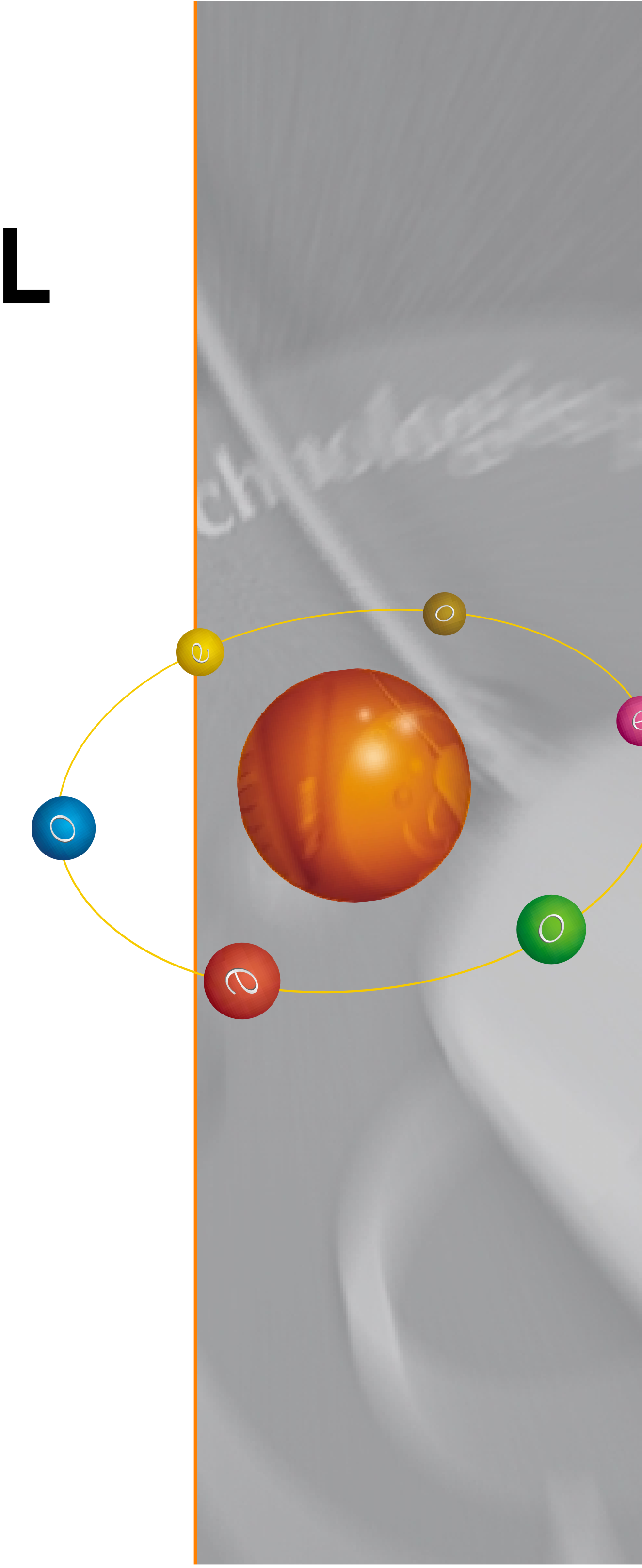
Objecteering/UML

Objecteering/J Libraries User Guide

Objecteering/Metamodel User Guide

Version 5.2.2



Objecteering

Software

www.objecteering.com

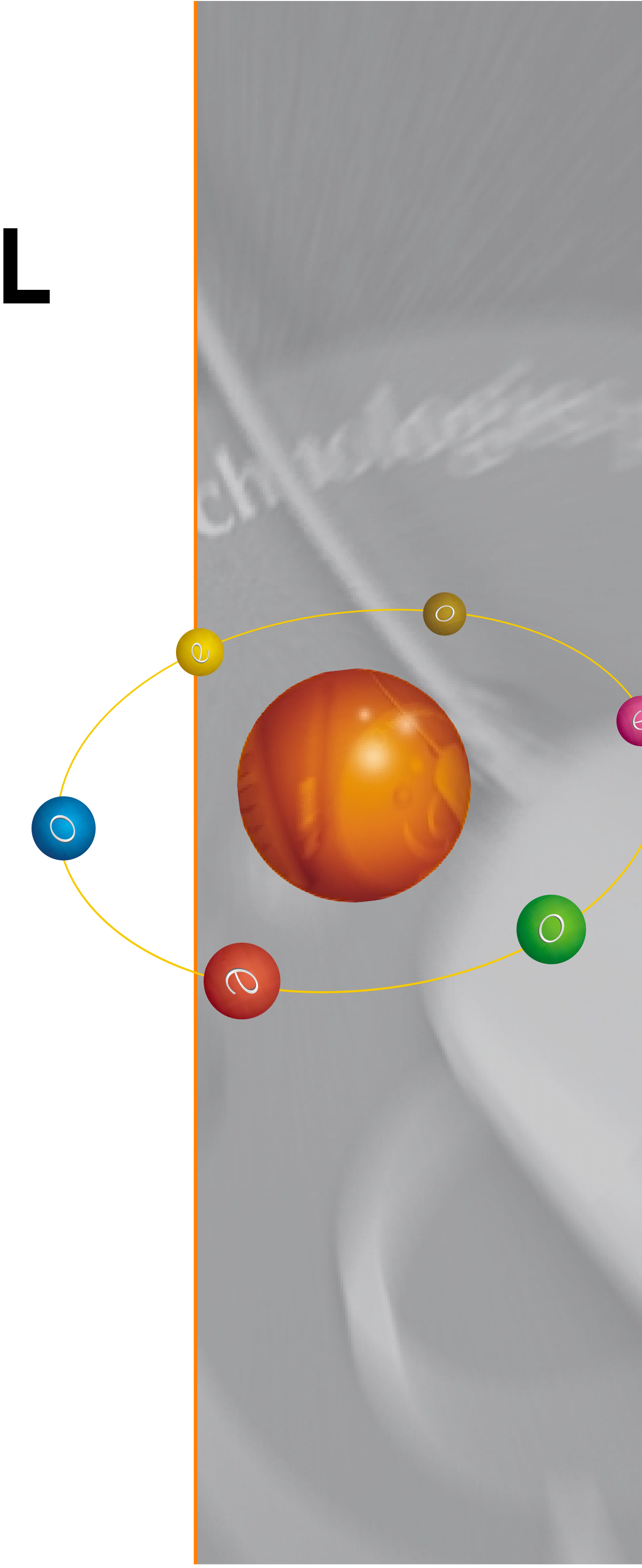
Taking object development one step further

Objecteering/UML version 5.2.2 - CODOBJ 001/001

Objecteering/UML

Objecteering/J Libraries User Guide

Version 5.2.2



Objecteering Software

www.objecteering.com

Taking object development one step further

Contents

Chapter 1: Presentation

Chapter 2: Modifying a UML model with J

Chapter 3: Diagrams and view elements

Chapter 4: System and execution environment services

Chapter 5: Module management facilities

Chapter 1: Presentation

Overview

Presentation

Welcome to the *Objecteering/J Libraries User Guide*!

