


























# OBJECT PLANT

## User's Manual

A Macintosh tool for Object Oriented Analysis and Design

by Mikael Arctædius

Introduction to the Object Plant.....	2
Registration.....	2
Reference Manual.....	4
Views.....	4
Cursors.....	6
Document preferences.....	7
General.....	7
Data types.....	7
Exporting diagram pictures.....	8
General Preferences.....	9
Code Generation Preferences.....	10
Copy and Paste.....	11
Undo.....	11
Drag-and-Drop.....	12
Non-obvious keyboard shortcuts.....	13
Joining line segments.....	13
Object Model Diagrams (Class Diagrams).....	14
Pages .....	14
Adding a new toplevel page.....	14
Changing the name of a toplevel page.....	14
Adding a sublevel page.....	15
Changing the name of a subpage.....	15
Elements in an Object Model.....	15
 Classes.....	15
The class dialog window.....	16
Add an attribute.....	17
Add an operation.....	18
Adding a parameter to an operation.....	19
Changing a parameter.....	19
Delete a parameter.....	19
Delete an attribute or operation.....	20
Inherit abstract operations.....	20
Add a new exception.....	20
Moving a class box.....	20
"Nudging".....	20
Resizing a classbox.....	20
Several copies of a single class.....	20
 Interfaces.....	22
  Associations.....	22
The association dialog window.....	23
Qualifiers.....	24
Moving associations.....	24
 Aggregations.....	25
The aggregation dialog window.....	25
Creating multibranch aggregations.....	26
Moving aggregations.....	26
  Generalizations.....	26
The generalization dialog window.....	26
Creating multibranch generalization.....	27
Moving generalizations.....	28
 Suppliers.....	28

	Packages.....	28
	Dependency.....	28
	Cut Tool.....	28
	Frame Tool.....	28
	Text Tool.....	29
	Notes.....	29
	Inspect Tool.....	29
	Event Trace Diagrams (Message Trace Diagrams).....	30
	Pages .....	30
	Elements in an Event Trace diagram.....	30
	Threads.....	30
	Resizing a thread.....	31
	Boxes.....	31
	Moving a box.....	32
	Resizing a box.....	32
	Interactions.....	32
	The event dialog window.....	32
	State Diagrams.....	34
	Pages .....	34
	Changing the name of a State Diagram page.....	34
	Elements in a State Diagram.....	34
	States.....	35
	The state dialog window.....	35
	Add an event.....	36
	Delete an event.....	36
	Moving a state box.....	37
	Resizing a state box.....	37
	Events.....	37
	The event dialog window.....	37
	Initial state.....	39
	Final state.....	39
	Use Case Diagrams.....	40
	Elements in a Use Case Diagram.....	40
	Use Case.....	40
	The use case dialog window.....	40
	Actors.....	41
	Communicate association.....	42
	The communicates dialog window.....	42



Generalization, extends, uses.....	42
Project Window.....	43
Drag-n-drop.....	44
Importing Code (CodeRetriever).....	46
C++ known bugs.....	47
Java known bugs.....	47
CodeRetriever Options.....	47
Code Generation.....	49
The template files.....	51
TAGS.....	53
Top-level tags.....	53
{CLASS}-level tags.....	54
{INTERFACE}-level tags.....	58
{USECASE}-level tags.....	58
{OPERATION}-level tags.....	58
{PARAMETER}, {INPARAMETER}.....	60
{ATTRIBUTE}-level tags.....	60
{INITIAL VALUE}-level tags.....	60
{FILENAME}-level tags.....	61

I apologize for any out of date information and incomplete areas in this documentation. It is a constant battle keeping it in synch with the development of Object Plant itself. I know that the Use Case diagram section is a bit thin. It will be expanded when I extend the tool's Use Case functionality.

Object Plant is a tool for object-oriented analysis and design. Originally it was based on the Object Modeling Technique development methodology by Rumbaugh, Blaha, Premerlani, Eddy and Lorensen but nowadays it also supports a subset of the Unified Modeling Language (UML) developed by Booch, Rumbaugh and Jacobson.

With the Object Plant you can...

- make Object Model diagrams (Class Diagrams in UML notation)
- make Event Trace diagrams (Message Trace diagrams in UML notation)
- make State diagrams
- make Use Case diagrams
- create data dictionaries
- generate C++ code or Java code, HTML documentation
- reverse engineer C++ or Java code (CodeRetriever 10 USD extra)

To use the Object Plant it may help to have at least a passing familiarity with...

- object-oriented analysis/design/programming, especially the OMT method
- or the Unified Modeling Language (by Booch, Rumbaugh and Jacobson)

In this manual I usually stick to the OMT notation. When there are differences in the notation, e.g. different tool icons, I will show them both but not all dialogs are shown in both versions of notations.

## Acknowledgements and References

Thanks to the active beta testers and registered users.

The Object Plant is written in C++ and some plain old C. No class library is used. It was modeled in the oldfashioned style using a pen and paper.

The floating windows are created with the Infinity Windoid 2.6 by Troy Gaul and some modified code from Apple's Develop issue 15.

The Java templates were originally written by Jeff Moore.

The Word 5.1 RTF template is written by Lars Thomasson.

The menu shortcut extensions (e.g. shift-command-A) are created with Mercutio MDEF by Ramon M. Felciano.

(Mercutio MDEF from Digital Alchemy Copyright " Ramon M. Felciano 1992-1997, All Rights Reserved)

Some of the lists are handled with The A List:

The A List © 1997 Kyle Hammond

The splash-screen picture is rendered with Persistence of Vision.

Pictures in this document uses buttons of the "Greg's Buttons" type. Some pictures have been created in System 7.5 and others with MacOS 8 or MacOS 8.1.

Information about Object Plant (known bugs, new releases etc) can be found at:

<http://www.softsys.se/ObjectPlant/>

## Introduction to the Object Plant

*Why ObjectPlant?*

In OOA/OOD diagrams of different kinds are a very important part. Much of the modeling of a system is described in diagrams and shorter text paragraphs. One could of course use a general purpose drawing tool like for example the picture editor in Word.

The OMT notation includes

- Object Model diagrams (called Class Diagram in the Unified Modeling Language)
- State diagrams and Event Trace diagram in the Dynamic Model in OMT (called State diagrams and Message Trace diagrams in the Unified Modeling Language)

## Registration

The Object Plant is distributed as shareware. You are permitted to use it on a trial basis for up to 30 days. If you wish to continue using the product beyond that period, you are expected to pay a registration fee to obtain a license to use Object Plant.

Entering the license code into the program will remove shareware popups and "Not registered" texts in printouts and exported pictures.

To register use the Register program to create a register form which then shall be sent to Kagi Shareware using mail, email or fax.

The price for a single license is 25 USD, there are also site and world licenses available for 500 USD resp. 2000 USD.

<http://www.kagi.com>

Email: [sales@kagi.com](mailto:sales@kagi.com) (1 to 3 day processing time delay)

FAX: +1 510 652 6589 (4 to 8 day processing time delay)

Postal address: (4 to 8 day delay plus transit time to Kagi)

Kagi

1442-A Walnut Street #392-MU

Berkeley, California, 94709-1405

USA

**Note:** a non-registered copy of Object Plant cannot create documents with more than 20 classes and 20 interfaces.

The CodeRetriever (reverse engineering part of ObjectPlant) is distributed as shareware. You are permitted to use it on a trial basis for up to 30 days. If you wish to continue using the product beyond that period, you are expected to pay a registration fee to obtain a license to use CodeRetriever.

Entering the license code into the program will remove shareware popups.

To register use the Register program to create a register form which then shall be sent to Kagi Shareware using mail, email or fax.

The price for a single license is 10 USD, there are also site and world licenses available for 200 USD resp. 800 USD.

<http://www.kagi.com>

Email: [sales@kagi.com](mailto:sales@kagi.com) (1 to 3 day processing time delay)

FAX: +1 510 652 6589 (4 to 8 day processing time delay)

Postal address: (4 to 8 day delay plus transit time to Kagi)

Kagi

1442-A Walnut Street #392-MU

Berkeley, California, 94709-1405

USA

# Reference Manual

## Views

When doing Object Oriented Analysis and Design one often starts with dealing with a small amount of detail information, e.g. only class names are defined in the analysis phase and decisions on class attributes and operations are left until the Design phase. And it is normally first in the implementation phase that data types are introduced.

To support these different levels of abstraction, the Object Plant lets the user define a set of different Views. A View defines what information that shall be visible in the four main windows.

For the different views you have different settings for the Object Model diagrams, Event Trace diagrams, State Diagrams and Use Case diagrams. In the Object Model you can control:

- if the attribute type shall be visible
- if the operation return type shall be visible
- if the operation arguments shall be visible
- if the visibility (public, protected, private, none) shall be visible
- if the static checkbox shall be visible in dialog windows
- if the notes shall be visible
- if all, none or a selected set of the attributes shall be visible
- if all, none or a selected set of the operations shall be visible
- if all, none or a selected set of the rolenames shall be visible
- if all, none or a selected set of the qualifiers shall be visible
- if all, none or a selected set of the attributes and operations shall be visible for reference classes/interfaces

Attributes, operations, rolenames and qualifiers all have a "Show" checkbox in their dialog windows which controls if they belong to the selected set or not.

For example, an attribute is visible if either

- the current view specifies that all attributes shall be visible or
- the current view specifies that selected attributes shall be visible and the attribute's show checkbox (in the class dialog window) is checked.

In the Event Trace diagrams you can control:

- if notes shall be visible
- if all or none of the event names shall be visible

In the State Diagrams you can control:

- if notes shall be visible
- if all, none or a selected set of the entry and exit actions shall be visible
- if all, none or a selected set of the event actions and do activities shall be visible
- if all, none or a selected set of the event extras shall be visible

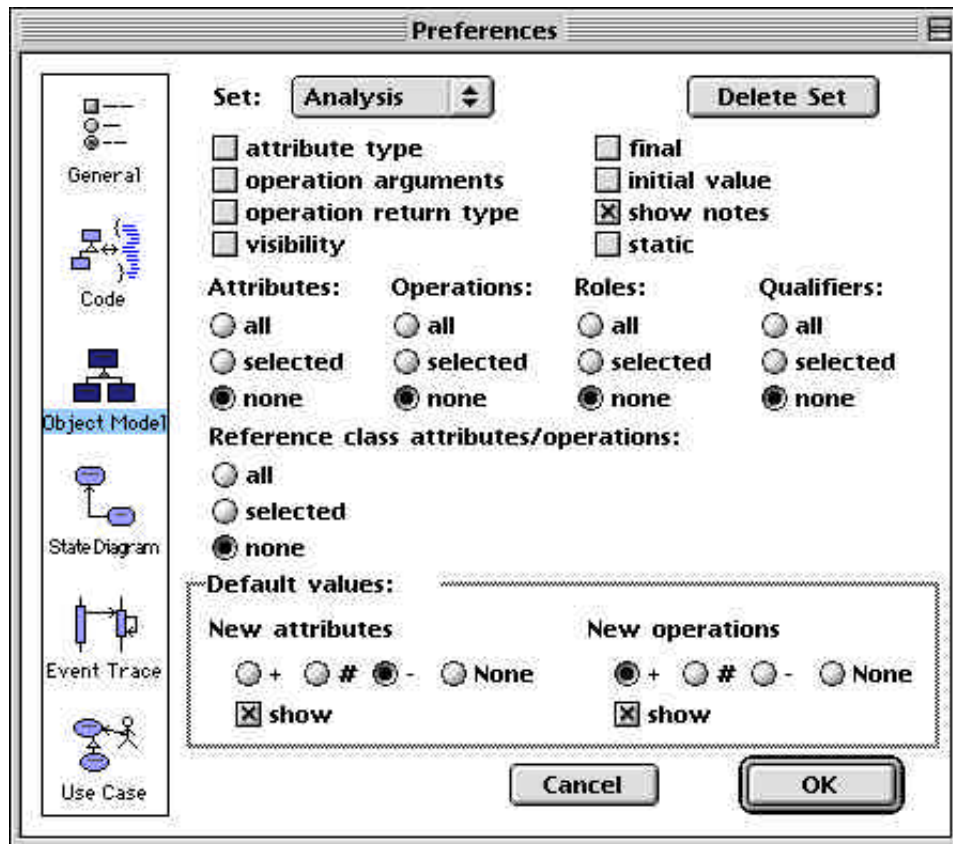
In the Use Case diagrams you can control:

- if notes shall be visible



Please note that if you choose that notes shall not be visible, the Notes tool is still available and you can create Notes but you can't see them or select them.

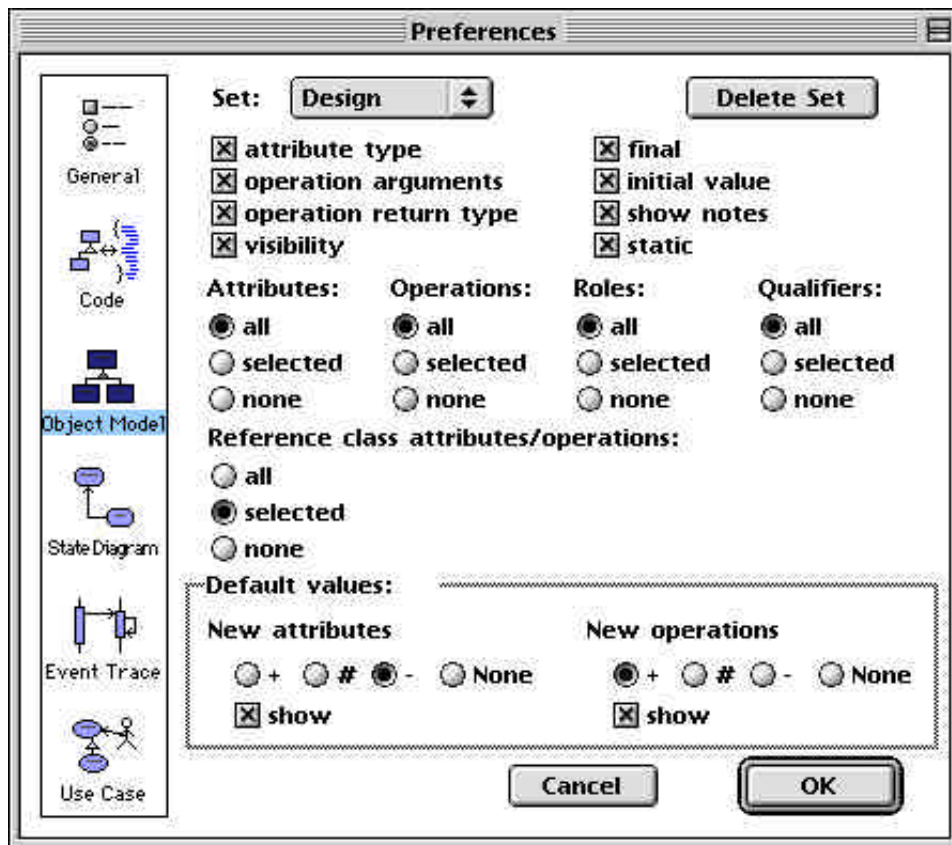
An analysis view could for example have the following settings for the Object Model:



**Figure 1. An example of an Analysis Object Model view setting**

Then only empty classboxes and associations would be visible and no attributes, operations, rolenames or qualifiers would be displayed but the class dialog windows would still include attribute and operation information.

And a design view could have the following settings for the Object Model:

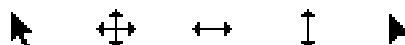


**Figure 2. An example of a Design Object Model view setting**

In the view preferences, there is also a section for default values. In this section (which is not related to the view sets) you can specify some default values that will be used when creating new attributes/operations. You can specify the visibility and the status of the show checkbox.

## **Cursors**

The Object Plant uses different cursors to indicate different states of the toolbox. When the select tool is active, the cursor can either be an ordinary arrow, a move cursor or a resize cursor (shown below).

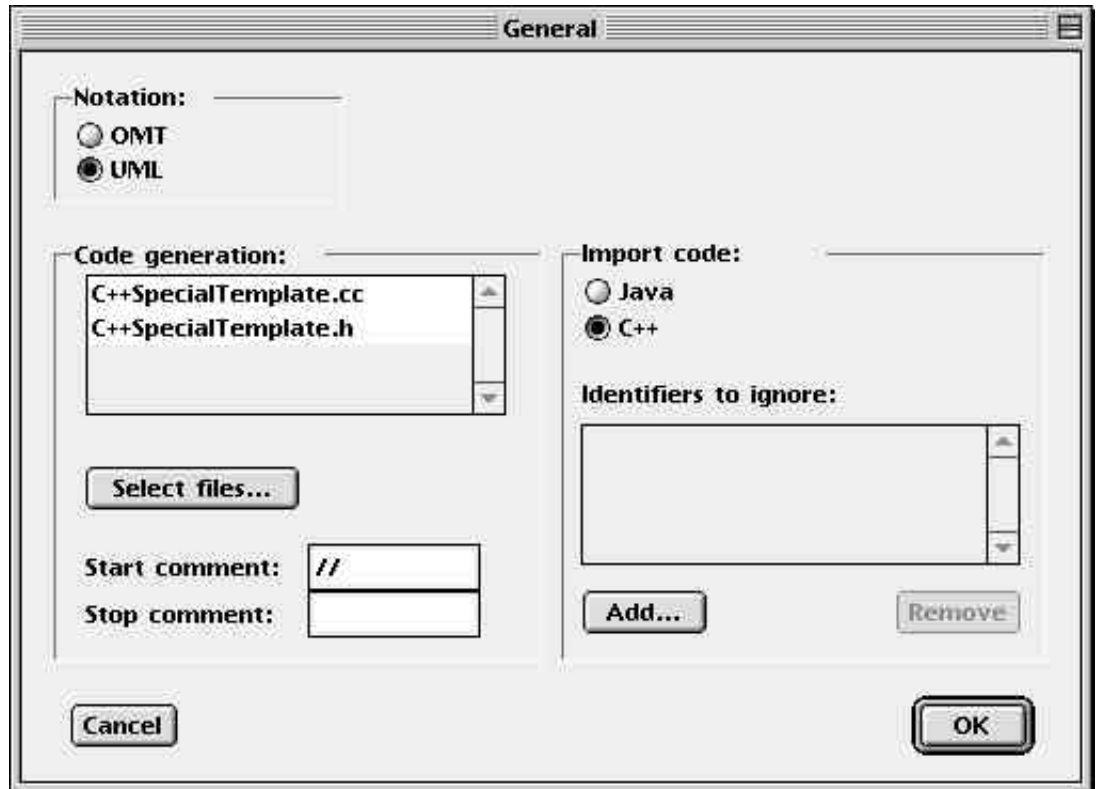


**Figure 3. Different select tool cursors**  
(select, move, horizontal move, vertical move, resize)

## **Document preferences**

### **General**

The File->Document Preferences->General/Code... can be used to select the notation to be used with the document and the language that is used when importing code (reverse engineering) into the document. Furthermore, the code generation templates and start and stop comments for the code generation of the project are specified here.



