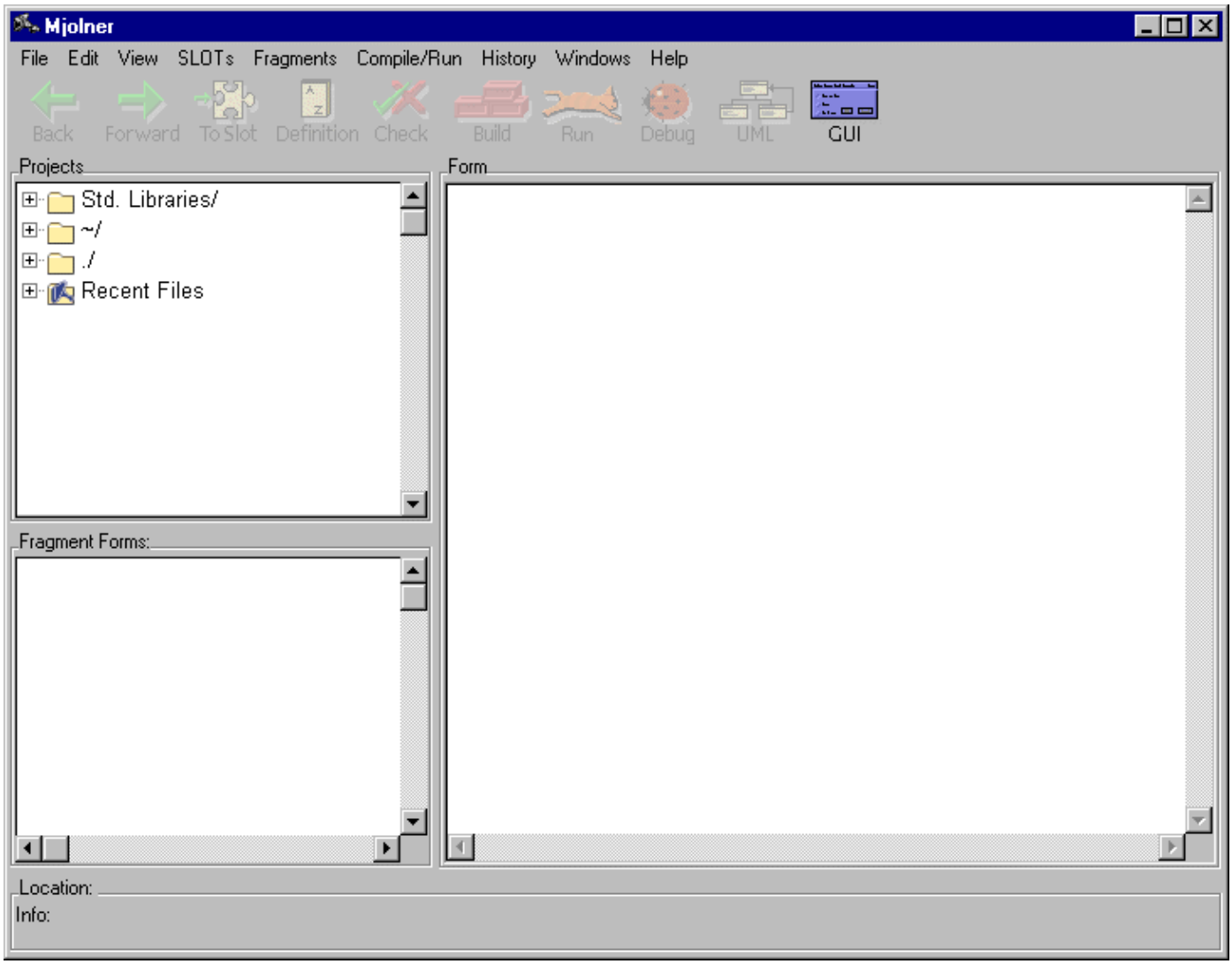


It is also possible to search for a text string or all applications of a name in the whole dependency graph. See Dependency Graph − Domain and Extent Projects for how to select the dependency graph for a fragment group. For a given fragment group, select the Extent of the dependency graph as project in the Project list pane and search in 'Current Project'.

4 Editor

The editor consists of basically two types of editors: a group editor and a code editor.

Figure 17



The Group editor

The group editor is only relevant if the language in question is BETA. It is used for browsing or editing BETA programs. It is used to present and modify the structure of a group, i.e. the properties (ORIGIN, BODY, MDBODY, INCLUDE etc.) and fragments (Descriptor forms, DoPart forms or Attributes forms).

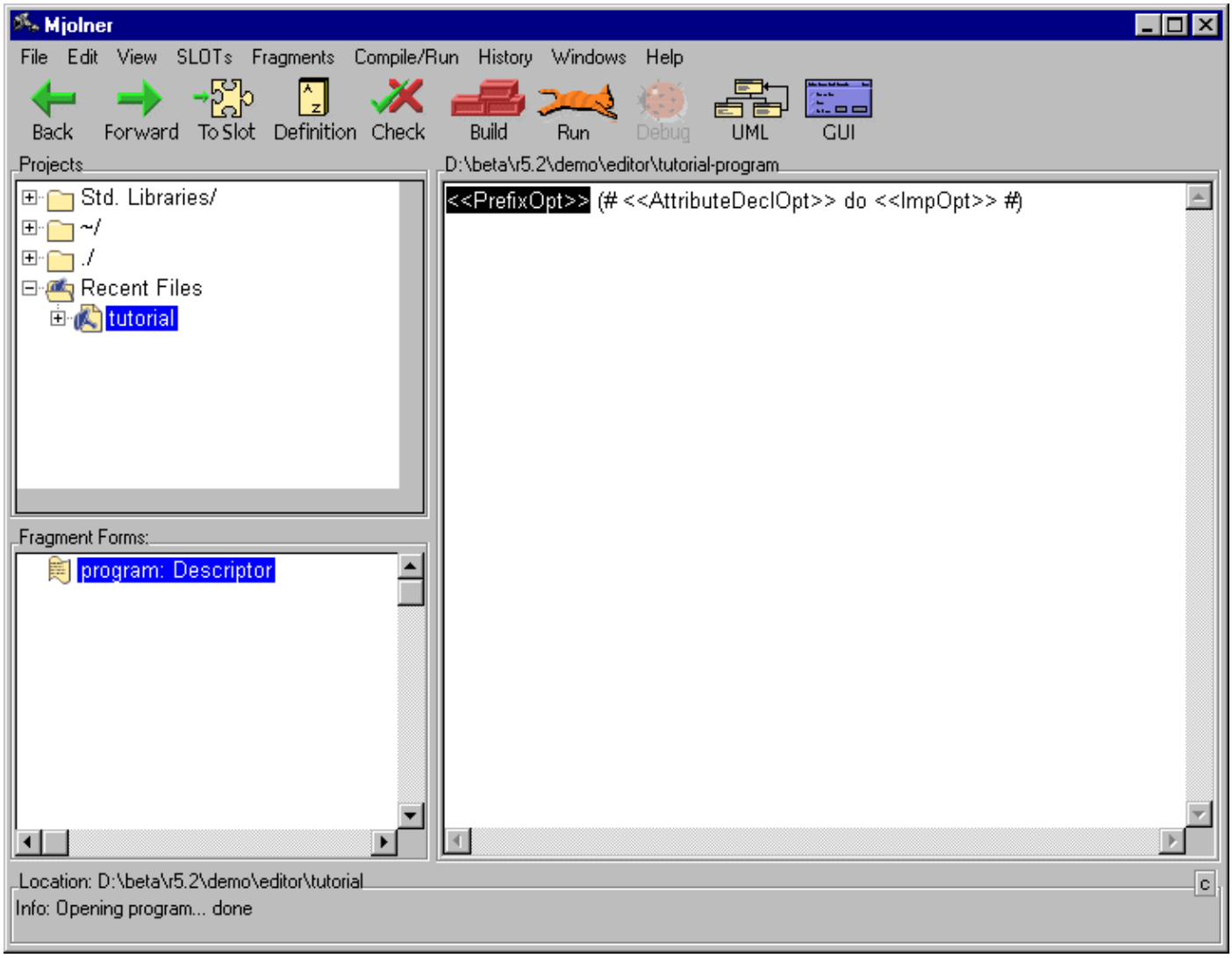
The Code editor

The code editor provides structure editing on each fragment form.

4.1 Creating a New Program

When mjolner is started without arguments a template for a BETA program is automatically created.

Figure 18



This program is called untitled1.

Alternatively the 'New BETA Program...' command of the File menu can be used.

4.2 Editing at Code Level

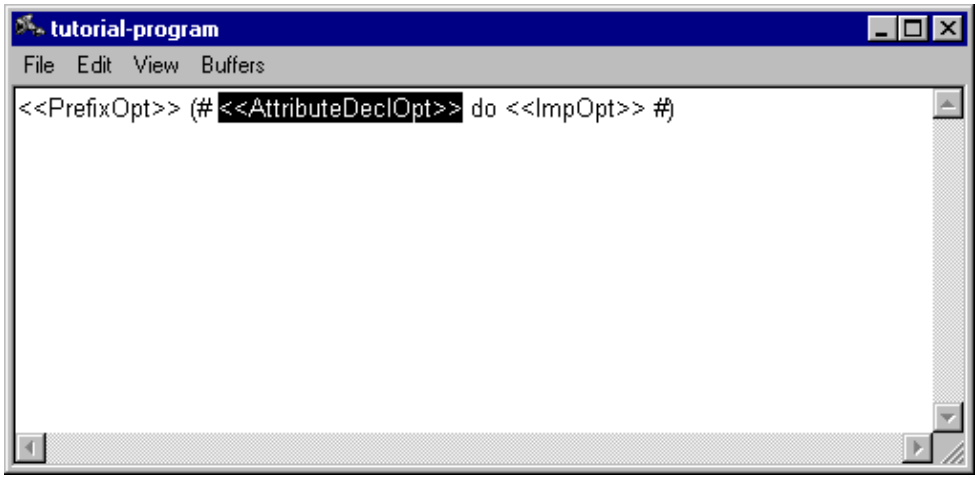
4.2.1 Code editor

The code editor provides structure editing on each fragment form. In the following a separate code editor is chosen by shift−double−clicking on the program fragment form in the group editor (or using the pop−up menu in the code editor) of Fig. 13.

4.2.2 Structure Editing

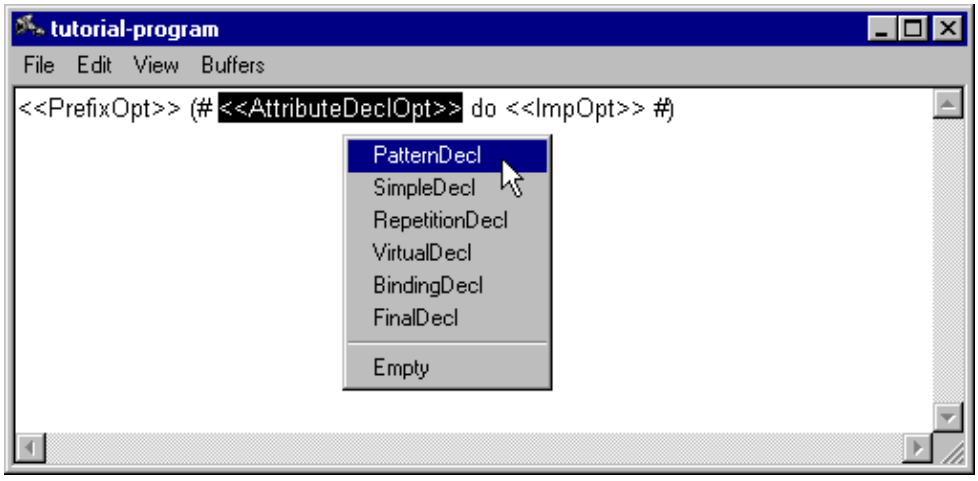
The basic idea of structure editing is that the program is manipulated in terms of its logical structure rather than the textual elements such as characters, words and lines. The advantage of this approach is that only logically coherent parts can be inserted or deleted and thereby preserving the syntactical rules of the language at any time.

Figure 19



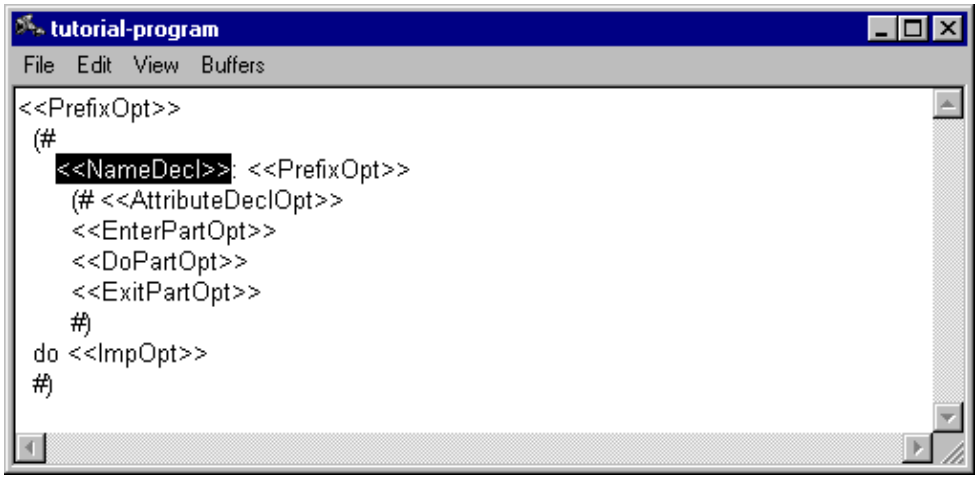
The window above shows an example where a template for a BETA program has been derived. The template includes placeholders (nonterminals) and keywords (terminals). If a nonterminal is selected the mouse pointer is changed to an icon that indicates that the rightmost button of the mouse must be pushed to pop−up a menu (only Unix version, in general the Pop−up Menu Button;.;; is used, see Basic User interface principles). In the example, the <<AttributeDeclOpt>> placeholder has been selected and the legal declarations (according to the BETA grammar) is shown in the pop−up menu.

Figure 20



The PatternDecl entry is selected and the result is:

Figure 21



4.2.3 Text Editing and Parsing

Structure editing has its greatest force at the higher levels of editing, i.e. for creating the overall structure of the program or for moving around large chunks of code. At the detailed level the textediting technique is more useful.

Text editing can at any time be used as an alternative to structure editing. Text editing is activated either by just starting to type or by selecting the Textedit command in the Edit menu. If you start typing at the keyboard, the typed characters will replace the current selection in the code editor window. Text editing mode may alternatively be entered without deleting the current selection, by means of the Textedit command. In that case the text cursor will be positioned in the start of the current selection.

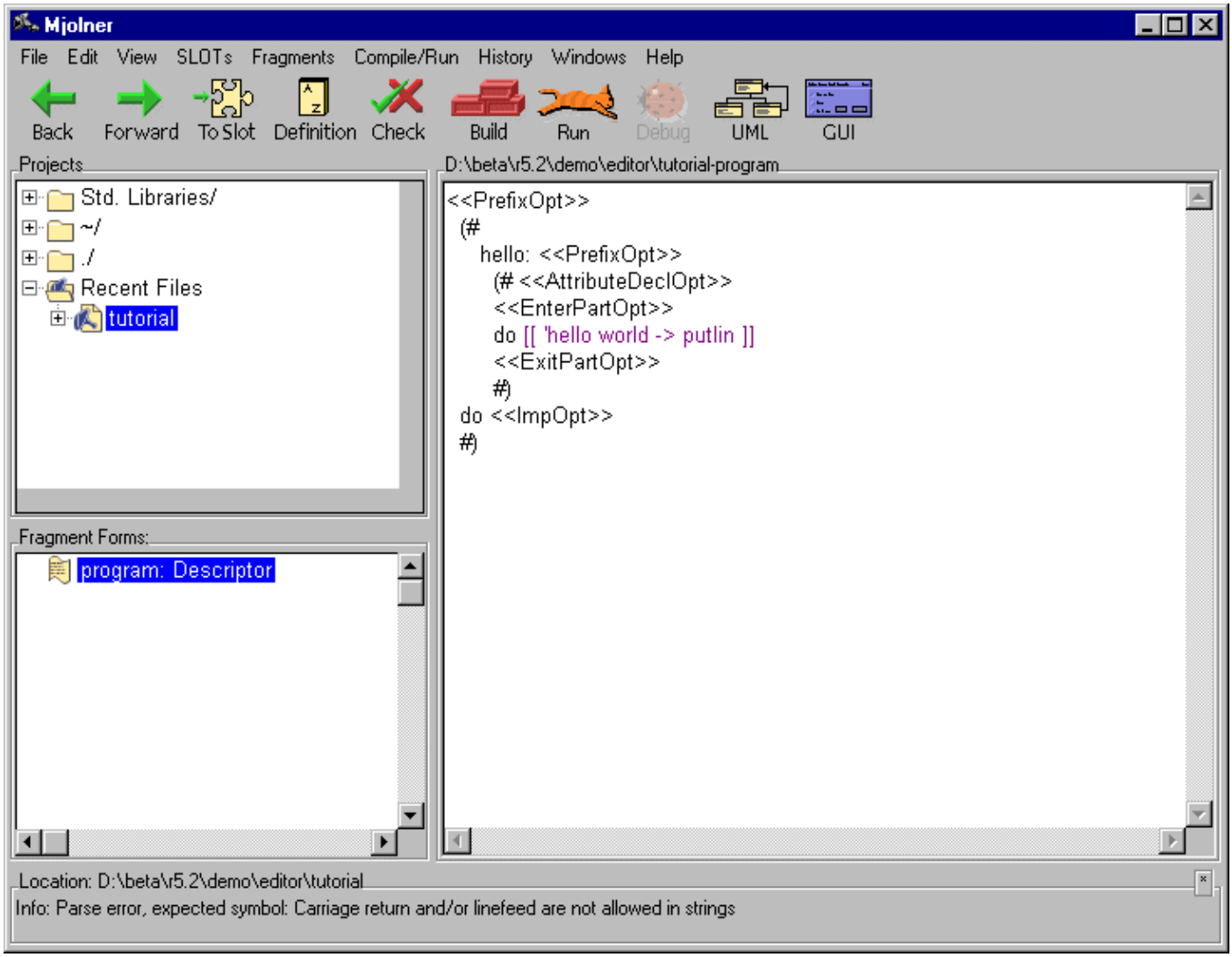
Text editing mode can be terminated by selecting the Parse Text command in the Edit menu. The possibly modified text will immediately be parsed and any parse errors will be reported (but only one at a time). Note that semantic checking is not done by the editor. In this example a parse error is detected.

In Fig. 17 the placeholder <<NameDecl>> is selected and the name hello is typed. After that the <<DoPartOpt>> is selected and the text

```
'hello world -> putline
```

is typed. When textediting is exited the syntax error is immediately reported:

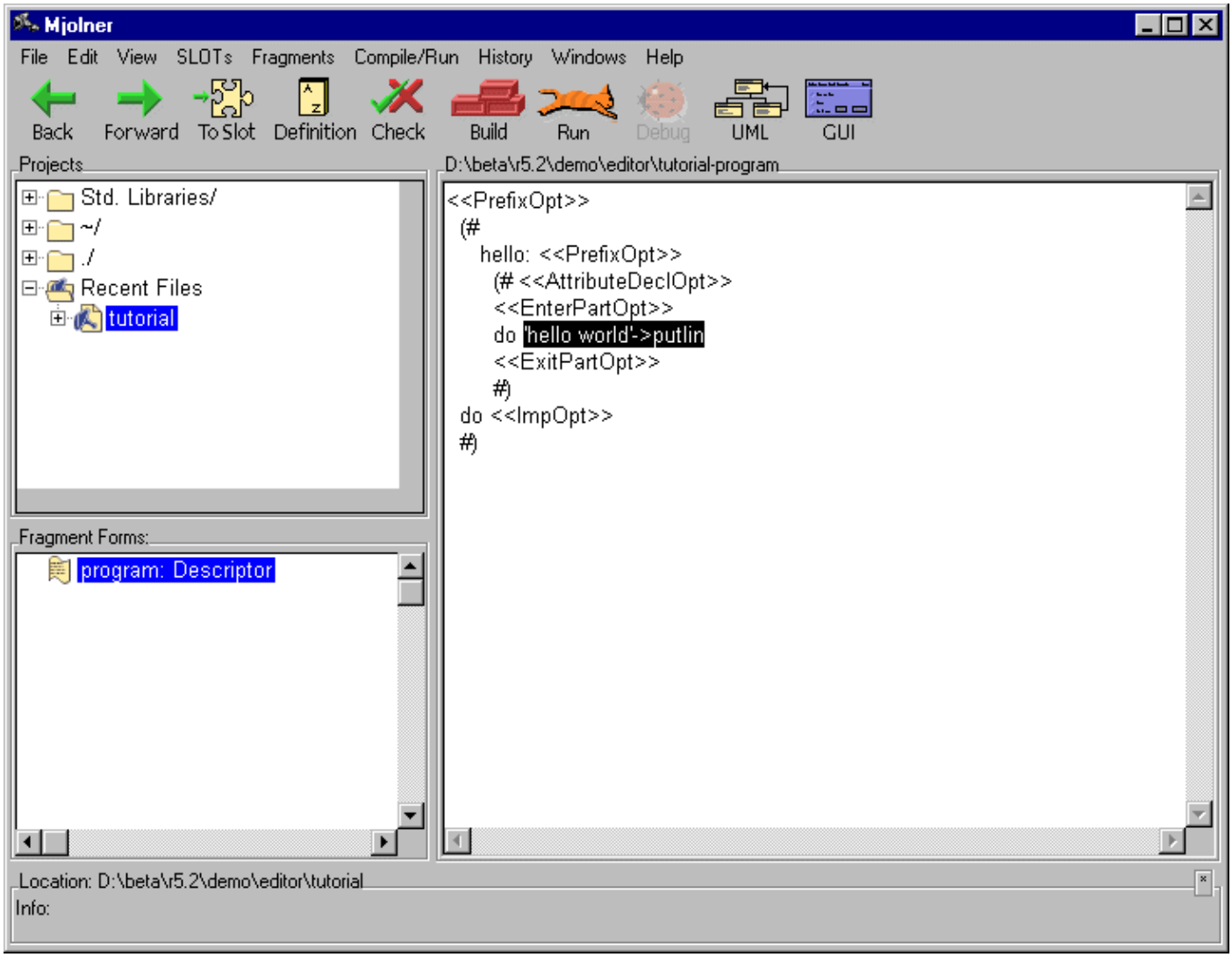
Figure 22



4.2.4 Checking

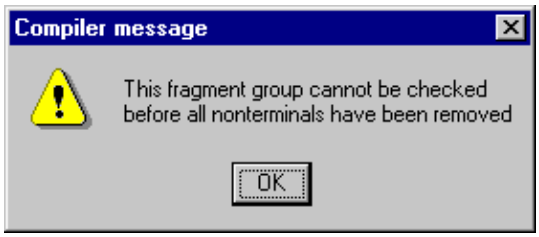
After correcting the syntax error the program looks like below:

Figure 23



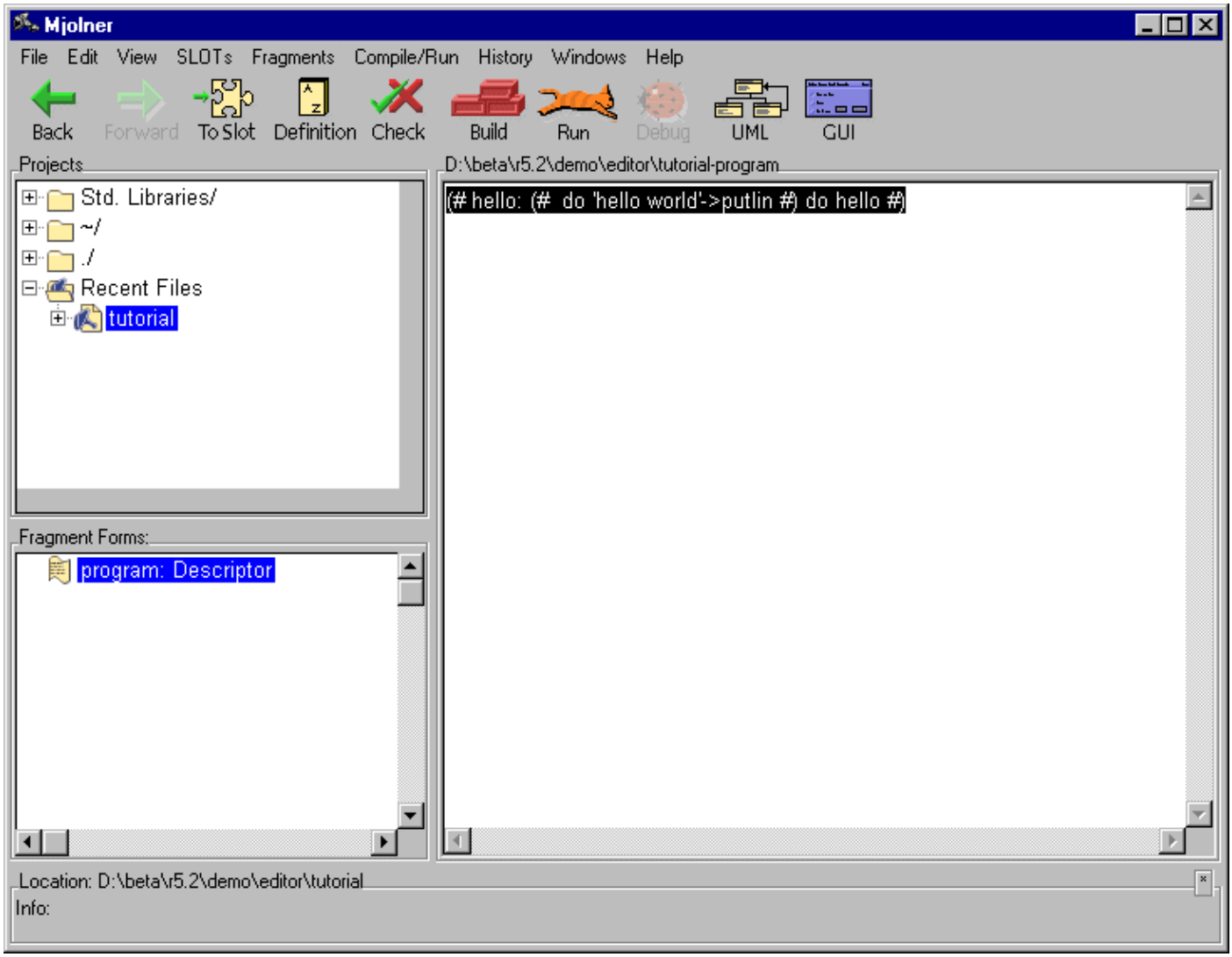
Now we want to call the checker. This is done by means of the Check Current command in the Tools menu. But the compiler does not accept unexpanded nonterminals. Therefore the following dialog is popped up:

Figure 24



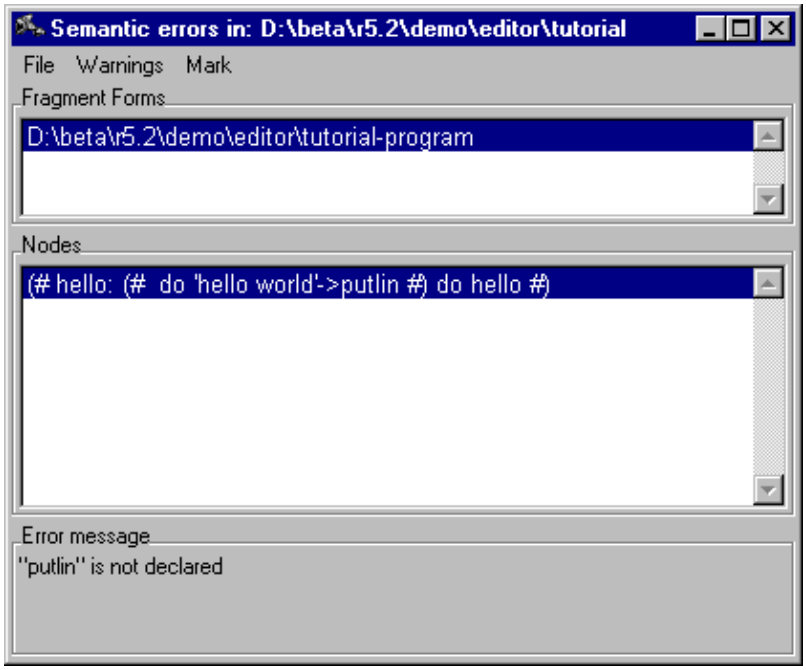
Notice that in this example all nonterminals are optionals (is indicated by the Opt suffix). An easy way to remove unexpanded optionals is to select the whole program and use the Remove Optionals in the Edit menu. The result is:

Figure 25



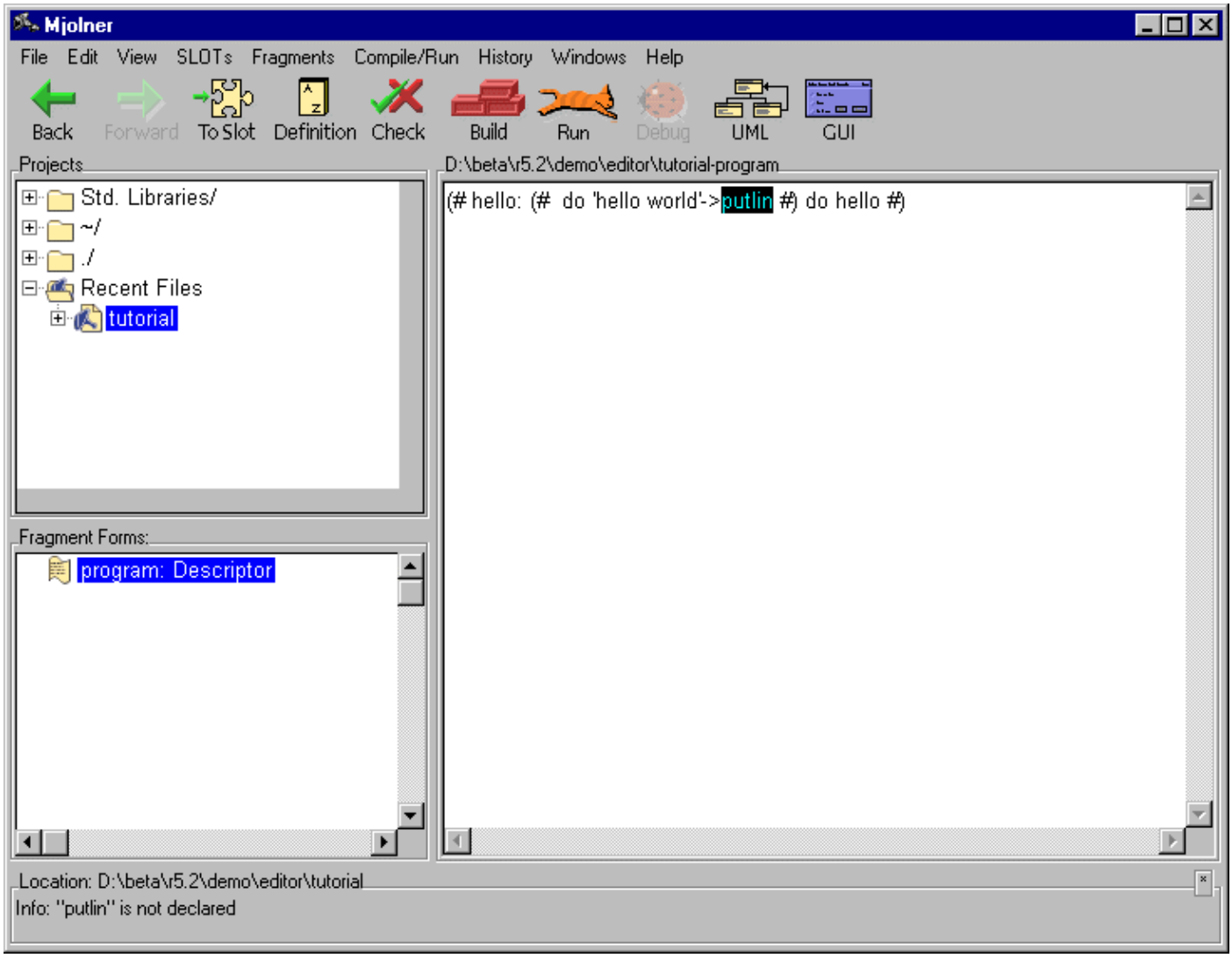
Now the checker is called again and the semantic error is detected and shown by means of the semantic error viewer:

Figure 26



In this case there is only one semantic error, but in general the Fragment Forms pane will contain a list of fragment forms with semantic errors and the Semantic Errors pane will for each fragment form in the upper pane show a list of semantic errors. By selecting in the two panes the different semantic errors can be inspected. By clicking in the Semantic Errors pane the code editor will select the corresponding structure. The first semantic error will always be selected automatically. In the example the following selection is made:

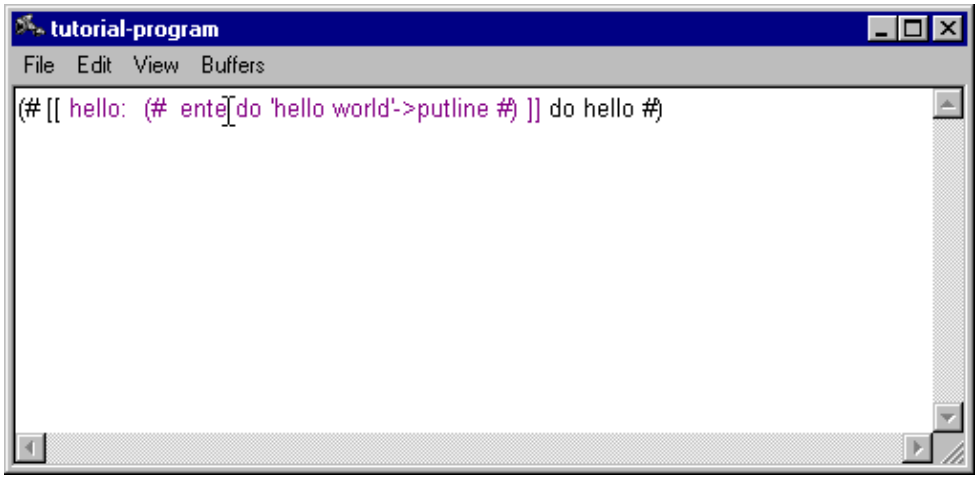
Figure 27



4.3 Modifying a Program

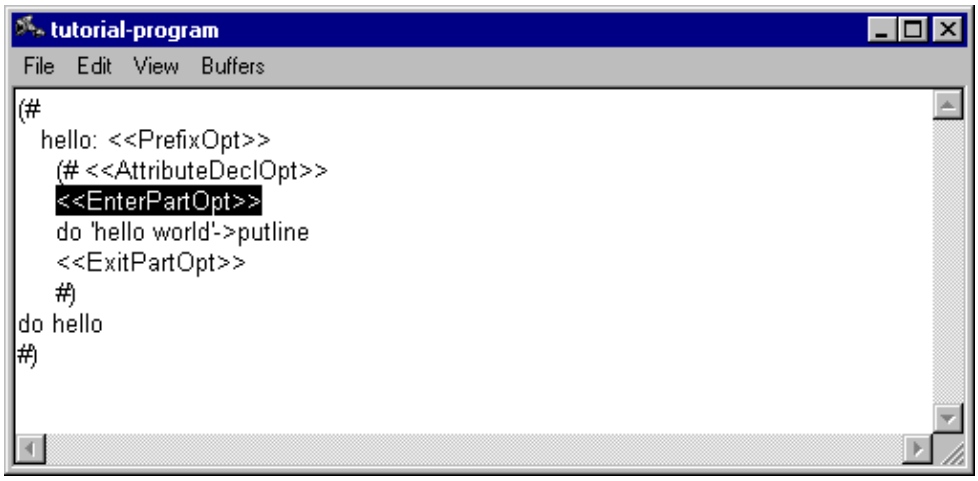
Consider the program of Fig. 22. We now want to add an enter part that should enter a text to be printed after the 'Hello world' string. To add an enter part you can either select for example the descriptor of the hello pattern and type the enter part using text editing.

Figure 28



Another technique is to select the whole pattern declaration and use the Show optionals command in the Edit menu. The latter has been done below:

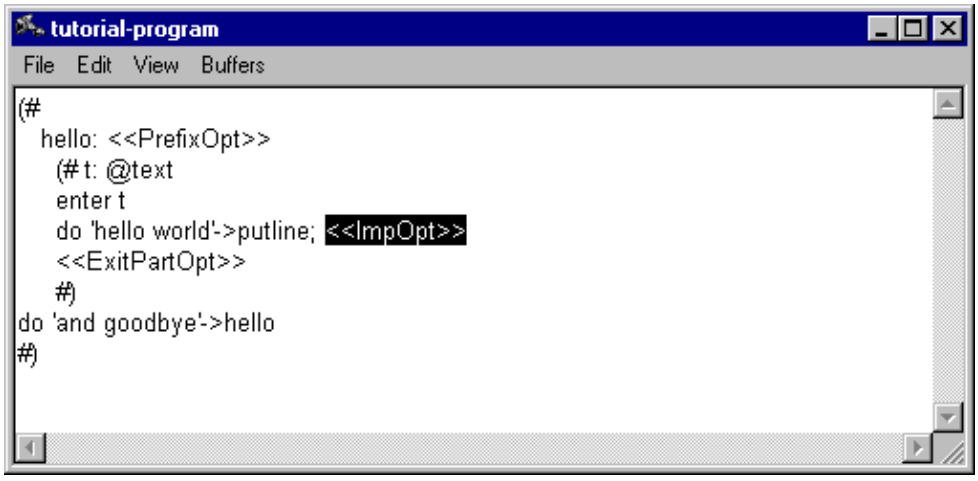
Figure 29



We select the EnterPartOpt nonterminal and via the pop−up menu we choose 'EnterPart' which gives the result: enter <<Evaluation>>. Then we select <<Evaluation>> and 't' is typed. Likewise we select the AttributeDeclOpt nonterminal and type: t: @text. Now we want to add an imperative after the putline. To insert a new list element in a list element is selected and the Insert After or Insert Before command of the Edit menu is used. An alternative to Insert After is to press the <cr> key.

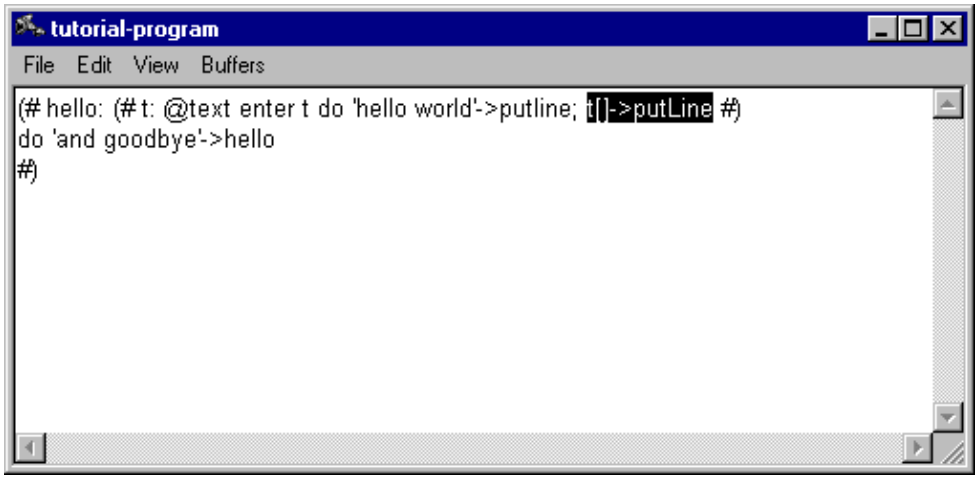
In Fig. 23 the imperative 'hello world'−>putline is selected e.g. by clicking on the arrow (−>). Then the <cr> key is pressed and the result is:

Figure 30



The final result is:

Figure 31



4.4 Editing at Group Level

4.4.1 Group Editing

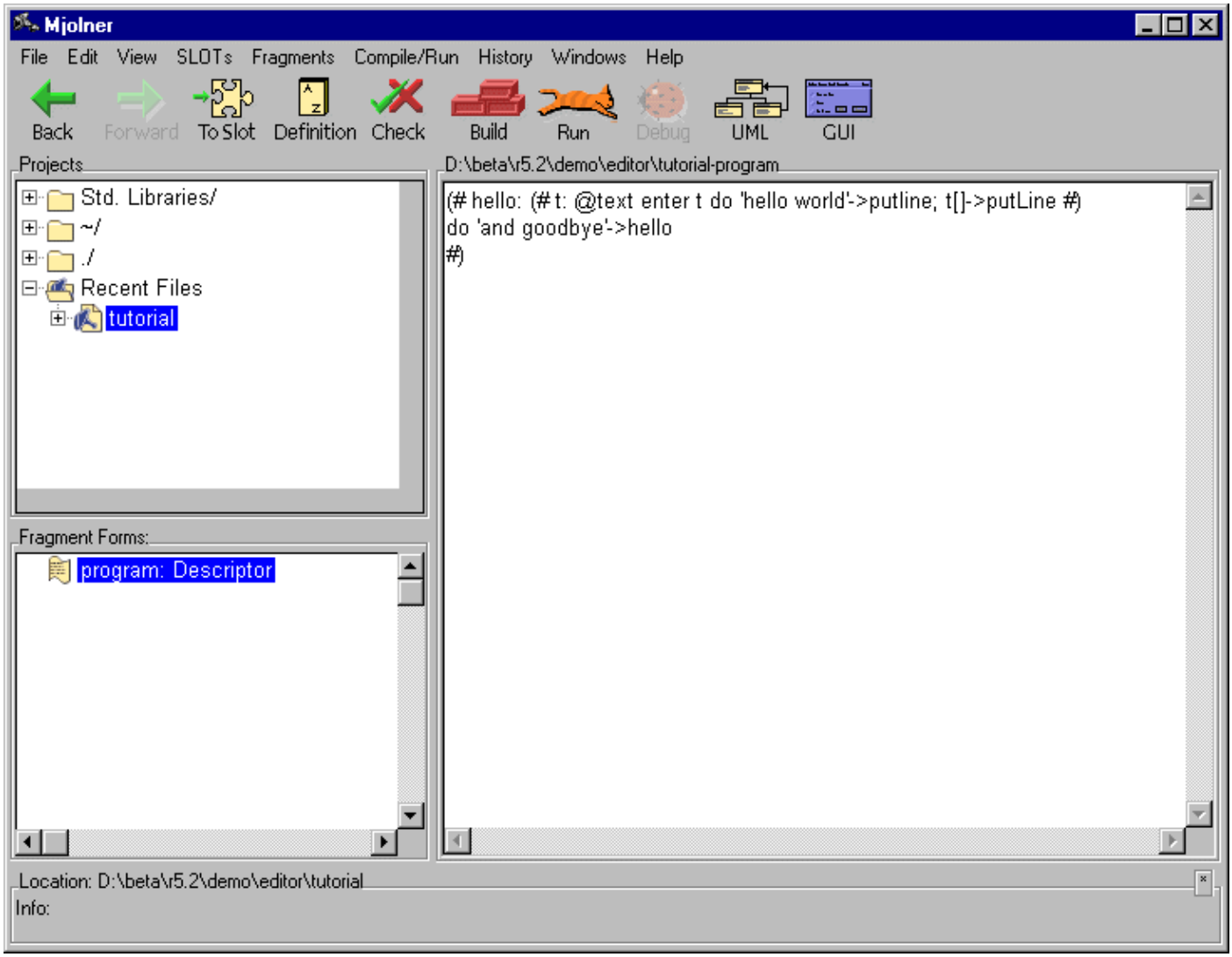
Manipulation of the fragment group structure is done in the group editor. It is possible to insert or delete fragment forms, modify the names of the fragment forms, or edit the properties of the fragment group e.g. the ORIGIN, INCLUDE and BODY properties. Editing of properties is done using a property editor that is a structure editor on the properties. This editor is activated by means of the 'Fragments:Edit ORIGIN, INCLUDEs etc...' command.

4.4.2 Fragmenting

The fragment system provides facilities for splitting a BETA program into several parts in order to support separation of interface and implementation, variant configuration or separate compilation. The code editor supports this kind of fragmenting.

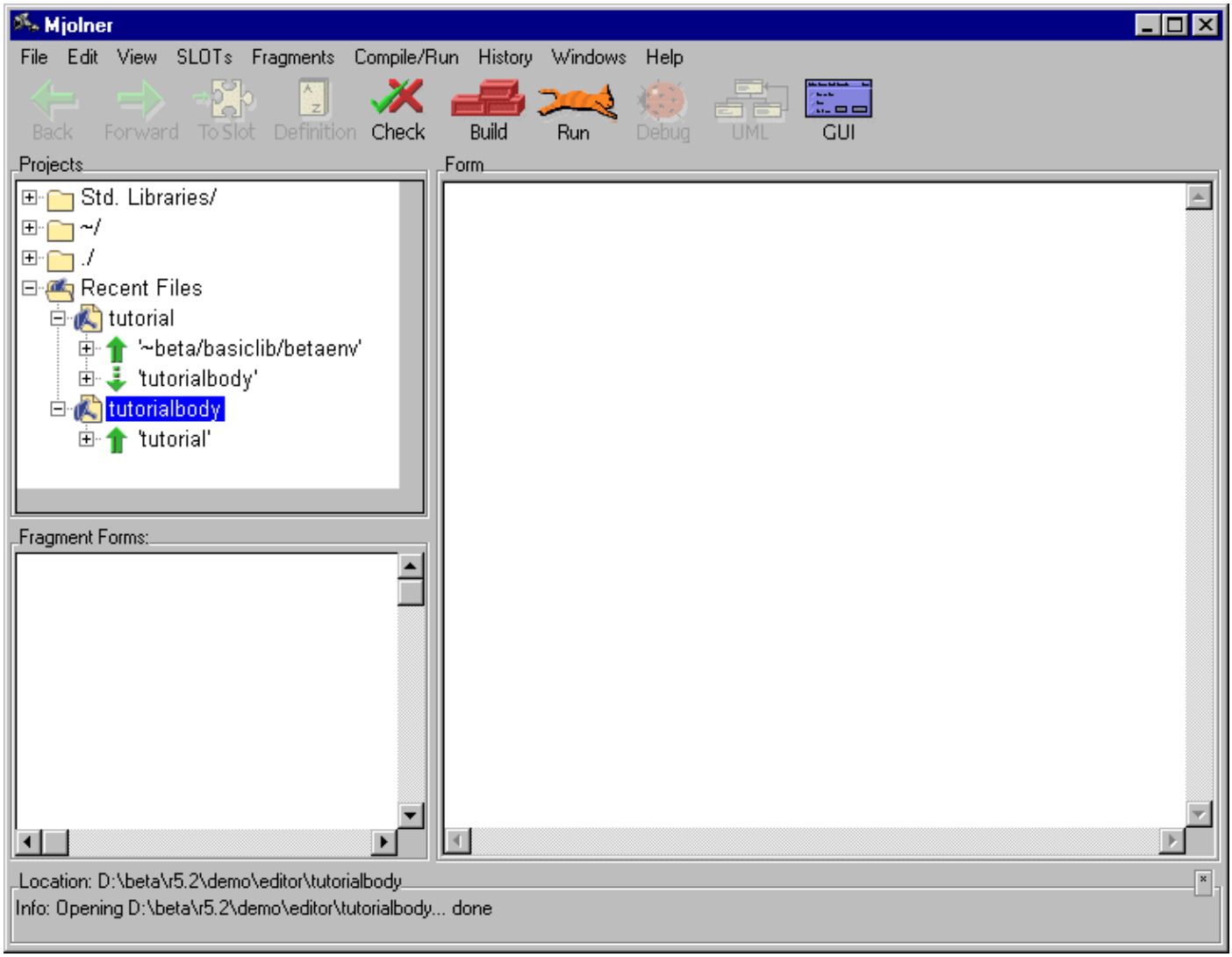
If, for example, a fragment form is going to be divided into an interface part and an implementation part the, SLOTs menu is very useful. Consider the small program hello:

Figure 32



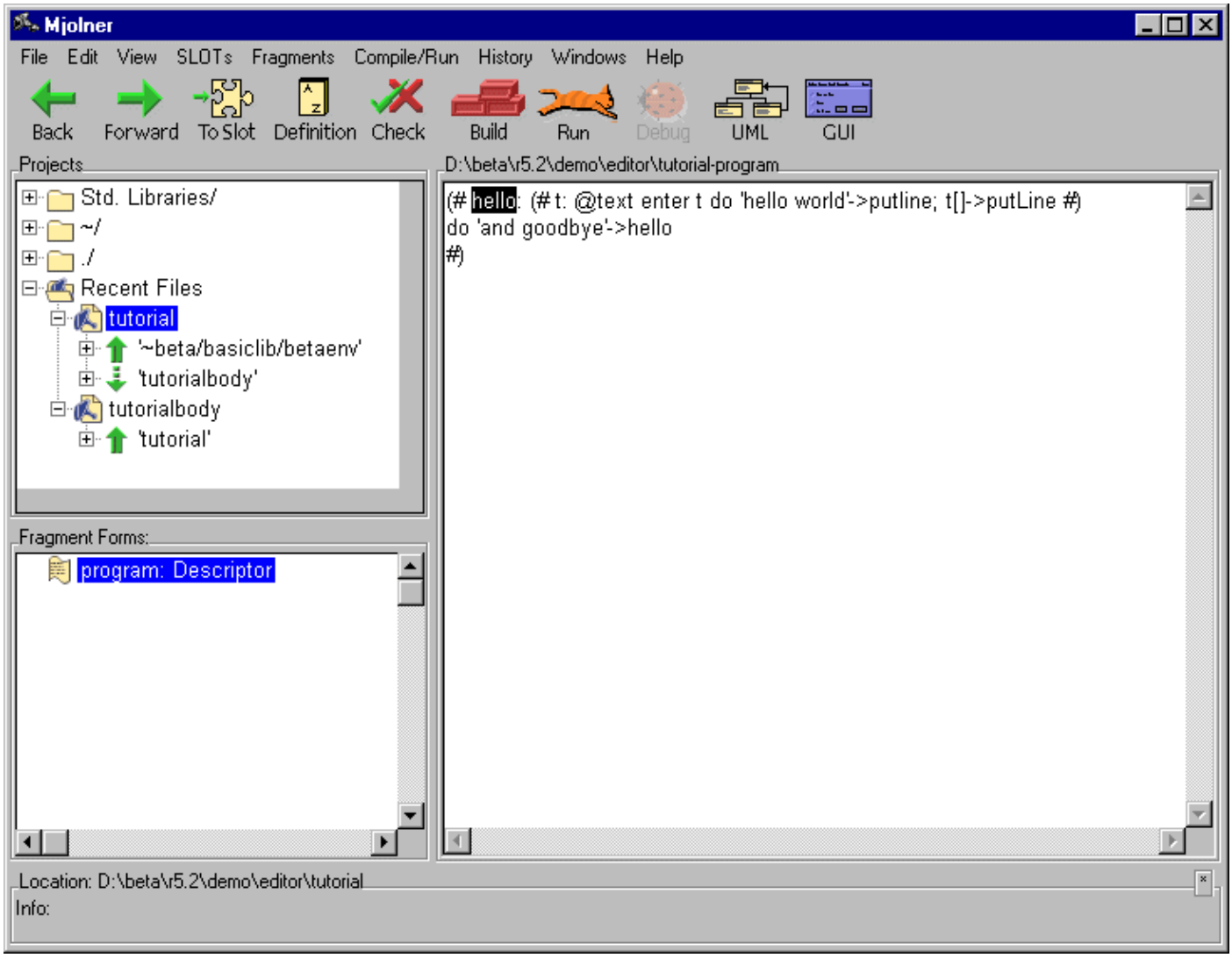
Let say we want to create a BODY file called 'hellobody' that contains the DoPart of the hello pattern. First we create the BODY file using the 'SLOTs:Create Implementation File...'.

Figure 33



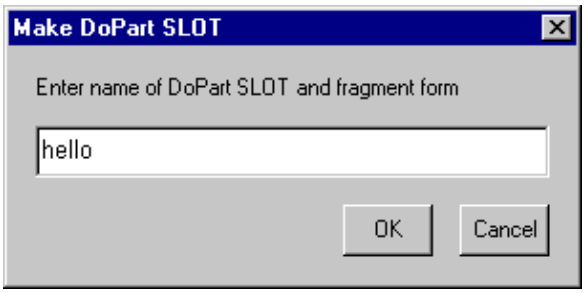
Notice that this new file has ORIGIN in hello. If we double−click on this ORIGIN, we return to the hello file, and we see that the property: 'BODY hellobody' has been inserted.