J is a language dedicated to the management of UML modules in Objecteering/UML. This language is documented in:

- The *Objecteering/The J Language* user guide, which provides basic J mechanisms
- The *Objecteering/MetaModel User Guide*, which explains the different information accessible from J
- The *Objecteering/UML Profile Builder User Guide*, which explains how to enter, execute, test and package J rules

The *Objecteering/J Libraries User Guide* completes our extensive range of user guides and tools, thus providing a full set of utilities, which reinforces still further the power of J.

Examples

An example is provided in the form of the "*J_Examples*" UML profiling project. Commands which execute this example on the test project are defined in the "pop-up" menus. J methods are defined in the "default#external#J_Example" UML profile.

J library services

The Objecteering/UML J libraries provide a set of utility services which are used to:

- dynamically modify a model with the session mechanism
- create a simplified graphical user interface
- build a module, by realizing its own work products and its own initialization processing, or by using text edition with links on the model, etc.
- couple the operating system, by launching processing of files and directories, etc.
- create, modify and automatically position diagrams
- realize model exchanges in standardized ASCII form or specific to Objecteering/UML.

The ... library	is used to...
Module management services	execute J operations during the installation of a module.
System and environment	control system processes, and get information on the environment.
File management	manage the typical services which an Objecteering/UML system provides on files and directories.
Scan services	facilitate navigation of the metamodel.
Stream exchange services	provide model stream exchange in the XMI standard or in the "Externalization" format specific to Objecteering/UML.
Work product	customize the mechanisms predefined for work products in Objecteering/UML.
Generated files management and edition	provide the consistency mechanisms and edition facilities used to generate files.
J dialog box	provide elementary GUI facilities.
J session management	provide mechanisms which are used to modify a model with J.
Handling diagrams with J	create, change or lay out UML diagrams with J.
Properties editor	provide services to manage the properties editor

J service syntax conventions

J language service syntax

J services have the following syntax (Figure 1-1):

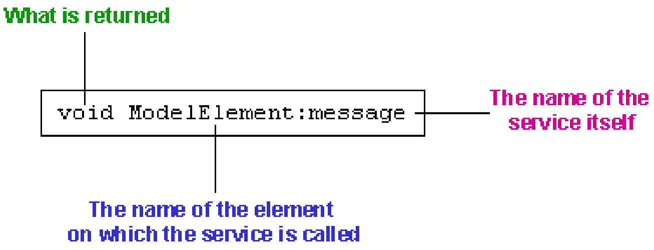

What is returned
void ModelElement:message
The name of the service itself
The name of the element on which the service is called

Figure 1-1. J service syntax

This indicates that the "*message*" service can be called on any "*ModelElement*" and will return a "*void*".

For example:

```
boolean Object:moduleInstall()
```

The "*moduleInstall*" service, which is called after a module is installed, can be called on any "*Object*" and will return a boolean.

Chapter 2: Modifying a UML model with J

Overview of model transformation

Model transformation

J can be used to access model values for all sorts of applications, and is at its most powerful when transforming a model. Starting with an initial model, it is used to create associations, operations, generalization, links, etc. The model is automatically transformed, just as if a designer, seeking to implement an analysis model, had manually carried out the transformation. J a particularly high performance tool for automating the building of design patterns.

"High level" primitives

It is important to note that any modification of the model must be carried out in a session (please see "*Managing the Session*" section in chapter 2 of this user guide for further information).

For most of the metaclasses, metaclass creation or modification services (please see the "*Constructor*" rubric for each metaclass) are provided.

High level primitives use information from the metamodel, to create all the necessary and easily deducible links automatically. We strongly recommend using them instead of "low level" primitives.

The advantages of high level primitives are as follows:

- easier creation of elements
- upward compatibility is guaranteed. Future developments linked to our model will be supported by these primitives, something which is not guaranteed for low level primitives.

For example, each element must be attached to a given set of "parent" elements. If there are omissions, there will be rejects at the end of a session. All classes must, for example, be attached to a package or a class. This must be remembered during "low level" creation of a class.

"Low level" primitives

In general, J has a model element creation mechanism called "low level". It is implemented simply by using the names of classes, roles, attributes, etc. These primitives are called "low level", since all necessary links must be defined, and all necessary intermediary elements created. We recommend that wherever possible "high level" primitives be used. Furthermore, "high level" primitives are stable, regardless of whether or not the metamodel evolves in future versions.

Example of usage

In the example below, a "*print*" method will be created on the current class. Low level primitives are used here, but more convenient high level primitives should be used instead.

Note: The "*add a "print" operation*" command on a class is used to execute this program on classes.

```
Class:addPrint()
{
Operation M ;
sessionBegin  ("addPrintOperationExample", true) ;
  // beginning of the "persistent" session
M = Operation.new ();
  // creation of a method
M.setName ("print") ;
  // Assignment of the Name attribute
this.appendPart(M) ;
  // the current class is attached to the method
sessionEnd ();
  // end of the session,
  // the current class has an attached method
  //after execution of Objecteering/UML's consistency
  //check
}
```

Managing the session

Overview

Model transformation is carried out within a session which has the following characteristics:

- It can be interrupted during processing ("*sessionAbort*").
- It must end with a consistency check, since any transformation must be consistent, otherwise it is refused. If removable consistency checks have been deactivated, the check will not be applied.
- Sessions must not be embedded.
- The final user can do "*undo*" for a session.
- It is the final user who makes a definite backup of a session, except when the transformation mode is performed with *objingcl*. In this case, if the consistency check is sound, a back-up is made.

Starting a session

```
sessionBegin (in String sessionName, in boolean
isPersistant);
```

This starts a session. The name of the session (chosen by the programmer) is fixed as a parameter. The "*isPersistent*" parameter determines whether the user model should be modified (*isPersistent* = *true*) or whether transformation should be ignored at the end of a session.