**Figure 4. The Document Preferences->General/Code dialog**

If you change the notation, this will not show immediately. You have to close the document and open it again.

### **Data types**

You can define a number of data types that is used within the document. This can be useful when starting to decide upon attribute and operation types. The predefined datatypes will then be included in popup menus to allow easy access to them.

You can also save a set of datatypes which can be used in several documents.

To define a document's data types use the File->Document Preferences->Data Types... command. In the data types dialog, you can add and remove single datatypes but also load datatype files and create datatype files.

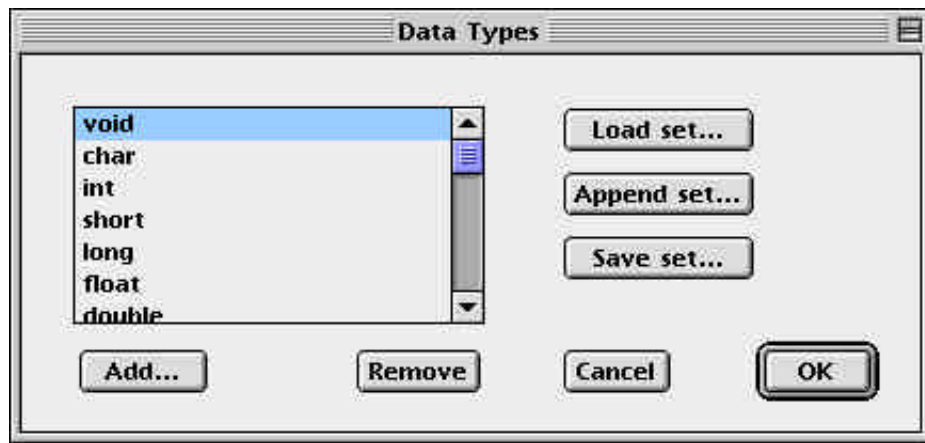


Figure 5. The Document Preferences->Data Types dialog

### Exporting diagram pictures

You can export any diagram page in your model. The page will be exported in either PICT or EPS (Encapsulated PostScript) format. Select the File->Export Page as... command and the following dialog box will show up. (Note that this menu is enabled only when one of the four main windows, Object Model, Event Trace, State Diagram or Use Case, is frontmost.)

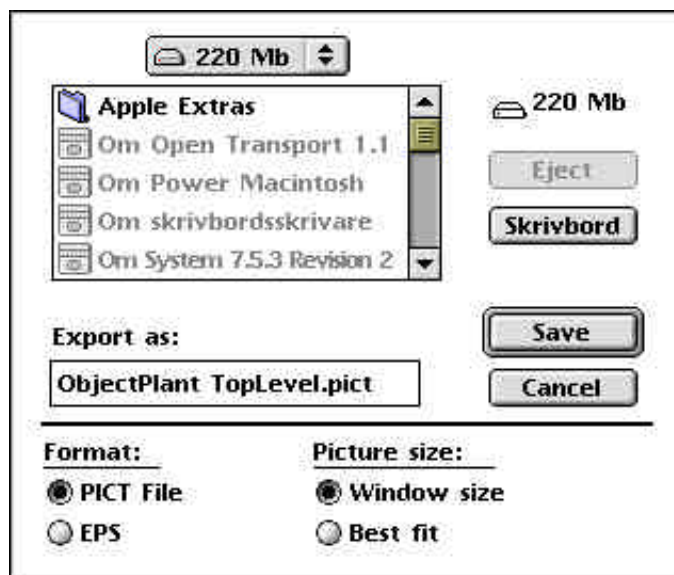


Figure 6. The Export Page dialog

The name of the exported file will default to the name of exported page with a .pict or .eps extension dependent upon the current setting of the format radio buttons.

## **General Preferences**

In the General Preferences dialog you can select:

- if the selected tool will stay active after first use. If this checkbox is not checked, the tool will automatically revert to the Select tool after using any other tool.
- if you want a dialog window to automatically be opened when you create a new item in a diagram, e.g. when you create a new class, the class dialog window will automatically open up if this option is checked.
- if you want to be able to move items immediately when selecting them. If this option is not checked, you will have to wait for a while with the mouse button pressed before the move cursor appears and you are allowed to move the item.
- if you want items to snap to an invisible grid (6x6) (N.B. This is only implemented in the Use Case diagram!)
- if you want a dashed line in the main windows indicating the size of a printed page
- the size of the black select squares
- the size of the generalization arrows
- if the system font shall always be used (in diagrams and dialog windows)
- and if the system font shall not be used, you can specify the font to be used in the main diagram windows.
- the default notation used when creating new documents
- if the type popup-menus shall be sorted alphabetically
- if type popup-menus shall automatically include all classes in document
- if class boxes shall contain an empty attribute compartment if the class does not have any attributes.
- if information about input/output operation parameters shall be displayed in the parameter info dialog (necessary for IDL code generation).

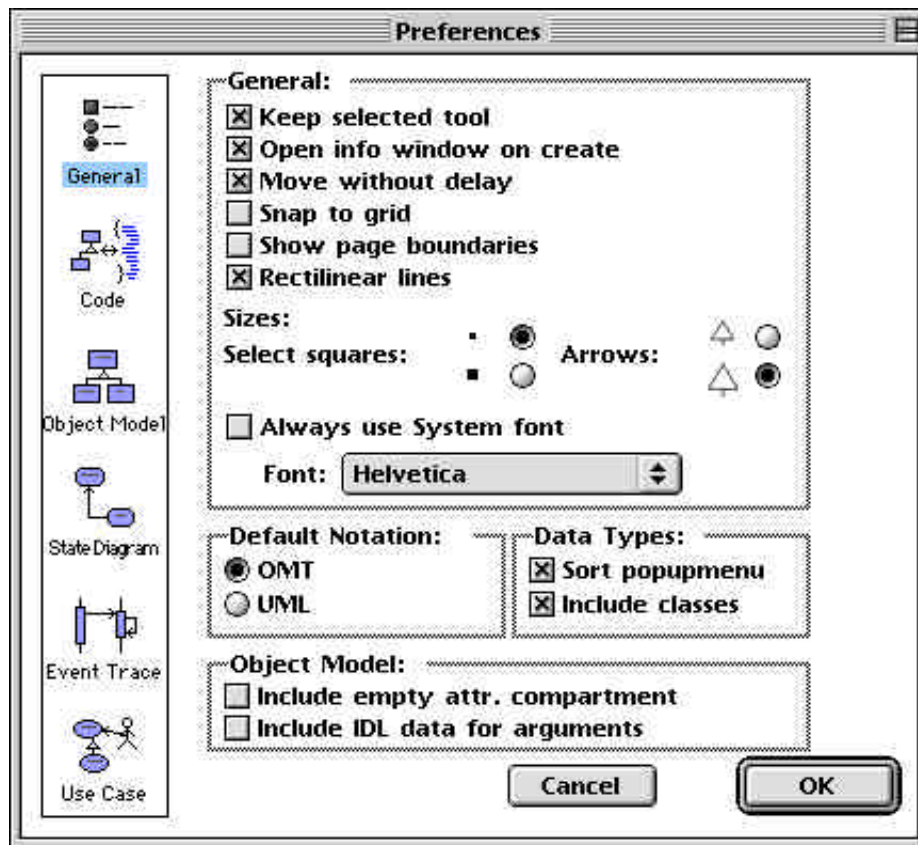
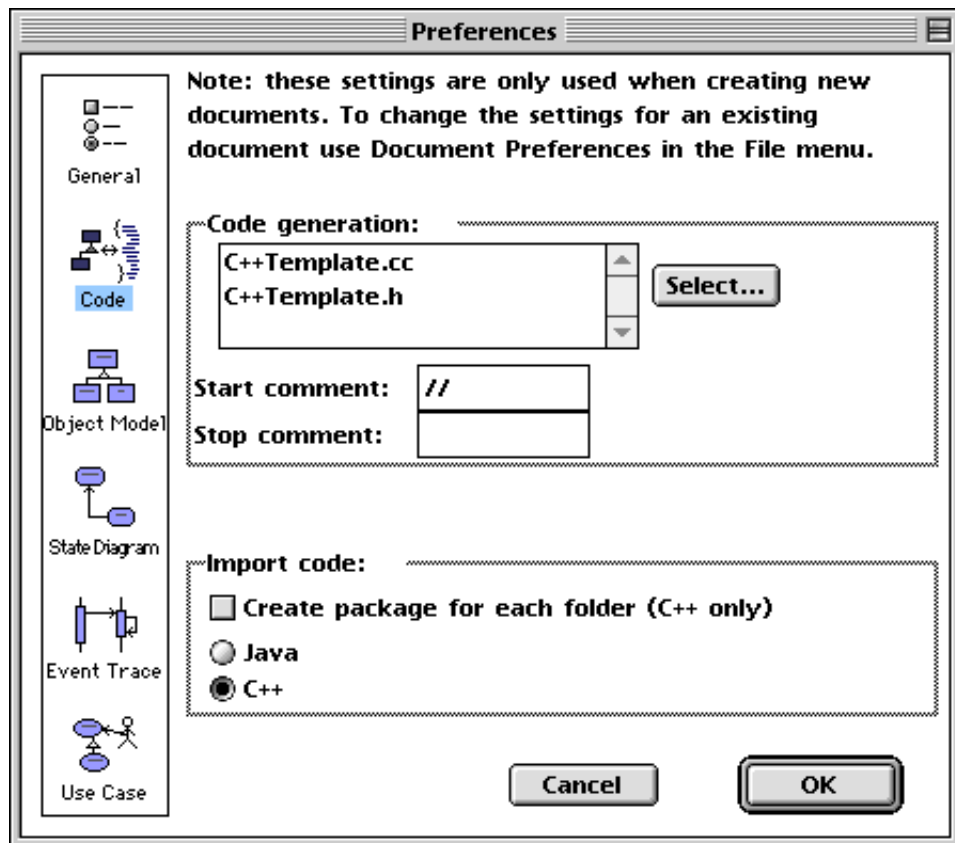


Figure 7. The General Preferences

### Code Generation Preferences

In the code generation preferences dialog you can select what templates files to be used when generating code and also specify how to enclose comments in the generated code. The stop comment field can be left empty if the start comment "tag" works till the end of the line, e.g. the start comment could be set to "/\*" and the stop field left blank for C++.

Please note that these preferences are used when creating a new document. The document may have its own set of code generation preferences which can be accessed through the File->Document preferences->General... menu.



**Figure 8. The Code Generation Preferences**

### **Copy and Paste**

Copy, Cut and Paste is implemented in the Object Plant for all textfields. Copy and Paste is also implemented for the following diagram items:

- Classes (Object Model)
- Interfaces (Object Model)
- Attributes (Object Model)
- Operations (Object Model)
- Parameters (Object Model)
- States (State Diagram)
- State Events (State Diagram)
- Threads (Event Trace Diagram)
- Actors (Use Case diagrams)
- Use Cases (Use Case diagram)

As you can see, references, associations, events and packages cannot be copied. This will be implemented in later versions of the Object Plant.

### **Undo**

Undo is only implemented for most delete operations. The exceptions are:

- no undo for delete in text fields.
- no undo when deleting packages or classes with subsystems

## **Drag-and-Drop**

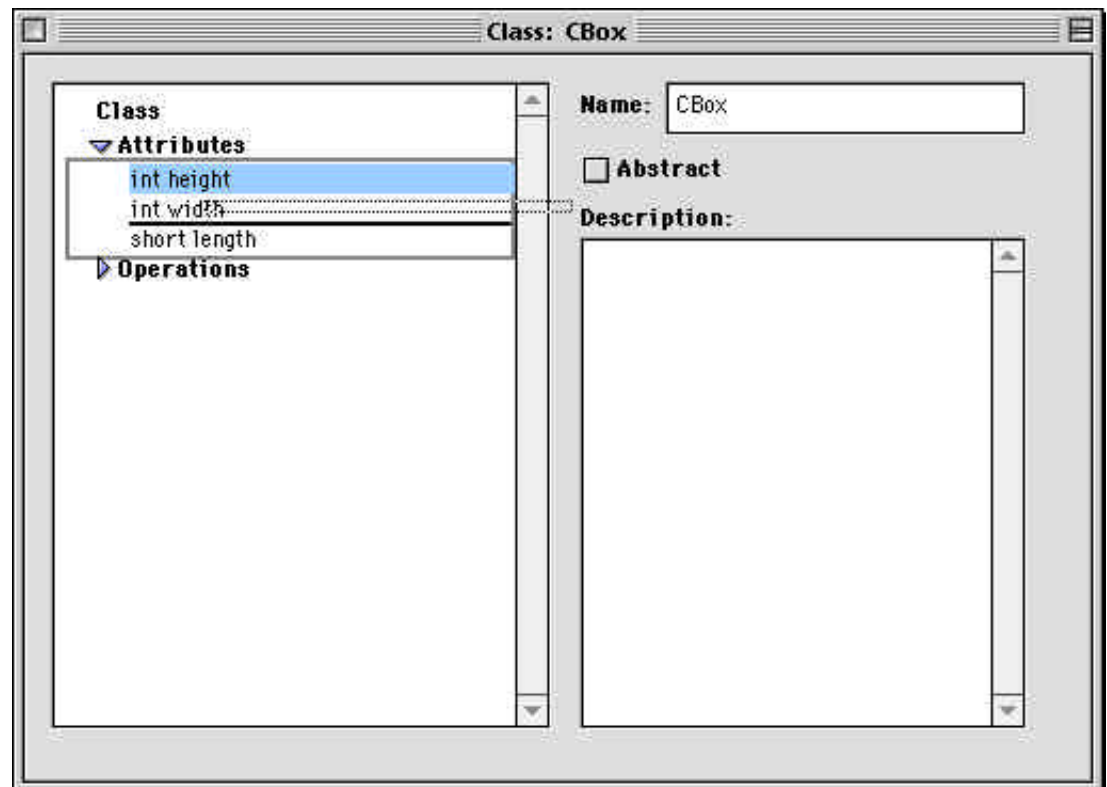
Drag-and-Drop is partly implemented in the Object Plant. It can be used in the Class, Interface, State, UseCase info windows and in the Project window.

In the Class and Interface info windows, drag-and-drop can be used to re-order the attributes, operations and parameters.

In the State info window, drag-and-drop can be used to re-order the events.

In the UseCase info window, drag-and-drop can be used to re-order the extension points.

An example: If you have a list of three attributes, height, width, length and you want to change the order to width, height, length, select the height attribute by clicking on the attribute's line, drag the height attribute without releasing the button and drop it on the width attribute. Voilà, the attributes have been re-ordered.



**Figure 9. Drag-and-Drop in a class dialog window**

It is only possible to drag within a window and not between two different windows.

In the datatypes dialog, drag-and-drop can be used to re-order the data types.



## **Non-obvious keyboard shortcuts**

The following keyboard shortcuts are available.

When any of the three main windows is active:

- TAB-key steps through the tools of the toolbox
- shift-TAB steps through the toolbox in the reverse order
- escape-key selects the Select tool
- command-option-left arrow: align left
- command-option-right arrow: align right
- command-option-top arrow: align top
- command-option-bottom arrow: align bottom
- command-option-A: select association tool
- command-option-C: select class tool
- command-option-G: select generalization tool
- command-option-H: select aggregation (has a) tool
- command-option-I: select interface tool

When the textcursor is in a textfield that has a show checkbox, ctrl-S can be used to toggle the checkbox.

In the Class, Interface and State info windows, the delete key can be used to delete selected attributes, operations or events. If there aren't any selected items, the delete key will be applied to the active textfield in the info window. The backspace is always applied to the active textfield.

In dialog windows with textfields, the TAB key can be used to step between textfields. Shift-TAB steps between textfields in the reverse order.

Holding the option key while closing a dialog window (with mouse click) will close all open dialog windows.

## **Joining line segments**

Lines are used by the Object Plant in associations between classes (and interfaces) and events between states. A line can consist of one or several segments. When a line is selected the endpoints of all segments are marked with a filled square. A segment can be divided into two smaller segments using the cut tool and two segments can be joined into one segment by using the select tool and moving the filled square that links the two segments and drop it on the end of either of the two segments. You have to move the middle point to (almost) the exact position of either endpoint. I know that this can be tricky why you also can use the cut tool to remove handles. If you have the cut tool selected and option-click on a line's handle it will be removed.

## **Object Model Diagrams (Class Diagrams)**

### **Pages**

An Object Model can consist of one or several pages. The page metaphor is close to how one works with a pen and paper, you cannot simply draw a complete Object Model diagram in one single sheet of paper, you must normally have several pages, each page describing a subsystem of the complete system being modeled. Or sometimes having several pages describing parallel systems.

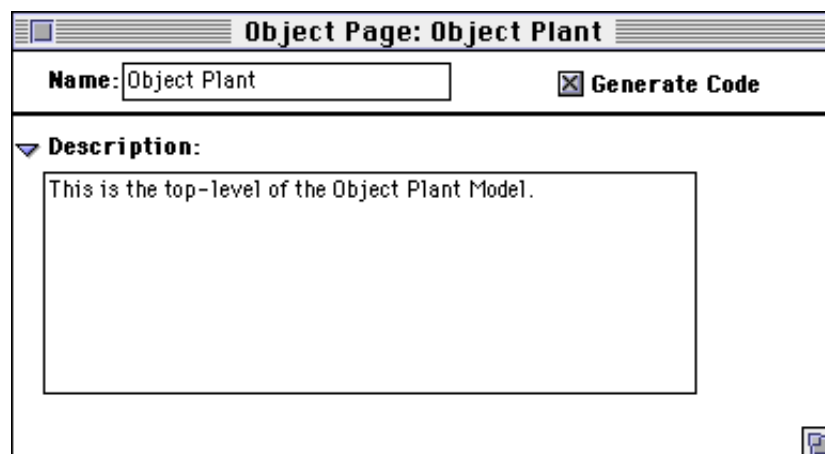
For each Object Model page you can specify if you want that page to generate code when using the "Generate Code..." command. If the "Generate Code" checkbox in the Page dialog window is checked, code will be generated for that particular page.

#### **Adding a new toplevel page**

Adding a new page at the toplevel can be done with the Special->New Page menu command.