Figure 34



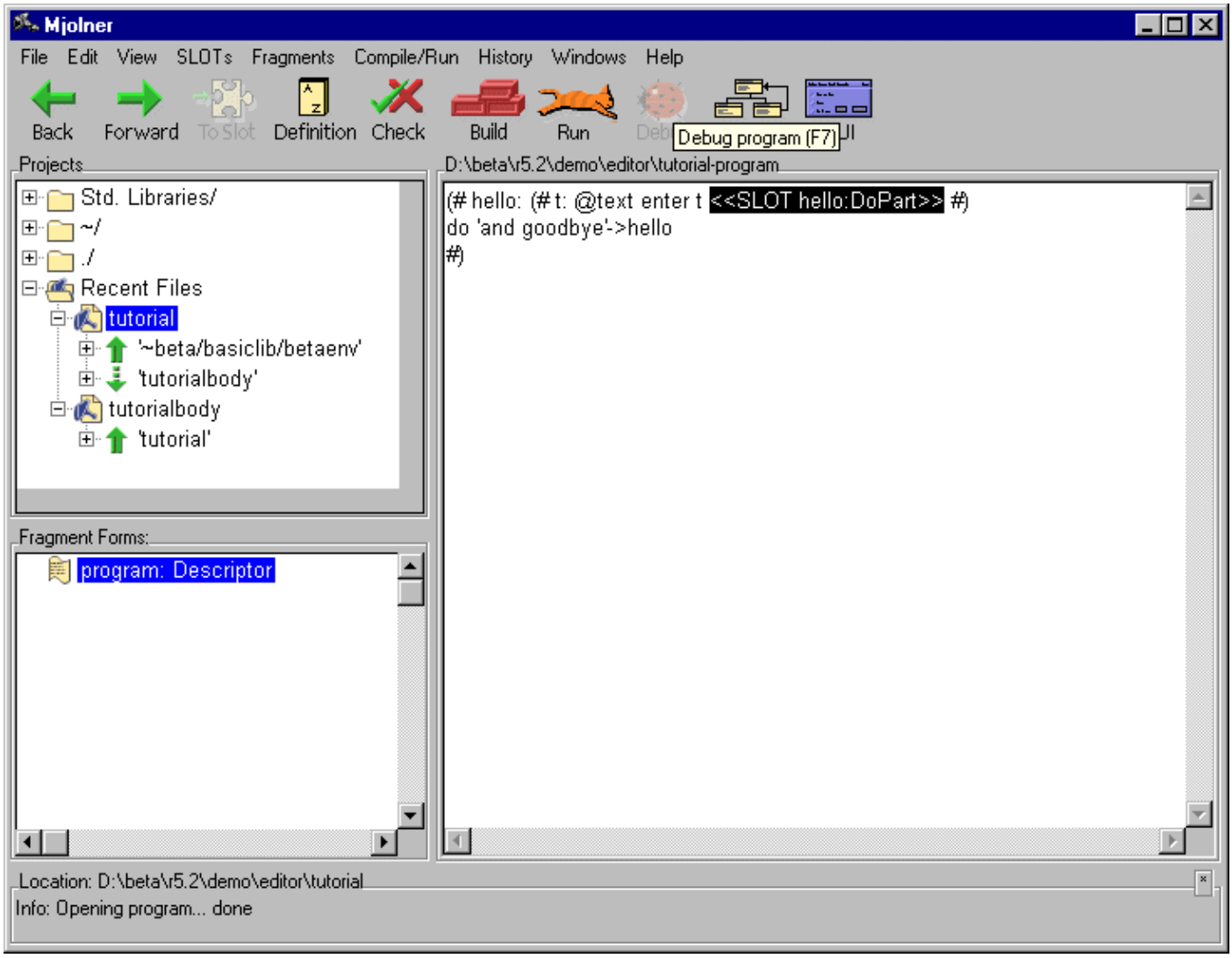
Now we are ready to hide the implementation of the hello pattern. This is done by means of the 'SLOTs:Hide Implementation...' command. For each DoPart in the selection you are asked whether to create a DoPart SLOT and move it to the BODY file.

Figure 35



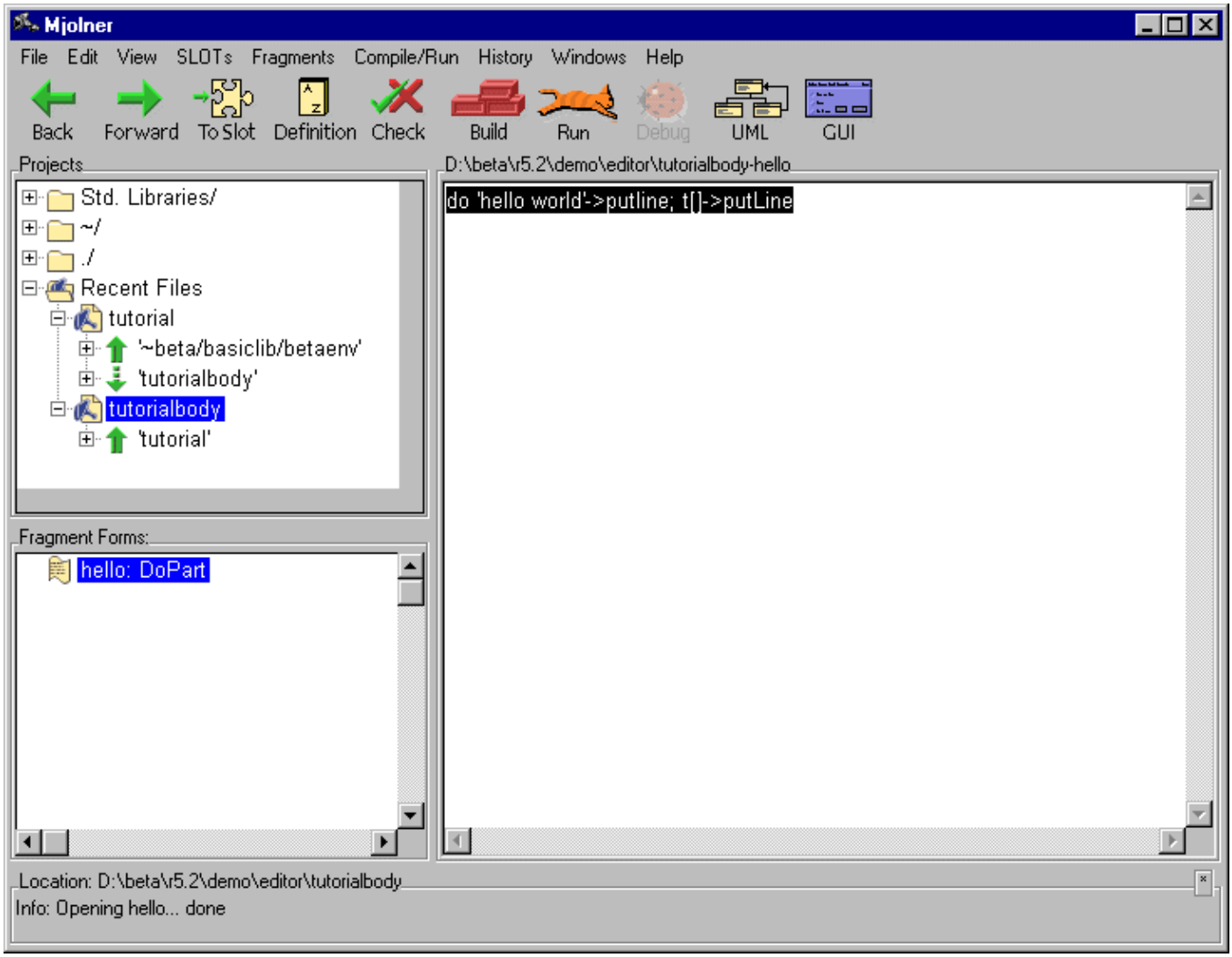
We accept and the result is:

Figure 36



and the BODY file looks like:

Figure 37

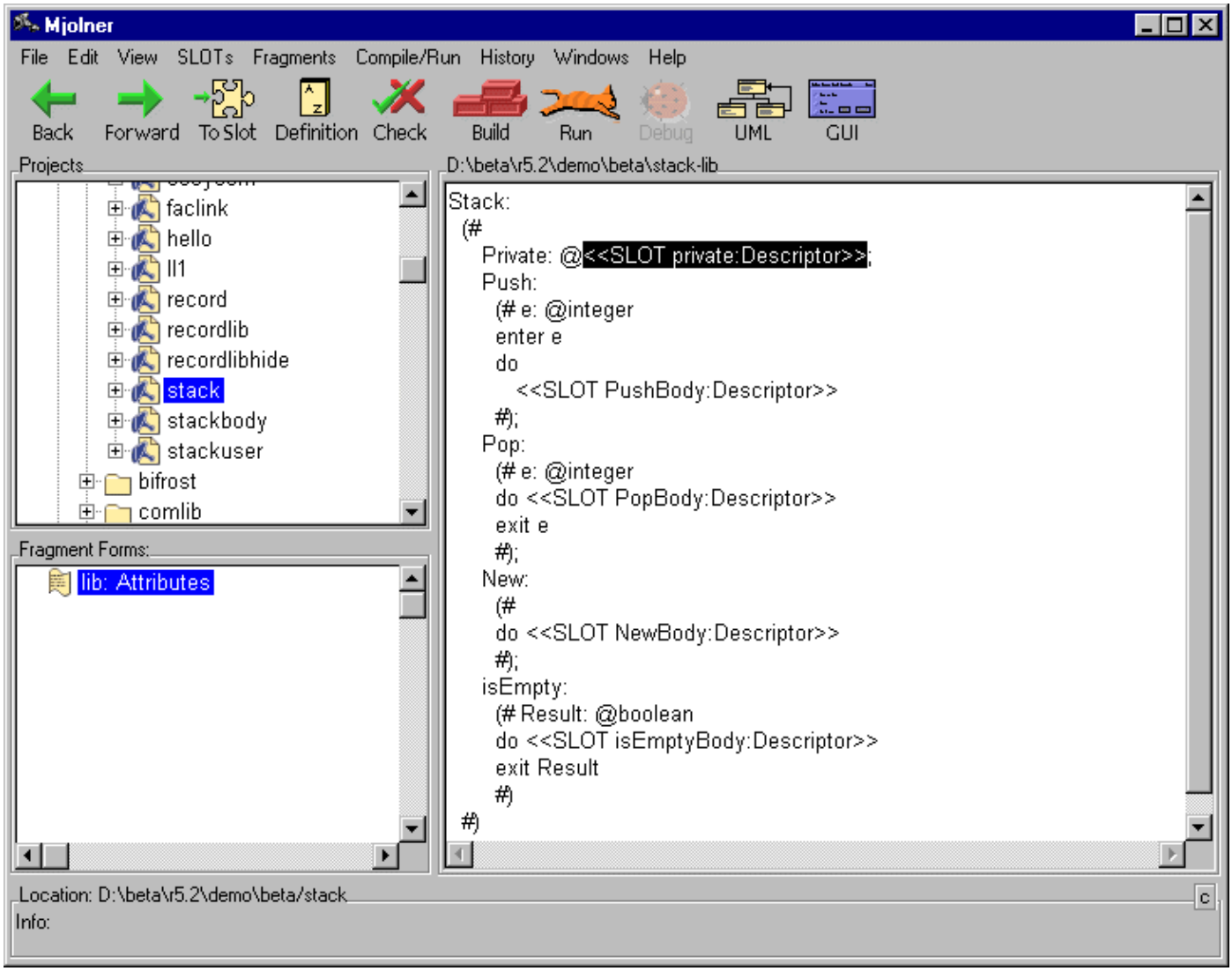


There are alternative ways to fragmentize BETA programs in mjolner.

4.5 Work Space

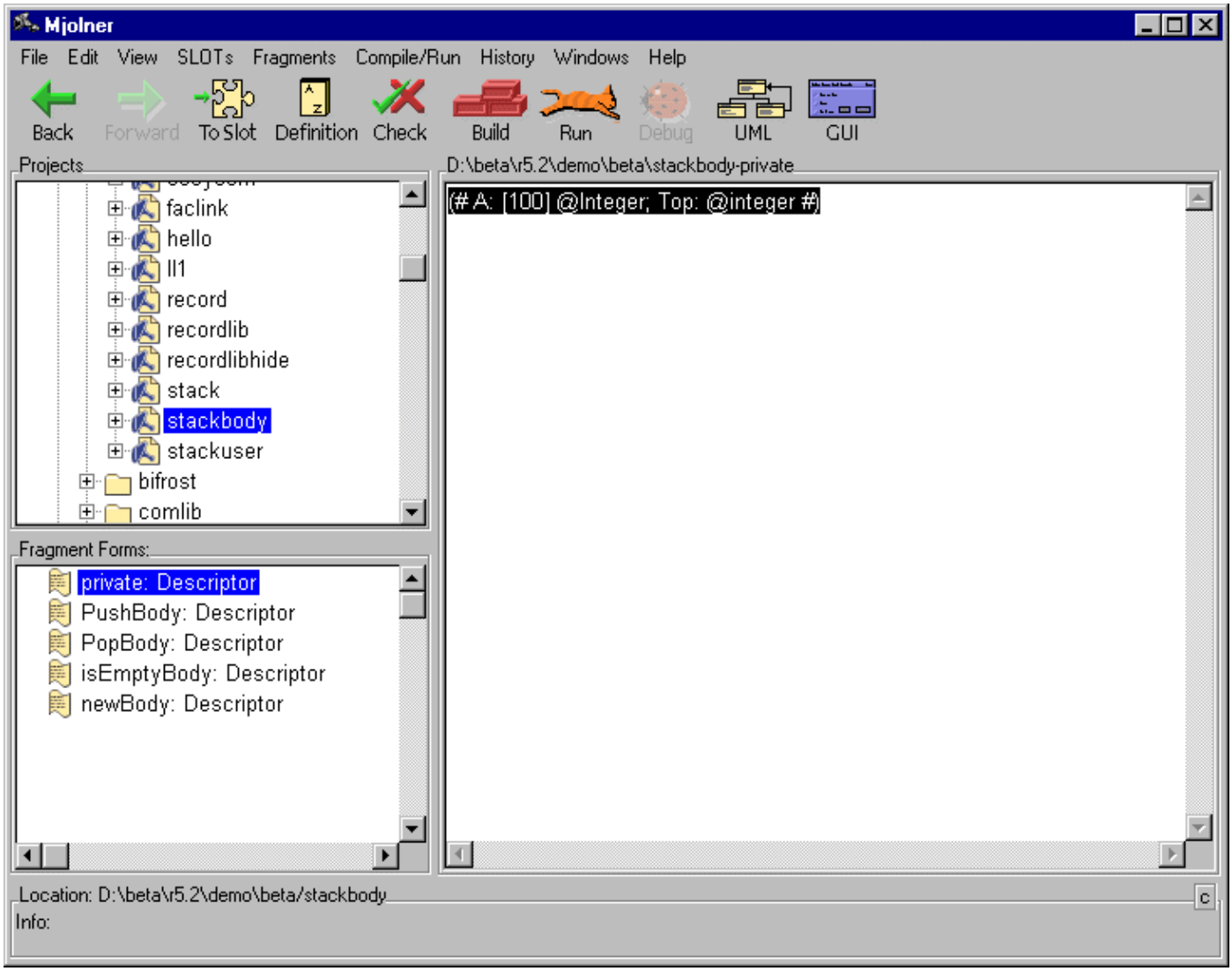
Consider the following interface file:

Figure 38



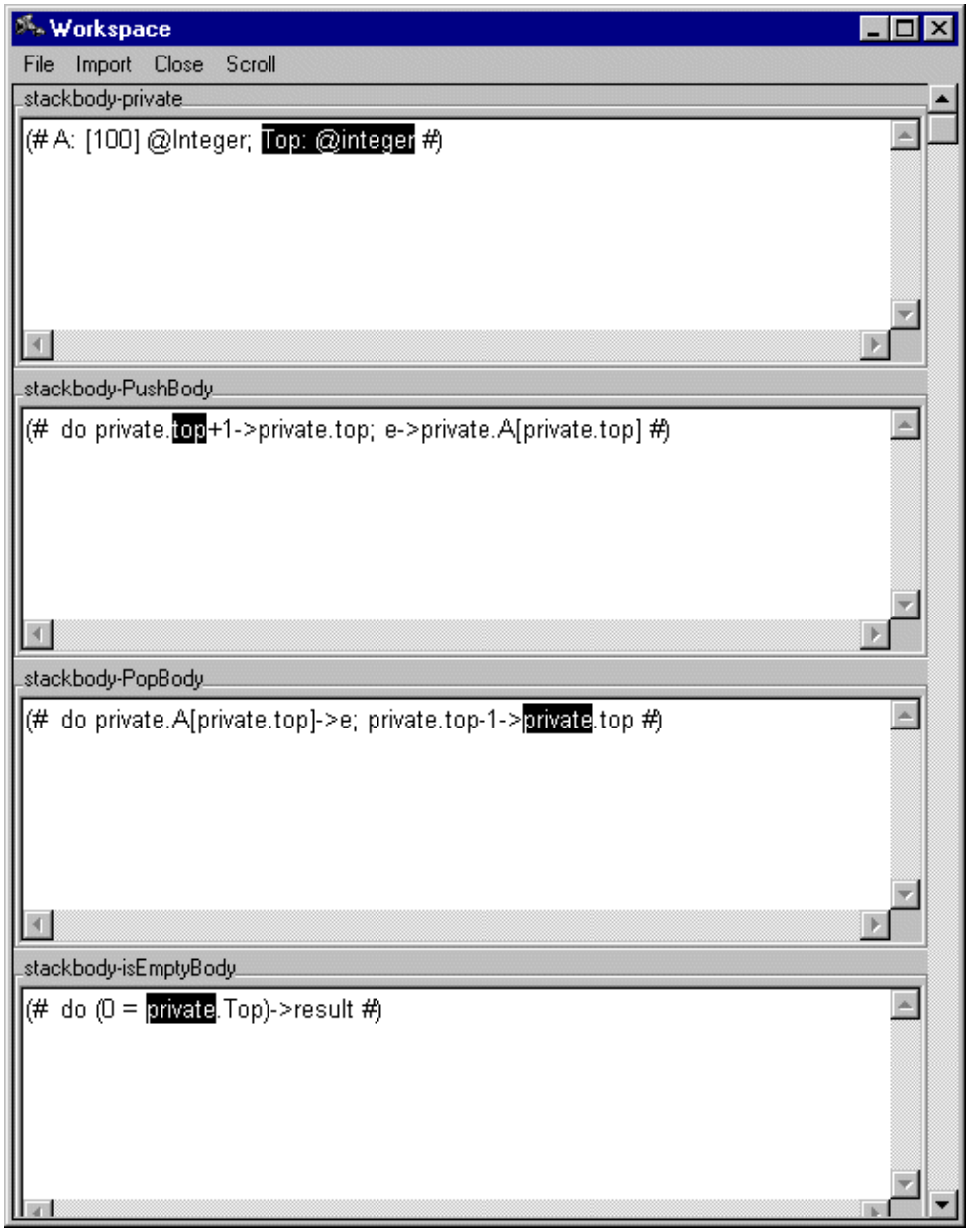
Each fragment form in the implementation file can be shown one at a time in the browser either by double−clicking on the SLOTs or by double−clicking on the BODY property in the group editor window to get the implementation file and then selecting them one at a time:

Figure 39



but an easier way is to get all or a subset of the fragment forms in a so−called work space window by using the Workspace command of the Windows menu. The Import menu is used to open a code editor on all or a subset of the fragment forms in the workspace window:

Figure 40


Workspace
File Import Close Scroll
stackbody-private
(# A: [100] @Integer; Top: @integer #)
stackbody-PushBody
(# do private.top+1->private.top; e->private.A[private.top] #)
stackbody-PopBody
(# do private.A[private.top]->e; private.top-1->private.top #)
stackbody-isEmptyBody
(# do (0 = private.Top)->result #)

5 CASE Tool

5.1 How to Get Started

5.1.1 Class diagrams

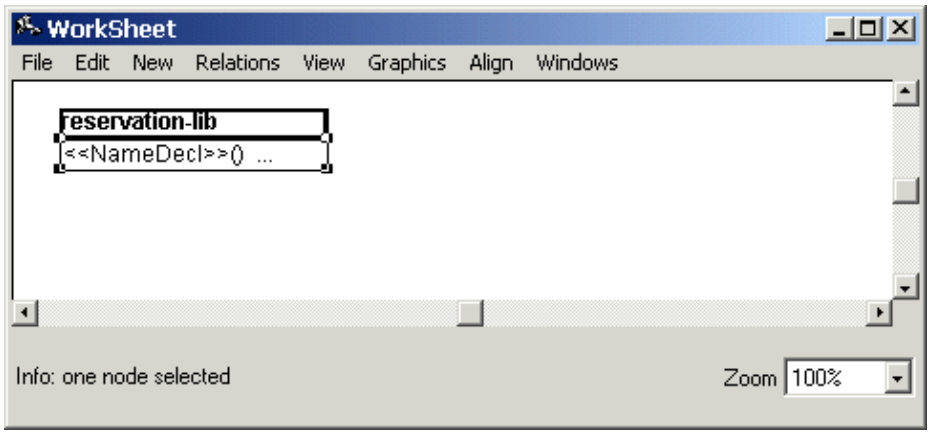
Freja is launched from within the Mjølner tool one of the following ways:

- In the source browser select the *UML* toolbar button . If the code editor is empty a new diagram with corresponding code may be created. If the code editor contains an Attributes fragment form or a Descriptor fragment form with declarations, the fragment form will be shown in Freja as a reverse engineered diagram of the code in the fragment form. Note that relations, like specializations and associations, will only be reverse engineered if the fragment has been checked (e.g. using the *Check* toolbar button). Also note that if a diagram already exists for the selected fragment this will be opened and possibly be updated.
- In the source browser open a BETA fragment form. Right−click anywhere in the code editor and select *Show UML Class Diagram* from the popup menu. This will have the same effects as mentioned above.

5.2 Editing

5.3 Creating a New Diagram

To create a new diagram use *Diagrams:New Diagram...* in the source browser. Below the command New BETA program has been used:

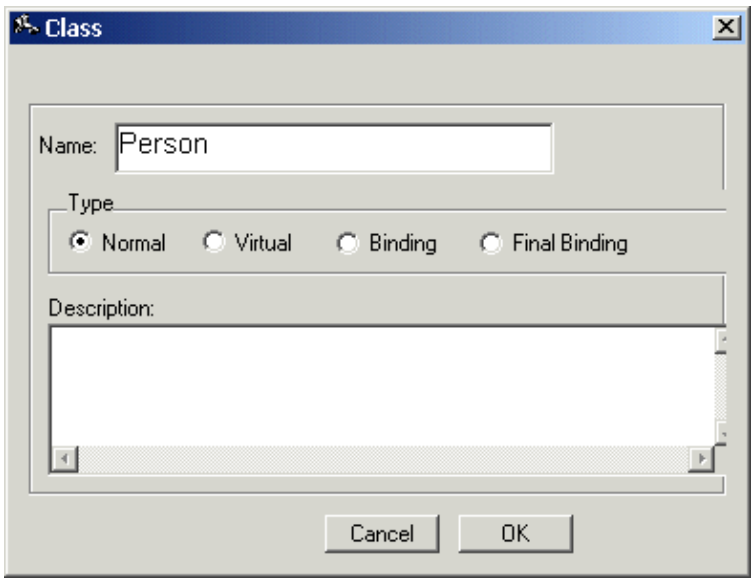


The title of the diagram corresponds to the name and the category (attributes) of the new BETA fragment:

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/freja/associations';
-- lib: Attributes --
<<NameDecl>>: (#  #)
```

5.3.1 Creating a New Class

When an attribute or a title of a diagram is selected it is possible to create a new class using the Class command in the *New* menu [**tip:** you may also right−click the class or attribute and select *New:Local Class...* from a popup−menu]. Doing this you get a dialog where the name of the new class can be typed in.

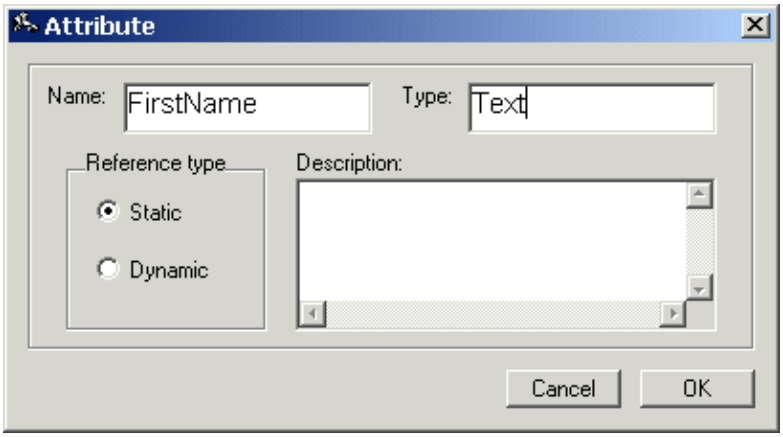


The BETA code corresponding to new class looks like this:

```
Person:(# #)
```

5.3.2 Adding Attributes and Operations

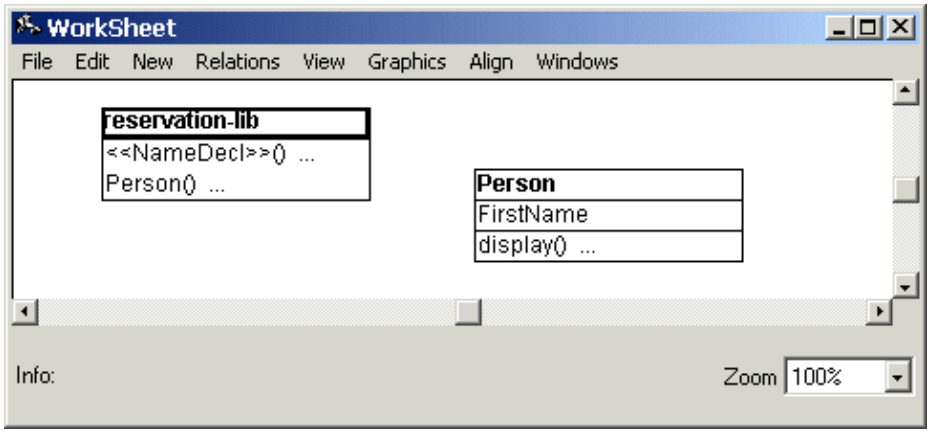
To add an attribute to a class, select the class and use *New:Attribute...* [**tip:** you may also right−click the class and select *New:Attribute...* from a popup−menu]. As above a dialog is presented and you can now type in the name of the new attribute.



The corresponding BETA code looks like this:

```
Person(# FirstName: @Text #)
```

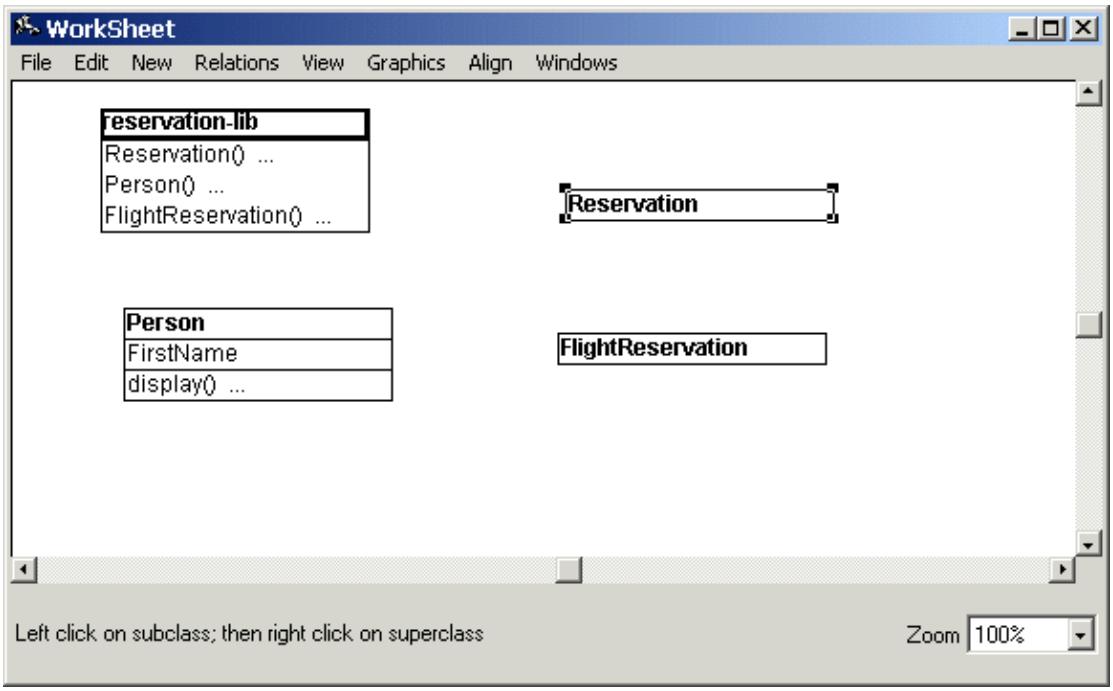
In the same manner as with attributes a new operation "display" can be added to the class using *New:Operation...*. This makes the diagram appear as follows:



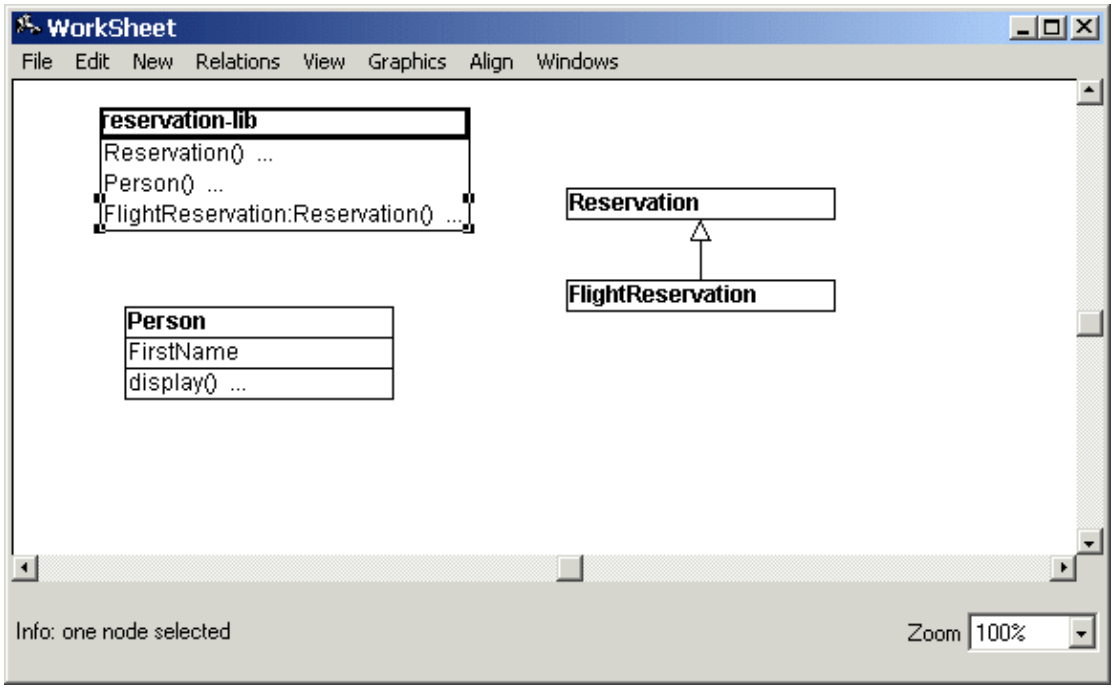
Notice the first item in the diagram, <<NameDecl>>() ..., is still there. This is meant as first template for a new class (the <<NameDecl>> part is a placeholder for the name of the class). In this case, where we have already created a new class using *New:Class...*, we could just delete this first item by selecting it and using *Edit:Clear*. However, we can also choose to use it as a template for a new class. To do this, select it and use the *Edit...* entry of the right−click popup−menu to replace "<<NameDecl>>" with the name "Reservation".

5.3.3 Specifying Specialization

In the following situation one class further, FlightReservation, has been added to the diagram. To specify that FlightReservation is a subclass of Reservation the *Specialization* command of the *Relations* menu is used. As the status message of the menubar states in the example below it is now possible to create a specilization by clicking the left mouse button on the subclass and hereafter clicking the right mouse button on to the superclass.



The result will look like this:



The code correspondingly changes from:

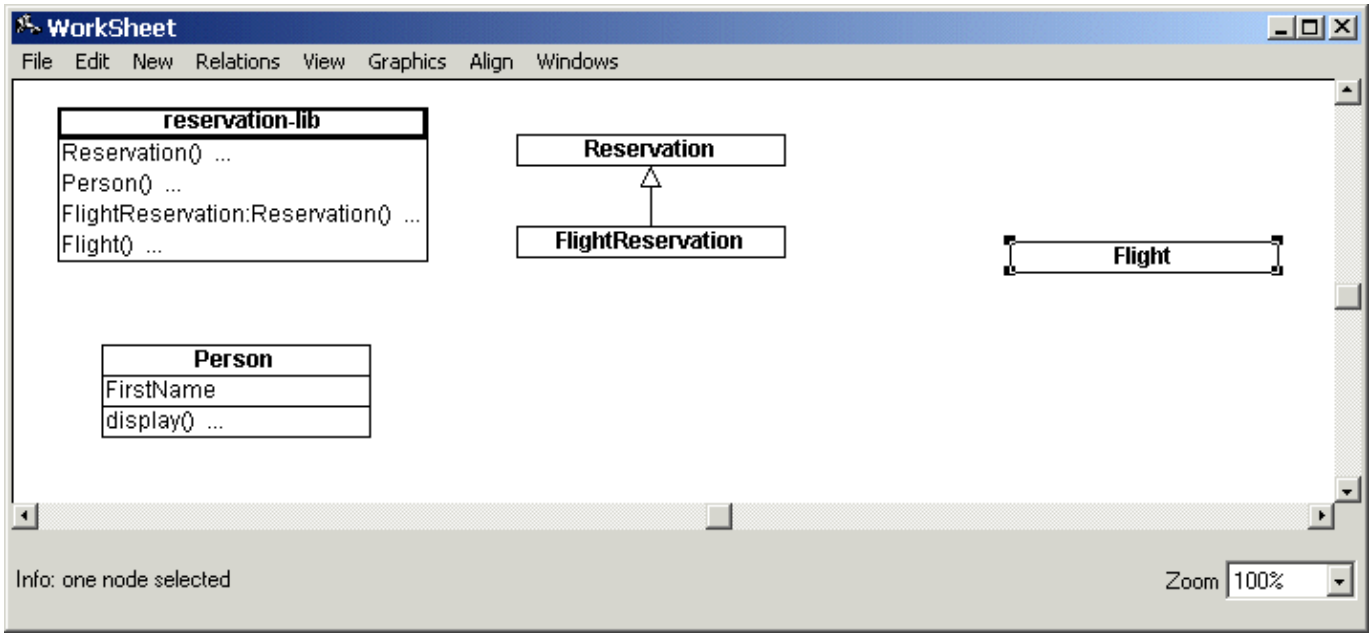
```
Reservation: (# ... #);
FlightReservation: (# ... #);
```

to:

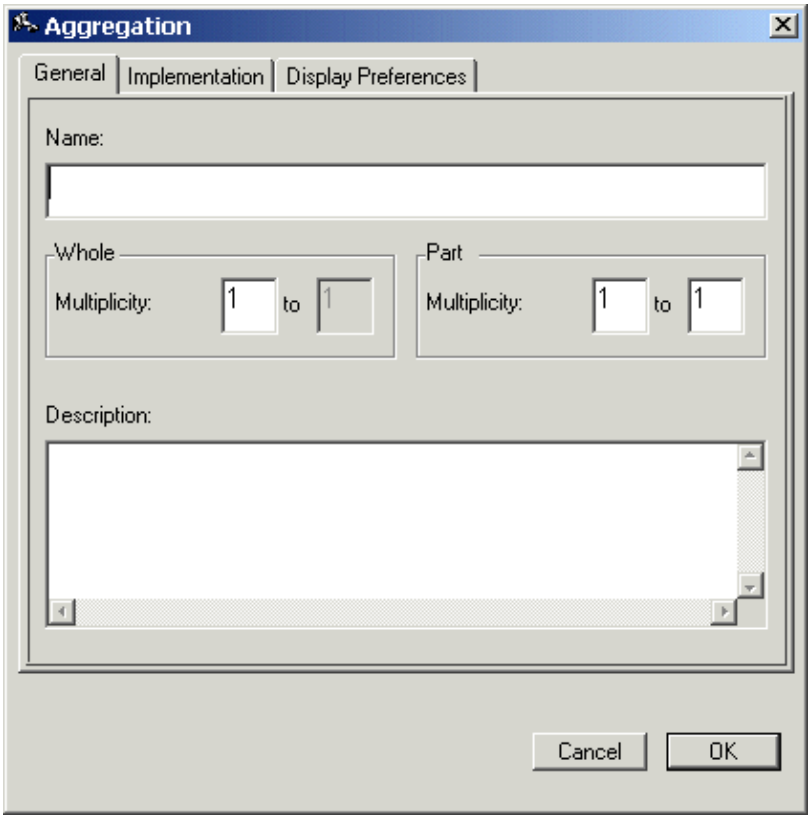
```
Reservation: (# ... #);
FlightReservation: Reservation(# ... #);
```

5.3.4 Specifying Aggregation

Below the class Flight has been added to the diagram.



We wish to specifify a by−reference aggregation between FlightReservation and Flight. To do this select *Aggregation* in the *Relations* menu. This brings up the following dialog:

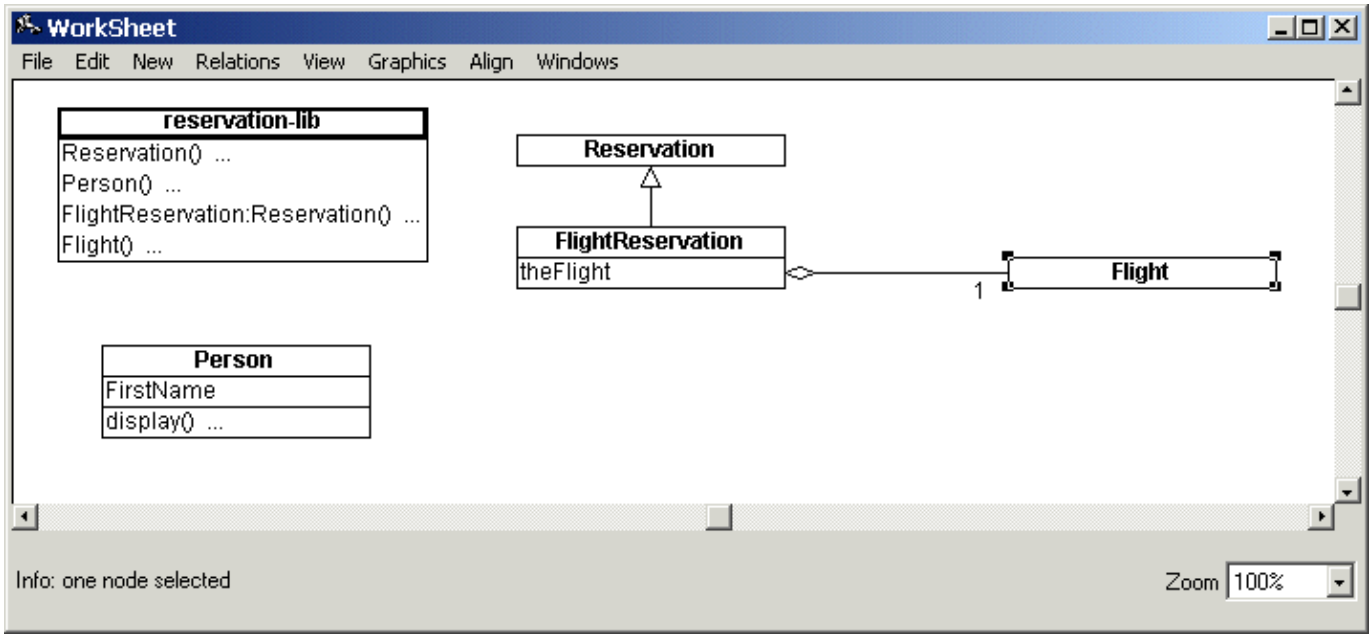


The aggregation dialog makes it possible to:

- give the aggregation a name
- specifify the multiplicities of the whole and the part
- specify the implementation of a one−to−many [1] aggregation; that is to specify if a repetition or one of the basic container classes should be used in the resulting code (using the *Implementation* tab)
- specify if the aggregation is to be implemented by−reference or by−value (using the *Implementation* tab)
- give a textual description of the aggregation
- set a number of preferences for the visual presentation of the aggregation (using the *Display Preferences* tab)

In this example we only specify the name of the aggregation (defaults are such that we get a one−to−one by−reference aggregation).

After choosing *Ok* in the dialog left click on the class designating the whole (FlightReservation) and, after that, right click on the class designating the part (Flight).



The BETA code correspondingly changes from:

```
Flight: (# #);
FlightReservation: Reservation(#  #)
```

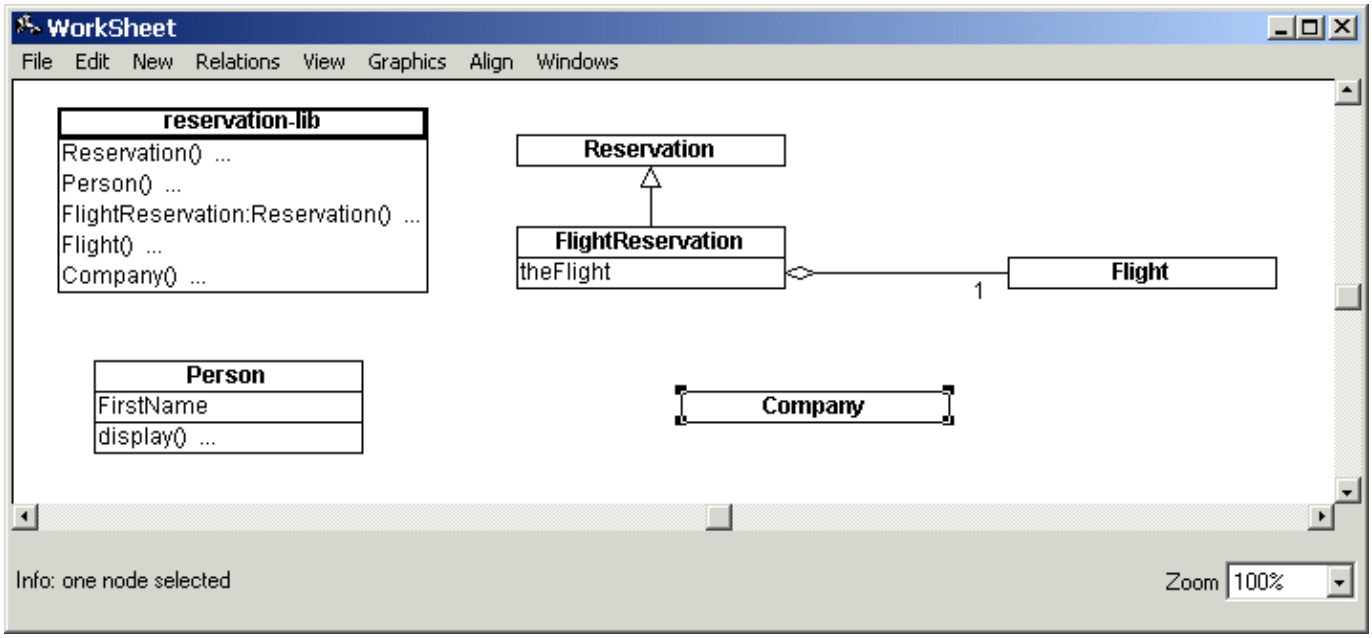
to:

```
Flight: (# ... #);
FlightReservation: Reservation(# theFlight: ^Flight #)
```

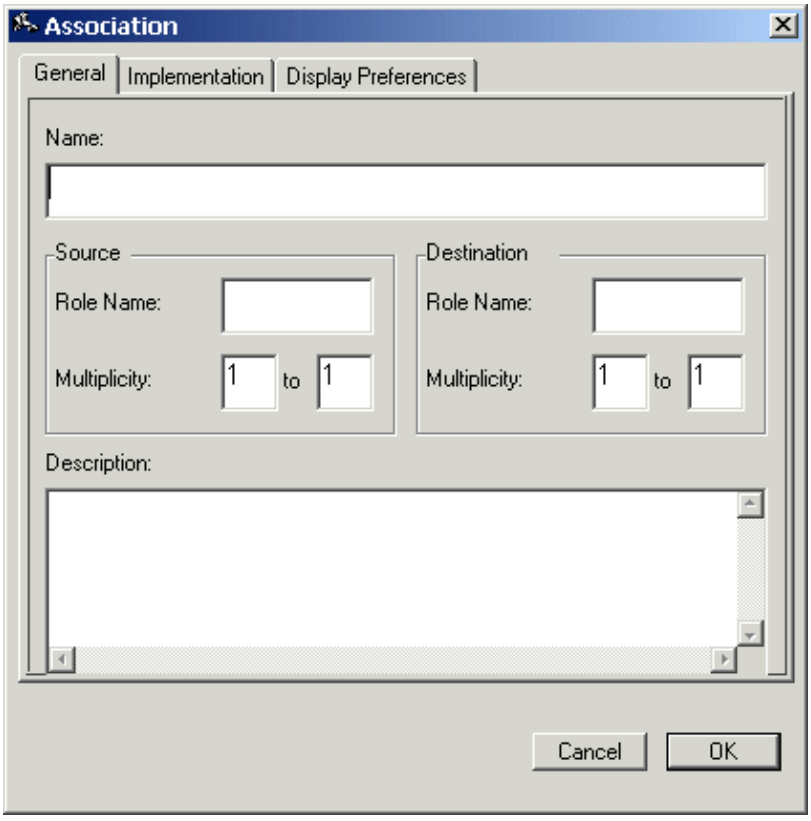
As can be seen the name of the aggregation is reused in the code and the fact that we created a by−reference aggregation is reflected by the dynamic reference implementation of the new declaration (had we instead chosen by−value a static reference would have been declared).

5.3.5 Specifying Association

Below a new class Company has been added to the diagram.



We wish to specify a one−to−many association between Company and Person. To do this select *Association* in the *Relations* menu. This brings up the following dialog:



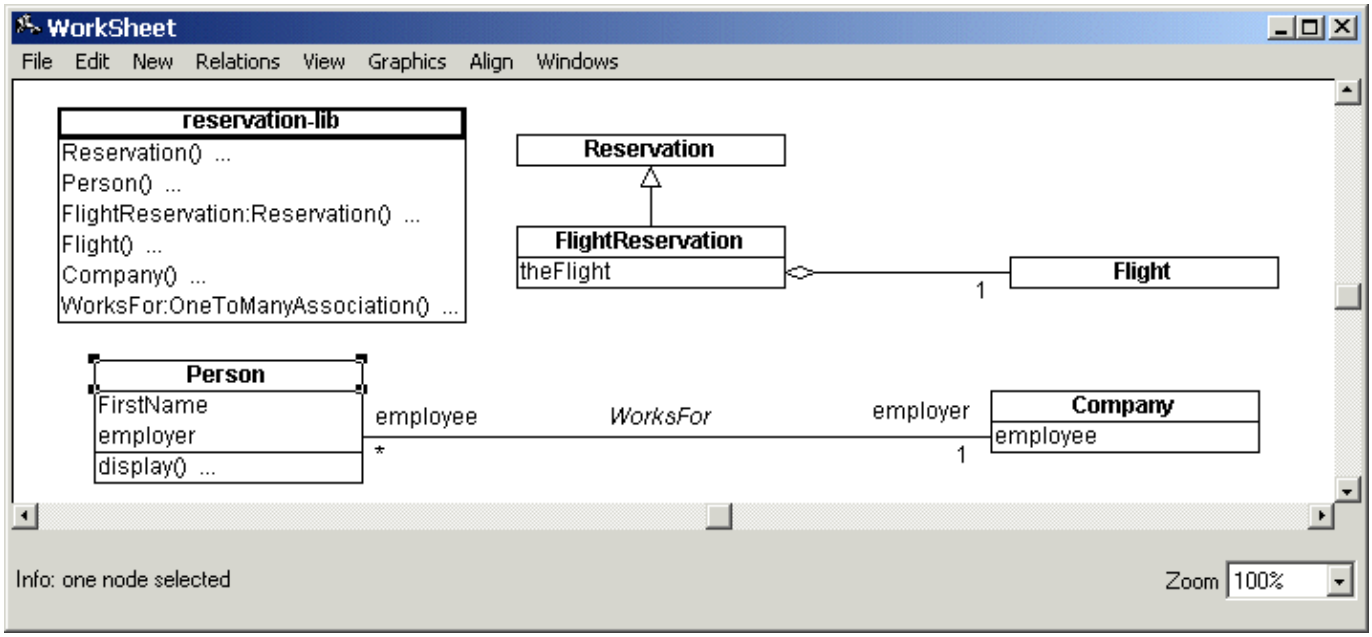
The Association dialog makes it possible to:

- give the association a name
- specify role names for both sides of the association
- specify multiplicities for both sides of the association
- specify if the association should be embeded or not; that is to specify if the generated code is to result in a separate association class or if it is to be embedded in the source and destination classes (using the *Implementation* tab) (see below)
- give a textual description of the association

- set a number of preferences for the visual presentation of the association (using the *Display Preferences* tab)

In this example we specify the name of the association, WorksFor, the name of the source role, employee, the name of the destination role, employer, and the multiplicities. Specifying that the multiplicity of the employee end of the association is more than one (in this case '*') automatically sets the implementation to the default, List.

After choosing *Ok* in the dialog left click on the class designating the source (Person) and right click on the class designating the destination (Company).



The resulting code looks like this:

```
Person: (# ...; employer: ^WorksFor #);
Company: (# employee: ^WorksFor #);
WorksFor: OneToManyAssociation
  (# oneType:: Company; manyElmType:: Person #)
```

Where the things added to code are the employer and employee declarations and the WorksFor class.

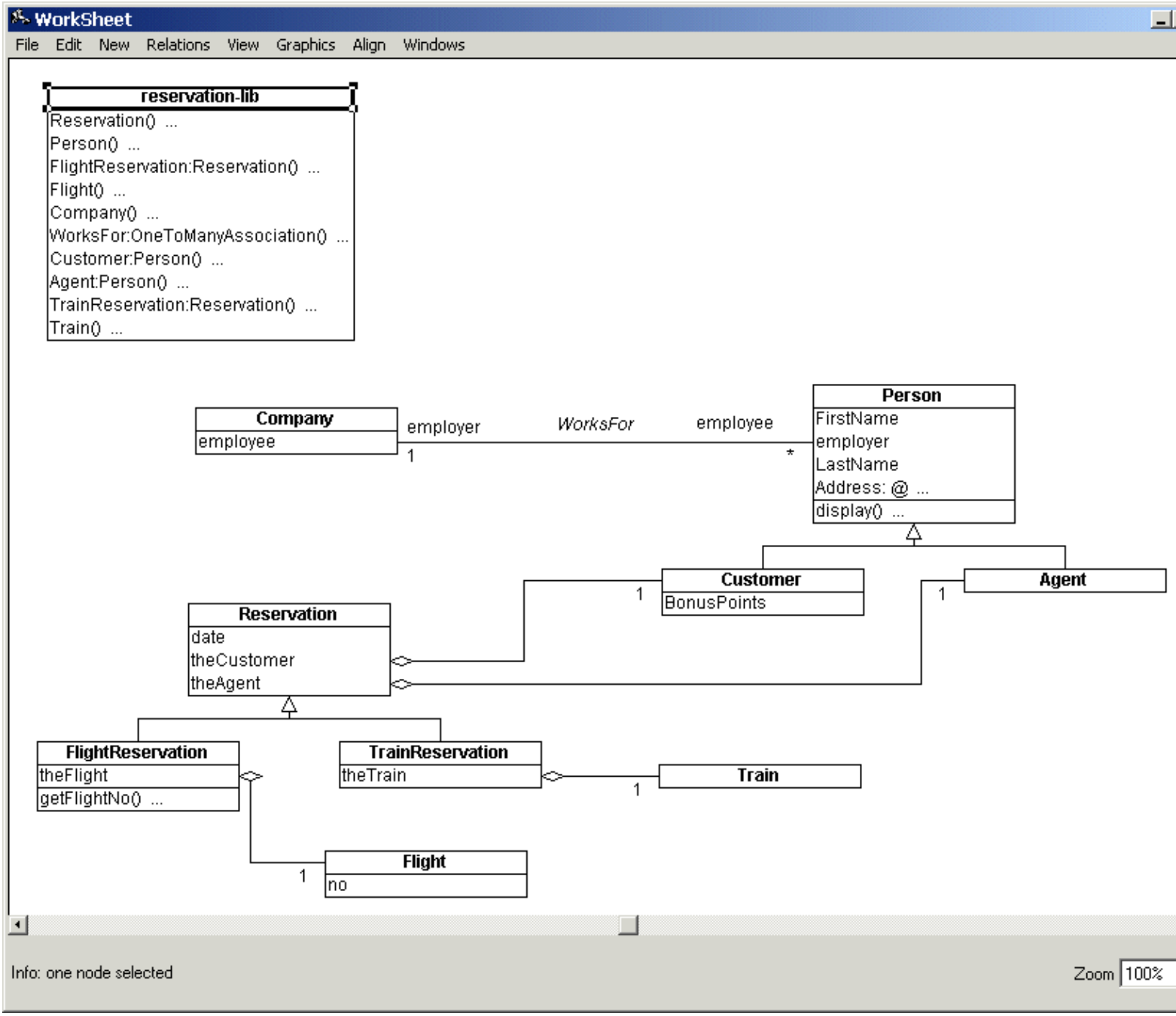
Had the embed option been chosen the code would have looked like this:

```
Person: (# ...; employer:@AssociationOne(# element:: Company #) #);
Company: (# employee:@AssociationMany(# element:: Person #) #)
```

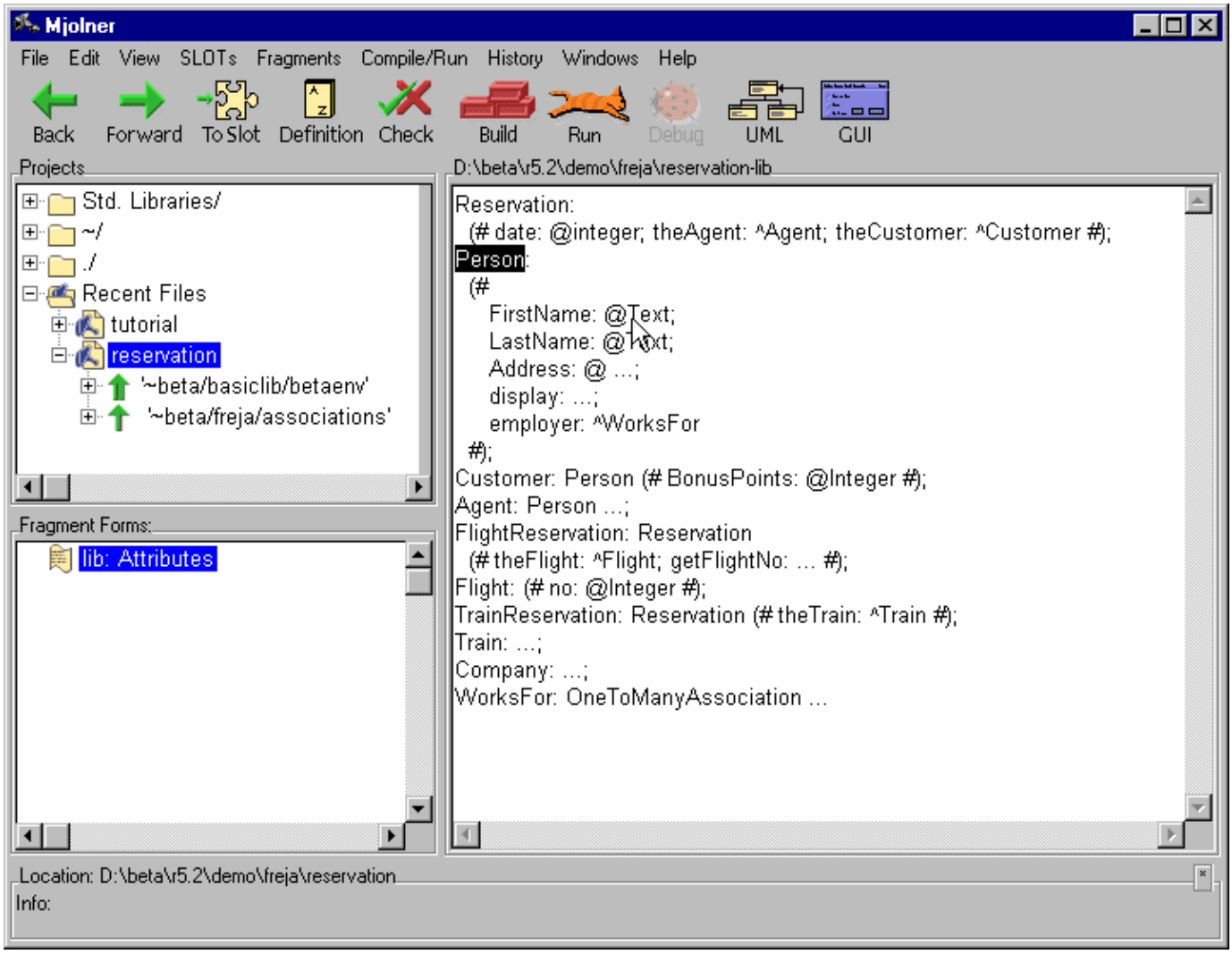
Notice that in this case no separate association class is generated.

5.3.6 Completing the Code in Code Editor

Consider the following design diagram which could be considered as complete at the design level:



To fill out the code details, the code editor, the textual representation editor must be activated:



The textual representation of the design might, however, not be complete. For example, it still contains operations with empty do−parts. To fill out the do−part of the getFlightNo operation in the FlightReservation Class select the entire getFlightNo pattern in the code editor. Enter text edit mode and type in the appropriate action sequence. For example:

```
getFlightNo:
  (#
     do theFlight.no->putint
   #)
```

Notice: When entering text edit mode with the entire getFlightNo pattern selected we are warned that such editing *might* have the consequence that parts of the diagram have to be rebuildt. In this case, however, we choose to ignore the warning and enter text edit mode anyway. As it turns out the diagram was not affected at all by the edit because only a do−part was changed (action sequences do not have a visual representation in the diagrams; only static structures have).

After filling out more code in the code editor, e.g. further action sequences of the objects, the compiler is activated in the *Compile/Run* menu of the code editor.

If the compiler reports a semantic error, the corresponding node is also selected in Freja.

When the program is semantically correct and it can be executed and tested. In this phase Freja will typically not be activated. When the program has been tested and considered fulfilling the requirements, the design diagrams may have become obsolete, and need to be updated, i.e.

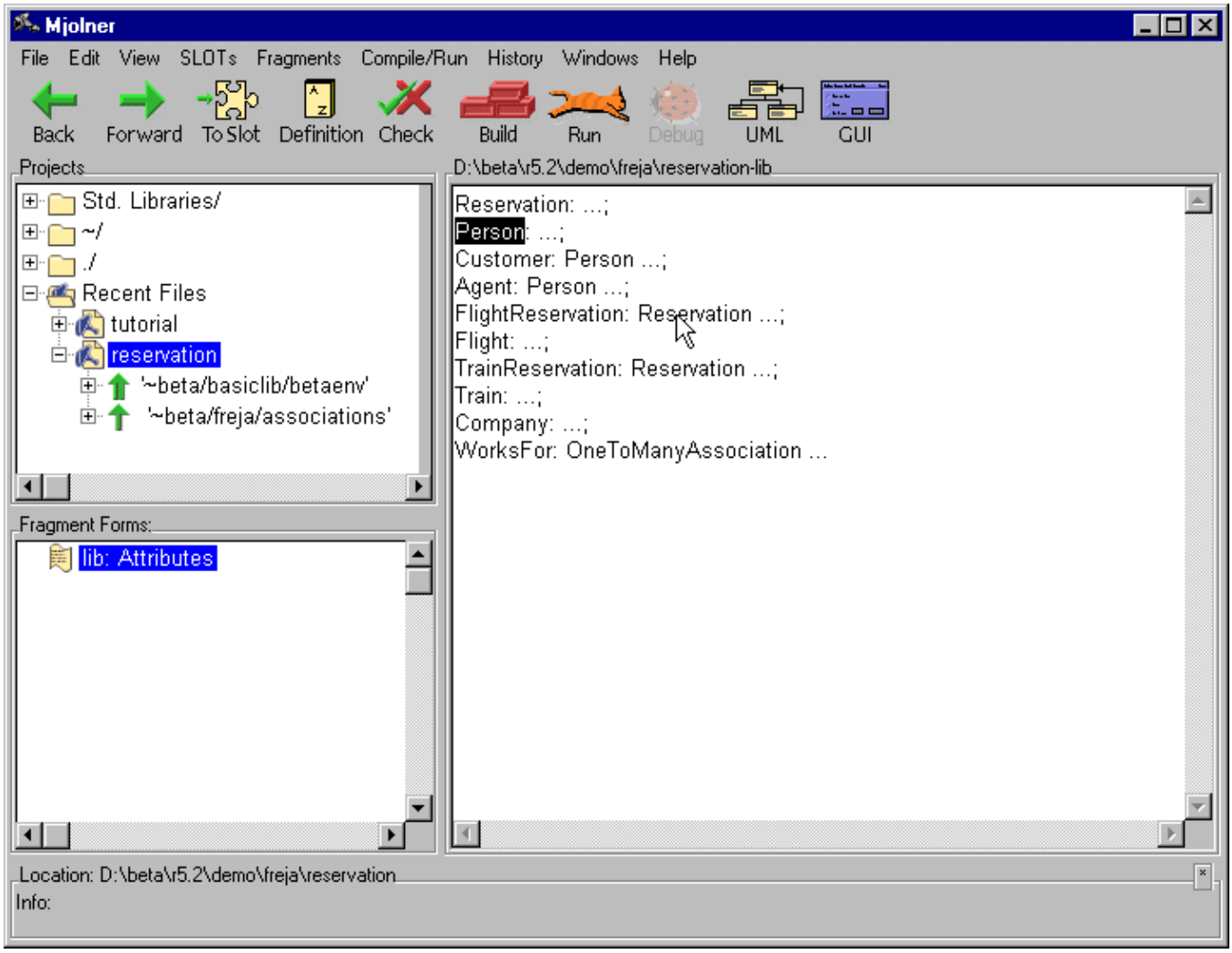
reverse engineering is necessary. This is done by activating Freja on the BETA program.

[1] An aggregation is said to be one−to−many if the multiplicity of the part is one and the multiplicity of the whole is more than one

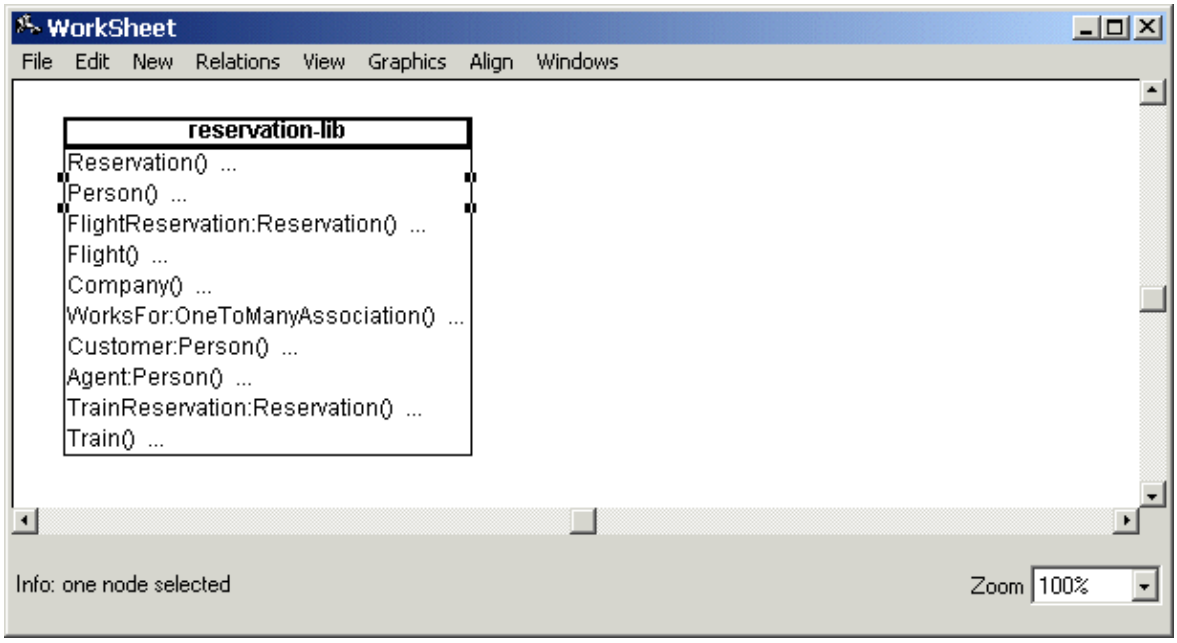
5.4 Reverse Engineering

5.5 Class Diagrams

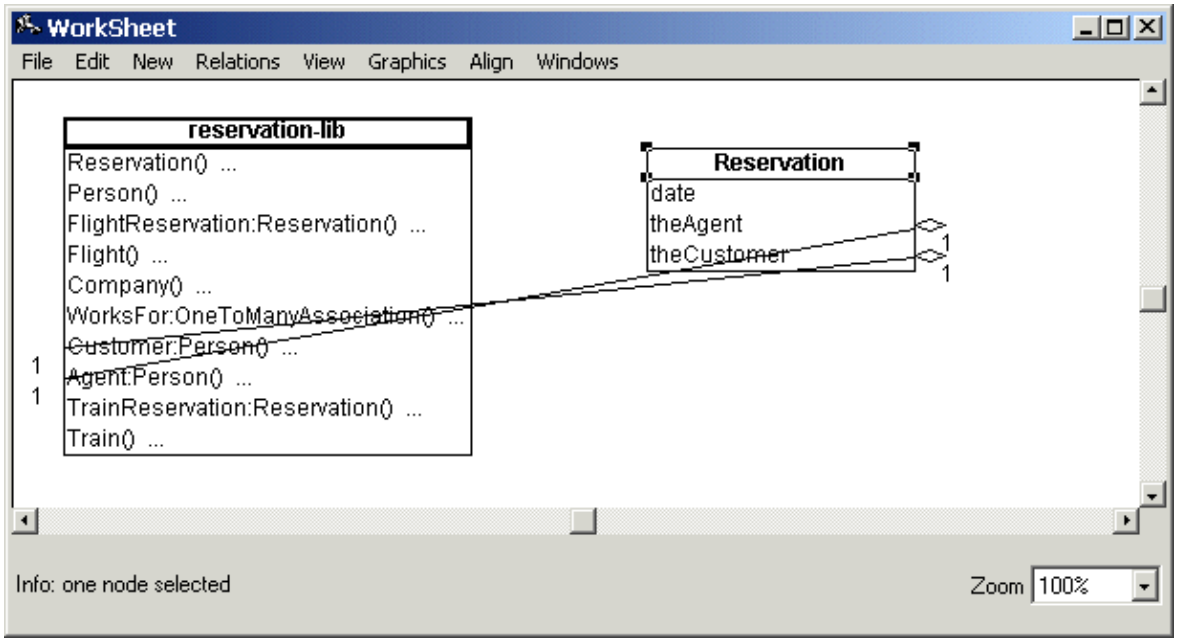
To extract the design diagrams from the BETA code, the program is opened in the CASE tool. Consider the following program shown in the code editor:



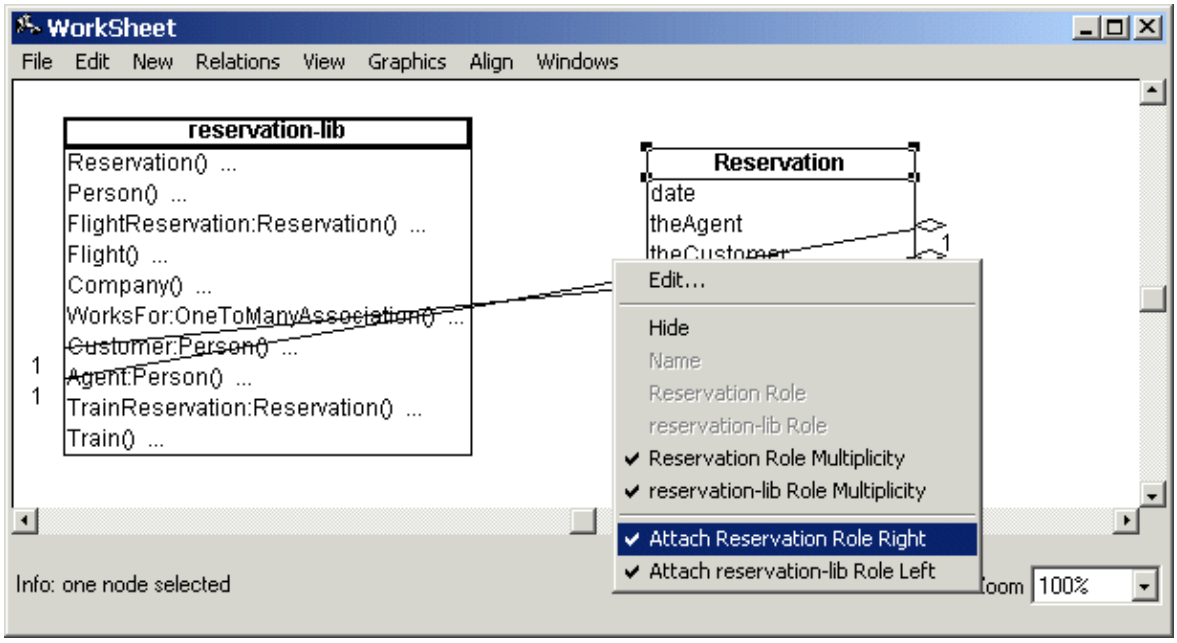
When opening this program in Freja the following diagram initially appears:



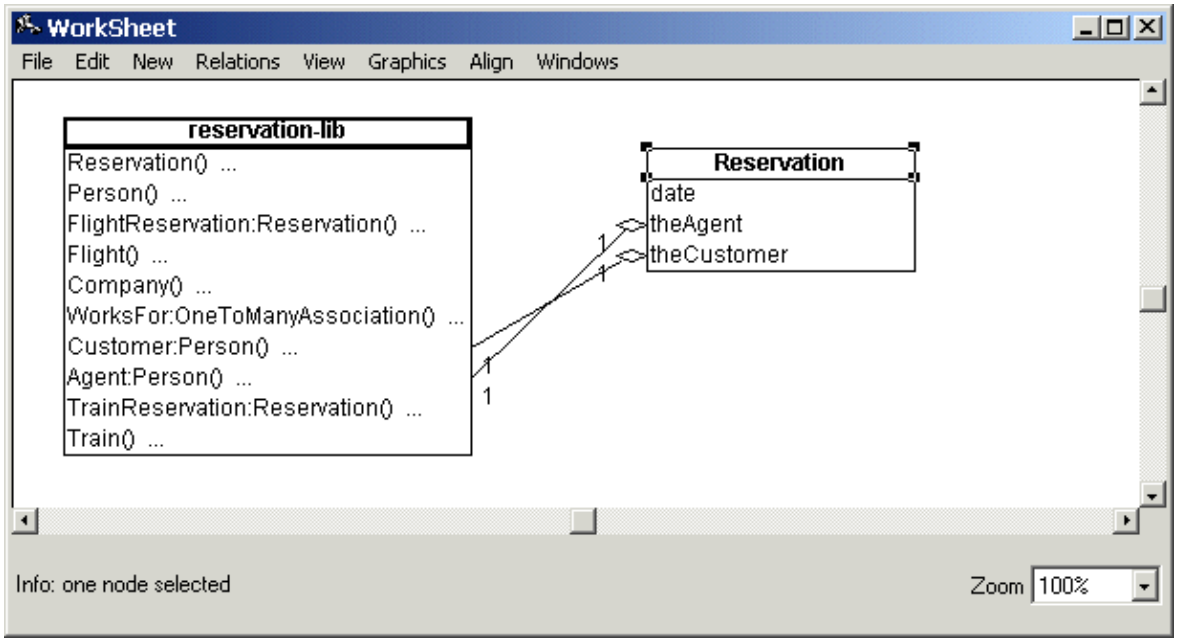
This is an abstract design view of the program. A detailed design diagram can be obtained by detailing selected parts of the diagram. If an entry contains three dots (...) it can be detailed. Below the class Reservation is detailed by double−clicking on the Reservation attribute:



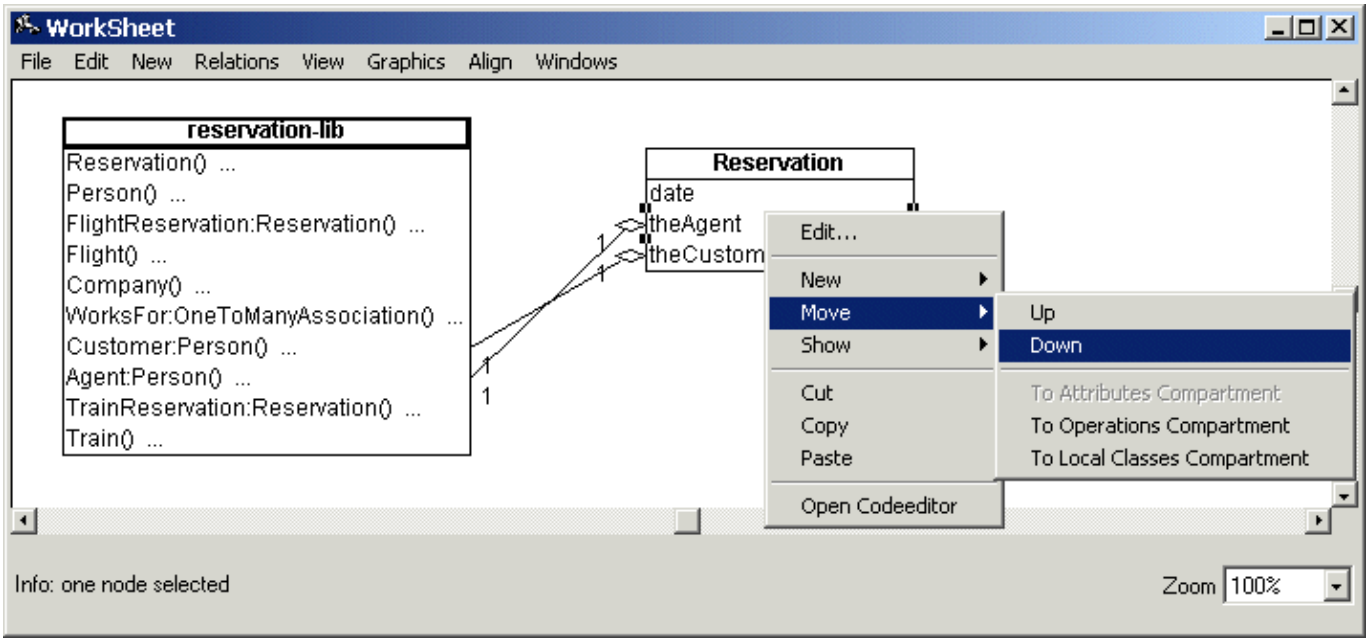
This intial layout of course can be changed to make the diagram appear graphically more appealing. To prevent the shown association connectors from crossing over the class boxes, point the mouse cursor to a connector and right−click it to get a popup−menu:



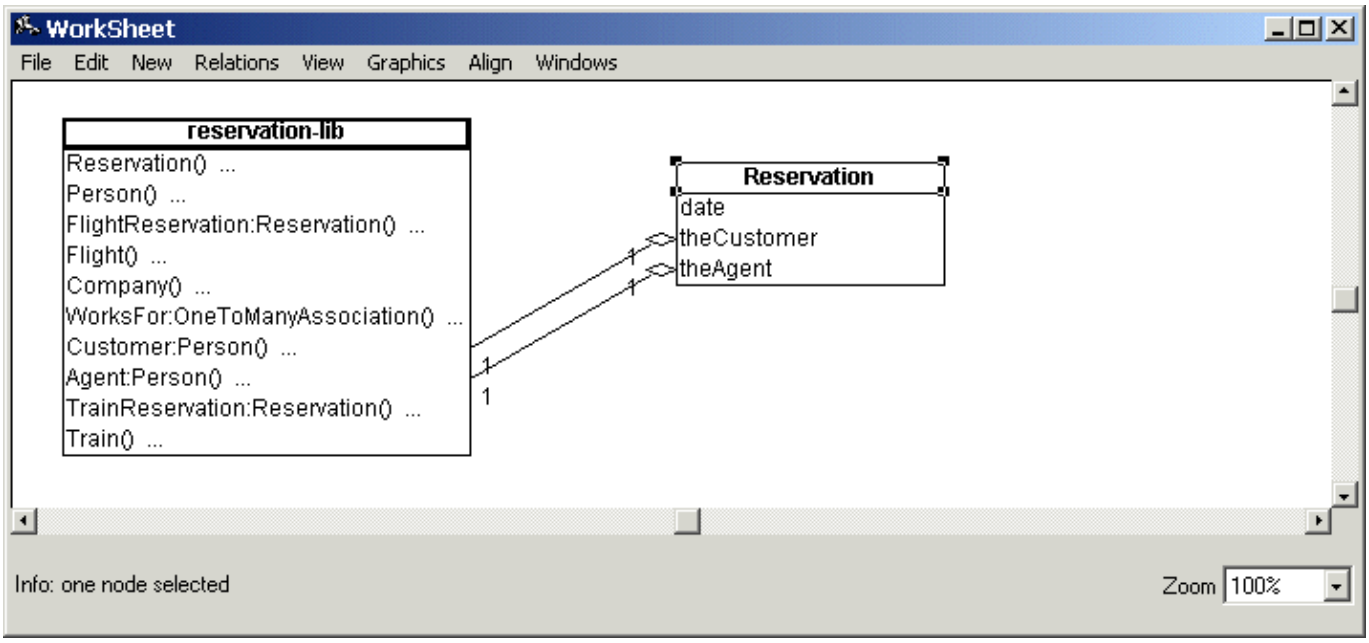
Select the checked item, *Attach Reservation Role Right*, to make one end of the connector attach itself to the left side of the class instead of the right. Repeat this operation to change the other points of attachment of the two connectors, to make the diagram look like this:



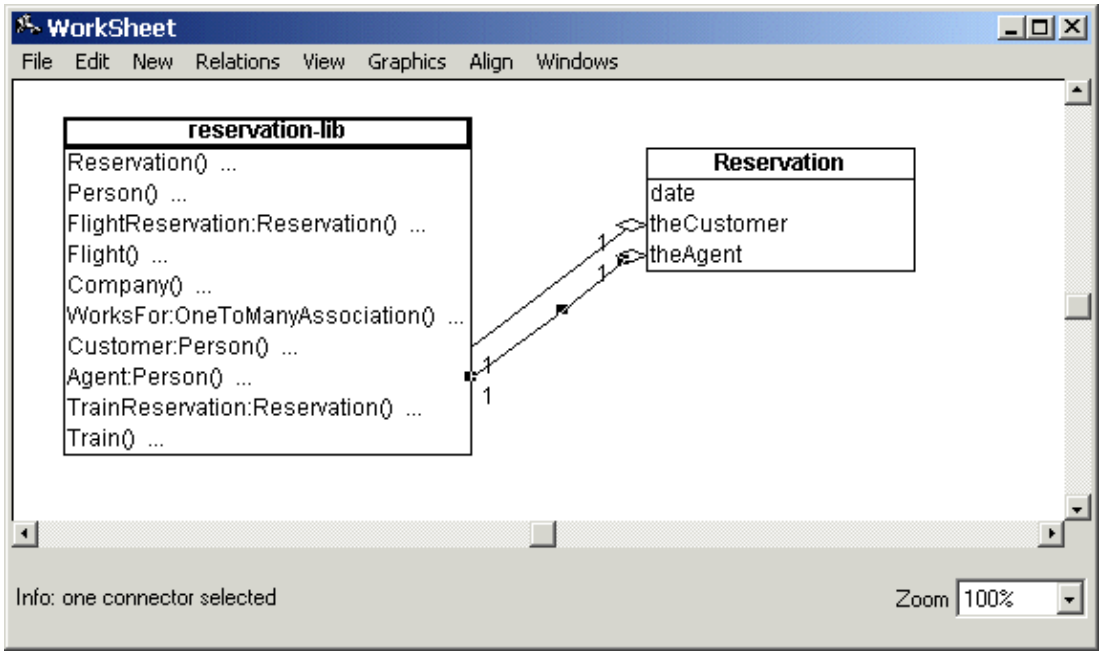
To prevent the two connectors themselves from intersecting one of the endpoints has to be moved. Right−click the attribute theAgent of the Reservation class:



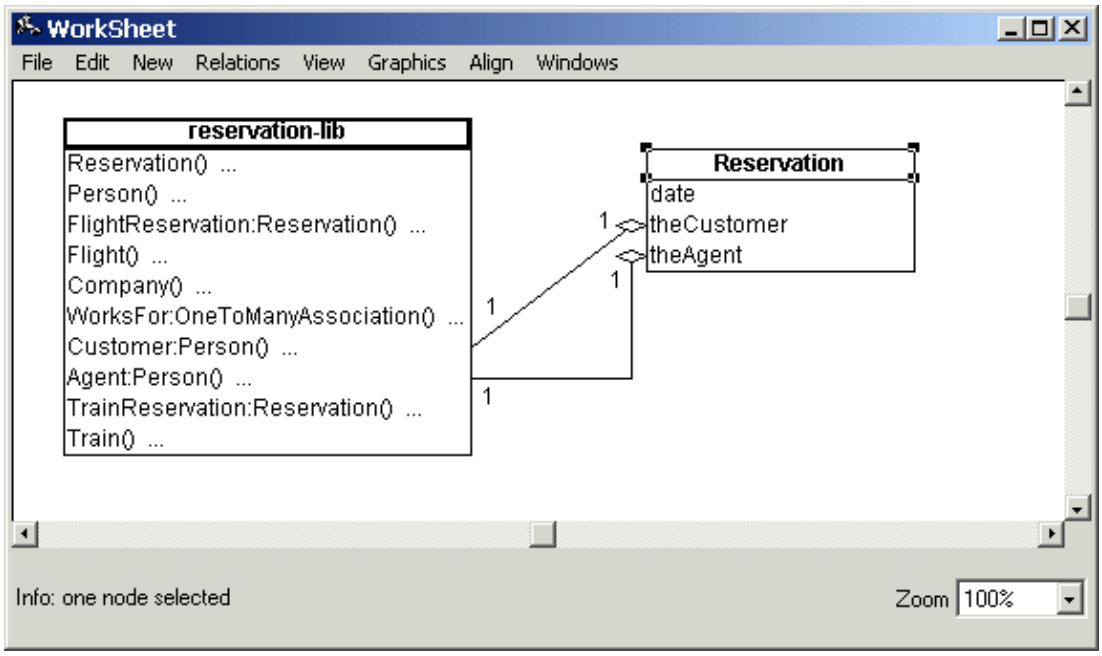
Select *Move:Down* to move the attribute one position down in the class box:



To further visually seperate the two connectors, you might want to bend one connector away from the other. This can be done by inserting a new edge somewhere on the line constituting the connector you wish to bend. To do this, double−click the connector in the place where you want the new edge inserted:



A small rectangular dot appears where the edge was inserted and this new dot can now be dragged to a new position to obtain the desired bending of the connector:



Further classes may be detailed and the layout adjusted in ways similar to above.

5.6 The Notation

The graphical design notation used in the CASE tool, Freja, is UML.

UML is the result of the efforts of Jim Rumbaugh, Grady Booch and Ivar Jacobson [2] to unify their methods: OMT (Object Modeling Technique),Booch and OOSE (Object−Oriented Software Engineering).

The current version of Freja supports Class diagrams.

The class diagrams illustrate the static structure of the model with symbols for class, aggregation, association and specialization. In Freja design diagrams and code are tightly integrated. As a consequence class diagrams can also be seen as an illustration of the static structure of a BETA program. In this sense UML class symbols correspond to BETA patterns, UML aggregations to BETA whole−part and reference compositions and UML specializations to BETA specializations. A symbol that is not directly found as a language construct in BETA is an association.

The most recent updates on the Unified Modeling Language are avialable via the world wide web: http://www.rational.com.

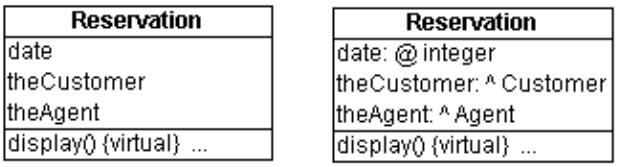
---

[2] All employees of Rational Software Corporation

5.6.1 Class Diagrams

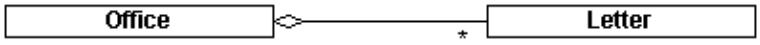
5.6.1.1 Class

A class is shown as a box with a title (the name of the class) and an optional list of local attributes, operations and classes. The two class symbols shown below are actually two different views on the same class (Reservation). The left symbol showing only the names of the attributes and the right symbol showing full type information of the attributes. In the following examples one or the other view will be used depending on the illustrative purposes of the example.



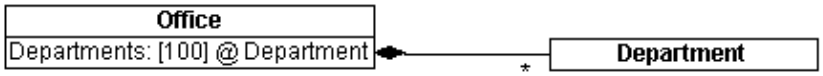
5.6.1.2 Aggregation

Aggregation is shown as a line connecting two class symbols. A diamond is attached to one end of the line. The diamond is attached to the class that is the aggregate.



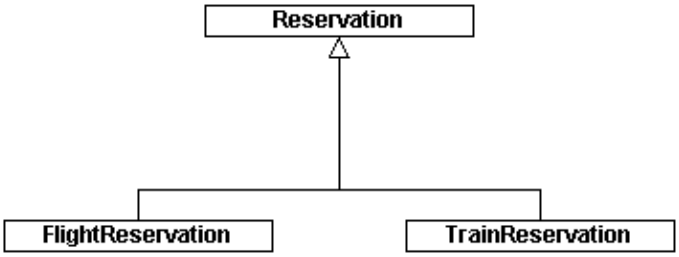
Multiplicities are shown at each end of the aggregation. Multiplicities may either by an integer value or an integer interval on the form lower−bound...upper−bound. In addition, the star character (*) may be used for upper−bound, denoting an unlimited upper bound. In the above example the multiplicities mean that an office may contain zero or more letters.

The diamond attached to the aggregate may either be hollow or filled. If it is hollow the implementation of the aggregation is said to be by−reference and if it is filled it is said to be by−value. The first case corresponds to reference composition in BETA, the second to whole−part composition.



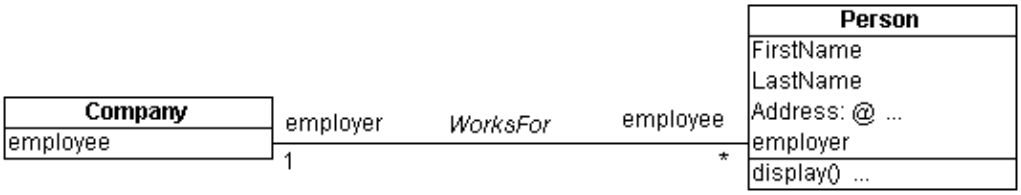
5.6.1.3 Specialization

Specialization is illustrated as a tree of class symbols. A line is drawn between the subclass and the superclass with an arrow at the end connected to the superclass.



5.6.1.4 Association

An association is shown as a line connecting two class symbols.



Multiplicities are shown on each end of the association. The syntax of the multiplicities on associations is exactly the same as the syntax of the multiplicities on aggregations. In the above example the multiplicities mean that a company may be associated to zero or more persons through the relation works−for and a person is related to exactly one company (i.e. in this example a person may not have more than one employer).