Ending a session

```
sessionAbort ();
```

This cancels the current session. All modifications are ignored.

```
sessionEnd ();
```

This ends the current session. If the session is persistent, then consistency checks are run. If the model is consistent, Objecteering/UML informs each of the updated "*clients*" (graphic editors, explorers). If the session is transient, it ends with no impact on the user model and the editors. This is not the case for removable consistency checks.

Note: any ending of a J program that does not use these primitives (end of the program, or "*exit*" primitive) corresponds to an "*Abort*".

State of a session

The following methods give the information related to the current session.

```
boolean sessionHasBegun()
```

This determines whether a session is in progress.

```
String get_sessionName()
```

This returns the name of the current session, or "" (empty) if there is no active session.

Managing consistency

If removable consistency checks are active, the consistency check function is activated automatically at the end of a persistent session. It can be launched by a program using the following method:

```
boolean sessionCheck()
```

which returns true if the model is consistent.

Model transformation primitives

Overview

Model transformation primitives are used to:

- create a metaclass instance
- change attribute values
- add a link
- destroy an element
- delete a link
- sort part of the model (for example, the attributes of a class)

It is possible, for example, to create a class, name it and add methods with parameters, using *Objecteering/UML Profile Builder*.

Low level primitives

The primitive ...	is used to ...	example ...

new	create an element	... creation of a class (*)
		Class C;
		C = Class.new;

set<AttributeName>	assign the value of an attribute.	C.setName("client");
		C.setVisibility(Public);

append<RoleName>	create a link in the metamodel	M :Operation ;... C.appendPart(M); ...add an operation to a class
delete	destroy an element. Its links are updated and its components are destroyed	C.delete; ...destruction of C and its components (therefore M)
erase<RoleName>	destroy a link	C.erasePart(M); ...deletes the link between the C class and the M operation (note : M is not destroyed)
sortSemanticAssociation (in String pSortedRoleName, in boolean pAscendingSort)	sort elements accessible from the current object, through the pSortedRoleName role. Elements are sorted according to their names. Elements with no name are not handled. This primitive requires a persistent J session.	currentPackage.sortSemantic Association ("OwnedNameSpace", true) ...sorts the accessible objects by name through the "OwnedNameSpace" role from the "currentPackage" package.

(*): it is necessary to attach this class to a UML modeling project, so that the creation be definite.

Example

We are going to create accessors with "*set<AttributeName>"* and "*get<AttributeName>*" for each class attribute.

Note: The "*addAccessor*" command on a class is used to execute the program below.

```
Class#addAccessor
{
   Operation M;
   Parameter P;
   Class C;
   sessionBegin ("addAccessor", true);

   C = this;
   PartAttribute
   {
       M = Operation.new;
       M.setName("set_" + Name);
       C.appendPart(M);
       P=Parameter.new;
       P.setName("AttValue");
       M.appendIO(P);
       M = Operation.new;
       M.setName("get_" + Name);
       C.appendPart(M);
       P = Parameter.new;
       P.setName("R");
       M.appendReturn(P);
   }
sessionEnd();
```

Chapter 3: Diagrams and view elements

Overview of diagrams and view elements

Presentation

Objecteering/UML grants access to the *diagram* and *ViewElement* metamodel, thus allowing you to handle diagrams and layout using the *J language*. Therefore, it is possible to automatically layout, animate and create diagrams with J, and to provide any kind of automated service related to *UML diagrams*.

This chapter will describe how to handle *diagrams* and *ViewElements*, by describing the principle, providing examples, and describing the *metamodel*.

Metamodel synthesis

A *Diagram* is made up of *ViewElements* that can be either Boxes (*ViewBox*) or *Links* (*ViewLink*). *Boxes* and *Links* are defined with one or several geometrical *Points*, and have various graphical resources that can be managed through J. A *ViewElement* is generally a representation of an *Element*, defined in the UML metamodel. Most elements can be represented by an unlimited number of *ViewElements*, depending on presentation needs. For example, a *Package* can have a related Class diagram, that will contain several *ViewBoxes* representing classes that belong to the *Package*, and *ViewLinks* representing dependencies, *Associations*, *Generalizations*, etc.

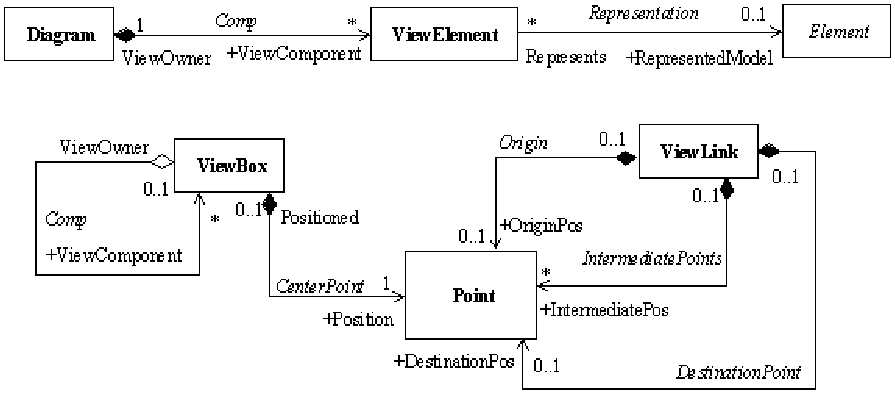


Figure 3-1. Diagram metamodel

Several *Elements* can have associated *Diagrams*, that will contain a set of *ViewElements*. This is typically the case of *NameSpaces*.

Diagrams and view elements model metaclasses

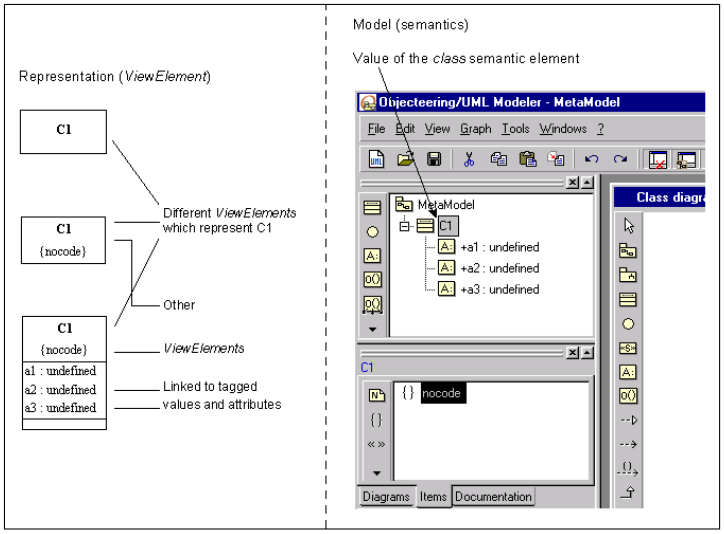
- *Diagram*: Graphic representation of a model.
- *Point*: Graphic point in a diagram.
- *ViewBox*: Graphic Boxes.
- *ViewElement*: Graphic representation of a *ModelElement*.
- *ViewLink*: Graphic link.