#### **Changing the name of a toplevel page**

If you doubleclick on a page name in the Project window you will get a dialog window where you can change the name of the page and also enter a short description of the page's content. There are no controls that the name you select is properly chosen. You can even enter the same name as another page.



**Figure 10. Object Model Page Dialog Window**

### Adding a sublevel page

When drawing Object Model diagrams on ordinary paper it's quite common to link two pages by adding a box with the subsystem's name inside it on the main page and then put that name as the header of the linked page. You can almost do the same way when using Object Plant. To add a new subsystem to the current page, use the class tool and create a class box by clicking somewhere in the Object Model window. Then option-doubleclick the class box and a new page will be created. The name of that new page will be identical to the name of the class box. A subsystem is treated exactly as an ordinary class. The new page will automatically be added to the page palette beneath the current page's entry. The name of the subpage will be indented relative its parent to indicate that it is a subpage.

### Another way of adding sublevel pages

A better way to add a sublevel page is to use the package tool. Adding a package will automatically create a sublevel page.

### Changing the name of a subpage

If you want to change the name of a subpage you cannot doubleclick inside the Project window like you do with toplevel pages, but you have to change the name of either the package or the class representing the sublevel page.

## **Elements in an Object Model**

In an Object Model page you can have the following kinds of elements:

- classes
- interfaces
- supplier associations
- associations
- aggregations
- generalizations
- packages
- dependencies
- notes

The Object Model toolbox contains fourteen different tools:

- |                       |                    |                    |
|-----------------------|--------------------|--------------------|
| • select tool         | • class tool       | • interface tool   |
| • supplier tool       | • association tool | • aggregation tool |
| • generalization tool | • package tool     | • dependency tool  |
| • cut tool            | • frame tool       | • text tool        |
| • note tool           | • inspect tool     |                    |



## **Classes**

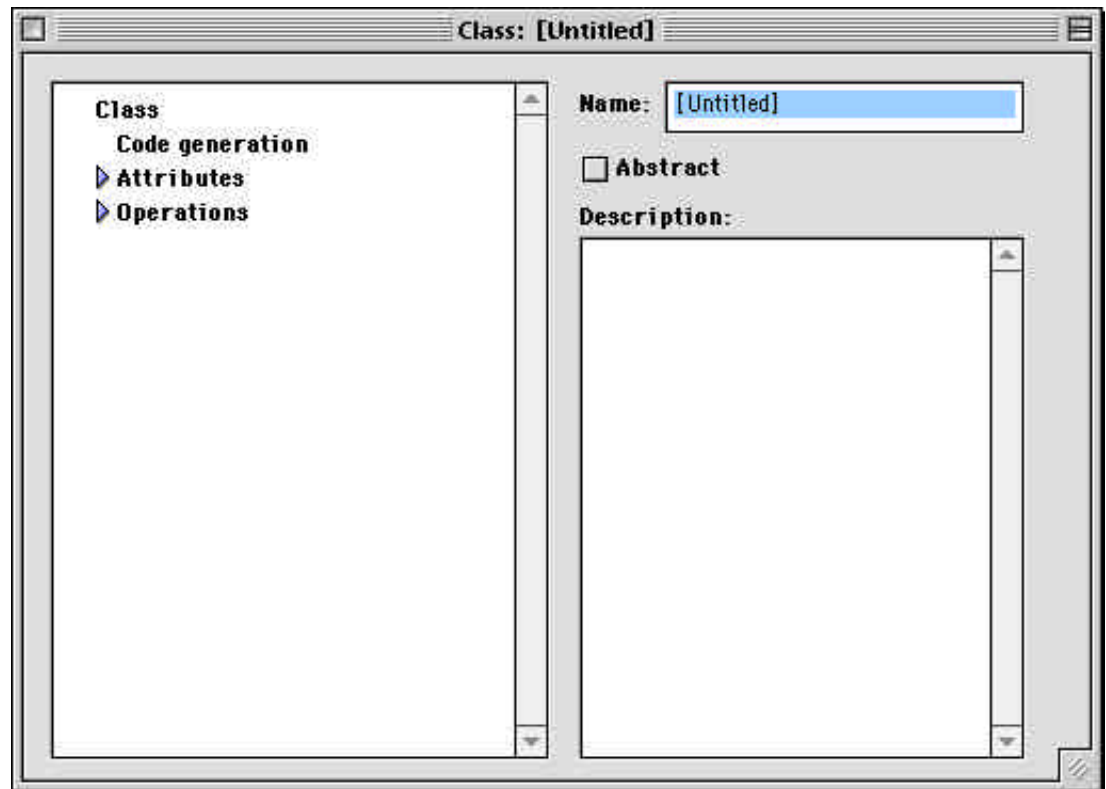
The class tool is used to create classes. Select the class tool and click in the Object Model diagram page where you want the upper left corner of the class box to be positioned. A class box is drawn and the name of the new class is [Untitled]. If you have selected the "Open info window on create" option in Edit->Preferences... General, the dialog window will automatically be displayed when a class is created.

The class dialog window can also be displayed by doubleclicking inside the class box or selecting the class box and then use the Special->Info... menu command.

### The class dialog window

The class dialog window is divided into two sections:

- Class overview (left pane)
- Selected item information (right pane)



**Figure 11. The dialog window of a new class**

In the Information pane (right side) there is a Name field, where you can enter the name of the class. There is also a checkbox where you can select if the class is abstract or not. Abstract classes have all text written in italics. (This is not standard OMT notation, but has been adopted by the Unified Modeling Language.)

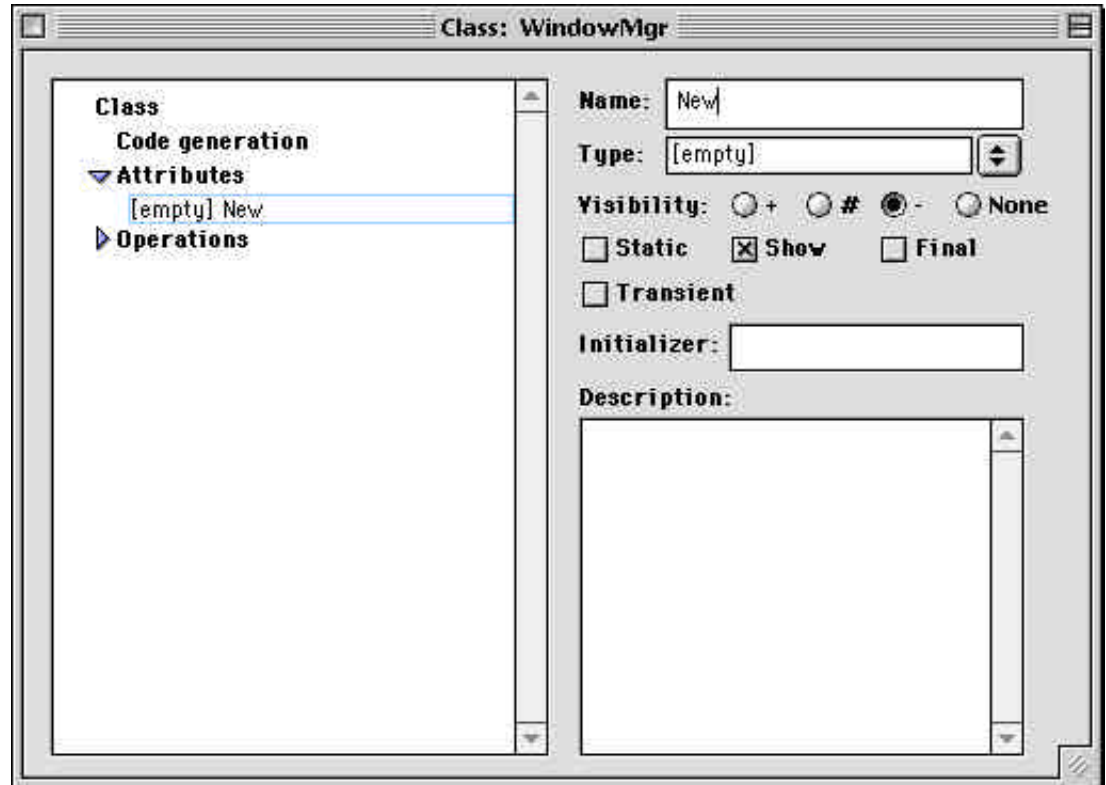
Description is a text field where you can enter a short description of the class. The attributes and operations sections are empty in a newly created class.

The Code Generation section is revealed if you select it by clicking on the Code Generation string in the left hand pane. In that section you can specify one or two filenames for the class. The meaning of these two filenames is determined by the code generation template files use of the FILENAME1 and FILENAME2 tags.

In the two code generation templates C++SpecialTemplate.cc and C++SpecialTemplate.h (included in the ObjectPlant package) the first field (Filename1) is used to build the generated file's name. If you specify CLibrary in the Filename1 field for one or several classes, two files will be generated named CLibrary.h and CLibrary.cc which will contain the code skeletons for these classes.

### Add an attribute

To add an attribute to a class you must have the class dialog window as the front window. Then a new menu called Class becomes active. In this menu there are five items: New attribute, New operation, New parameter, Inherit abstract operations and New Exception. Select the "New attribute" command to add a new attribute. Then the attribute section opens up and the new attribute is displayed with the name of the attribute set to "New".



**Figure 12. A class dialog window when adding a new attribute**

Below the Name of the attribute there are several textfields and controls some of which are not used during the analysis phase but later on (in design and implementation) is used to refine the attribute's properties.

There is a Type textfield with a popup menu to the right of it. In this field you shall enter the datatype of the attribute, e.g. float or CWindow\*. The popup menu can be used to enter a type from a predefined set of datatypes. The popup menu contains a list of all the datatypes that have been specified for this document in the File->Document Preferences->Datatypes dialog. If you created a new document, no data types are defined hence the popup menu is very short. A new attribute has a type called "[empty]". If you need a datatype that is not listed in the popup menu you can enter the data type directly in the textfield to the left of the popup menu. The textfield has an auto-completion feature that will try to auto-complete the type. This is a good reason for actually define the data types to be used in the File->Document Preferences->Datatypes dialog.

Below the Type information you can find a set of Visibility controls. Four radio buttons are used to select a "visibility level", public (+), protected (#), private (-) or none, for the attribute.

Below the Visibility row, there is a checkbox labeled "Show". Refer to the Different Views section for an explanation.

The Static, Final and Transient checkboxes are used to control code generation, Static shall be checked for static attributes (C++ and Java) and Final shall be checked for final attributes (Java).

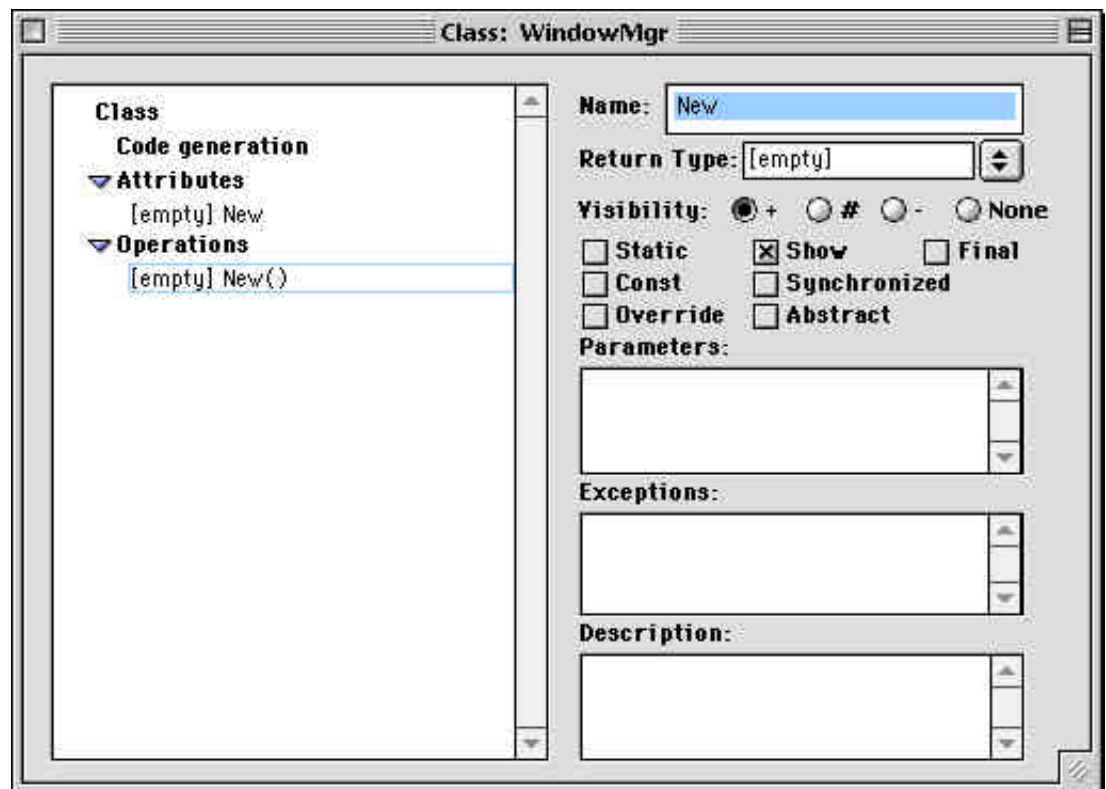
The Initializer textfield is also used to control code generation. In C++ only static attributes can have an initializer but for Java you can use this field all attributes.

At the bottom there is a textfield where you may enter a description of the attribute.

### Add an operation

Adding a operation is as simple as adding an attribute. Be sure that the class' dialog window is in front and then select Class->New operation. The rest is very much like adding an attribute.

The popup menu (with its associated textfield) to the right of the Return Type specifies the return type of the operation.



**Figure 13. A class dialog window with new operation**

A new operation has a return type called "[empty]". The parameters section will show a small list with the operation parameters, their types and names.

In the picture above the operation does not have any parameters.

### Adding a parameter to an operation

Adding a parameter is done by selecting the Class->New Parameter menu command. Then a dialog window will appear where you have to specify the type of the parameter and the parameter's name. In the picture below a parameter with type 'int' and name 'height' has been added.

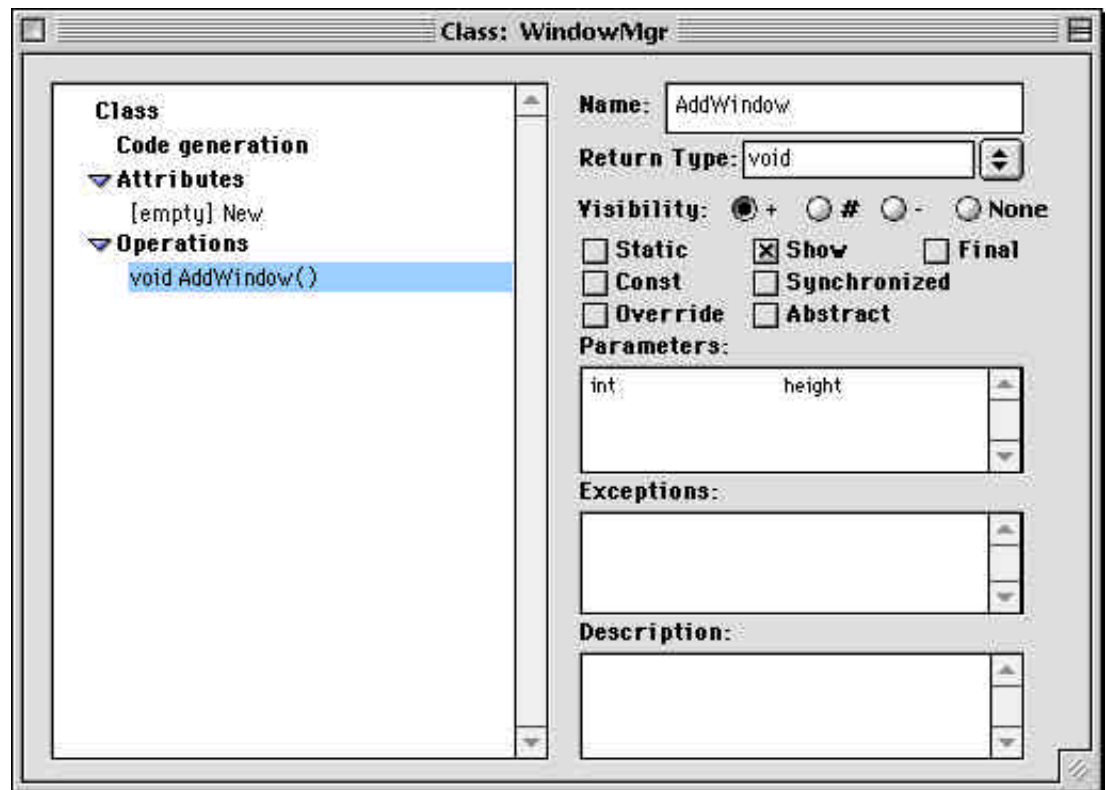


Figure 14. A class dialog window with an operation and its parameter

### Changing a parameter

To change a parameter doubleclick on the parameter in the parameter list and the parameter dialog window will show up where you can change the type and the name of the parameter.

### Delete a parameter

To delete a parameter select the parameter in the parameter list and then choose the Edit->Clear (or the keyboard shortcut: the delete key) menu command.

When an operation is selected the right hand pane shows a lot of code generation specific data, e.g. const, static, final data.

If the Static checkbox is checked, the {STATIC} tag in the code generation template files will be used. The same goes for the Const, Synchronized, Final, Override and Abstract checkboxes. Refer to the section describing the code generation.

Reordering parameters can be done by drag-n-drop.

### **Delete an attribute or operation**

To delete an attribute (or operation) select the attribute you want to delete by clicking at the attribute's line in the left hand pane. The attribute line then gets selected. If you choose Edit->Clear (or the keyboard shortcut: the delete key) the attribute is deleted.

### **Inherit abstract operations**

This menu item will be enabled only when the class has a parent class. By selecting this command, all abstract operations of the class' ancestors will be added to the class with the exception of abstract operations having an identical signature as an already existing operation.

### **Add a new exception**

This menu item will be enabled only when an operation is selected. By selecting this command (in the Class menu), a new Exception item will be inserted in the list of exceptions. A dialog will appear where you will be asked to enter the name of the exception thrown by the operation.

### **Moving a class box**

To move a classbox, use the Select tool. Click and hold the mouse inside the classbox. The box can then be moved.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the move cursor and the classbox can be moved.

### **"Nudging"**

All items in the drawings can be moved by using the arrow keys. They will then move one pixel for each key press. Using option-arrow key gives a coarser move.