Apart from multiplicities role names may be attached to each end of the association. A role name indicates the role played by the class attached to the end of the line near the role. For instance in the above example the role of a company seen from the view of a person is an employer (and vice versa).

Finally the association may be given a name. In the above example the name of the association is WorksFor.

6 Interface Builder

6.1 An Example Application

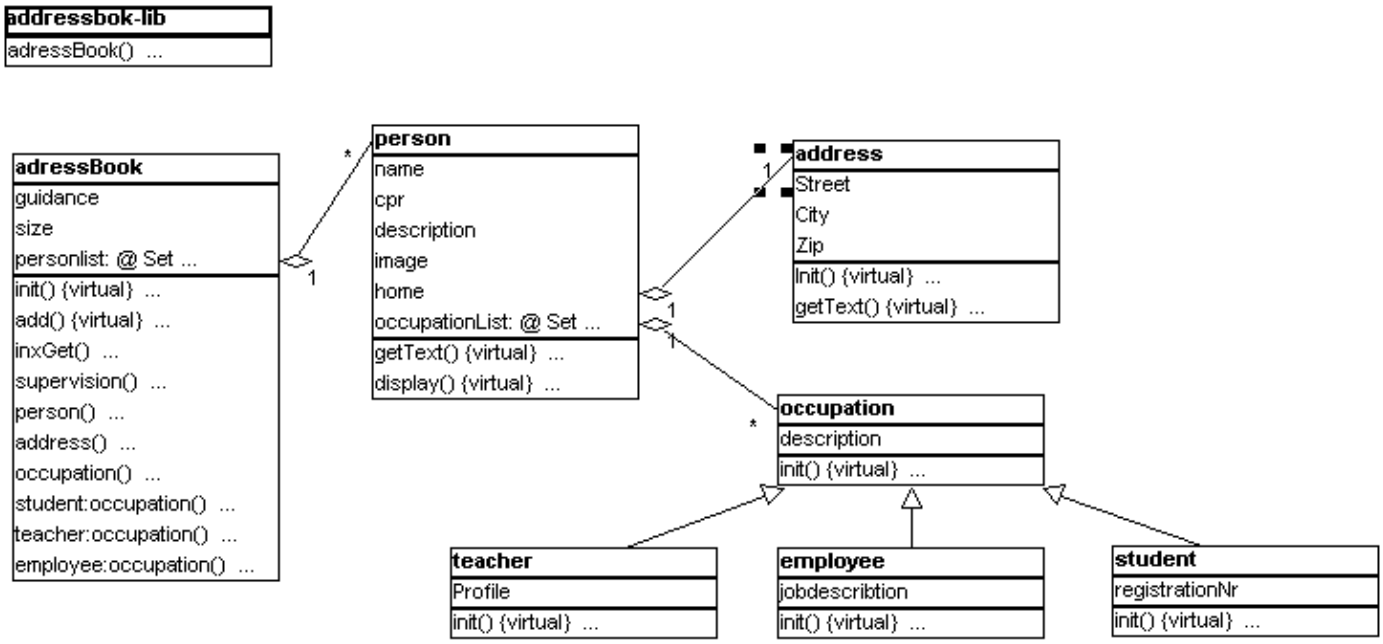
This tutorial will demonstrate how to use Frigg to make a small address book application. The application is structured into three parts:

addressbook The classes that defines the data objects in the application.

addressbookgui The classes that defines the graphical user interface.

addressbookappl The controlling application that ties the data model to the graphical user interface.

The data model is created in Freja (Mjølner BETA CASE tool). The following diagram describes the data model for the addressbook application:



The Freja manual can be consulted to get an explanation of the graphical notation used in the diagram.

Here is a short description of the classes in the diagram:

addressbook Has a list of persons.

person Has a number of simple attributes: name, CPR etc., a reference to the current address, and a list of occupations.

address Simple attributes: Street, city etc.

occupation Can be teacher, student and employee. This part of the model is left out of the

application for simplicity.

The addressbook model does not have visibility to the addressbookgui (i.e. it does not INCLUDE the addressbookgui fragmentgroup). This ensures that the addressbook data objects can be made persistent objects.

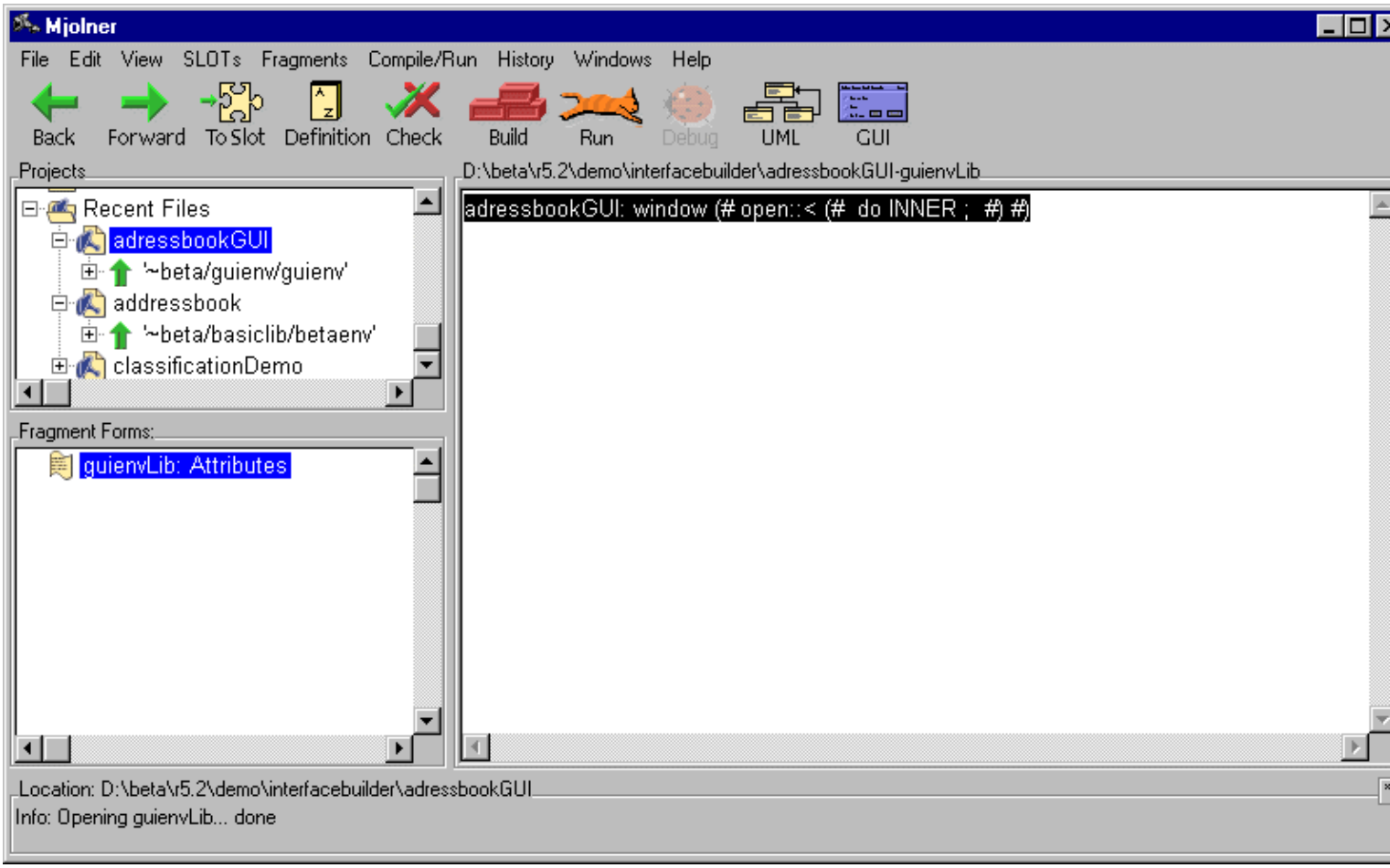
The addressbookgui does not know about the addressbook data model. This allows the addressbook user interface to be reused in other applications.

The addressbookappl includes both the data model and the user interface, and ties the two together.

The following sections will explain how to create the user interface and the application.

6.2 Creating the User interface

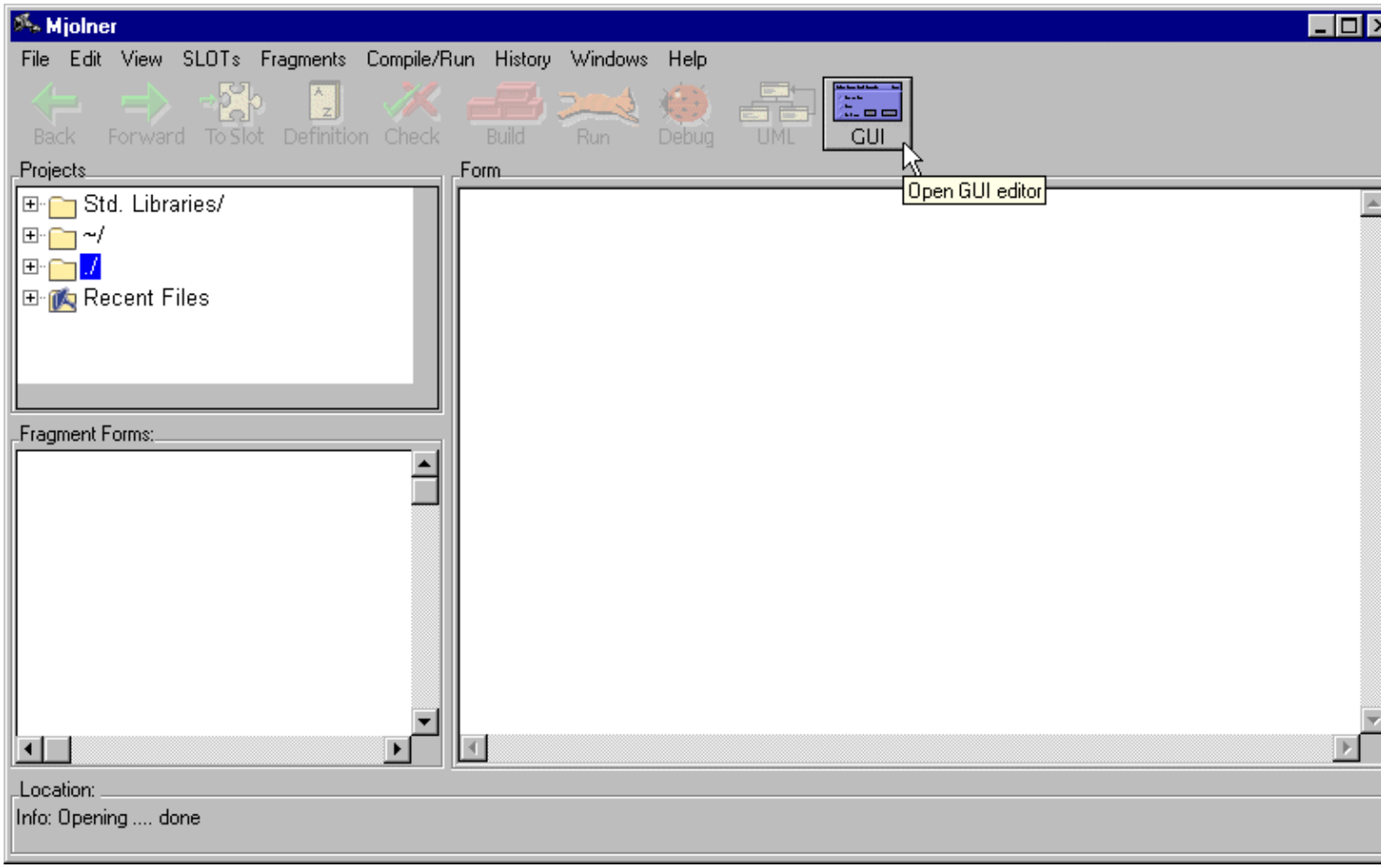
6.2.1 Creating the adressbookgui fragmentgroup



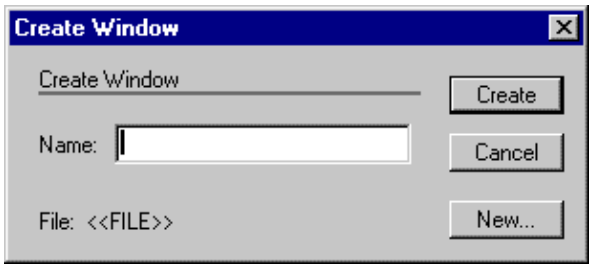
Here the addressbookGUI file is already created. This was done automatically by the interface builder. The file has ORIGIN in guienv and contains a GUIenvLib fragmentform. The GUIenvLib fragmentform can contain specializations of the window class in Lidskjalv (The Mjølner BETA user interface framework).

6.2.2 Creating the addressbook window

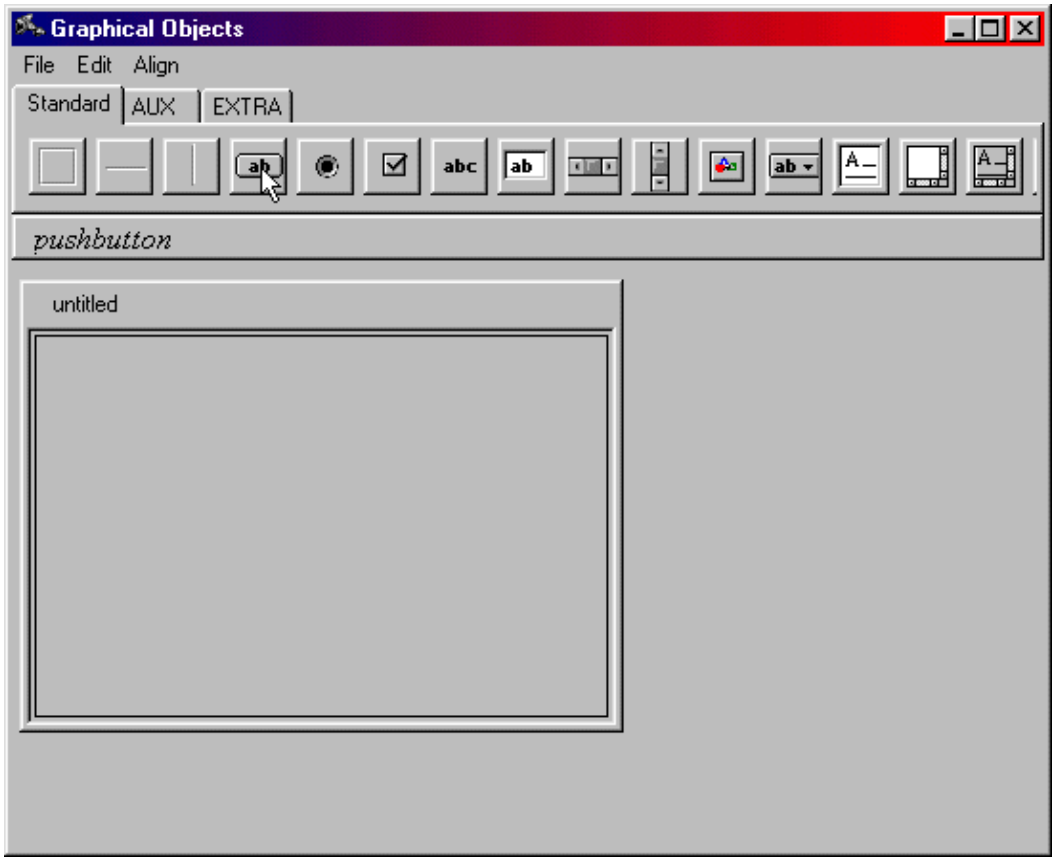
The addressbook window is created as a specialization of window via the UML command in the toolbar:



In the create dialog the name of the window are entered.



Pressing Create in the dialog the graphical editor window appears:

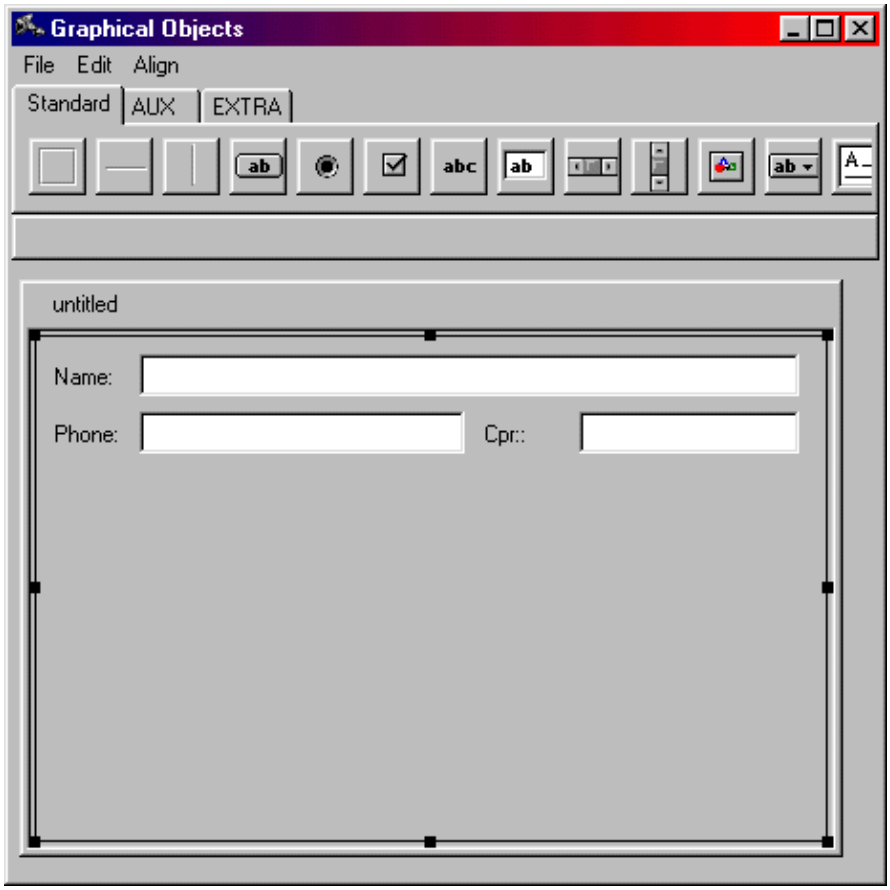


The empty area with a border is the content area of the window. This area has been resized to the desired size by dragging the border.

The palette at the top of the window contains the standard Lidskjalv window items, such as push−button and text field.

6.2.3 Adding items

The first items that will be added to the window, is simple text label and text fields. This is the fourth and the fifth item on the simple item palette. The items are added by dragging the items from the palette to the content area of the window.

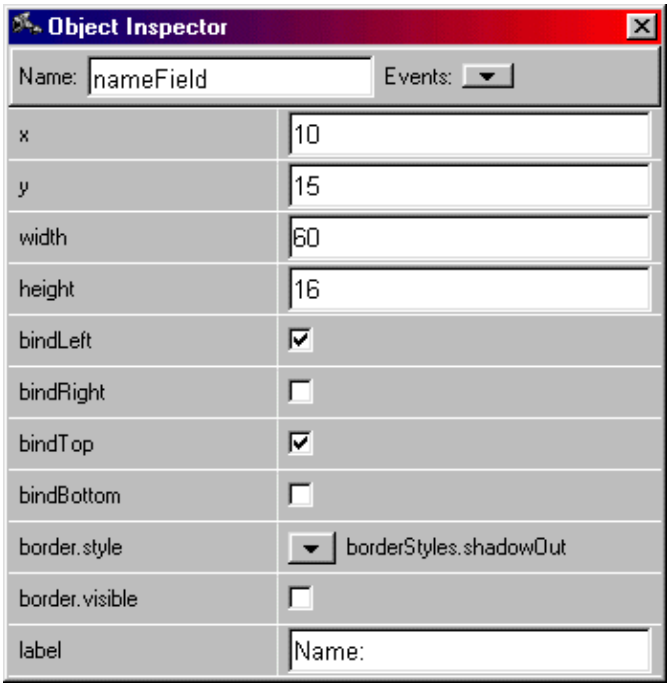


Here the items are added, and the text in the labels are changed to 'Name:', 'Phone:' and 'CPR:'. These changes are made via the 'Object Inspector' that are invoked by selecting an item and then choosing the **Object Inspector** in the **Edit** menu:

The items are arranged in the window by using the alignment commands in the 'Align' menu.

6.2.4 Changing the Name

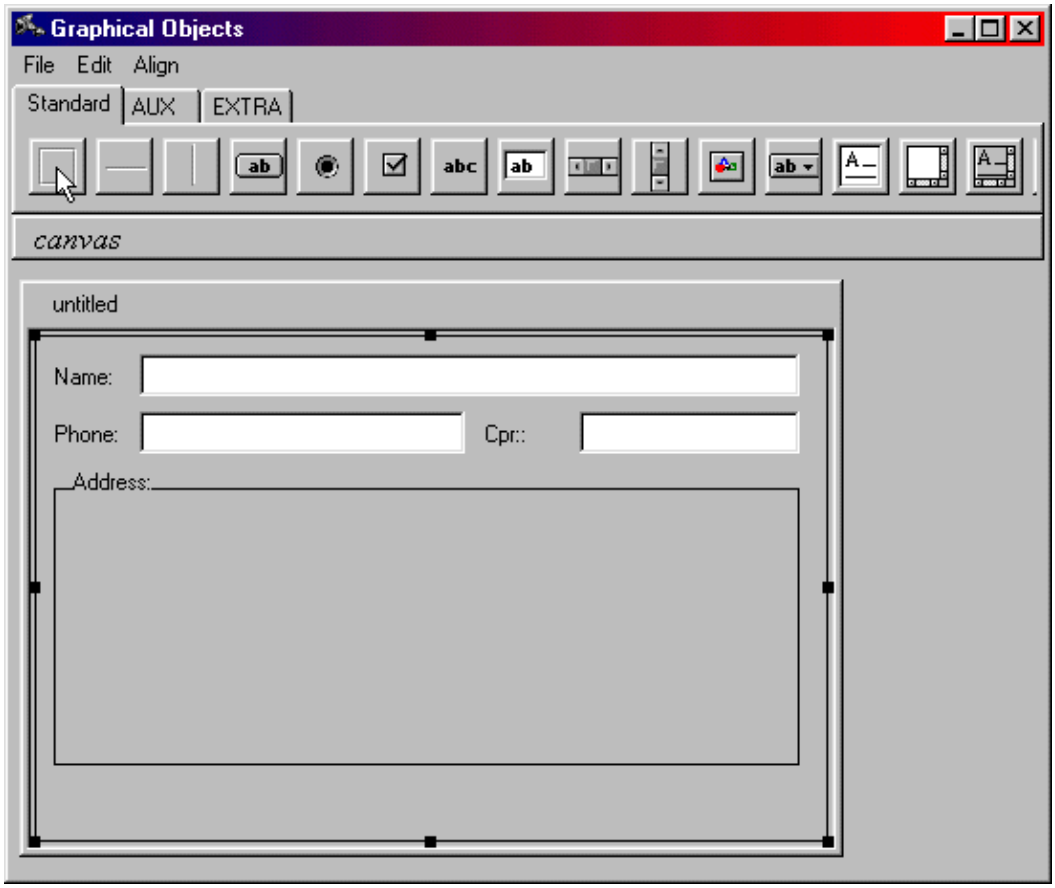
The items are given default names in the source code that looks like 'editText1' and 'staticText2', but since these items needs to be referred to in the application (in order to tie the user interface to the data model) it is a good idea to change the names. This is done by choosing the 'Edit Name' command in the 'Edit' menu:



Here the name of one of the text fields are changed to 'nameField', which will be easier to remember later.

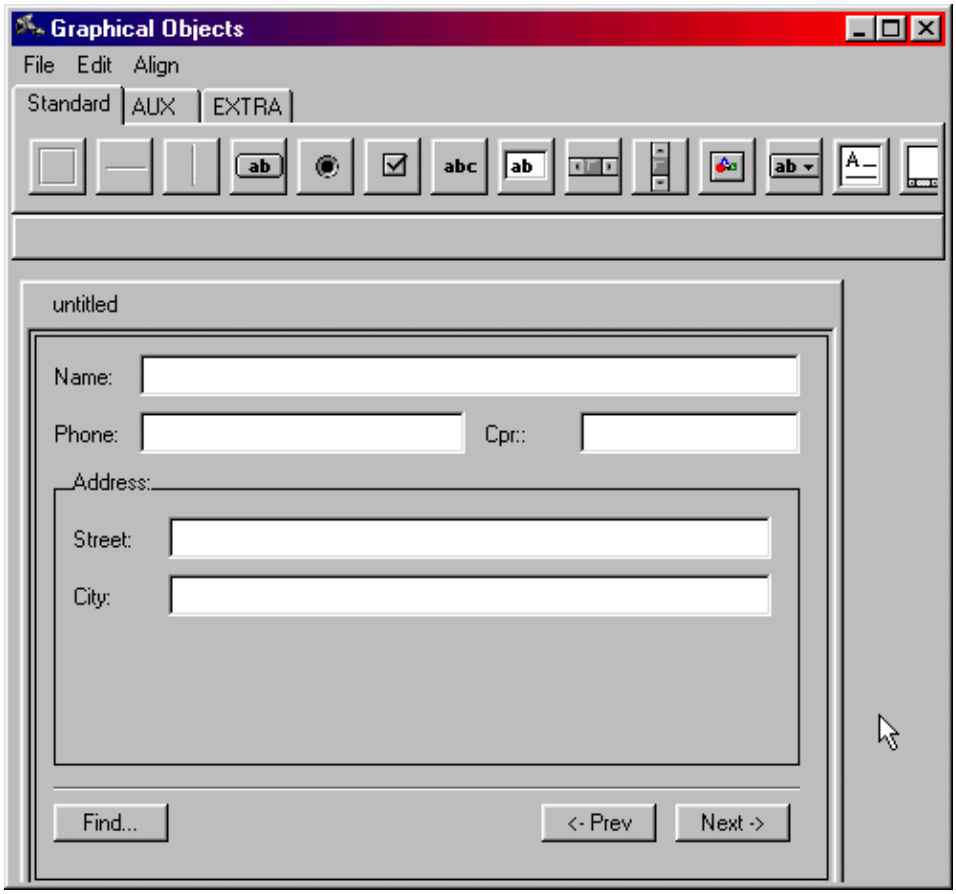
6.2.5 Compound items

The address part of the addressbook window are going to be a canvas, which is a compound item. This is the first item from the right on the palette. A canvas contains other items in its local coordinate system. Initially the canvas is empty, and then items can be added to the canvas. These objects move when the canvas are moved.



Here the address canvas has been added. The object info dialog has been used to give the canvas an 'etched in' border. The 'Address:' label are not part of the canvas, but are added to the window.

The address fields 'street' and 'city' can now be added to the canvas by dragging from the palette to the canvas.



Now the address fields are added. At the bottom of the window three push−buttons are added. They perform the main functions in the addressbook window. The other functions can be put in the menubar.

NOTE: The menubar can not be specified in the graphical editor. It will have to be coded in Sif, see the Lidskjalv manual for information about doing this.

Furthermore a separator is placed between the buttons and the 'address' canvas.

Now the user interface is complete. In the next section one way to tie the user interface and the data model together in an application, will be explained.

7 Debugger

The Debugger in the Mjolner System is called Valhalla.
The user interface of Valhalla consists of a main window containing a menu and a number of different windows (views) displaying different aspects of the debugged program (called the Valhalla Universe). The Valhalla Universe is a top−level window, containing different internal windows for displaying different aspects of the execution state of the debugged process. Below a very short description of the different window types in the Valhalla Universe is given before we go on to describe how to get started using Valhalla.

- The object views display the state of BETA objects and components [3].
- The stack views display the runtime stack of the debugged process.
- The stackbrowser view, which is a combined code, stack and object view.

Details on the functionality of the window types follows in the tutorial and in the reference manual following the tutorial.

---

[3] Components are BETA objects that have their own thread of control

7.1 Getting Started

In this tutorial, the program record will be used as an example. This program consists of the files