Handling diagrams and view elements with J: Principles

Model and representation

Objecteering/UML, in accordance with the UML standard, separates the model (its semantics) from its representation. The set of metaclasses presented up until now corresponded to the model (*Package*, *Class*, *Operation*, etc.). Representation allows the construction of graphical views associated to the model.

The *ViewElement* metaclass is a representation of a model element. A model element can be associated with several *ViewElements*. For example, a *Class* can be represented in several different *Diagrams* in several forms (with its attributes) which present its tagged values, in color, etc.). A *Class* will, therefore, have several *ViewElements* which represent it within several *Diagrams* (Figure 3-2).



Representation (*ViewElement*)
C1
C1
{nocode}
C1
{nocode}
a1 : undefined
a2 : undefined
a3 : undefined
Different *ViewElements* which represent C1
Other
ViewElements
Linked to tagged values and attributes
Model (semantics)
Value of the *class* semantic element
Objecteering/UML Modeler - MetaModel
File Edit View Graph Tools Windows ?
MetaModel
C1
+a1 : undefined
+a2 : undefined
+a3 : undefined
C1
{} nocode
Diagrams
Items
Documentation
Class diagr

Figure 3-2. Different representations (*ViewElement*) of a class (Model)

In UML, this distinction reflects the separation of *ModelElement* and *ViewElement*.

Graphic management

The graphic model is very general, and only presents the *ViewBox* and *ViewLink* generic classes.

The order of creation for a graphic element in J must always indicate the associated semantic element. Objecteering/UML, knowing the context of the diagram and of the presented model element, will then create the correct graphic form. For example, the introduction:

```
aBox = adiagram.createAddAndMoveViewBox
       (aClass1, 300,30,70,75);
```

will create in the "*aDiagram*" diagram a representation of the "*aClass1*" class, in the attached coordinates (see the "*Point*" class for the description of the system of coordinated graphics).

The three important notions to be dealt with are, therefore, the Diagram, the model element (Element) and the representation of the elements (*ViewElement*).

When a *ViewElement* is created, it is possible to modify its graphic resources in J (color, width, coordinates), by modifying the attributes provided by the *ViewElement*, *Point*, *ViewBox* and *ViewLink* classes.

For example, the instruction below will present the "*aBox*" box, with a red contour and a green background :

```
aBox.bgGreen=255;
aBox.fgRed=255;
```

First steps

Overview

We are now going to create a J method for a *Package*, which will automatically create a class diagram showing all the package's classes with their links (generalizations, associations etc).

Diagram handling services are used to create graphic elements. The layout service is then called, thus avoiding tedious calculation of coordinates in J.

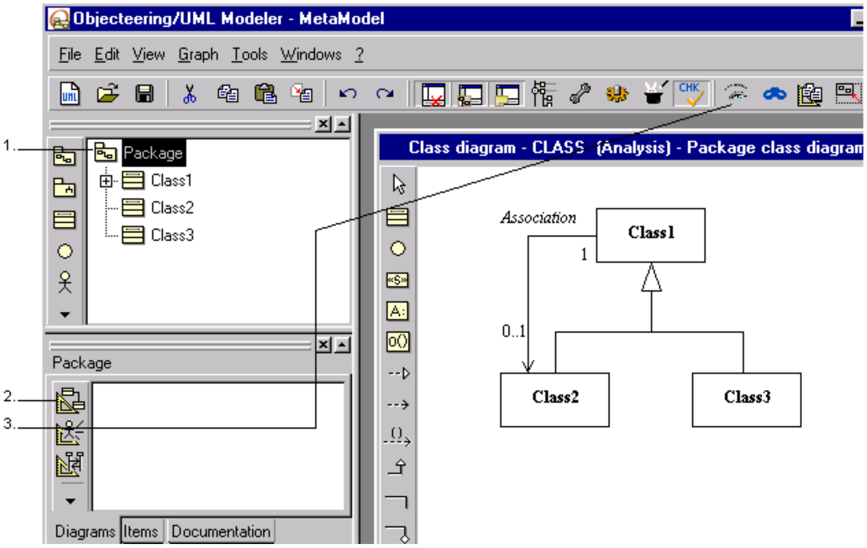

Objecteering/UML Modeler - MetaModel
File Edit View Graph Tools Windows ?
1.
2.
3.
Package
Class1
Class2
Class3
Package
Diagrams
Items
Documentation
Class diagram - CLASS (Analysis) - Package class diagram
Association
Class1
1
0..1
Class2
Class3

Figure 3-3. Diagram created automatically

Steps:

1 - Select the package in the explorer.

2 - Click on the "*Create a class diagram*" icon in the "*Items*" tab of the properties editor. The class diagram is then automatically opened.

3 - Click on the "*Show contents*" button. The contents of the package (the *Class1*, *Class2*, *Class3* classes) then appear.

To make the links (*associations*, *generalizations*, etc) visible, select the classes using the right-mouse button, and choose the "*Show links*" option from the context menu which appears.

Entering and executing

Edit the "*MetaExample*" example UML profiling project.

In the model explorer, run the "*create Class Diagram*" command on a package which owns classes with some links (associations, etc). You will then see the diagram appear.