### **Resizing a classbox**

The size of a classbox is automatically computed by Object Plant. Hence you cannot change the size of a classbox.

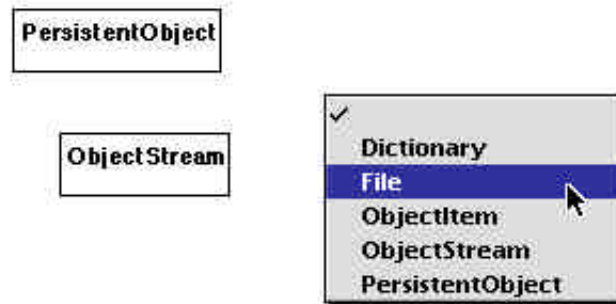
### **Several copies of a single class**

Sometimes it is necessary to have a class displayed in several pages of an Object Model, but they all refer to one and the same class in the model, i.e. there are several class box icons in the Object Model diagrams but only one class storing the information. For example if you have a kind of a base class CPersistence that all other objects that need to be persistent shall inherit from, you want to define the CPersistence class once and then have several references to that class, possibly one reference in every single page in your model.

Creating several CPersistence classes would not give the desired result if you consider code generation. In that case several CPersistence class files would be generated and maybe you have to merge them manually to get a single CPersistence class file.

The solution to this is to create a class reference, ie. a class box that refers to an already existing class. A class reference is created by using the class tool but holding down the shift key when clicking in the Object Model diagram. If you do that a popupmenu will appear listing all existing classes to choose from. Select the class that you want this class reference to refer to.

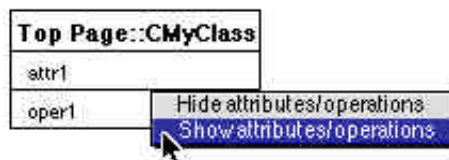




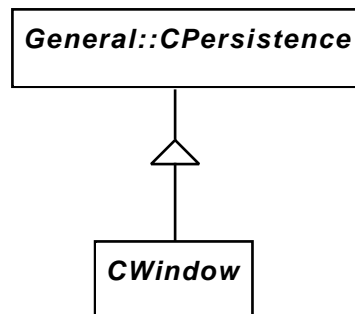
**Figure 15. Creating a class reference**

A class reference box looks much like an ordinary class, but attributes and operations are normally not displayed, only the class name preceded name of the page where the original class is located. (This is in fact a user option that can be found in the View menu.)

If you want the attributes and operations to be displayed you should select this in the Preferences dialog and, unless you have the "All" option selected, for every reference class which should display the attributes and operations option-click on the class with the select tool to bring up a popup menu where you select whether the attributes/operations shall be displayed or not.



**Figure 16. The class reference popup menu**



**Figure 17. A class reference**

If you delete a class reference box, only that reference is deleted and the main class and all other references are unaffected. If you try to delete an ordinary class which have reference classes you will get an popup confirming the delete operation. If you confirm the main class and all of the references will be deleted.



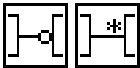
## Interfaces

The interface tool is used to create interfaces. Select the interface tool and click in the Object Model diagram page where you want the upper left corner of the interface box to be positioned. An interface box is drawn and the name of the new interface is [Untitled].

An interface is very much the same as a class. It will behave almost identically in the Object Plant. There are, however, some differences:

In the interface dialog window, the attributes and operations never have any visibility radio buttons since an interface (and its attributes and operations) by default is public.

As with classes, an interface may have references, ie. several interface box icons representing the same interface model item. Using the interface tool while holding down the shift key creates an interface reference box.



## Associations

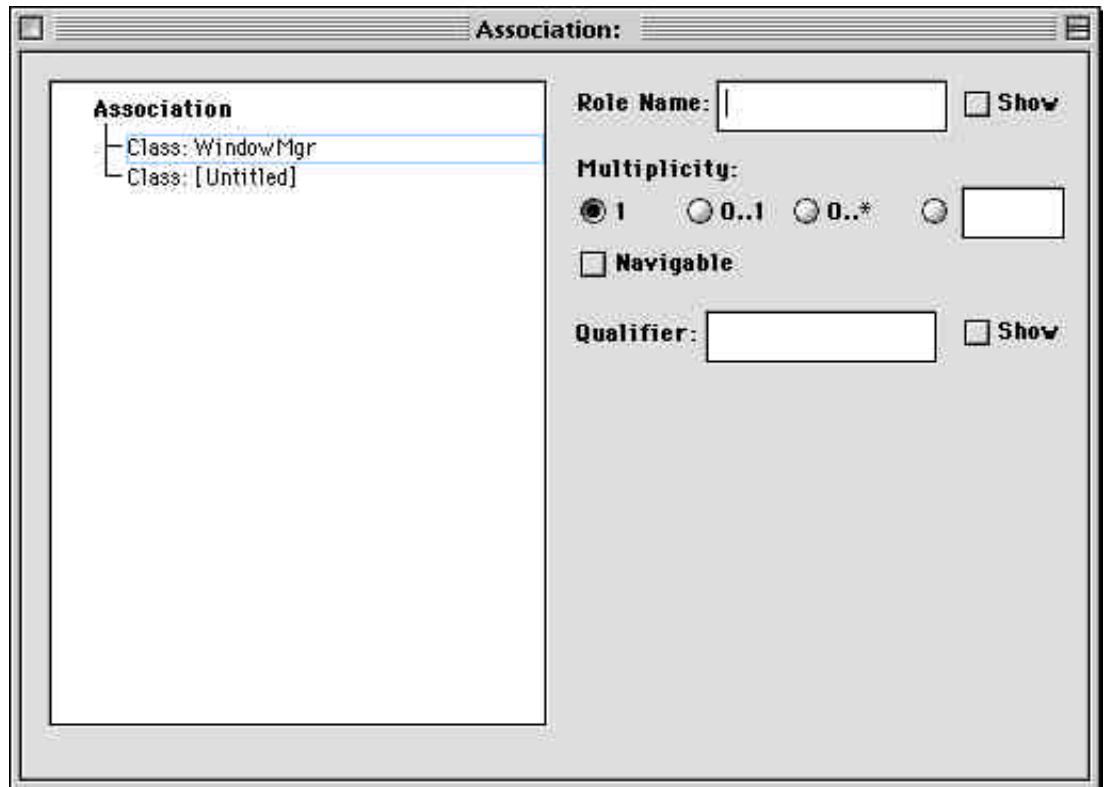
The association tool is used to create associations between classes. Select the association tool and click and hold the mouse inside a class that shall be linked to another (or the same) class with an association. Drag the mouse to point at the other class and then release the mouse button. An association is now drawn between the two classes. It is either a single straight line or any combination of horizontal and vertical lines that connect the classes.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the dialog window will automatically be displayed when an association is created. The association dialog window can also be displayed by doubleclicking on the association line or selecting the association and then use the Special->Info... menu command.

### The association dialog window

The association dialog window is divided into two sections:

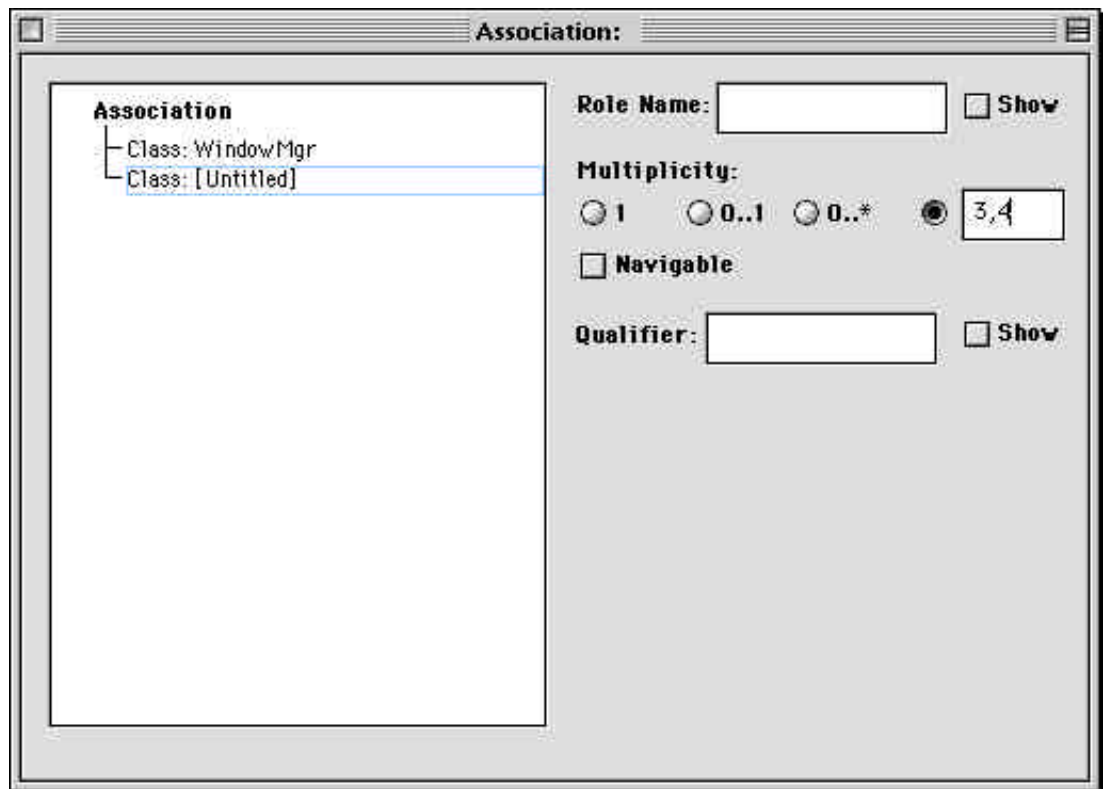
- Association overview (left pane)
- Selected item information (right pane)



**Figure 18. The association dialog window**

Each end of an association is attached to a class which is listed in the left pane. Each association end can have a rolename that can be specified in the Role Name text field. You can also specify the multiplicity of each end of the association. In the picture above, there is exactly one WindowMgr object connected to objects of the [Untitled] class.

The last of the four radiobuttons gives you the possibility to specify any range in an extra text field, e.g. a car has three or four wheels (shown in the picture below).



**Figure 19. The association dialog window with specified multiplicity**

### Qualifiers

Furthermore you can specify a qualifier for each end of an association. A qualifier is shown only if either:

- the current view specifies that all qualifiers shall be visible or
- the current view specifies that selected qualifiers shall be visible and the qualifier's show checkbox (to the right of the qualifier's textbox) is checked.

### Moving associations

You can move endpoints of a linesegment, but only one point at the time. To move a line segment you need to do two move operations, one for each endpoint. The Select tool is used to move an endpoint. Click and hold the mouse on the black square at the endpoint and when the cursor changes into the move cursor you can move the endpoint. If the endpoint is connected to a class, the move will be constrained to the classbox. Other endpoints can be moved freely.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the move cursor and the endpoint can be moved.



## Aggregations

The aggregation tool is used to create aggregations between classes. Select the aggregation tool and click and hold the mouse inside a class that shall be linked to another (or the same) class with an aggregation. Drag the mouse to point inside the other class box and then release the mouse button. An aggregation is now drawn between the two classes. It is either a single straight line or any combination of horizontal and vertical lines that connect the classes.

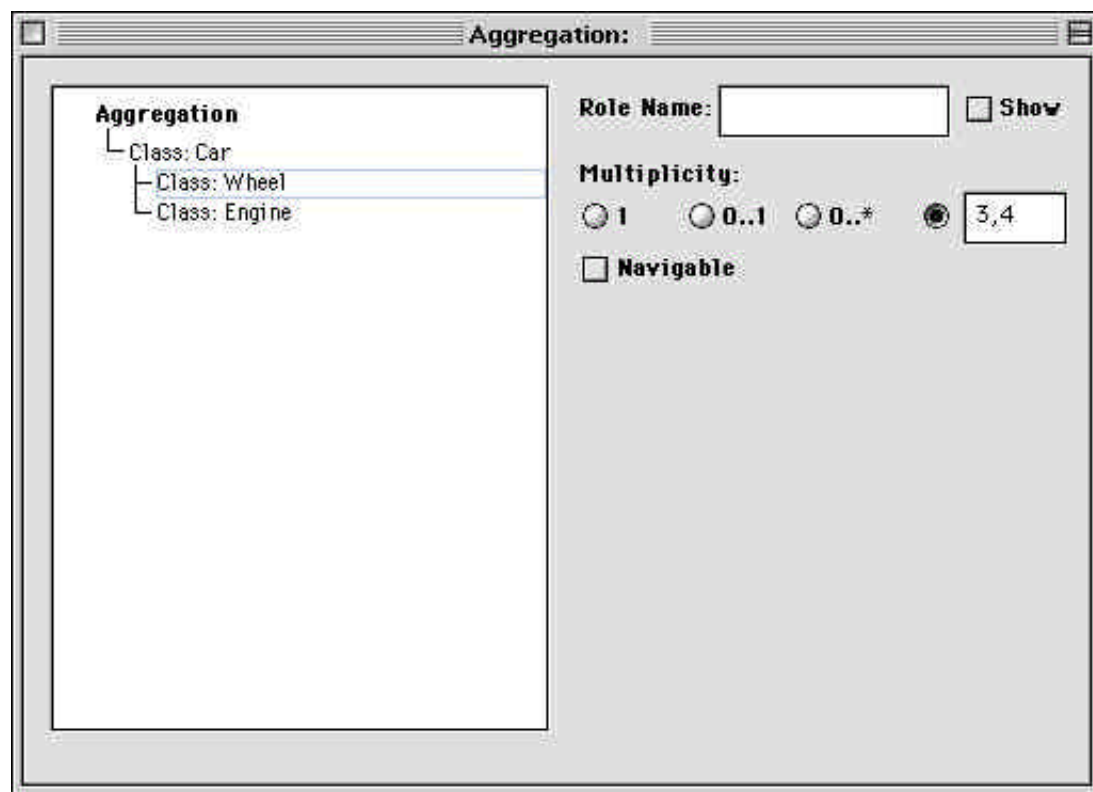
If you have selected the "Open info window on create" option in Edit->Preferences... General, the dialog window will automatically be displayed when an aggregation is created. The aggregation dialog window can also be displayed by doubleclicking on the aggregation line or selecting the aggregation and then use the Special->Info... menu command.

### The aggregation dialog window

The aggregation dialog window is divided into two sections:

- Aggregation overview (left pane)
- Selected item information (right pane)

Below is an example of a dialog window for a multibranch aggregation.



**Figure 20. The aggregation dialog window with specified multiplicity**

In the Role, Multiplicity & Qualifier section we find the names of the classes that are connected with the aggregation. The first class listed is always the top class which "consists of" the other classes.

Each aggregation branch can have a rolename that can be specified in the Role Name text fields. Furthermore you can specify the multiplicity of each branch.

### Creating multibranch aggregations

It is possible to make multibranch aggregations by first making an ordinary aggregation and then when the second branch is to be created, click and hold the mouse inside the diamond of the first aggregation, drag the mouse and release it inside the class of the second of the second branch.

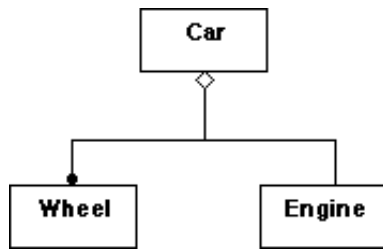
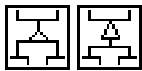


Figure 21. A multibranch aggregation

### Moving aggregations

Aggregations are moved in the same way as associations are moved but with one exception, the endpoint of the stem of a multibranch aggregation can only be moved if the stem itself is selected, i.e. you cannot move the "branchpoint" if you select a branch and then tries to move the branchpoint.



### **Generalizations**

The generalization tool is used to create generalization between classes. Select the generalization tool and click and hold the mouse inside a class that shall be the superclass (ancestor). Drag the mouse to point inside the generalized class box and then release the mouse button. A generalization is now drawn between the two classes. It is either a single straight line or any combination of horizontal and vertical lines that connect the classes.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the generalization dialog window will automatically be displayed when a generalization is created. The generalization dialog window can also be displayed by doubleclicking on the generalization line or selecting the generalization and then use the Special->Info... menu command.

### The generalization dialog window

The generalization dialog window is divided into two sections:

- Generalization overview (left pane)
- Selected item information (right pane)

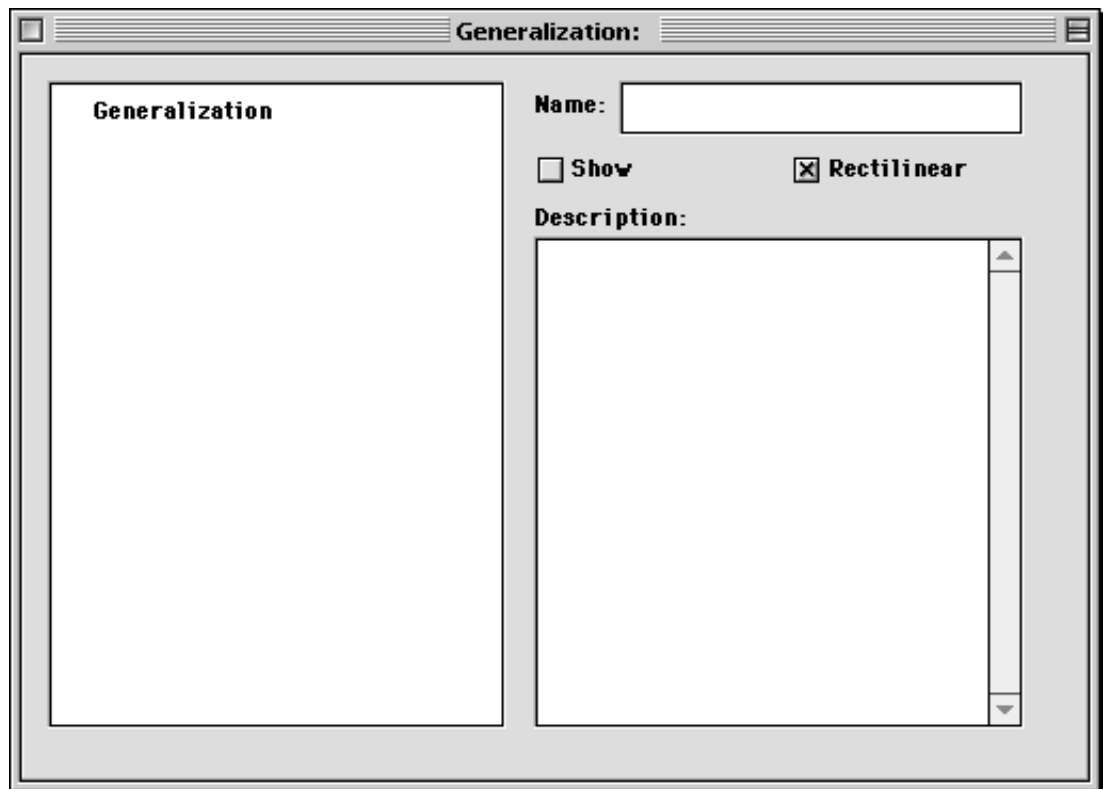


Figure 22. The generalization dialog window

#### Creating multibranch generalization

It is possible to make multibranch generalization by first making an ordinary generalization and then when the second branch is to be created, click and hold the mouse inside the pyramid of the first generalization, drag the mouse and release it inside the class of the second of the second branch.