```
~beta/debugger/demo/record
~beta/debugger/demo/recordlib
```

and uses

```
~beta/containers/hashTable
```

The files record and recordlib are listed in appendix A. To get hands−on experience using Valhalla, you should copy record and recordlib to a directory of your own, compile them, and then read the tutorial while strolling along on your own workstation:

```
> cd ~beta/debugger/demo
> cp record* myDir
> cd myDir
> beta record
```

Or on windows:

```
> copy %BETALIB\demo\beta\record* myDir
> cd myDir
> beta record
```

It is not neccessary to compile the program, you can also do that from within the Mjolner tool.

Start the Mjolner tool by typing:

```
mjolner
```

at the UNIX prompt or selecting the icon if you are using a Window system. Open the record example (could be specified at commandline). If you have not compiled the record.bet program allready select "Compile record" in the "Compile/Run" menu. When you have a compiled program, select "Debug recode" in the "Compile/Run" menu. Valhalla is not started. It will initialy perform a check of all involved files, this can take from a few seconds to a minut depending on program size and your computer. Valhalla will then open the Valhalla Universe.

7.2 An Example Usage

This section assumes the record program has been compiled as described in the previous section. Running record results in a runtime error:

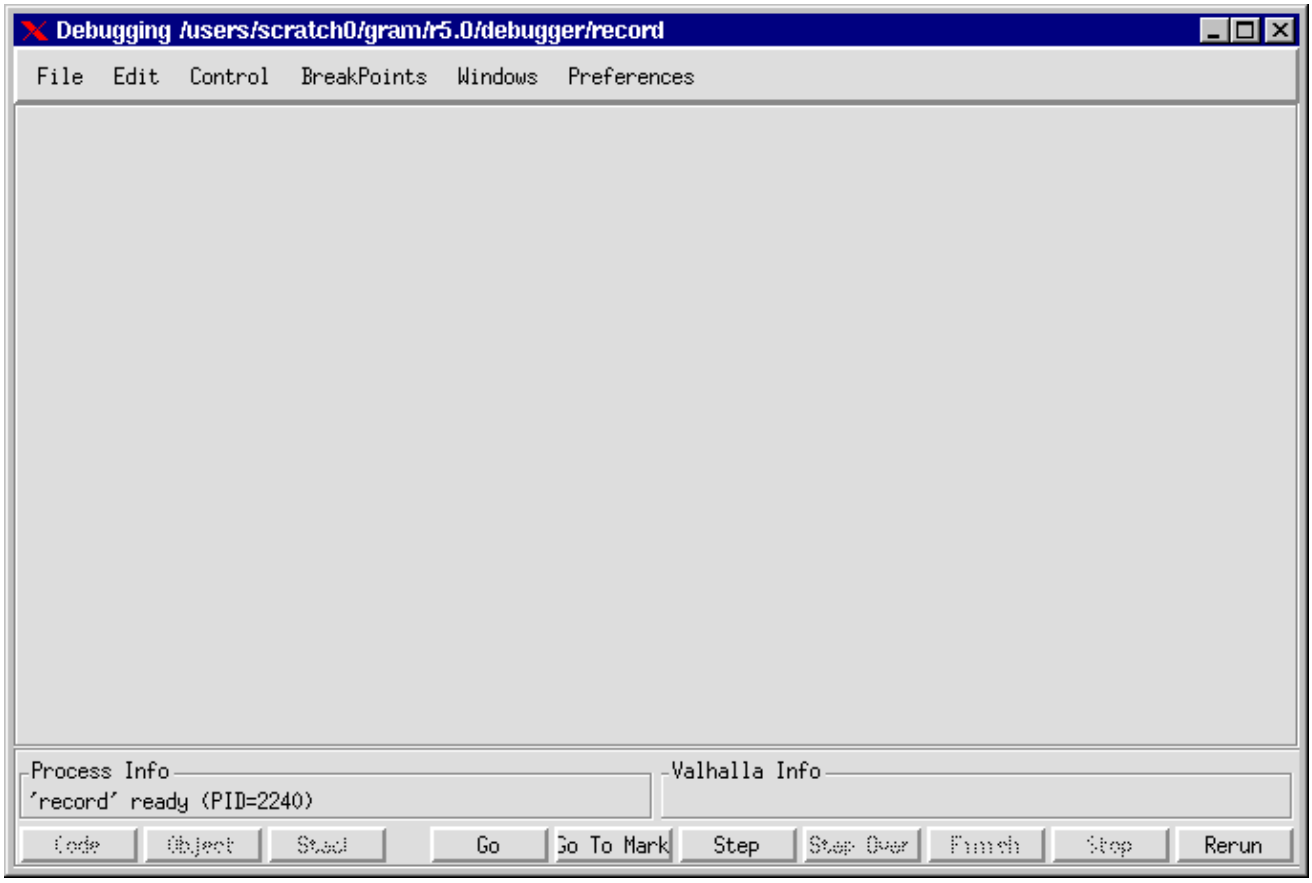
```
> cd myDir
> record
# BETA execution aborted: Reference is none
# Look at 'record.dump'
```

If you have the Mjolner tool running, you can run record by choosing "Run record" from "Compile/Run" menu.

Now let's use Valhalla to locate the cause of the error. Start Valhalla by typing choosing "Debug Record"

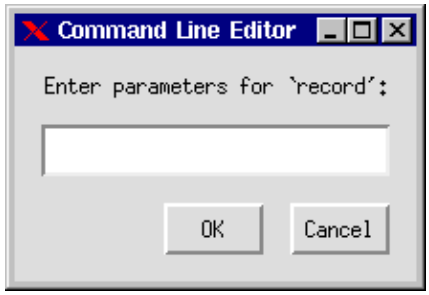
Valhalla will initialize and open the Valhalla Universe as shown [4] in .

Figure 41



If the record program took any commandline arguments, we could specify these using the command−line editor. Choose "Command−line" from the "Edit" menu in the Valhalla Universe.

Figure 42



The Valhalla Universe defines a number of menus. Most commands in these menus operate on the state of the debugged process, or enables control over the debugger process.

The middle pane of the Valhalla Universe contains a view area in which the different local views will be displayed.

Finally the bottom pane contains three areas: the Process Info Area, the Valhalla Info Area, and the Buttons Area.

In the Process Info area, you will see different messages related to the debugged process. In , you see the message: No Process, indication that no process is being debugged at this point.

In the Valhalla Info Area, you will find different messages related to the operation of Valhalla (status messages, error messages, etc.). In , no Valhalla messages are displayed.

In the Buttons areas, you find a number of buttons, which are short−cuts to often used commands, also found in the menus of the Valhalla Universe.

After we now have presented the Valhalla Universe, we continue the record example. When Valhalla has finished initializing, the program code for the debugged program (debuggee) is displayed in the sourcebrowser. Note that the program−descriptor is shown. If you had any other code opened you can find back to it using the "History"−menu.

We are now ready to start execution of the record application. We do this by pressing the *Go* button in the Buttons Area. The application will now begin execution, and since there in this case is a runtime error, the debugged process will not complete. Since the application is being debugged, the application does not terminate as usual, but will signal the error to Valhalla.

Note that a new code view is now visible in the sourcebrowser. This new code view displays the code being executed at the time of the runtime error, and highlights the exact source code that gave rise to the runtime error.

The name of the code view opened is Fragment: lib and the pattern in the source code is newBook, implying that the newBook pattern is defined in a fragment, called lib. From here, you can now use the semantic browsing facilities described in the Mjølner Tool manuals ([MIA 99−39], [MIA 99−40], [MIA 99−34]) to find the definitions of the different names in the displayed source code. If the semantic links refer to source code, not in the current code view, new code views are created, displaying the proper source code (similar to Open Separate in Sif). If, during browsing, you forget what imperative caused the error, or cannot find the window containing that imperative, just press the *Code* button, and Valhalla will raise (and wriggle) the window with the offending imperative selected.

A number of other informations about program state at time of error would be useful in order to decide what caused the error:

- What is the state of these objects.
- What was the call chain at time of error.

In the following sections we will consider how to obtain these informations.

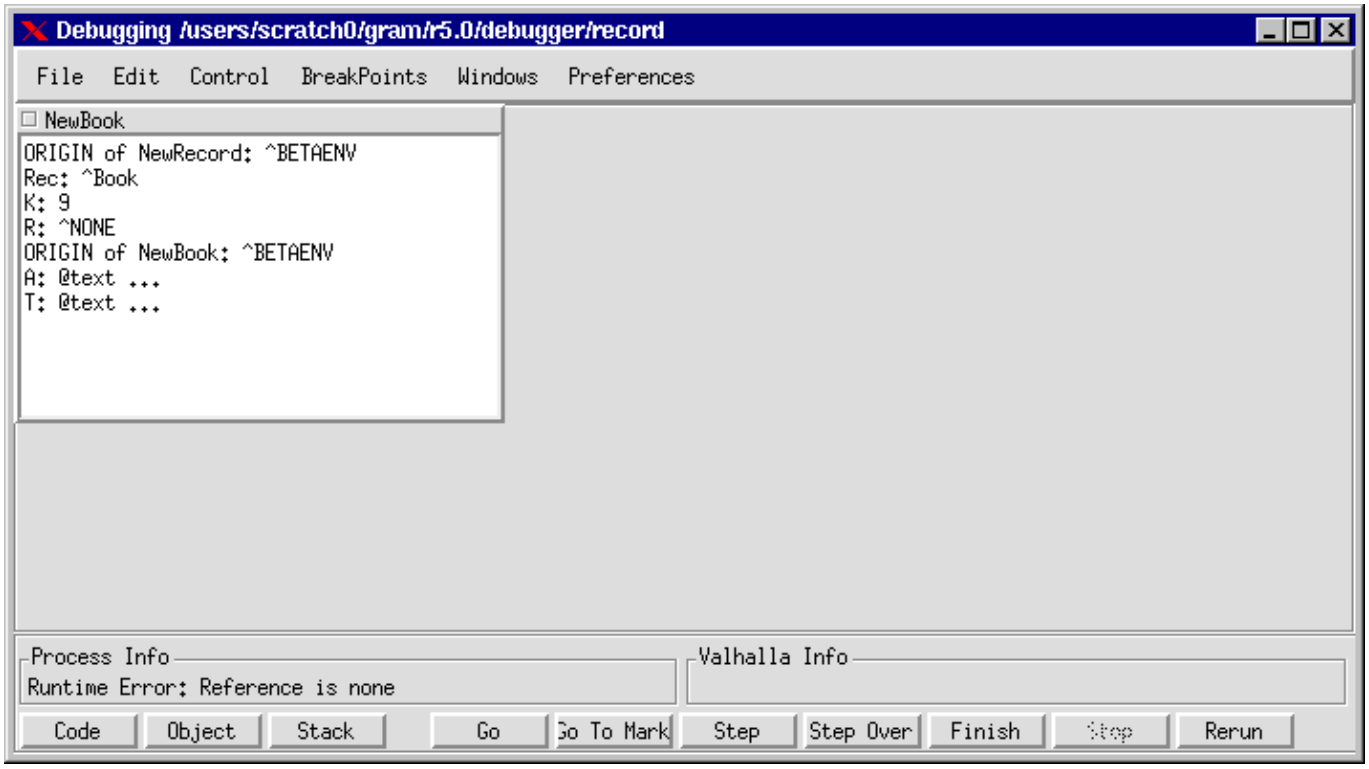
7.2.1 Inspecting object state

Above we found that Valhalla automatically displayed the source code containing the offending imperative. Usually, in order to determine the cause of the error, one has to consider the state of the objects in the executable to understand what caused the error. To browse the state of the objects, Valhalla implements so−called object views.

The immediately most interesting object, related to the error is the so−called Current Object, which is the object that executed the offending imperative. We can gain access to the current object by pressing the *Object* button in the Buttons Area. This will result in an object view being displayed in the Universe. This object view will display the state of the Current Object at the time of the error.

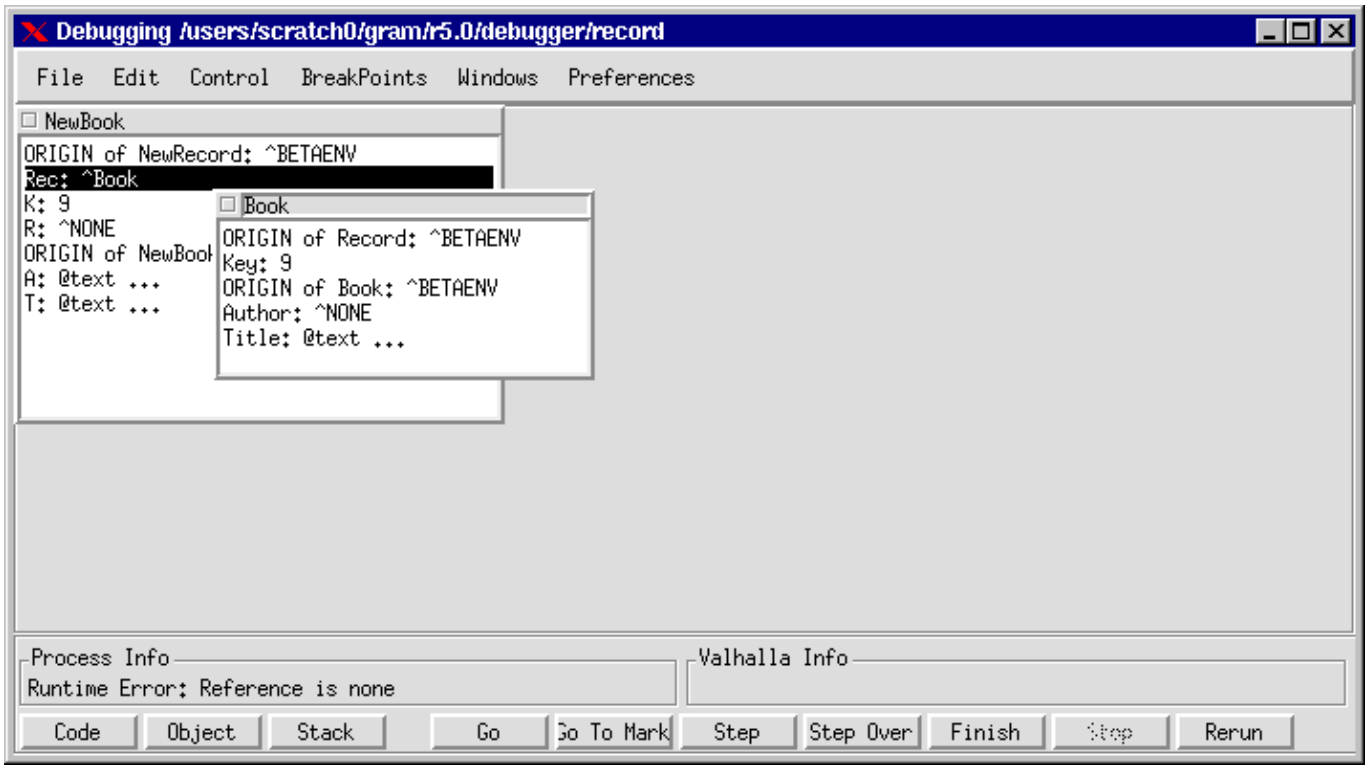
More generally, whenever the debugged process is stopped, pressing the *Object* button in the Buttons Area of the Valhalla Universe opens an object view displaying the current object if already open in the Universe, the object view will be raised (and wriggled). If we press the *Object* button, we get the following contents of the Universe:

Figure 43



From the object view in figure 3, we see that the current object at time of error was an instance of the pattern NewBook. The state of the object is displayed in the window. The Rec attribute is a reference to an object and is therefore described only by the name of its pattern, followed by ~ and a number. If we double−click the line Rec: ^Book~3, a new object view is opened, displaying the state of the object referred to by Rec (as shown in figure 4):

Figure 44



As it can be seen, the author attribute of Rec is NONE. This is actually the reason for the runtime error in record. As you probably remember, the failing imperative was A−>Rec.Author and because

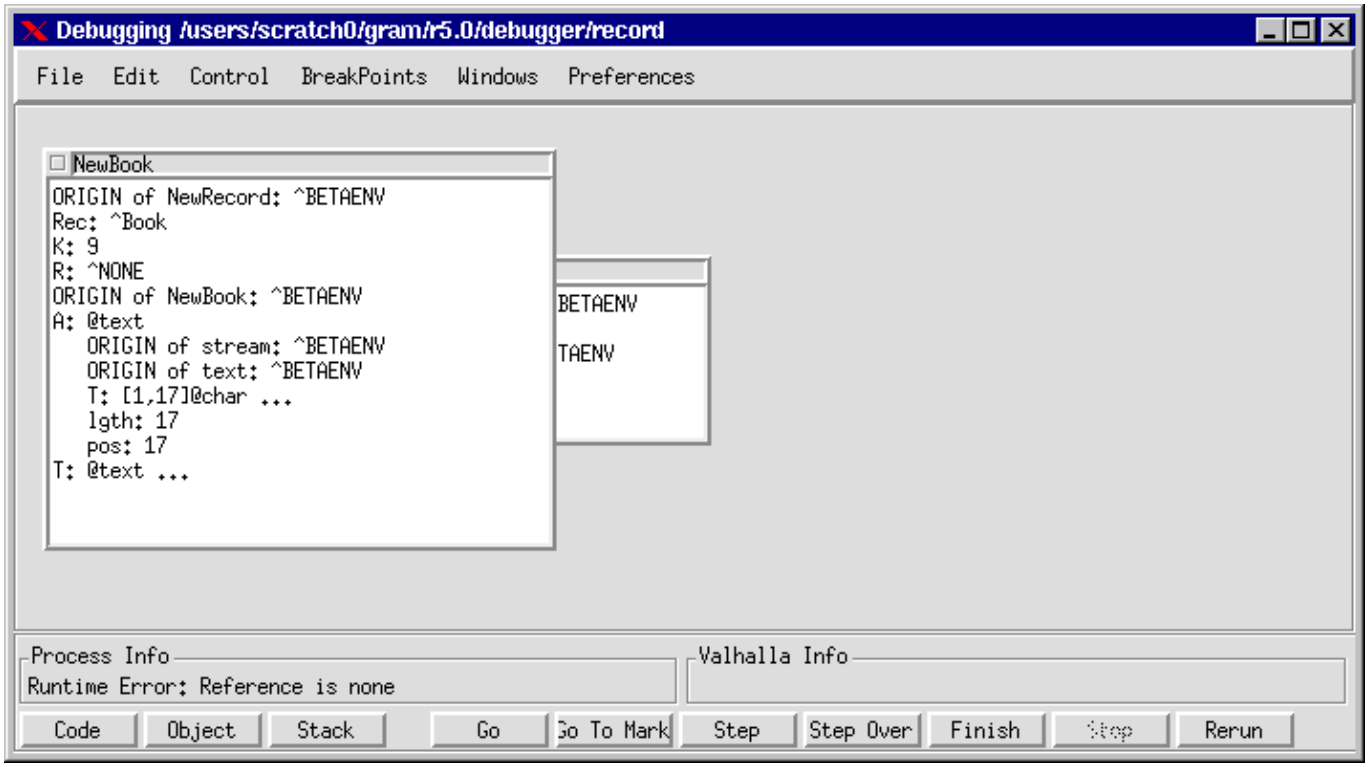
Rec.author is NONE, this results in a runtime error. If you do not remember, simply press the *Code* button in the Buttons Area of the Universe.

Since author is a dynamic reference to a Text object, we could correct the error by changing author to become a static instance of Text (exchanging ^ with @).

Eventhough we have now found the source of error, there is still a number of Valhalla features that have not yet been demonstrated. As a consequence we just continue this tutorial to introduce you to more of the Valhalla functionality.

Some attributes are shown contracted (shown by the ...). These ... indicate that these objects are complex objects, with inner structure. You can see the inner structure (and state) by double clicking on the attribute. If we do this on e.g. the A attribute, we get the following screen:

Figure 45



Note, that in contrast to when we followed the Rec attribute in , no new object view is opened, but the state of the A attribute is shown inline. This is due to the fact, that A is a static object, whereas Rec is a dynamic object reference.

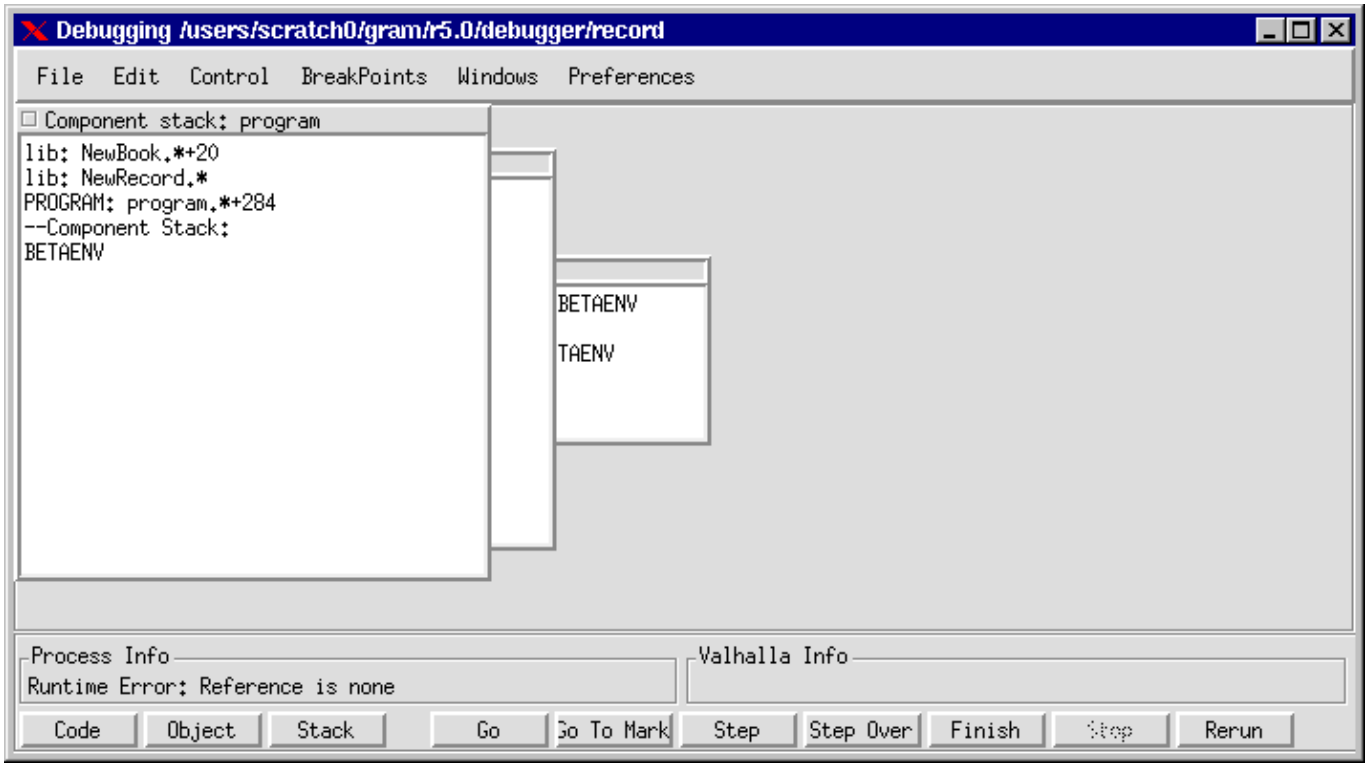
Object views are updated every time the program stops by hitting a breakpoint, receiving a signal or in case of a runtime error.

7.2.2 Inspecting the call chain

Now we know where and why the error happened. But how did we get there? To answer this question we use the stack view: We can get a stack view by pressing the *Stack* button in the Button Area.

The stack view shows the process stack at time of error. Each line in the stack view refers to a stack frame, with the most recent as the top−most line:

Figure 46



By double−clicking on the lines in the stack view, code views are opened, displaying the code related with this stack frame (with the active imperative selected). By holding the right mouse button down on a line, you get a small menu from where you can create object views, displaying the state of the corresponding stack frame.

In the bottom of the stack we can see that there is only one component present. This is the main component corresponding to the main program pattern, which is allways present.

7.2.3 Rerunning the program

If it during a debugging session becomes necessary to restart the debugged process (e.g. to try to trace the location of the error after having inserted some breakpoints), we can rerun the debugged process by pressing the *Rerun* button in the Buttons Area. The debugged process will then be reinitialized and prepared to start from the beginning again. All existing program state is cleared by the rerun, but existing breakpoints are maintained.

7.2.4 Setting a Breakpoint

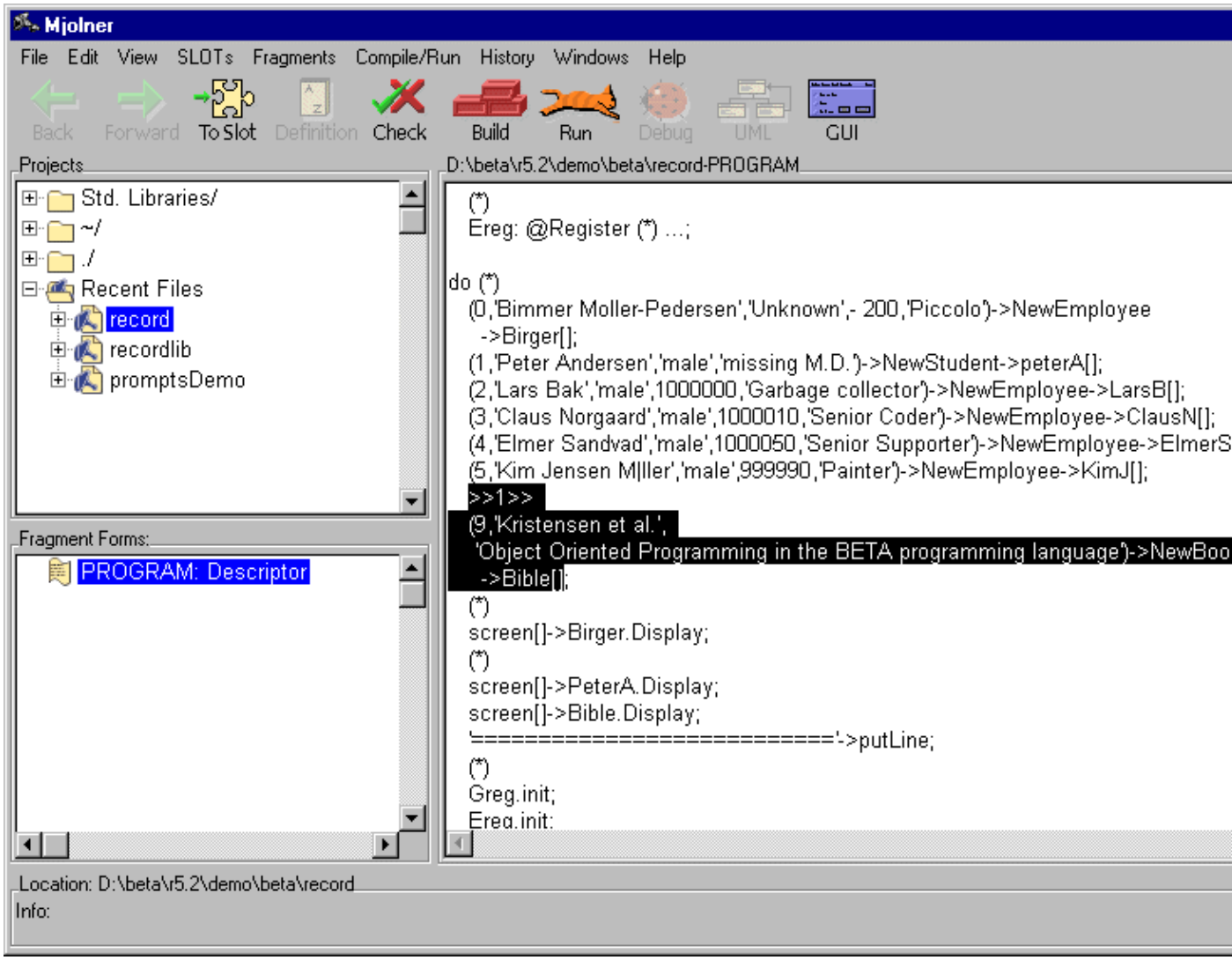
Until now, Valhalla has decided when to interrupt the debugged process, doing so because of a 'Reference is NONE' error. We have not yet seen an example of controlling the program execution in greater detail. To do this we are now going to restart the debugged process and trace the program flow until time of error

We press the *Rerun* button, and Valhalla now restarts the process and makes it ready to be restarted [5].

We now examine the code view, locating the invocation of the erroneous NewBook pattern. We now want to make the process continue execution until it is about to execute NewBook. We can do this by setting a breakpoint immediately before the invocation.

We click at the BETA imperative containing the generation and execution of a NewBook object. Then select *Set Breakpoint* from the *Breakpoints* menu in the Universe. An breakpoint marker (>>1>>) appears in front of the imperative to mark the presence of a breakpoint (the 1 indicates that this is the first breakpoint). You can also set the breakpoint by using the popup−menu associated with the right mouse button. The process will thus be interrupted just before this imperative is about to be executed. Figure 7 displays the look of the code view at this point.

Figure 47



Having set the breakpoint we make the debugged process begin execution by pressing the *Go* button. The process now runs until the breakpoint is hit. Then Valhalla updates all open code, stack and object views to display the current state of the debugged process.

Alternatively, we could have selected the *Step Over* button a number of times. *Step Over* executes the imperatives one by one, returning control to Valhalla after each imperative. This would bring the debugged process to exactly the same imperative, but by executing an imperative at the time in the PROGRAM fragment [6].

Now we would like to continue until the first imperative executed by NewBook.

From here we might want to trace the execution more closely. We can do this by using the *Step* button. *Step* is a single step facility, which executes one single BETA imperative at a time. This implies, that *Step Over* executed entire patterns in a single step, since *Step Over* is intuitively

single stepping at the imperative level of the visible code in the code view, whereas *Step* will stop execution e.g. immediately after a pattern invocation have been initiated, setting a breakpoint before the first imperative in the invoked pattern.

If we now press the *Step* button repeatedly, we can now follow the execution closely, until we reach the point immediately before the offending imperative.

7.2.5 The End

You have now concluded a tour of the most important Valhalla facilities. To get a more detailed description of these facilities as well as others not covered in this tutorial, please consult the reference manual.

---

[4] Screen dumps shown in the tutorial shows the views as they would have been if you had not copied the example program to a directory of your own, but simply compiled them in the ~beta/debugger/demo directory
[5] All breakpoints set before rerunning the program will continue being set
[6] *Step Over* sets a breakpoint at the imperative following the current imperative in the current pattern and thus skips procedure calls that might be embedded in the current imperative

7.3 Appendix A

This appendix contains the source code for the BETA program used as example in the tutorial. The source consists of the files record.bet and recordlib.bet.

Program 1: record.bet

```
ORIGIN '~beta/basiclib/betaenv';
--- INCLUDE './recordlib'
--- PROGRAM: descriptor ---
(# (* This fragment is an example of using virtual patterns in BETA.
    * The following classification hierarchy is defined in the
    * library fragment 'recordlib'
    * 1.  Record
    * 2.  Person
    * 3. Employee
    * 3. Student
    * 2. Book
    * Record has a virtual (procedure) pattern:
    * Display, that displays the attributes of a record.
    * Display is further bound in the sub-patterns.
    * The patterns NewRecord, NewPerson, etc may be used for
    * generating new instances of the record-patterns.
    * The pattern Register has two virtual attributes:
    * regCat:< record, the elements in the register.
    * Display, that displays the elements of the register.
    *
    * In this application two instances of Register are generated:
    * Greg is a general register where regCat is not further bound
    * Ereg is an employee register where only Emplyoee-records may
    * be inserted
    *)

   (**** The following are declaring a number of reference variables ****)

   Birger, PeterA, Bible, LarsB, ClausN, ElmerS, KimJ: ^Record;
```

```
   Greg: @Register; (* Greg is a general register for all types of Records *)
   Ereg: @Register (* Ereg is a specific register only for Employee records
                     (and subpatterns of Employee) *)
     (# RegCat::< Employee; (* a further binding of RegCat to restrict to
                               Employee *)
        Display::< (# (* Displays the restriction on this particular register *)
                    do '/Employee ' -> s.putText; INNER #);
     #);
do (* the following creates and instantiates a number of Record objects *)
   (0 ,'Bimmer Moller-Pedersen','Unknown',- 200,'Piccolo')
     -> NewEmployee -> Birger[];
   (1,'Peter Andersen','male','missing M.D.')
     -> NewStudent -> peterA[];
   (2,'Lars Bak','male',1000000,'Garbage collector')
     -> NewEmployee -> LarsB[];
   (3,'Claus Norgaard','male',1000010,'Senior Coder')
     -> NewEmployee -> ClausN[];
   (4,'Elmer Sandvad','male',1000050,'Senior Supporter')
     -> NewEmployee -> ElmerS[];
   (5,'Kim Jensen M|ller','male',999990,'Painter')
     -> NewEmployee -> KimJ[];
   (9,'Kristensen et al.','Object Oriented Programming in the BETA programming language'
     -> NewBook -> Bible[];
   (* the following displayes the Birger, PeterA and Bible objects on the
    * screen *)
   screen[] -> Birger.Display; (* not a nice view (: -) *)
   screen[] -> PeterA.Display;
   screen[] -> Bible.Display;
   '==========================' -> putLine;
   (* initialization of the Greg and Ereg registers *)
   Greg.init;
   Ereg.init;
   (*inserts all Record objects in the Greg and/or Ereg registers *)
   Birger[] -> Greg.insert;
   Bible[] -> Greg.insert;
   PeterA[] -> Greg.insert;
   ClausN[] -> Ereg.insert;
   LarsB[] -> Ereg.insert;
   ElmerS[] -> Ereg.insert;
   KimJ[] -> Ereg.insert;
   (* displays the Greg and Ereg registers on the screen *)
   screen[] -> Greg.display;
   screen[] -> Ereg.display;
   (if (LarsB[] -> Ereg.has) (* test if LarsB is in the Ereg register *)
    // true then 'LarsB in employee register' -> putLine
    // false then 'LarsB not in employee register' -> putLine
   if);
   (if (LarsB[] -> Greg.has) (* test if LarsB is in the Greg register *)
    // true then 'LarsB in general register' -> putLine
    // false then 'LarsB not in general register' -> putLine
   if);
   (* end of program *)
#)
```

Program 2: recordlib.bet

```
ORIGIN '~beta/basiclib/betaenv';
-- lib: Attributes --
(* This fragment is an example of using virtual patterns in BETA.
 * The following classification hierarchy is defined
 * 1.  Record
 * 2.  Person
```

```
 * 3. Employee
 * 3. Student
 * 2. Book
 * Record has a virtual (procedure) pattern:
 * Display, that displays the attributes of a record.
 * Display is further bound in the sub−patterns.
 * The patterns NewRecord, NewPerson, etc may be used for
 * generating new instances of the record−patterns.
 * The pattern Register has two virtual attributes:
 * regCat:< record, the elements in the register.
 * Display, that displays the elements of the register.
 *
 * This fragment is a library containing the declarations
 * It is used in the program "record"
 *)
Record:
(* Record objects contain two attributes: key and Display.  Key contains
 * the ID of this record (supplied by the programmar).  Display is a
 * virtual, enabling printing Records on the screen
 *)
  (#
     Key: @integer;
     Display:< (* declaration of a virtual (procedure) pattern  *)
       (#
          s: ^stream (* the input parameter is where to display this record *)
       enter s[]
       do
          s.newline;
          '−−−−−−−−−−−−−−−−−−'−>s.putLine;
          'Record: Key      = '−>s.putText;
          Key−>s.putInt;
          s.newline;
          INNER
       #);
  #);
Person: Record
  (#
  (* Person is a suppattern of Record, declaring two additional attributes:
   * Name and Sex.  Furthermore Display (inherited form Record) is extended
   * to print the Name and Sex attributes as well as the Key attribute.
   *)
     Name,Sex: @text;
     Display::< (* a further binding of Display from Record *)
       (#
       do
          'Person: Name     = '−>s.putText;
          Name[]−>s.putLine;
          '             Sex = '−>s.putText;
          Sex[]−>s.putLine;
          INNER
       #);
  #);
Employee: Person
  (# (* analog til Person *)
     Salary: @integer;
     Position: @text;
     Display::<
       (#
       do
          'Employee: Salary   = '−>s.putText;
          salary−>s.putInt;
          s.newline;
          '          Position = '−>s.putText;
          Position[]−>s.putLine;
          INNER
```

```
      #);
  #);
Student: Person
  (# (* analog til Person *)
     Status: @text;
     Display::<
       (#
       do 'Student: Status   = '−>s.putText; Status[]−>s.putLine; INNER
       #)
  #);
Book: Record
  (# (* analog til Person *)
     Author,Title: @text;
     Display::<
       (#
       do
          'Book: Author   = '−>s.putText;
          Author[]−>s.putLine;
          '         Title = '−>s.putText;
          Title[]−>s.putLine;
          INNER
       #)
  #);
doc0: (*** Temporary initialization ***) (# #);
NewRecord: (* creation and initialization procedure for Record objects *)
  (# RegCat:< Record; Rec: ^RegCat; K: @integer; R: ^Record
  enter K
  do &RegCat[]−>Rec[]; K−>Rec.Key; INNER ; Rec[]−>R[]
  exit R[]
  #);
NewPerson: NewRecord
(* creation and initialization procedure for Person objects *)
  (# RegCat::< Person; N,S: @text
  enter (N,S)
  do N−>Rec.Name; S−>Rec.Sex; INNER ;
  #);
NewEmployee: NewPerson
(* creation and initialization procedure for Employee objects *)
  (# RegCat::< Employee; S: @integer; P: @text
  enter (S,P)
  do S−>Rec.Salary; P−>Rec.Position; INNER ;
  #);
NewStudent: NewPerson
(* creation and initialization procedure for Student objects *)
  (# RegCat::< Student; S: @text enter S do S−>Rec.Status; INNER ; #);
(* This is a declaration of a register pattern.  Register objects will be able
 * to contain Records (or instances of suntarrerns of Record.
 *)
NewBook: NewRecord (* creation and initialization procedure for Book objects *)
  (# RegCat::< Book; A,T: @text
  enter (A,T)
  do A−>Rec.Author; T−>Rec.Title; INNER ;
  #);
Register:
(* Register is a container pattern with operations insert (insert an object)
 * scan (traverse the register), has (test for presence of an object in the
 * register), display (display the objects in the register).
 *
 * Register may contain objects, that are instances of Record (of subpatterns
 * hereof).  Specializations of Register may restrict the classes of objects
 * allowed in the specialized register pattern by further binding the regCat
 * virtual pattern
 *)
  (#
     regCat:< Record;
```

```
    regLst (* private pattern *) : (# succ: ^regLst; elm: ^regCat #);
    head: ^regLst;
    init:
    (* initialization pattern to be invoked before first usage fo an
     * register object
     *) (# do none −>head[] #);
    scan:
    (* walks through the register, executing INNIR for each element in the
     * register.  P will refer to the current element in the register.
     *)
      (# elm: ^regCat; p: ^regLst
      do
         head[]−>P[];
         search:
         (if (P[] = none )
          // false then
              P.elm[]−>elm[]; INNER ; P.succ[]−>P[]; restart search
         if)
      #);
    Display:<
    (* display the entire register by printing header and trailer text,
     * and scanning the entire register in between, invoking display on
     * each element in the register.
     *)
      (# s: ^stream
      enter s[]
      do
         s.newline;
         '############ Register Display '−>s.putText;
         INNER ;
         s.newline;
         scan
           (# do s[]−>elm.display #);
         '############ End Register Display #######'−>s.putLine
      #);
    Has:
    (* takes an object reference, and checks whether that object is in the
     * register
     *)
      (# E: ^regCat; found: @boolean;
      enter E[]
      do
         false−>found;
         search: scan
           (#
           do (if E.key // elm.key then true−>found; leave search if)
           #)
      exit found
      #);
    Insert:
    (* Takes an object reference and inserts that object in the register
     * (if not already in the register)
     *)
      (# E: ^regCat; P: ^regLst
      enter E[]
      do
         (if (E[]−>Has)
          // false then
              &regLst[]−>P[]; head[]−>P.succ[]; E[]−>P.elm[]; P[]−>head[]
         if);
      #);
  #)
  (* Register *)
```

Index

The entries in the alphabetic index consists of selected words and symbols from the body files of this manual − these are in **bold** font − as well as the identifiers defined in the public interfaces of the libraries − set in regular font.
In the manual, the entries, which can be found in the index are typeset like this. This can help localizing the identifier, when the link from the index if followed − especially in the case where the browser does not scroll the line to the top, e.g. because there is less than a page of text left.
In the small table of letters and symbols below, each entry links directly to the section of the index containing entries starting with the corresponding letter or symbol.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

B

C

D

E

F

G

H

I

N

O

P

R

S

U

V