The "*createClassDiagram*" J method can be modified to adapt graphical behavior (for example by displaying classes in red).

```
Package:createClassDiagram()
{
  StaticClassDiagram aDiagram;
  sessionBegin("Generate diagram",true);

  // create the diagram :
  aDiagram = StaticClassDiagram.new ;
  aDiagram.setName("generatedClassDiagram") ;
   appendproduct(aDiagram) ;

  // show package's components in the diagram :
  // open a graphic session
  aDiagram.open() ;

  // show classes and package :
  OwnedElementNameSpace
  {

    aDiagram.createAndAddViewBox(this) ;
    // show generalizations of the current class
       package:
    ParentGeneralization
    {
      aDiagram.createAndAddViewLink(this) ;
    }

    // show uses :
    DestinationUse
    {
      aDiagram.createAndAddViewLink(this) ;
    }
    // show associations :
    if (ClassOf.Name=="Class")
    PartAssociationEnd
    {
      aDiagram.createAndAddViewLink(this) ;
    }
  }

  // layout :
  aDiagram.layoutViewElements
(aDiagram.ViewComponentViewBox,
  aDiagram.ViewComponentViewLink) ;

  // close the graphic session :
  aDiagram.close() ;
  sessionEnd () ;
}
```

"Diagram" class

Diagram overview

```
class Diagram extends InternalProduct;
```

Graphical representation of a model. A *Diagram* is a kind of product that is attached to a *ModelElement*. It contains *ViewElements* that constitute the representation of a model. For example, a *Package* can be attached to *Class Diagrams*, that will be made up of *ViewElements* representing some of the elements of the *Package*, their *Links*, properties, etc.

The subclasses of *Diagram* represent each kind of diagram specified by UML and supported by Objecteering/UML.

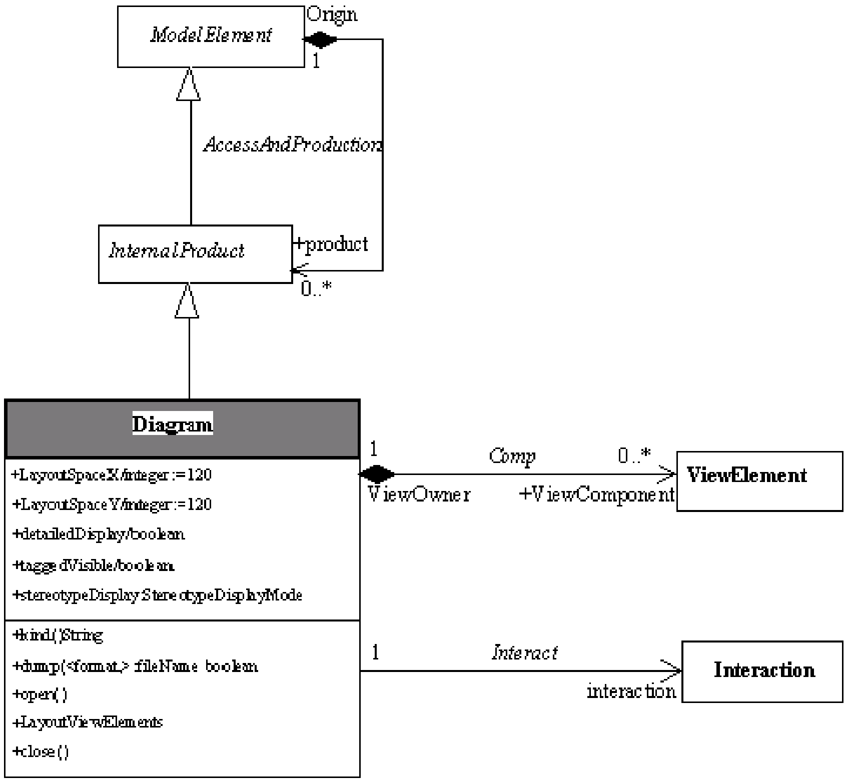


Figure 3-4. Detailed class diagram : *Diagram*

<u>See also</u>: *ModelElement*, *ViewBox*, *ViewLink*.

Diagram properties

The *Diagram* class has the following associations:

- *ViewComponent:ViewElement*: Link to the graphical elements that constitute the *Diagram*. This link is only valid after the *open* operation has been called.
- *interaction:Interaction*: Link allowing diagrams associated with an interaction to be retrieved. This link is only valid for sequence diagrams.

It has the following attributes:

- *LayoutSpaceX*: *integer*
- *LayoutSpaceY*: *integer*

X and Y spacing between boxes is taken into account, particularly for automatic positioning.

- *detailedDisplay*: *boolean*. The *detailedDisplay* attribute is used to indicate the level of detail shown in the diagram.
- *taggedVisible: boolean*. The *taggedVisible* attribute indicates whether or not tagged values are visible when elements are created (default values depend on general parameterization, which is carried out through the *Formalism* section of *UML Modeler* parameterization).
- *stereotypeDisplay*: This can use one of the following values:
    - NoneStereotypeDisplayMode: Stereotypes are not displayed.
    - IconStereotypeDisplayMode: Stereotypes are displayed using an icon.
    - LabelStereotypeDisplayMode: Stereotypes are displayed using a label.
- *showSystemBoundary*: *boolean*. The *showSystemBoundary* attribute indicates whether or not the boundaries of use cases (in other words, their oval encirclement) in use case diagrams are to be visible.

Diagram consistency rules

Not every type of diagram can be created for every type of element. The following table resumes what it is possible to create on different elements.

Model element ...	possible associated diagram ...
Package, Sub-system, Class	Class Diagram
Package, Sub-system	Use Case Diagram
Package, Sub-system, Class, Collaboration, UseCase	Sequence Diagram
Package, Sub-system, Class	Object Diagram
Collaboration	Collaboration Diagram
ActivityGraph	Activity Diagram
StateMachine	State Diagram
Package, Sub-system	Deployment Diagram
Package, Sub-system	Deployment Instance Diagram

Diagram constructors

```
Diagram ModelElement:createUseCaseDiagram
                     (in String pName)
```

This operation creates a *UseCaseDiagram* whose name is given.

```
Diagram ModelElement:createAndAddUseCaseDiagram (in
String pName)
```

This operation creates and adds to the current element a *UseCaseDiagram* whose name is given.

Example:

```
UseCaseDiagram MyDiag
=MyPackage.<createAndAddUseCaseDiagram
           ("MyUseCaseDiag");
```

```
Diagram ModelElement:createClassDiagram
                     (in String pName)
```

This operation creates a *ClassDiagram* whose name is given.

Example:

```
ClassDiagram MyDiagram =createClassDiagram
                        ("MyDiagram");
```

```
ModelElement:addDiagram (in Diagram pDiagram)
```

This operation adds a *Diagram* to the current *Element*.

Example:

```
MyPackage.<addDiagram (MyDiagram);
```

```
Diagram ModelElement:createAndAddClassDiagram
                     (in String pName)
```

This operation creates and adds to the current element a *ClassDiagram* whose name is given.