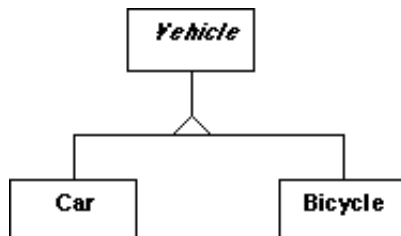


Figure 23. A multibranch generalization

## Moving generalizations

Generalizations are moved in the same way as aggregations are moved.



### **Suppliers**

The supplier tool is used to create supplier relations between a class and an interface. Select the supplier tool and click and hold the mouse inside a class that shall be the supplier. Drag the mouse to point inside the interface box and then release the mouse button. A supplier line is now drawn between the class and the interface. It is either a single straight line or any combination of horizontal and vertical lines.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the supplier dialog window will automatically be displayed when a supplier relation is created. The supplier dialog window can also be displayed by doubleclicking on the supplier relation line or selecting the supplier relation and then use the Special->Info... menu command.



### **Packages**

The package tool is used to create a package. A package is a group of model items and every package is associated with an own page which name is the same as the package's name.

When you create a package a new page will automatically be created and the new (and empty) page will be displayed right after creating the new package.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the package dialog window will automatically be displayed when a package is created. In the package dialog you can name the package and also enter a textual description of the package.



### **Dependency**

The dependency tool is used to create dependency relations between packages. Select the dependency tool and click and hold the mouse inside a package that shall be the dependent. Drag the mouse to point inside another package and then release the mouse button. A dependency line is now drawn between the packages. It is either a single straight line or any combination of horizontal and vertical lines.



### **Cut Tool**

With the cut tool a line segment can be split into two segments. Just click on the line where you want the line segment to be split. You can join two segments by dragging their connect point and drop it at either end point of the two segments or by using the cut tool and option click on a handle.



### **Frame Tool**

With the frame tool a simple outlined frame can be created. This can be useful to visually group items.





### **Text Tool**

With the text tool a text of selected font and size can be created.



### **Notes**

A note is general text field. Notes are supported by the Unified notation but not in OMT. The note dialog window only includes one section, the description, which is the text that is displayed inside the note. The size of a note will not be computed automatically like the class box. You can change the size of the note box by clicking the mouse in one of the corners, wait for the cursor to change into the resize cursor and stretch the box by moving the mouse.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the resize cursor.



### **Inspect Tool**

The inspect tool can be used to read the description of any item (class, association, generalization or aggregation) without opening a dialog window. Select the inspect tool and position the magnifier glass above an item and the description text will be displayed.

If you double-click on any item in the Object Model diagram, the item's dialog window will be displayed with the Description box opened.

## **Event Trace Diagrams (Message Trace Diagrams)**

### **Pages**

An Event Trace diagram normally describes a scenario that has been chosen for some reason. Perhaps because the scenario involves parallel activities which is easily described in an Event Trace diagram.

For a complete system a set of Event Trace diagrams can be drawn. How many Event Trace diagrams that is drawn depends on the nature of the system being described.

As with the Object Model diagram the page metaphor is close to how one works with Event Trace diagrams. The difference to Object Model diagrams is that you normally don't nest Event Trace diagrams. Or at least you can't nest them in Object Plant. You can always use the package tool to group event trace diagrams into logical units.

### **Elements in an Event Trace diagram**

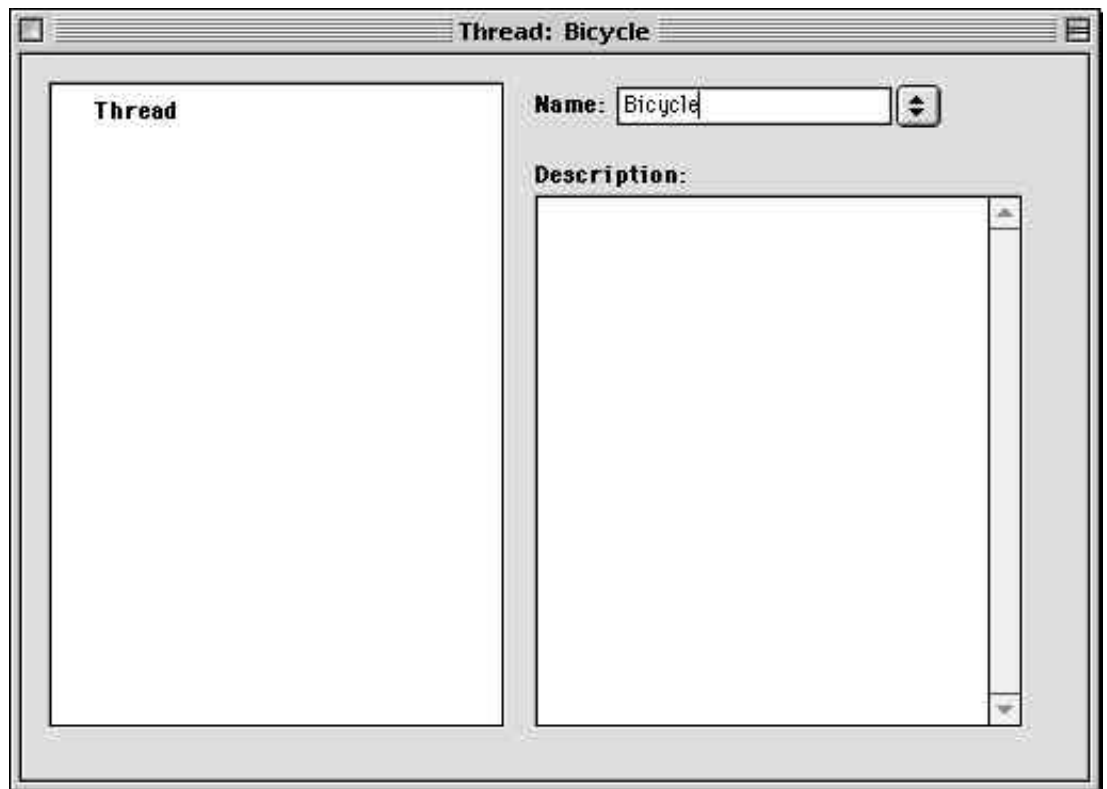
In an Event Trace diagram page you can have the following kinds of elements:

- threads
- boxes
- events
- packages
- notes



#### **Threads**

When drawing Event Trace diagrams a thread is a specific object instance or any instance of a specific class. To guide the user towards this use of thread-class "cohesion", the dialog window for a thread looks a bit different from other dialog windows. Instead of a free text field for the name of the thread, a popup menu is displayed containing a list of the classes that the complete system contains.



**Figure 24. The thread dialog window**

If you want to name your thread to something else you can enter any name in the textfield to the left of the popup menu.

If you don't start with the Object Model and define the classes but rather start with the Event Trace diagram you will get an almost empty list when selecting the popup menu. This is one good reason for starting with the Object Model.

#### **Resizing a thread**

To change the length (or height) of a thread use the select tool and position the cursor above either the top end of the thread or the lower end of thread. Click and hold the mouse on the black square (if the thread is already selected) at the endpoint and when the cursor changes into the resize cursor you can move the endpoint.

All the threads in a page must have the same length, hence if you modify the length of one thread, all the others will also change length.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the resize cursor and the endpoint can be moved.



#### **Boxes**

With box tool you can place frames on threads that indicates when a thread is active. The definition of "active" is a bit vague, in a multiprocess system it may indicate that a process is running, in a single-process system it can indicate that code belonging to the thread is executed, or waiting for other threads to complete a task.

When the box tool is active and you move the cursor above a thread, there will be a box hanging from the cursor ready to be placed on the thread. Click once to create a box where the mouse is positioned.

A box does not have a dialog window since no information is needed for a box.

### **Moving a box**

To be filled in.

### **Resizing a box**

To change the height of a box use the select tool and position the cursor above either the top end of the box or the lower end of the box. Click and hold the mouse on the black square (if the box is already selected) at either endpoint and when the cursor changes into the resize cursor you can move the endpoint. The move of an endpoint will be constrained by event lines, thread limits and other boxes.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the resize cursor



### **Interactions**

The interaction tool is used to create interactions between threads. Select the interaction tool and click and hold the mouse inside a box that shall interact with another (or the same) box. Drag the mouse to point at the other box and then release the mouse button. An interaction is now drawn between the two boxes. Note that you can only make "horizontal" interactions, i.e. it must be possible to connect the two boxes with a horizontal line.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the dialog window will automatically be displayed when an interaction is created.

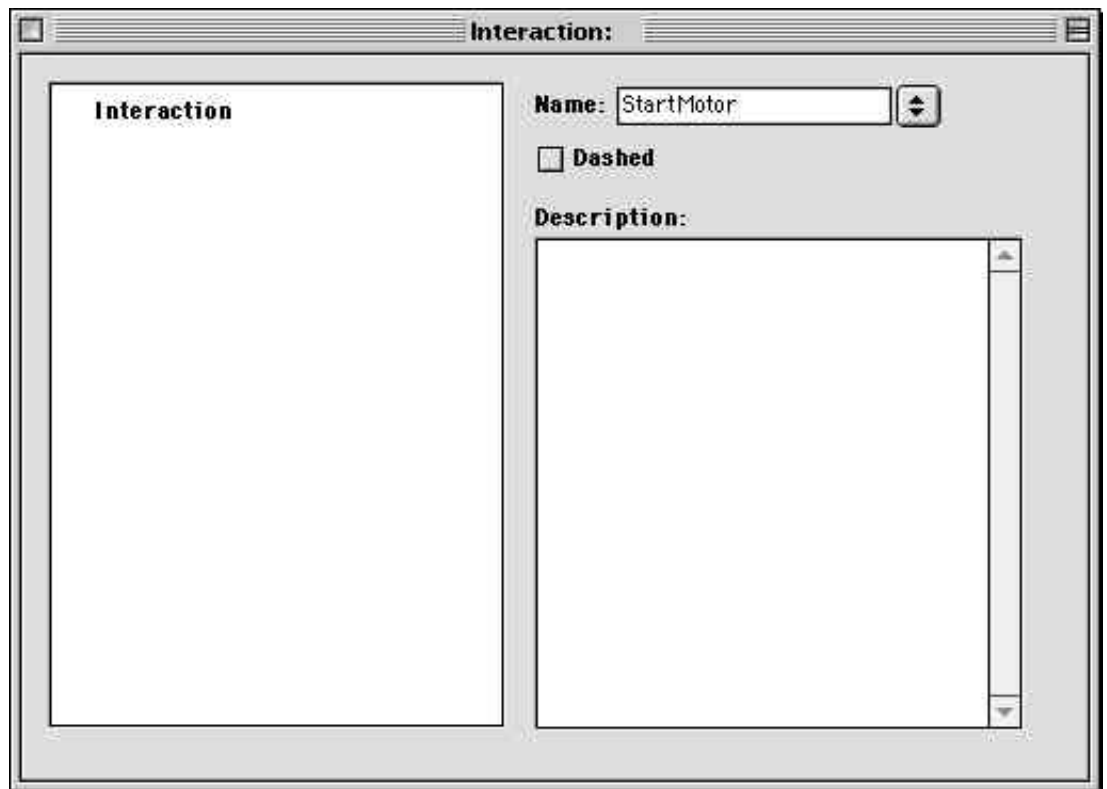
### **The event dialog window**

The interaction dialog window is divided into two sections:

- Interaction overview (left pane)
- Selected item information (right pane)

The interaction dialog window looks different depending upon if the receiving thread is a class thread or some other kind of thread.

If the receiving thread is a class, the dialog window looks like this:



**Figure 25. The interaction dialog window**

Where the Name section has a popup menu containing a list of operations that the receiving thread (class) has.

If the receiving thread is not a class, the Name section contains an ordinary text field without the popup menu for the name. Below the name, there is a checkbox labeled "Dashed". If you check it, the interaction line will be dashed. This is not standard OMT, but is sometimes used to indicate creation of the receiving thread object.

Below the checkbox there is an ordinary text field for a description of the interaction.

### **Moving an interaction**

The Select tool is used to move an interaction. It can only be moved up and down within the limits of the sender and receiver thread boxes. Click and hold the mouse on the interactionline and when the cursor changes into the move cursor you can move it.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the move cursor and the interaction can be moved.

## State Diagrams

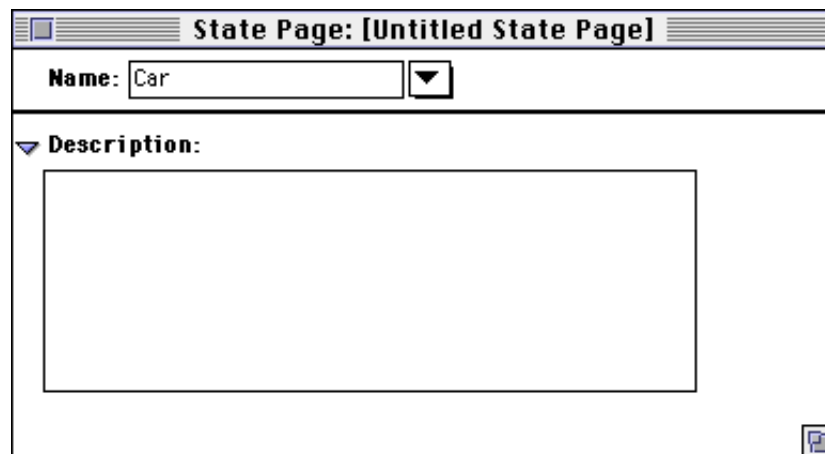
### Pages

A State Diagram normally describes the different states that an object instance of a specific class can take. Not all objects have internal states, but the classes that do have internal state handling can be described with a State Diagram each.

As with the Object Model and Event Trace diagrams the page metaphor is close to how one draws State Diagrams on paper. You describe the behaviour of a class in a separate State Diagram (a diagram/page). Sometimes State Diagrams are used to described complex behaviour and then nested State Diagrams is useful. The Object Plant does not support nested State Diagrams. You can use the package tool to create subpages.

#### Changing the name of a State Diagram page

If you doubleclick on the page in the page palette you will get a dialog window where you can change the name of the page and also enter a short description of the page's content. Instead of a free text field for the name of the page, a popup menu is displayed containing a list of the classes that the system contains. This will guide you into using a State Diagram page for each class you will describe the states for.



**Figure 26. The state page dialog window**

If you want to name your State Diagram page to something else there is an "Other..." alternative in the popup menu. If you select the "Other..." alternative, you will get a dialog where you can enter any name you want for the page.

If you don't start with the Object Model and define the classes but rather start with State Diagrams you will get an almost empty list when selecting the popup menu. This is another good reason for starting with the Object Model.

### Elements in a State Diagram

In a State Diagram page you can have the following kinds of elements:

- states
- events
- packages
- notes



## States

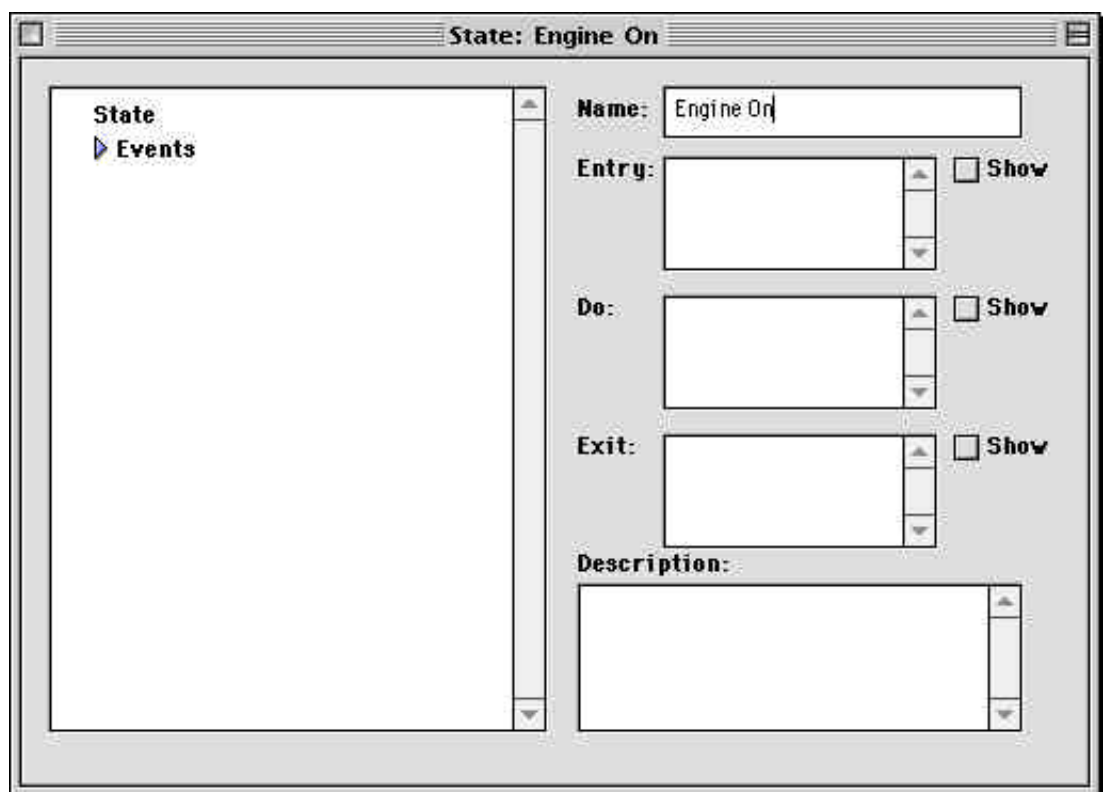
The state tool lets you create states. A state specifies the response of an object to input events. The response can include an action or a change of state. A change of state caused by an event is called a transition.