Example:

```
ClassDiagram MyDiag MyPackage.<createAndAddClassDiagram
                               ("MyClassDiag");
```

```
ModelElement:createAndAddObjectDiagram (String Name);
```

```
ModelElement:createAndAddCollaborationDiagram
             (String Name);
```

```
ModelElement:createAndAddSequenceDiagram (String Name);
```

```
ModelElement:createAndAddDeploymentDiagram
             (String Name);
```

```
ModelElement:createAndAddInstanceDeploymentDiagram
             (String Name);
```

```
Operation:createAndAddStateDiagram (String Name);
NameSpace:createAndAddStateDiagram (String Name);
```

```
Operation:createAndAddActivityDiagram (String Name);
NameSpace:createAndAddActivityDiagram (String Name);
```

These methods create a *Diagram* related to *ModelElement*, that will have the Name *Name*, and the *kind* related to the name of the method. This should be carried out only for authorized *ModelElements* (see table below).

Every *ModelElement* that can have diagrams (*Class*, *Package*, *UseCase*, *StateMachine*, *Collaboration*, and so on) provides the following service:

```
private Diagram[] getDiagrams();
```

This is the most thorough way of getting existing diagrams.

Diagram methods

`String kind()`

Returns the kind of diagram which can be: *Activity, Class*, *Collaboration*, *ComponentDeployment*, *InstanceDeployment*, *Object*, *Sequence*, *State*, *UseCase*.

`boolean dump (in String format, inout String fileName)`

Generates a diagram in a file. The format can be: "*gif*", "*ps*", or "*emf*". "*gif*" is the correct format for HTML, "*emf*" is the windows metafile format, and "*ps*" is PostScript.

`Object:setDiagramDumpPath (in String path)`

Sets the directory used by the documentation generation for the diagrams (dump()).

`open();`

This is an essential operation, used to handle the diagram's *ViewElements*.

`close();`

This closes a diagram.

```
layoutViewElements (inoutViewBox [] boxes,
                    inoutViewLink [] links);
```

A method which ensures the automatic positioning of links and boxes in a diagram through parameterization.

Example：

```
layoutViewElements (myDiagram .<ViewComponentViewBox,
                    myDiagram .<ViewComponentViewLink);
```

This command guarantees the automatic positioning of the entire "MyDiagram" diagram.

A diagram must always be open for its representation elements to be modified.

When the modification session is complete, the *close* instruction allows the modifications to be taken into account, and the diagram to be refreshed.

Note: An active editor is only taken into account once the execution of the J program has been completed.

"Point" class

Point overview

```
class Point;
```

Graphical *Point* in a diagram. A *Point* is defined through its x and y coordinates.

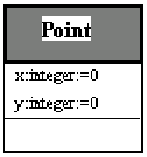


Figure 3-5. Detailed class diagram: *Point*

See also:*Diagram*, *ViewElement* .

Point properties

The *Point* class has the following attributes:

- x: Value of the x coordinate of the point.
- y: Value of the y coordinate of the point.

Point coordinate system

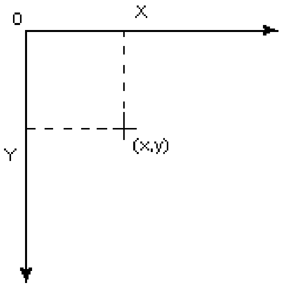


Figure 3-6. Coordinate system in a *Diagram*

Objecteering/UML uses a classic system of coordinates: the origin is in the top left corner of the diagram, and the coordinate unit is the pixel.

Note: Boxes can overlap other boxes and links (for example, packages which overlap classes and packages, composed states, etc). In this case, the origin of an overlapped element is the top left corner of the embedding box.

Point constructor

```
ViewLink:addPoint (in integer x, in integer y);
```

Adds an intermediary point to a link. We then have a "*broken*" link.

The points are generally implicitly created by *ViewElements*. They may be added explicitly on links.

"ViewBox" class

ViewBox overview

```
class ViewBox extends ViewElement;
```

Graphical Boxes. A *ViewBox* represents nodes in a graphic representation. For example, a *Class* is represented by a box. A box is graphically characterized by its center *Point*, its width and its height.

ViewBox can also be embedded. For example, an *Attribute* in a *Class* is represented by a *ViewBox* representing the *Attribute* in a *ViewBox* which represents the class.

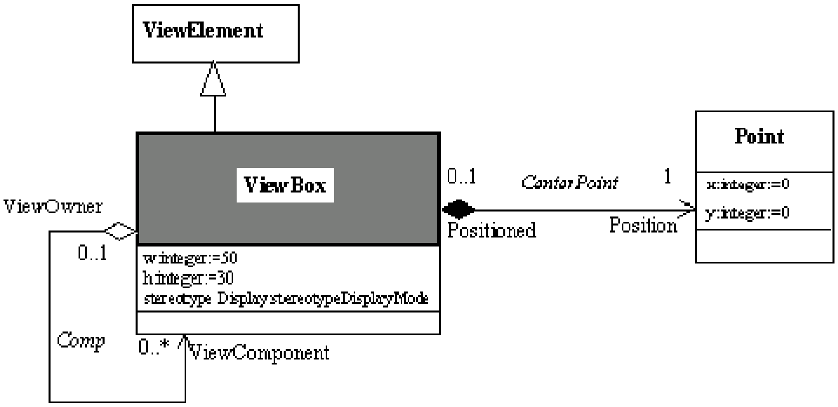


Figure 3-7. Detailed class diagram : *ViewBox*

<u>See also</u>:*ViewLink*, *Diagram*.

ViewBox properties

The *ViewBox* class has the following associations:

- *ViewComponent:ViewBox: ViewBox* contained in the owner *ViewBox.*
- *Position:Point: Point* being the center of the *ViewBox*.

The *ViewBox* class has the following attributes:

- w: Value of the width of the *ViewBox*., defined in pixel.
- h: Height of the *ViewBox*, defined in pixel.
- stereotypeDisplay: This can use one of the following values:
  - NoneStereotypeDisplayMode
  - IconStereotypeDisplayMode
  - LabelStereotypeDisplayMode

ViewBox constructor

```
ViewBox Diagram:createAddAndMoveViewBox
                (in Element model,
                 in integer x,
                 in integer y,
                 in integer w,
                 in integer h)
```

This adds a box representing the "*model*" element in the diagram.

```
ViewBox ViewBox:createAddAndMoveViewBox
                (in Element model,
                 in integer x,
                 in integer y,
                 in integer w,
                 in integer h)
```

This overlaps in a diagram box a new box which represents the "*model*" element into a diagram box.

A *ViewBox* can be created in a *Diagram*, or in a parent *ViewBox*. All the associations and attribute values are set at the time of creation.

"ViewElement" class

ViewElement overview

```
class ViewElement extends Element;
```

Graphical representation of a *ModelElement*. For example, the box representing a class in a diagram will be a *ViewElement* related to the *Class*.

A *ViewElement* belongs to a *Diagram*.

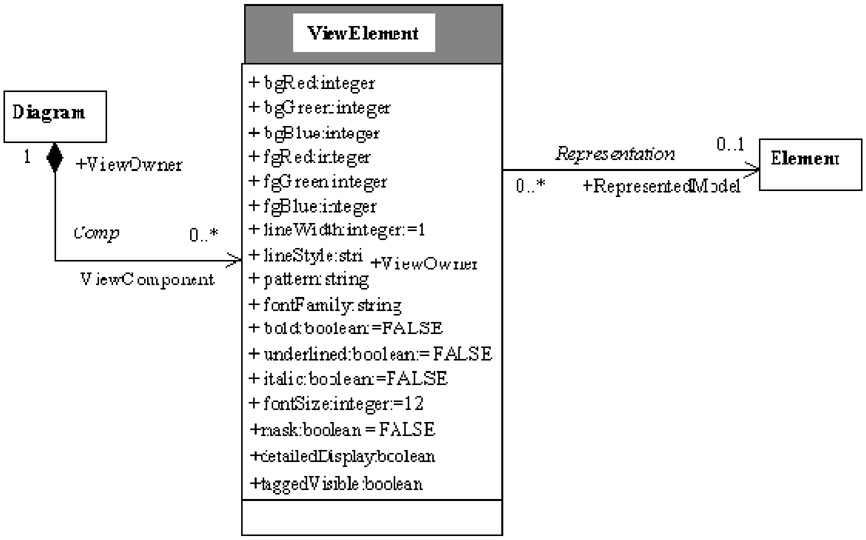
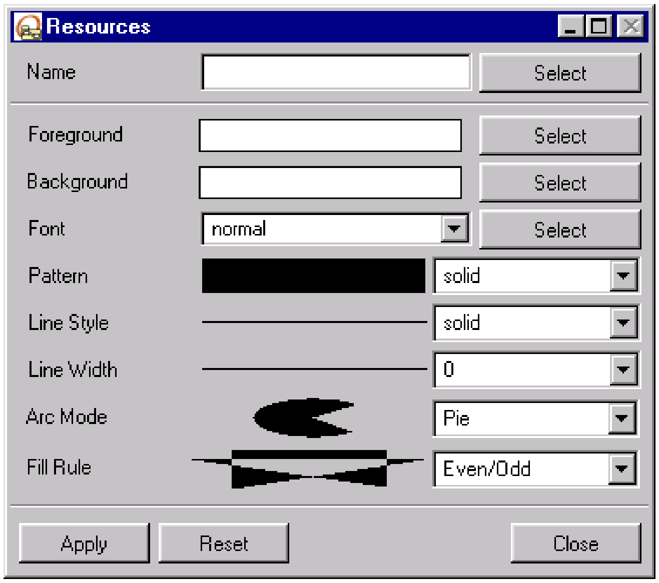


Figure 3-8. Detailed class diagram: *ViewElement*

Color management

The color of graphic elements is managed by *ViewElement*. The "fg" colors (foreround color) are those of the box lines and of its text, whilst the "bg" colors (background color) are those of the background of the box.



Name	
Foreground	
Background	
Font	normal
Pattern	solid
Line Style	solid
Line Width	0
Arc Mode	Pie
Fill Rule	Even/Odd

Figure 3-9. The "*Resources*" dialog box in Objecteering/UML

The RGB system allows you to specify colors. The values of the "*Red*" "*Green*" and "*Blue*" attributes are those which appear in the color definition box (as shown in Figure 3-9). Whole values between 0 and 255 are concerned here, and they are used to indicate the density of colors. When everything is set to 0, the color is black, and when everything is set to 1, the color is white.

ViewElement properties

The *ViewElement* class has the following association:

- *Model:Element*: Link to the element represented by the *ViewElement*

The class has the following attributes:

Attribute values for colors:

- *bgRed*: Level of red in the background color.
- *bgGreen*: Level of green in the background color of the *Element*.
- *bgBlue*: Level of blue in the background color of the *Element*.
- *fgRed*: Level of red in the foreground color of the *Element*.
- *fgGreen*: Level of green in the foreground color of the *Element*.
- *fgBlue*: Level of blue in the foreground color of the *Element*.

Attribute values for line styles:

- *lineWidth*: width of the line drawing the *Element*.
- *lineStyle*: style of the line drawing the Element : solid, dash, dot, dashdot, dashdoubledot, alternate, doubledot, longdash.

Attribute values for patterns:

- *pattern*: type of box fill-in (solid, clear, gray, cross, dark 1 to 4, light 1 to 4).

Attribute values for font:

- *fontFamily*: this attribute concerns the name of the font family, for example, times new roman, lucida, helvetica, etc. This can vary according to the platform used.
- *bold*: application of "bold" to characters
- *underlined*: underlining of characters
- *italic*: characters in italics
- *fontSize*: size of characters

Attribute values for display modes:

- *mask*: used to mask elements
- *detailedDisplay*: used to indicate the level of detail shown
- *taggedVisible*: indicates whether or not tagged values are visible
- *stereotypeDisplay*: indicates whether stereotypes should be represented by a small icon, a large icon or nothing

"ViewLink" class

ViewLink overview

```
class ViewLink extends ViewElement;
```

Graphical link. A *Link* is drawn between two *ViewElements*. It represents *Elements* such as *Dependencies*, *Associations*, *Transitions*, etc.

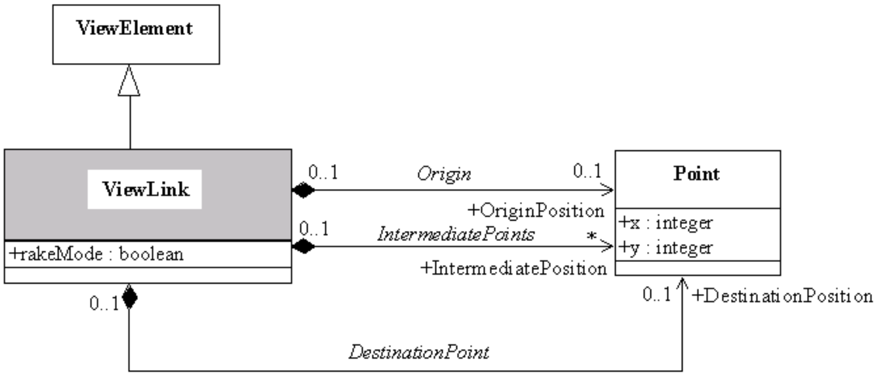


Figure 3-10. Detailed class diagram: *ViewLink*

ViewLink properties

The *ViewLink* class has the following associations:

- *OriginPosition:Point*: Origin point of the link.
- *IntermediatePosition:Point*: Set of points that defines the intermediate angles of the link.
- *DestinationPosition*:*Point*: Destination *Point* of the *Link*.