To create a state place the cursor where you want the upper left corner of the state box to be positioned. A state box is drawn and the name of the new state is [Untitled]. If you have selected the "Open info window on create" option in Edit->Preferences... General, the state dialog window will automatically be displayed when a state is created.

### The state dialog window

In the state dialog window there are two sections:

- State overview (left pane)
- Selected item information (right pane)



**Figure 27. The state dialog window**

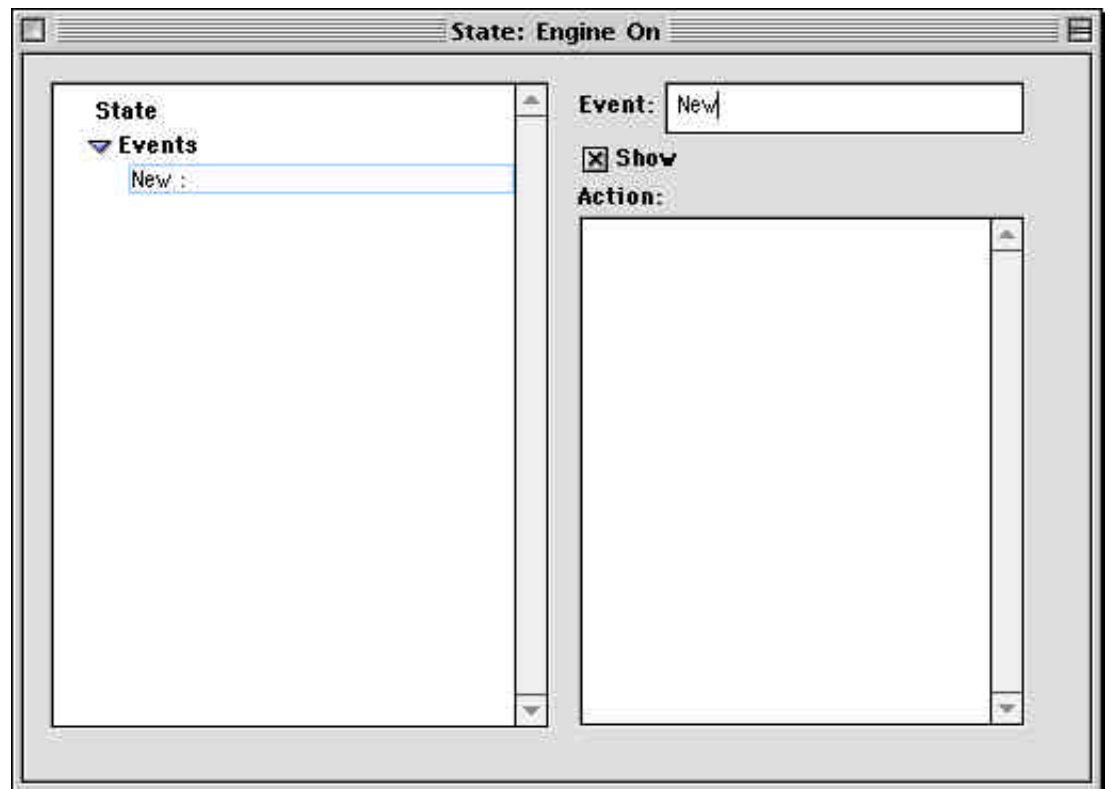
In the Name field, you can enter the name of the state. Description contains a text field where you can enter a short description of the state. The Events section is empty in a newly created state.

Below the Name field there are three textfields corresponding to:

- entry action:  
the action taken when entering the state.
- do activity:  
the activity (an activity takes time to complete while an action is (almost) an instantaneous operation.) that takes place while being in the state.
- exit action:  
the action that is performed when leaving the state.

#### **Add an event**

To add an event to a state you must have the state dialog window as the front window. Then a new menu called State becomes active. In this menu there is only one item: New event. Select the "New event" command to add a new event. Then the event section opens up and the new event is displayed with the name of the event set to "New".



**Figure 28. The state dialog window with a new event**

Below the event name text field, there is a checkbox labeled "Show". Refer to the Views section for an explanation.

There is also an Action text field. This is where you can describe the actions taken when receiving the event.

#### **Delete an event**

To delete an event select the event you want to delete by clicking on the event line. The event line then gets selected. If you then choose Edit->Clear (or the keyboard shortcut: the delete key) the event is deleted.



### **Moving a state box**

To move a state box, use the Select tool. Click and hold the mouse inside the state box. Then box can then be moved.

#### **Note:**

If you have the "Move without delay" option disabled, you must hold the mouse down a while before the cursor changes into the move cursor and the state box can be moved.

### **Resizing a state box**

The size of a state box is automatically computed by Object Plant. Hence you cannot change the size of a state box.



## **Events**

The event tool is used to create transitions between states. Select the event tool and click and hold the mouse inside a state that shall be linked to another (or the same) state with an event transition. Drag the mouse to point at the other state and then release the mouse button. An event is now drawn between the two states. It is either a single straight line or any combination of horizontal and vertical lines that connect the states.

If you have selected the "Open info window on create" option in Edit->Preferences... General, the dialog window will automatically be displayed when an event is created.

### **The event dialog window**

The event dialog window is divided into two sections:

- Event overview (left pane)
- Selected item information (right pane)

**Figure 29. The event dialog window**

An event can have an attribute, e.g. if the event is keypress the attribute could for example be key value=<cr>. This means that the transition would only take place if the event keypress with value <cr> is received by the state.

A guard is another type of condition, e.g. if the event is keypress, the attribute is <cr> and then lets say that the transition shall take place only if no modifier keys are used. This could then be entered into the Guard text field.

The action field is used when the transition itself causes an action to take place. This could often also be put into either the exit actions of the state we came from or the entry actions of the state that we're going to.



### **Initial state**

The initial state tool is used to place an initial state symbol in a state diagram. There can only be one initial state symbol in a state diagram. Events can be drawn only from an initial state to other states.

An initial state does not have a dialog window since no information is needed for an initial state.



### **Final state**

The final state tool is used to place a final state "bulls-eye" symbol in a state diagram. There can be several final states in a diagram. Events can be drawn only to a final state from other states.

A final state does not have a dialog window since no information is needed for a final state.

## Use Case Diagrams

### Elements in a Use Case Diagram

In a Use Case Diagram page you can have the following kinds of elements:

- use cases
- actors
- communications
- packages
- extends, uses relations
- notes



### Use Case

The use case tool lets you create use cases. "A use case is a set of sequences of actions a system performs to yield an observable result of value to an actor."

#### The use case dialog window

In the use case dialog window there are two sections:

- Use Case overview (left pane)
- Selected item information (right pane)

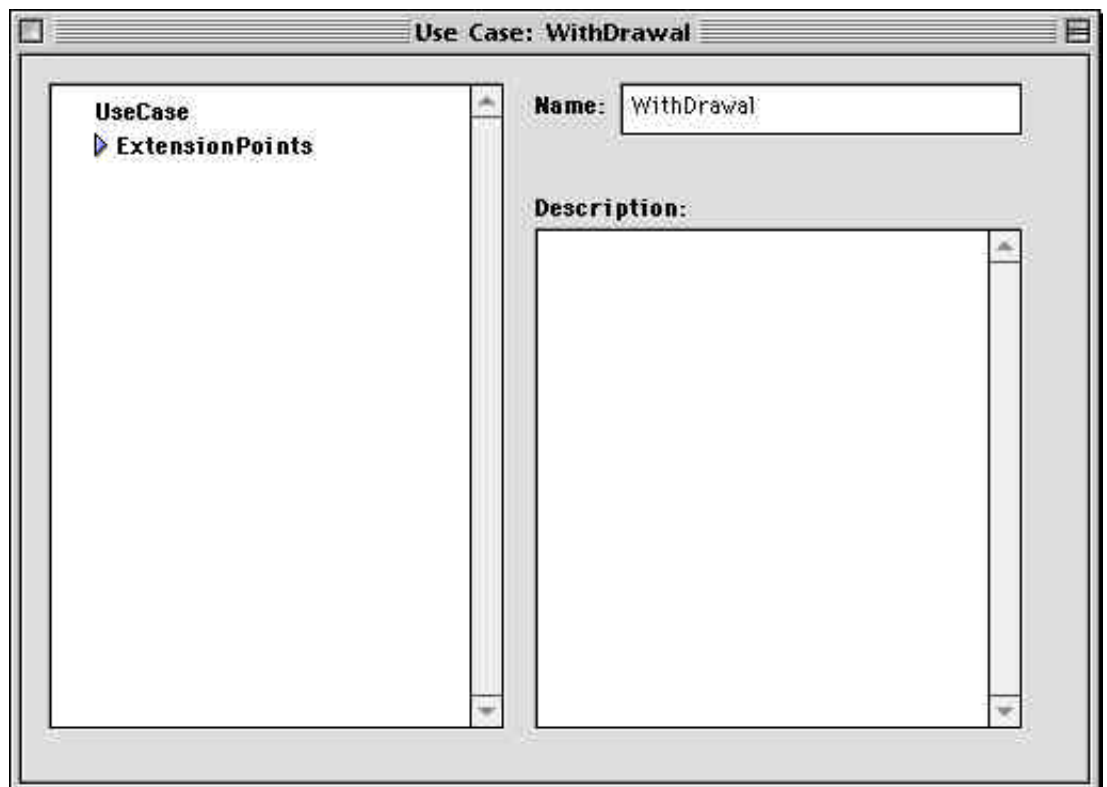


Figure 30. The use case dialog window

In the Name field, you can enter the name of the use case. Description contains a text field. ExtensionPoints can be added using the UseCase menu. The order of the ExtensionPoints can be rearranged with drag-n-drop.

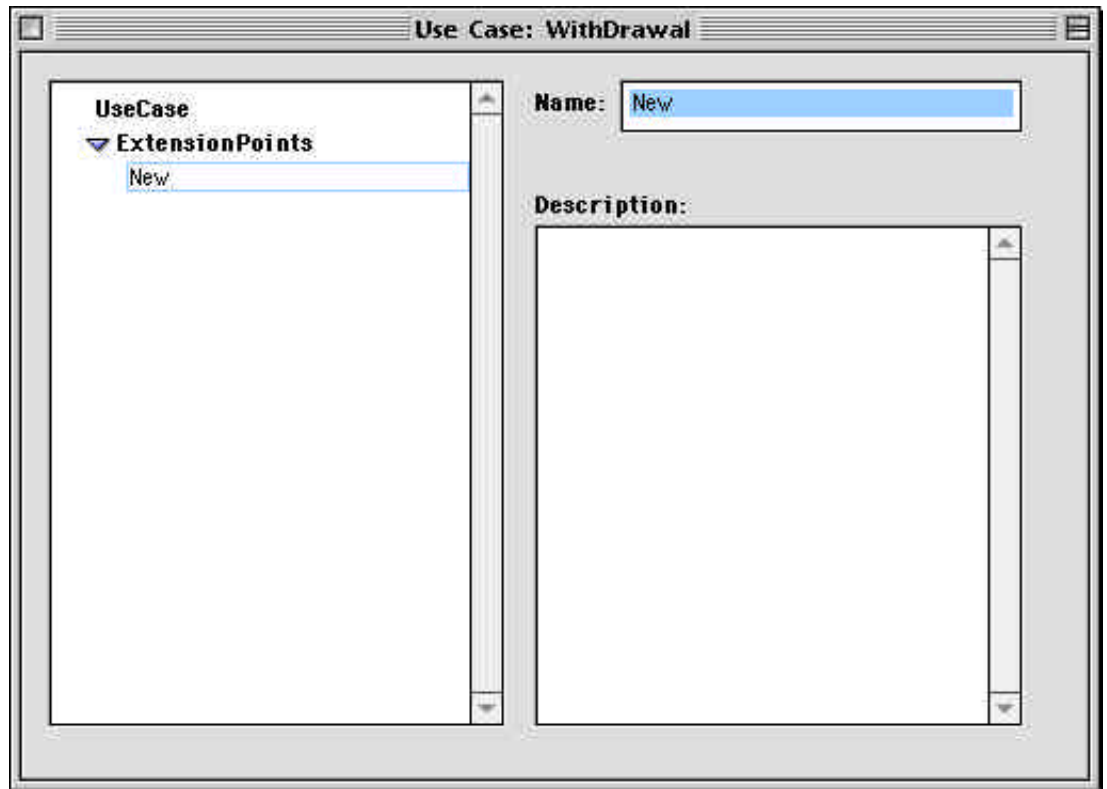


Figure 31. A UseCase dialog window when adding a new extensionpoint



## Actors

The actor tool is used to create actors. An actor is not necessarily a person but can also be other systems or equipment connected to the target system.

### The actor dialog window

In the actor dialog window there are two sections:

- Actor overview (left pane)
- Selected item information (right pane)

In the Name section, you can enter the name of the actor. Description contains an ordinary text field



## **Communicate association**

The communicate tool lets you create a communication association between an actor and a use case. A communication association shows the participation of an actor in a use case. The actor communicates with the use case.

Select the communicates tool and click and hold the mouse inside a use case or an actor. Drag the mouse to point at the other part (actor or use case) and then release the mouse button. A communication association is now drawn between the actor and the use case.

### **The communicates dialog window**

In the communicates dialog window there are two sections:

- Communicate overview (left pane)
- Selected item information (right pane)

In the Name section, you can enter the name of the communication association.

Description contains a text field.

In the current version of Object Plant, the name of a communication association is never shown.



## **Generalization, extends, uses**

The generalization tool lets you create an extends or use inheritance between two Use Cases or between two Actors.

## **Project Window**

Please note the difference between the name of the document and the name of pages within the document. The name of the document is shown in the title bar of the Project window and the name of pages is shown in the title bar of the four main windows (Object Model, State Diagram, Event Trace and Use Case).

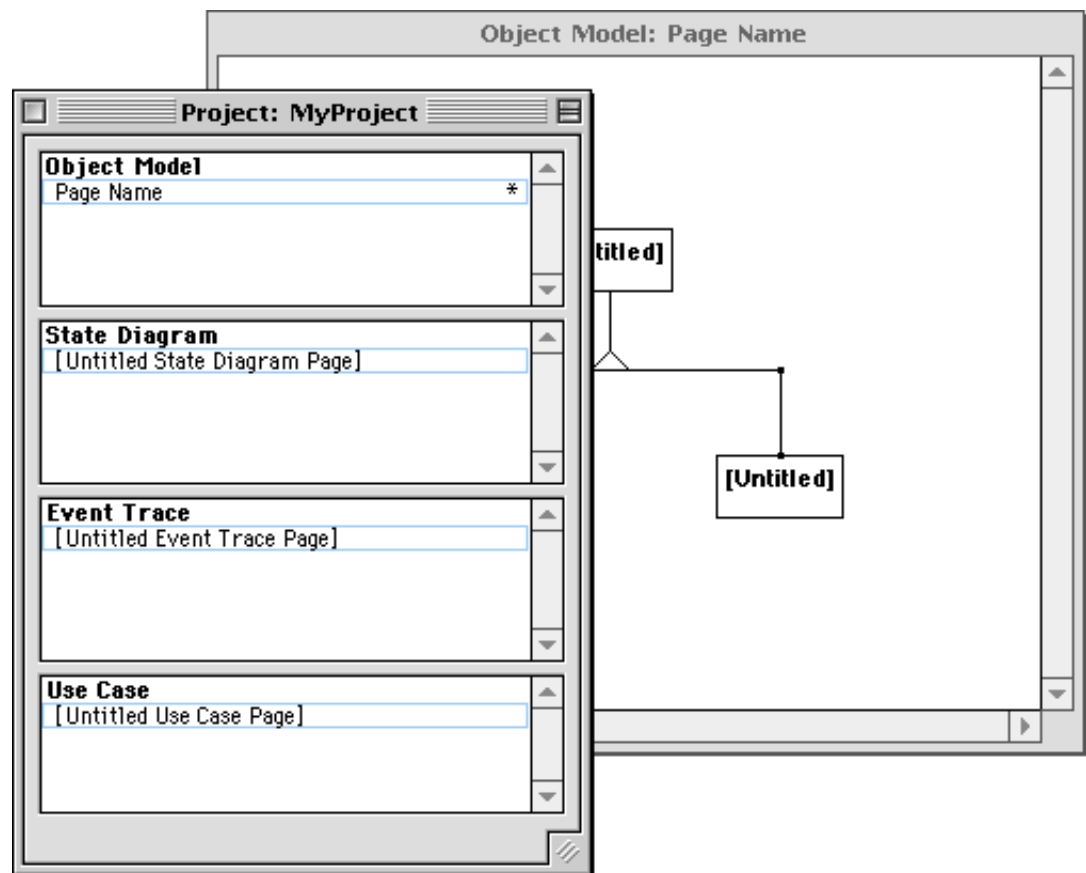
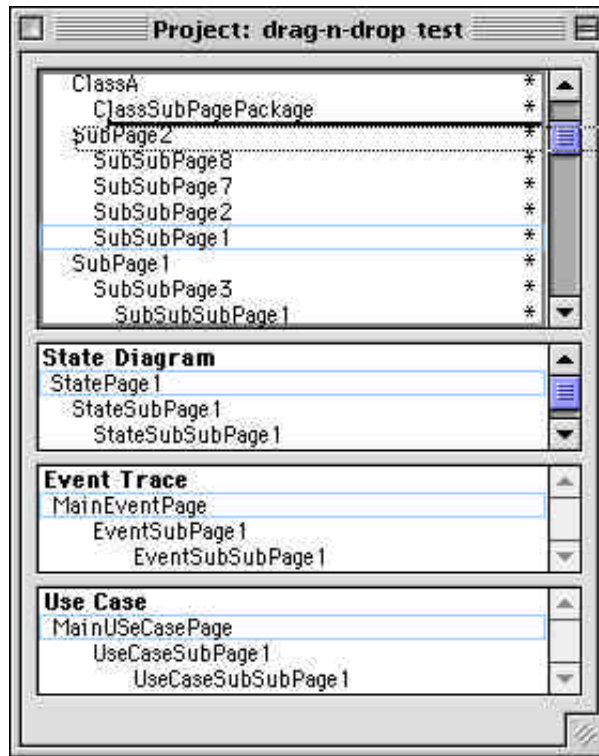


Figure 32. The project window and the object model window

## Drag-n-drop

The order and hierarchy of pages/packages can be changed in the Project window using drag-n-drop. When doing a drag-n-drop, the blinking caret showing the insertion point, does not only show the insert point in the vertical direction, i.e. the order of the pages, but it also indicates horizontally if the moved page/package will be inserted as a subpage to the previous page/package. In the picture below, SubSubPage1 is going to be inserted as a subpage ClassSubPagePackage. (The caret is indented relative the ClassSubPagePackage.)



**Figure 33. Drag-n-drop in the project window**

When selecting a page to drag, all the sub-levels to that page will also be dragged automatically. If you drag SubPage2 in the picture above, not only SubPage2 would be moved but also SubSubPage1,2,7,8.

In the picture below, SubSubPage1 will be placed in the ClassA page/package (the caret is indented relative the ClassA) and below the ClassSubPagePackage.



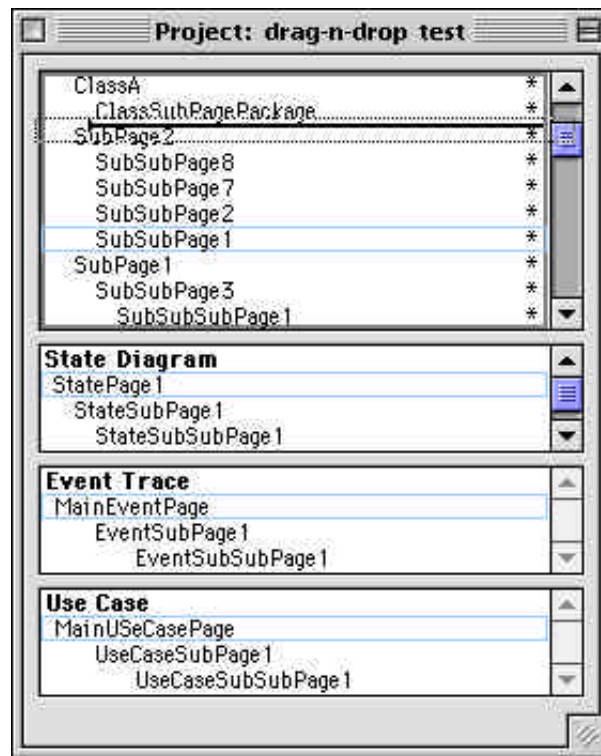
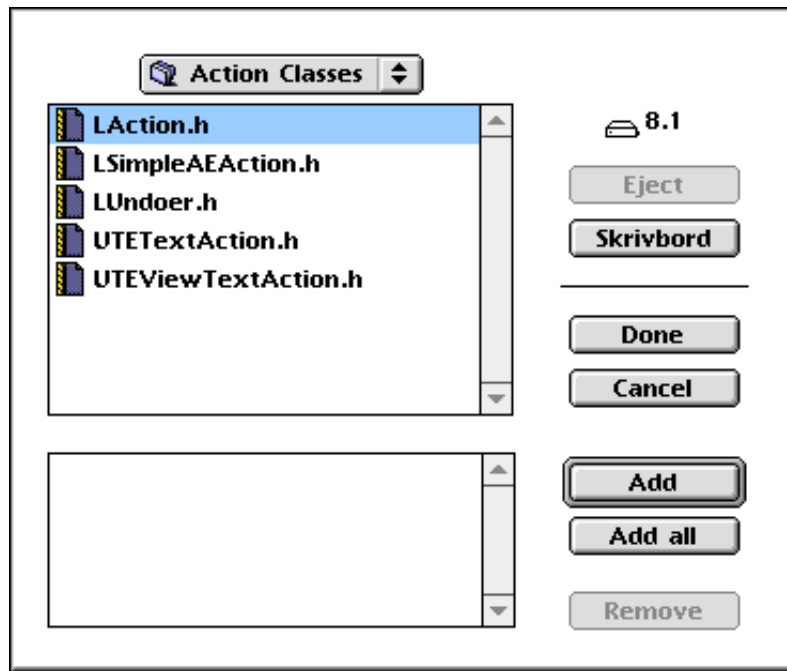


Figure 34. More drag-n-drop in the project window

## Importing Code (CodeRetriever)

CodeRetriever is the name of the Reverse Engineering feature of ObjectPlant. To use the CodeRetriever you should pay another 10 USD for a single user license.

With the CodeRetriever you can import C++ and Java code into your Object Model diagrams. In the File menu there is an item Import code... which will import code of the language selected in the Document preferences->General/Code dialog. The "Import code"-dialog will filter the available files so that only the relevant files are displayed. If C++ has been selected as the import language, only files with names ending in .h or .hpp will be displayed. If Java has been selected only files with names ending in .java are displayed.



**Figure 35. The "Import code" dialog**

If the code preference option "Create package for each folder" has been enabled and C++ files are imported, the created classes will be put in a newly created package(s) named as the folder the imported file is stored in. An example:

You select three files in the "Import code"-dialog, CObject.hpp (from a folder named General), CPersistentObject.hpp and CObjectStream.hpp from a folder named Persistence, two new packages will be created: General and Persistence. In the General package a new class will be created named CObject and in the Persistence package two new classes will be created; CPersistentObject and CObjectStream. If the packages already exists no new packages will be created but the imported classes will be created in the already available packages.

If you haven't enabled the "Create package for each folder"-option, all imported classes will be created in the currently selected ObjectModel page.

In the case of importing Java code, the "package" statement (if available) will be used to create the packages. An example:

You select two files in the "Import code"-dialog, Event.java (with a package statement: package package java.awt;) and ActionEvent.java (with a package statement: package java.awt.event;) three new packages will be created: java, awt and event. They will be created hierarchially so that awt is created inside the java package and event is created inside the awt package.

In the awt package a new class will be created named Event and in the event package a new class will be created named ActionEvent. If the packages already exist no new packages will be created but the imported classes will be created in the already available packages.

Since the Event class implements (inherits) an interface called java.io.Serializable, such an interface will be created in the same package as the Event class unless such an interface is already present in the ObjectModel in a package java/io. If the Serializable interface is already present a reference to that interface will be created in the awt package so that the supplier relation between Event and Serializable can be shown.

### C++ known bugs

- Due to the strange way function pointers are declared, these cannot be parsed in a simple way unless they have been typedefed, e.g.  
PersistentObject \*(\*functionPtr)( void );  
should be typedefed to  
typedef PersistentObject \*(\*Creator)( void );  
so that for example an attribute holding a function pointer is declared as  
Creator myAttr;
- There are a lot of more strange expressions in C++ that cannot be parsed.

### Java known bugs

- The old and deprecated way of declaring methods returning an array:  
ResultType Identifier( FormalParameterList ) []  
is **not** supported by CodeRetreiver.
- Inner classes and interfaces are ignored.

### CodeRetriever Options

In the File->Document Preferences->General/Code dialog window there is a possibility to add identifiers which the code parser will ignore. This can be useful when importing code containing macros, e.g. MacApp code:

```
class EXPORT_MA TCheckBox : public TCtrlMgr
// The TCheckBox class implements a check box using the Control
// Manager facilities of the Macintosh Toolbox.
{
    MA_DECLARE_CLASS;
```

The EXPORT\_MA is a macro that isn't useful importing into ObjectPlant (actually ObjectPlant will think that the class is named EXPORT\_MA) and the MA\_DECLARE\_CLASS macro is also not necessary for ObjectPlant. So if you're importing files which strange macros you should add them to the list of identifiers to be ignored.

## Code Generation

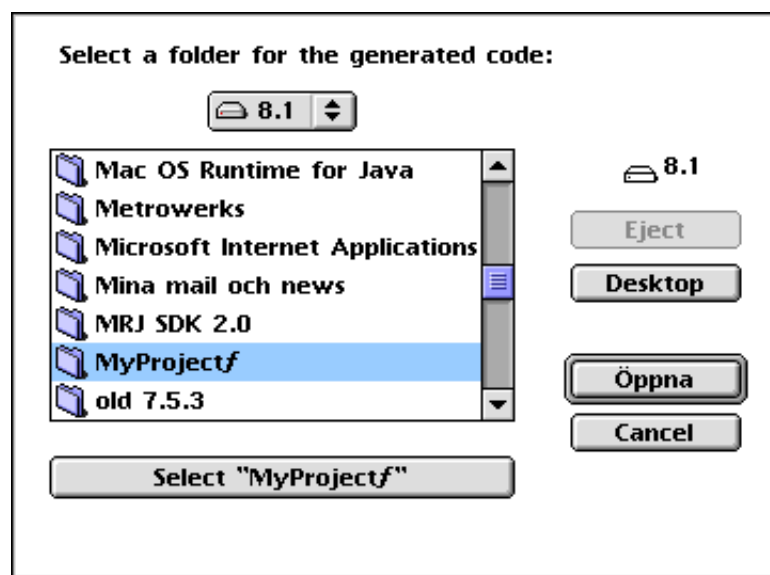
The Object Plant can generate C++ or Java code based on the information in the Object Model diagrams. If C++ is selected each class will generate two files, a source code file (.cp) and an include file (\*.h). The format of the generated files' format is based on two template files, C++template.cc and C++template.h. The name of the generated files will be "class name.cp" and "class name.h".

The Object Plant can also generate files based on the information about use cases stored in the Use Case diagrams. None of the provided code generation templates use this feature why you will have to write your own template if you need to generate documents based on the Use Case models.

The Java templates, provided by Jeff Moore, will generate one file for each class or interface. Two template files are used for Java generation a 'Class.java' file which generates code for classes and an 'Interface.java' file which generates code for interfaces.

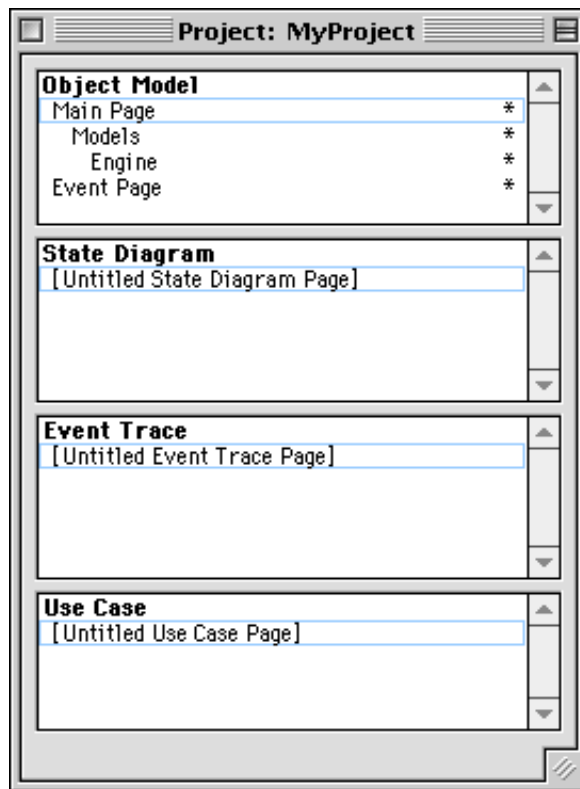
If the generated class name is longer than 31 characters, the name of the created files will be truncated. The first 25 characters of the file name will always be equal to the first 25 characters in the class name.

When generating code, by selecting the File->Generate code (all)... or File->Generate code (selected)...command, the following dialog appears where you are expected to select the folder where the generated files are to be stored.

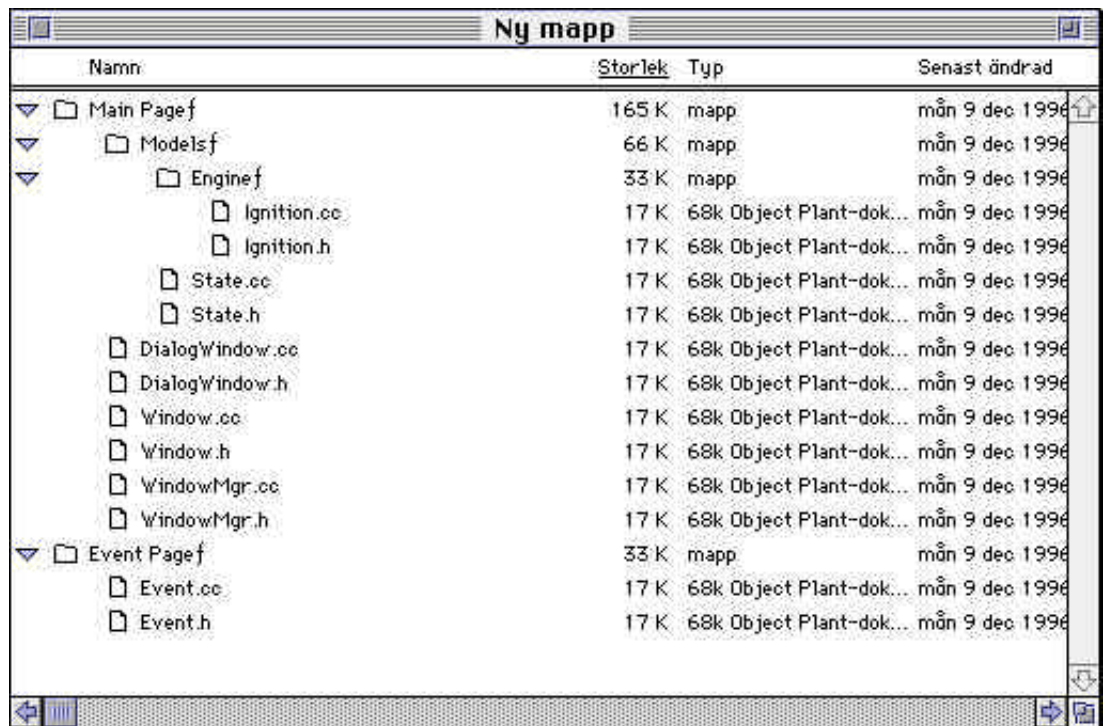


**Figure 36. The "generate code" dialog**

The generated files will be stored in the folder you select in the dialog. For each page included in your model a folder will be created within the folder selected in the dialog. Subsystem pages will create hierarchical folders. For example, if you have an Object Model with pages (and subsystems) looking like this in the page palette:



**Figure 37. The Object Model of a code generation example**  
the generated structure will look like this (if C++ code is generated):



**Figure 38. The generated folders and files**

If this doesn't suit your needs you will have to modify the template files.

## The template files

You can create an own set of template files if you like and use them by selecting your files in the "Preferences:Code generation" dialog.

The template files contains tags and plain text. Tags are always enclosed by curly braces, e.g. {CLASS}. Just like in HTML-code, some tags are only applicable in certain contexts, and they may have different meanings in different contexts.

Some tags have both a starttag and an endtag. The endtag looks like the starttag but the name starts with a '/' character, e.g. {CLASS}, {/CLASS}. Some tags don't have an endtag.

If you want to specify a string that shall be output you can just write the text without quotes unless the string contains any special character such as '"'. You can also enclose string with quote-characters (") if you encounter any problem with formatting of tabs etc.

The generated file will have the same creator type as the template file. The included template files all have 'ttx' (SimpleText) as creator. You can change the creator type of the template files using ResEdit or by copying the content of the template file into a file with the correct creator type. CodeWarrior, for example, has creator type 'CWIE'.

### A short note about the attribute tags

Within the attribute-tag {ATTRIBUTE}{/ATTRIBUTE} the following rules apply:

Consider the attribute field as consisting of four possible strings and a set of valid tags:

```
{ATTRIBUTE}start-string{NAME}string{TYPE}separator-string{END}end-  
string{/ATTRIBUTE}
```

where all strings are optional and can include other tags and the NAME and TYPE-tags can change places. The output from the code generator will be as follows (with the only exception that if there are no attributes, nothing will be output):

```
start-string ATTRIBUTE1_NAME string ATTRIBUTE1_TYPE separator-string  
ATTRIBUTE2_NAME string ATTRIBUTE2_TYPE separator-string  
ATTRIBUTE3_NAME string ATTRIBUTE3_TYPE end-string
```

Since the separator-string may span over several rows (see example below where "static" is a part of the separator-string but is related to the next attribute) there is a NEXT-tag that can be used to access information from the next attribute. If the NEXT-tag is not used, all information in the separator-string, e.g. {STATIC}, {DESCRIPTION} is taken from the current attribute. When the NEXT-tag has been parsed all references to information such as {STATIC} will be read from the next attribute.

**An example:**

**code generator template:**

```
{ATTRIBUTE}"// Attributes\r\r\t"
{STATIC}"static "{/STATIC}{TYPE}"\t\t"{NAME}";\r\t"{NEXT}
{STATIC}"static "{/STATIC}{END}";\r\r// End Attributes\r"{/ATTRIBUTE}
```

**output (with two attributes int x, static short y):**

```
// Attributes

    int          x;
    static short  y;

// End Attributes
```



## TAGS

At the top-level of a template file, the following tags are applicable:

### Top-level tags

<u>Tag</u>	<u>Meaning</u>
{DOCUMENTNAME}	Is replaced by the Object Plant document name.
{USERCODE}	Puts a start- and an end-mark in the generated code. Anything within these marks will remain when regenerating code.
{PROGRAMMER}	Is replaced by the text you entered in the registration dialog (also visible in the about dialog).
{TIMESTAMP}	Is replaced by the current time (in the format selected in the Date & Time control panel).
{DATESTAMP}	Is replaced by the current date (in the format selected in the Date & Time control panel).
{CLASS}	Text (and by other tags generated text) is written to the generated file for all classes in the Object Model. If no classes exist none of the text between the start and the endtag is written to the file. <b>Valid tags within the CLASS tag:</b> {DOCUMENTNAME}, {SUBSYSTEM}, {USERCODE}, {CLASSNAME}, {DESCRIPTION}, {FILENAME}, {PUBLIC}, {PROTECTED}, {PRIVATE}, {VIS_NONE}, {OPERATION}, {ATTRIBUTE}, {ABSTRACT}, {INTERFACELIST}, {ANCESTORLIST}, {DEEPANCESTORLIST}, {HEIRLIST}, {CONSISTOFLIST}, {PARTOFLIST}, {ASSOCLIST}, {PROGRAMMER}, {TIMESTAMP}, {DATESTAMP}, {DATECREATED}, {DATEMODIFIED}, {FILENAME1}, {FILENAME2}
{/CLASS}	
{INTERFACE}	Text (and by other tags generated text) is written to the generated file for all interfaces in the Object Model. If no interfaces exist none of the text between the start and the endtag is written to the file. <b>Valid tags within the INTERFACE tag:</b> {DOCUMENTNAME}, {SUBSYSTEM}, {USERCODE}, {INTERFACENAME}, {ANCESTORLIST}, {DEEPANCESTORLIST}, {HEIRLIST}, {DESCRIPTION}, {FILENAME}, {OPERATION}, {ATTRIBUTE}, {PROGRAMMER}, {TIMESTAMP}, {DATESTAMP}, {DATECREATED}, {DATEMODIFIED}, {FILENAME1}, {FILENAME2}
{/INTERFACE}	
{USECASE}	Text (and by other tags generated text) is written to the generated file for all use cases in the Use Case Model. If no use cases exist none of the text between the start and the endtag is written to the file. <b>Valid tags within the USECASE tag:</b> {DOCUMENTNAME}, {SUBSYSTEM}, {USERCODE}, {NAME}, {DESCRIPTION}, {FILENAME}, {PROGRAMMER}, {TIMESTAMP}, {DATESTAMP}
{/USECASE}	

{FILENAME} {/FILENAME}	<p>Text (and by other tags generated text) is used to create the generated file's name (and path).</p> <p>Valid tags within the FILENAME tag: {DOCUMENTNAME}, {SUBSYSTEM}, {CLASSNAME}, {ANCESTORLIST}, {INTERFACENAME}, {USERCODE}, {DATESTAMP}</p> <p>( {CLASSNAME} and {ANCESTORLIST} is valid only if {CLASS}{/CLASS} encloses {FILENAME}{/FILENAME} and {INTERFACENAME} is valid only if {INTERFACE}{/INTERFACE} encloses {FILENAME}{/FILENAME} )</p>
{SUBSYSTEM} {/SUBSYSTEM}	<p>This tag can be used to get information about nested Object Models. For example, in Java you can make the file belong to a certain package which is the actual Object Model page. The text between the start and the endtag is used as a separator , e.g.</p> <pre>package {SUBSYSTEM} . {/SUBSYSTEM};</pre> <p>makes all classed belong to a package named after the object model page name.</p>
{TRAILER} {/TRAILER}	<p>This tag can be used to get add information last in a generated file. This is particularly useful when generating code for several classes/ interfaces into a single document. The code generation will assure that the text within the TRAILER-tags will always be put last in the generated document.</p>

### **{CLASS}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{CLASSNAME}	Is replaced by the class' name, e.g. WindowMgr.
{DESCRIPTION}	Is replaced by the class' description.
{FILENAME1}	Is replaced by the FileName1 as specified in the class dialog window or if that field is empty by the class' name, e.g. WindowMgr.
{FILENAME2}	Is replaced by the FileName2 as specified in the class dialog window or if that field is empty by the class' name, e.g. WindowMgr.
{OPERATION} {/OPERATION}	<p>Text (and by other tags generated text) is written to the generated file for all operations of the class.</p> <p>If no operations exist none of the text between the start and the endtag is written to the file.</p> <p>Valid tags within the OPERATION tag: {USERCODE}, {NAME}, {DESCRIPTION}, {RETURNTYPE}, {CLASSNAME}, {PARAMETER}, {INPARAMETER}, {OUTPARAMETER}, {INOUTPARAMETER}, {OVERRIDE}, {ABSTRACT}, {STATIC}, {SIGNATURE}, {FINAL}, {IFNORETURNTYPE}, {ELSE}, {/IF}, {EXCEPTIONLIST}, {/EXCEPTIONLIST}</p>

{ATTRIBUTE} {/ATTRIBUTE}	<p>Text (and by other tags generated text) is written to the generated file for all attributes of the class.</p> <p>If no attributes exist none of the text between the start and the endtag is written to the file.</p> <p>Valid tags within the ATTRIBUTE tag: {NAME}, {DESCRIPTION}, {TYPE}, {CLASSNAME}, {OVERRIDE}, {STATIC}, {FINAL}, {INITIAL VALUE}, {END}, {NEXT}</p>
{ABSTRACT} {/ABSTRACT}	<p>Text (and by other tags generated text) is written to the generated file if the class is abstract.</p> <p>Valid tags within the ABSTRACT tag: {ANCESTORLIST}, {HEIRLIST}, {CONSISTOFLIST}, {PARTOFLIST}, {ASSOCLIST}</p>
{PUBLIC} {/PUBLIC}	<p>Text (and by other tags generated text) is written to the generated file for the enclosed operation and attribute tags if the corresponding operation/attribute has a "public" visibility.</p> <p>Valid tags within the PUBLIC tag: {OPERATION}, {ATTRIBUTE}</p>
{PROTECTED} {/PROTECTED}	<p>Text (and by other tags generated text) is written to the generated file for the enclosed operation and attribute tags if the corresponding operation/attribute has a "protected" visibility.</p> <p>Valid tags within the PROTECTED tag: {OPERATION}, {ATTRIBUTE}</p>
{PRIVATE} {/PRIVATE}	<p>Text (and by other tags generated text) is written to the generated file for the enclosed operation and attribute tags if the corresponding operation/attribute has a "private" visibility.</p> <p>Valid tags within the PRIVATE tag: {OPERATION}, {ATTRIBUTE}</p>
{VIS_NONE} {/VIS_NONE}	<p>Text (and by other tags generated text) is written to the generated file for the enclosed operation and attribute tags if the corresponding operation/attribute has a "none" visibility.</p> <p>Valid tags within the VIS_NONE tag: {OPERATION}, {ATTRIBUTE}</p>
{ANCESTORLIST} {/ANCESTORLIST}	<p>Text (and by other tags generated text) is written to the generated file for all immediate superclasses of the class. The ANCESTORNAME tag shall be used to get the ancestor's name. Any string before the ANCESTORNAME tag will be written once to the generated file. Any string after the ANCESTORNAME will be used as a separator if the class has several ancestors. The END tag can be used to specify text that shall follow the list of ancestors. The string written between the END tag and the /ANCESTORLIST tag will be output at the end of the ancestor list. If no ancestors exist none of the text between the start and the endtag is written to the file. C++ examples:</p>
{ANCESTORLIST} "#include \"\"{ANCESTORNAME}\".h\"" "#include \"\"{END}\".h\"\"{/ANCESTORLIST}	<p>This will create a set of include directives, one for each inherited class.</p>
class {CLASSNAME} {ANCESTORLIST}: public {ANCESTORNAME}, {/ANCESTORLIST}	<p>Valid tags within the ANCESTORLIST tag: {ANCESTORNAME}, {USERCODE}, {END}</p>

{DEEPANCESTORLIST} {/DEEPANCESTORLIST}	Text (and by other tags generated text) is written to the generated file for all superclasses of the class (and superclasses' superclasses a.s.o.). In all other respects it is compatible with the {ANCESTORLIST} tag.
{REVERSEDEEPANCESTORLIST} {/REVERSEDEEPANCESTORLIST}	Text (and by other tags generated text) is written to the generated file for all superclasses of the class (and superclasses' superclasses a.s.o.) but in the reverse order compared to the DEEPANCESTORLIST tag.
{HEIRLIST} {/HEIRLIST}	Text (and by other tags generated text) is written to the generated file for all subclasses of the class. The HEIRNAME tag shall be used to get the heir's name. Any string before the HEIRNAME tag will be written once to the generated file. Any string after the HEIRNAME will be used as a separator if the class has several heirs. The END tag can be used to specify text that shall follow the list of heirs. The string written between the END tag and the /HEIRLIST tag will be output at the end of the heir list. If no heirs exist none of the text between the start and the endtag is written to the file. Valid tags within the HEIRLIST tag: {HEIRNAME}, {USERCODE}, {END}
{CONSISTOFLIST} {/CONSISTOFLIST}	Text (and by other tags generated text) is written to the generated file for all aggregates of the class (i.e. other classes connected to the class with aggregations). The AGGREGATENAME tag shall be used to get the other class' name. Any string before the AGGREGATENAME tag will be written once to the generated file. Any string after the AGGREGATENAME will be used as a separator if the class has several aggregate classes. The END tag can be used to specify text that shall follow the list of aggregates. The string written between the END tag and the /CONSISTOFLIST tag will be output at the end of the aggregate list. If no aggregates exist none of the text between the start and the endtag is written to the file. Valid tags within the CONSISTOFLISTtag: {AGGREGATENAME }, {USERCODE}, {END}

{PARTOFLIST} {/PARTOFLIST}	<p>Text (and by other tags generated text) is written to the generated file for all class that the current class is an aggregate of (i.e. other classes connected to the class with aggregations). The <code>AGGREGATENAME</code> tag shall be used to get the other class' name. Any string before the <code>AGGREGATENAME</code> tag will be written once to the generated file. Any string after the <code>AGGREGATENAME</code> will be used as a separator if the class is a part of several other classes. The <code>END</code> tag can be used to specify text that shall follow the list of classes. The string written between the <code>END</code> tag and the <code>/PARTOFLIST</code> tag will be output at the end of the class list. If the class is not a part of any other class none of the text between the start and the endtag is written to the file.</p> <p>Valid tags within the <code>PARTOFLIST</code> tag: <code>{AGGREGATENAME}</code> , <code>{USERCODE}</code> , <code>{END}</code></p>
{ASSOCLIST} {/ASSOCLIST}	<p>Text (and by other tags generated text) is written to the generated file for all classes that the current class is associated with (i.e. other classes connected to this class with association). The <code>ASSOCCLASSNAME</code> tag shall be used to get the other class' name. Any string before the <code>ASSOCCLASSNAME</code> tag will be written once to the generated file. Any string after the <code>ASSOCCLASSNAME</code> will be used as a separator if the class is a associated with several other classes. The <code>END</code> tag can be used to specify text that shall follow the list of classes. The string written between the <code>END</code> tag and the <code>/ASSOCLIST</code> tag will be output at the end of the class list. If the class is not associated with any other class none of the text between the start and the endtag is written to the file.</p> <p>Valid tags within the <code>ASSOCLIST</code> tag: <code>{ASSOCCLASSNAME}</code> , <code>{ASSOCNAME}</code> , <code>{ASSOCDESC}</code> , <code>{BASEROLE}</code> , <code>{BRANCHROLE}</code> , <code>{BASEQUALIFIER}</code> , <code>{BRANCHQUALIFIER}</code> , <code>{USERCODE}</code> , <code>{END}</code></p>
{INTERFACELIST} {/INTERFACELIST}	<p>Text (and by other tags generated text) is written to the generated file for all interfaces that the class supplies. The <code>INTERFACENAME</code> tag shall be used to get the interfaces' name. Any string before the <code>INTERFACENAME</code> tag will be written once to the generated file. Any string after the <code>INTERFACENAME</code> will be used as a separator if several interfaces are supplied by the class. If no interfaces exist none of the text between the start and the endtag is written to the file. Java example:</p> <p><code>{INTERFACELIST}implements {INTERFACENAME}, {/INTERFACELIST}</code></p> <p>Valid tags within the <code>INTERFACELIST</code> tag: <code>{INTERFACENAME}</code> , <code>{DESCRIPTION}</code> , <code>{USERCODE}</code> , <code>{END}</code></p>
{DATECREATED}	Is replaced by the creation date of the class.
{DATEMODIFIED}	Is replaced by the date when the class was last modified.

### **{INTERFACE}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{ INTERFACENAME }	Is replaced by the interface's name, e.g. Storing.
{ FILENAME1 }	Is replaced by the FileName1 as specified in the interface dialog window or if that field is empty by the interface's name, e.g. Storing.
{ FILENAME2 }	Is replaced by the FileName2 as specified in the interface dialog window or if that field is empty by the interface's name, e.g. Storing.
{ DESCRIPTION }	Is replaced by the interface's description.
{ OPERATION }	Text (and by other tags generated text) is written to the generated file for all operations of the interface. If no operations exist none of the text between the start and the endtag is written to the file. Valid tags within the OPERATION tag: {USERCODE}, {NAME}, {RETURNTYPE}, {INTERFACENAME}, {PARAMETER}, {OVERRIDE}, {STATIC}, {SIGNATURE}, {FINAL}
{ /OPERATION }	
{ ATTRIBUTE }	Text (and by other tags generated text) is written to the generated file for all attributes of the interface. If no attributes exist none of the text between the start and the endtag is written to the file. Valid tags within the ATTRIBUTE tag: {USERCODE}, {NAME}, {TYPE}, {INTERFACENAME}, {OVERRIDE}, {STATIC}, {FINAL}, {INITIAL VALUE}, {END}
{ /ATTRIBUTE }	
{ ANCESTORLIST }	As described in the {CLASS} tag section.
{ /ANCESTORLIST }	
{ DEEPANCESTORLIST }	As described in the {CLASS} tag section.
{ /DEEPANCESTORLIST }	
{ HEIRLIST }	As described in the {CLASS} tag section.
{ /HEIRLIST }	

### **{USECASE}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{ NAME }	Is replaced by the use case's name, e.g. Withdrawal.
{ DESCRIPTION }	Is replaced by the use case's description.

### **{OPERATION}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{ RETURNTYPE }	Inserts the operation's return type, e.g. long.
{ NAME }	Inserts the operation's name, e.g. HideWindow.
{ DESCRIPTION }	Is replaced by the operation's description.

<code>{PARAMETER}</code> <code>{/PARAMETER}</code>	Text (and by other tags generated text) is written to the generated file for all parameters of the operation. If no parameters exist none of the text between the start and the endtag is written to the file. Valid tags within the <code>PARAMETER</code> tag: <code>{TYPE}</code> , <code>{NAME}</code>
<code>{INPARAMETER}</code> <code>{/INPARAMETER}</code>	Text (and by other tags generated text) is written to the generated file for all input parameters of the operation. If no input parameters exist none of the text between the start and the endtag is written to the file. Valid tags within the <code>INPARAMETER</code> tag: <code>{TYPE}</code> , <code>{NAME}</code>
<code>{OUTPARAMETER}</code> <code>{/OUTPARAMETER}</code>	Text (and by other tags generated text) is written to the generated file for all output parameters of the operation. If no output parameters exist none of the text between the start and the endtag is written to the file. Valid tags within the <code>OUTPARAMETER</code> tag: <code>{TYPE}</code> , <code>{NAME}</code>
<code>{INOUTPARAMETER}</code> <code>{/INOUTPARAMETER}</code>	Text (and by other tags generated text) is written to the generated file for all input/output parameters of the operation. If no input/output parameters exist none of the text between the start and the endtag is written to the file. Valid tags within the <code>INOUTPARAMETER</code> tag: <code>{TYPE}</code> , <code>{NAME}</code>
<code>{OVERRIDE}</code> <code>{/OVERRIDE}</code>	Text enclosed by these tags is written to the generated file if the operation has its "override" checkbox checked.
<code>{ABSTRACT}</code> <code>{/ABSTRACT}</code>	Text (and by other tags generated text) is written to the generated file if the operation is abstract.
<code>{!ABSTRACT}</code> <code>{/!ABSTRACT}</code>	Text (and by other tags generated text) is written to the generated file if the operation isn't abstract.
<code>{SIGNATURE}</code>	Inserts the operation's signature which is necessary for the regeneration of code to work properly.
<code>{STATIC}</code> <code>{/STATIC}</code>	Text enclosed by these tags is written to the generated file if the operation has its "static" checkbox checked.
<code>{FINAL}</code> <code>{/FINAL}</code>	Text enclosed by these tags is written to the generated file if the operation has its "Final" checkbox checked.
<code>{IFNORETURNTYPE}</code> <code>{ELSE}</code> <code>{/IF}</code>	Text (and by other tags generated text) between <code>{IFNORETURNTYPE}</code> and <code>{ELSE}</code> is written to the generated file if the operation has an empty return type. Text (and by other tags generated text) between <code>{ELSE}</code> and <code>{/IF}</code> is written to the generated file if the operation has a non-empty return type. The <code>{ELSE}</code> tag is optional. Valid tags within the these tag: <code>{RETURNTYPE}</code>
<code>{EXCEPTIONLIST}</code> <code>{/EXCEPTIONLIST}</code>	Text (and by other tags generated text) is written to the generated file for all exceptions of the operation. If no exceptions exist none of the text between the start and the endtag is written to the file. Valid tags within the <code>EXCEPTIONLIST</code> tag: <code>{EXCEPTIONNAME}</code>

**{PARAMETER}, {INPARAMETER},  
{OUTPARAMETER}, {INOUTPARAMETER}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{TYPE}	Inserts the type of the parameter, e.g. short.
{NAME}	Inserts the parameter's name, e.g. length. Any string after the first of either NAME or TYPE will be used as a separator if several parameters are used by the operation. If the operation has no parameters none of the text between the start and the endtag is written to the file.
{END}	Any string following this tag will be output at the end of all parameters and their associated output.

**{ATTRIBUTE}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{TYPE}	Inserts the type of the attribute, e.g. float.
{NAME}	Inserts the name of the attribute, e.g. Visible.
{DESCRIPTION}	Is replaced by the attribute's description.
{OVERRIDE}	Text enclosed by these tags is written to the generated file if the attribute has its "override" checkbox checked.
{/OVERRIDE}	
{FINAL}	Text enclosed by these tags is written to the generated file if the attribute has its "Final" checkbox checked.
{/FINAL}	
{TRANSIENT}	Text enclosed by these tags is written to the generated file if the attribute has its "Transient" checkbox checked.
{/TRANSIENT}	
{INITIAL VALUE}	Text enclosed by these tags is written to the generated file if the attribute has an "Initial value" specified. Valid tags within the INITIAL VALUE tag: {VALUE}
{/INITIAL VALUE}	
{END}	Any string following this tag will be output at the end of all attributes and their associated output.
{NEXT}	Can be used to step to the next attribute within the separator-string. Refer to "A short note about the attribute tags".
{STATIC}	Text enclosed by these tags is written to the generated file if the attribute has its "static" checkbox checked.  Valid tags within the STATIC tag: {NAME}, {TYPE}, {CLASSNAME}, {INTERFACENAME}
{/STATIC}	

**{INITIAL VALUE}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
{VALUE}	Inserts the value of the attribute initial value, e.g. 3.4.



### **{FILENAME}-level tags**

<b><u>Tag</u></b>	<b><u>Meaning</u></b>
<code>{SUBSYSTEM}</code> <code>{/SUBSYSTEM}</code>	<p>This tag can be used to get information about nested Object Models. For example, you can give the generated file a name which makes subfolders to be created for each nested diagram. The text between the start and the endtag is used as a separator between the folder levels, e.g.</p> <p><code>{FILENAME}{SUBSYSTEM}f:{/SUBSYSTEM}{CLASSNAME}.cc{/FILENAME}</code> puts all .cc files in own folders named after the object model page name. The folders will have a 'f' character at the end of the name. This is useful to avoid name conflicts between folders and files.</p>
<code>{FILENAME1}</code>	<p>Only allowed if the FILENAME appears inside a <code>{CLASS}{/CLASS}</code> or <code>{INTERFACE}{/INTERFACE}</code> pair of tags. Is replaced by the FileName1 as specified in the class/interface dialog window or if that field is empty by the class' or interface's name.</p>
<code>{FILENAME2}</code>	<p>Is replaced by the FileName2 as specified in the class/interface dialog window or if that field is empty by the class' or interface's name.</p>