The *ViewLink* class has the following attributes:

- *rakeMode*: Determines whether or not adjacent links of the same kind must be in rake mode. This only functions for "*generalization*" links.

ViewLink constructor

```
ViewLink Diagram:createAndAddViewLink
                 (in Element linkModel);
```

Adds to a diagram a link which represents the "*linkModel*" element. The link is, by default, a straight line between the centers of origin boxes and link end-points.

ViewLink method

```
ViewLink:addPoint (in integer x, in integer y);
```

Adds a links to an intermediary point. We then have a "*broken*" link.

```
ViewElement Diagram:GetLinkDestination
                    (in ViewLink link)
ViewElement Diagram:GetLinkOrigin (in ViewLink link)
```

These methods allow the origin and destination elements of a link to be found. In general, "*ViewBox*" is concerned, but links on links can also exist (for example, in the case of a *ClassAssociation*).

Editor management services

Overview

J provides services which are used to manage editors, for example to open explorers and graphic editors.

Commands

```
NameSpace:openInBrowser();
```

This service is used to open a browser in the current NameSpace.

```
Diagram:openInEditor();
```

This service opens the current diagram in edit mode.

Chapter 4: System and execution environment services

System and environment

Presentation

Running a UNIX or a Windows process, getting information on environment variables are necessary services for interacting properly with the current operating system.

System environment variables

Dos and Windows use the notion of a system variable, which is close to the notion of the UNIX environment variable. It is possible to take the value of these variables from J. For everything linked to internal Objecteering/UML parameterization, we strongly recommend the use of the "*Module parameters*".

```
getEnvironmentInfo(in String info,out String result);
```

Returns the value of the "*info*" environment variable in "*result*". If "*info*" does not exist, then "*result*" is an empty string.

Note: An undefined value or a value defined as empty produces the same result (result = "")

Example:To recapture the content of the PATH variable:

```
String path;
   getEnvironmentInfo ("PATH", path);
```

```
String getObjingPath ()
```

Returns the "*OBJING_PATH*" environment variable, which specifies where Objecteering/UML is located.

Running a process

```
boolean Object:checkedSpawn (in String command_i, inout
String result_o)
```

This operation spawns a command and returns results. It is simpler but provides less control than the following commands.

Example: `checkedSpawn ("ps", result);`

```
boolean Object:spawnProcess(in String commandLine, in
boolean withConsole, in boolean withShell, in boolean
wait, out int cmdStatus)
```

Creates a process which executes "*commandLine*". "*withShell*" indicates whether or not the command should be launched in a shell (*sh* for Unix, *command.com* for Win95 and *cmd* for NT). "*wait*" indicates whether or not you should wait for the end of the process. "*withConsole*" is only used when console Windows (NT/95) programs are launched, and indicates whether or not the MS/DOS console should be made to appear. "*cmdStatus*" indicates the outgoing status of the command, and is only valid if "*wait*" is true. This method returns a status which indicates whether or not it was possible to launch the command.

```
boolean Object:spawnOutputProcess(in string
commandLine, in boolean withConsole, in boolean
withShell, out int cmdStatus, out String output)
```

Create a process which executes "*commandLine*". "*withShell*" indicates whether or not the command should be launched in a shell (*sh* for Unix, *command.com* for Win95 and *cmd* for NT). "*withConsole*" is only used when console Windows (NT/95) programs are launched, and indicates whether or not the MS/DOS console should be made to appear. "*cmdStatus*" indicates the outgoing status of the command. "output" contains the set of standard and error outgoings of the command. This method returns a status which indicates whether or not it was possible to launch the command.

Notes on the two above calls:

- In Windows, when the "*withConsole*" parameter is equal to "*false*", the child process with the *SW_HIDE* attribute is launched. Consequently, when the program launched is a Windows application (*msdev*, for example) it is obligatory to set this parameter to "*true*", otherwise the application will never appear.
- "*commandLine*" accepts blanks in file names if inverted commas (" ") are placed around the file names.
- When the "*withShell*" parameter is equal to "*true*", it is possible to put any metacharacter (*, >, etc) known to the shell into "*commandLine*". The metacharacter obviously depends on the platform shell (be careful of Unix/Windows, Windows/NT and even Windows 95 portability).

Getting the day's date

```
String Object:getCurrentTime ()
```

This returns the current date.

```
String Object getFormatedTime (in String pFormat)
```

This returns the date in the format specified by the pFormat string. This carries out the equivalent of the "strftime" system call.

For example, the result of the following example:

```
String pFormat = "%d / %m / %y"
String ITime = getFormatedTime(pFormat);
StdOut.write (Itime);
```

is the current date, for example, 14 / 12 / 01.

The formatting codes for pFormat are as follows:

The code ...	represents ...
%a	the abbreviated weekday name.
%A	the full weekday name.
%b	the abbreviated month name.
%B	the full month name.
%c	the date and time representation appropriate to the local time-zone.
%d	the day of the month as a decimal number (01-31).
%H	the hour in 24-hour format (00-23).
%I	the hour in 12-hour format (01-12).
%j	the day of the year as a decimal number (001-366).
%m	the month as a decimal number (01-12).
%M	the minute as a decimal number (00-59).
%p	the current local time-zone's a.m./p.m. indicator for a 12-hour clock.
%S	the second as a decimal number (00-59).
%U	the week of the year as a decimal number, with Sunday as the first day of the week (00-53).
%w	the weekday as a decimal number (0-6, where Sunday is 0).
%W	the week of the year as a decimal number, with Monday as the first day of the week (00-53).
%x	the representation of the date for the current local time-zone.
%X	the representation of the time for the current local time-zone.
%y	the year without the century, as a decimal number (00-99).
%Y	the year with the century, as a decimal number.
%Z	the name or abbreviation of the time-zone (no characters if the time-zone is unknown).
%%	the percent sign.

As in the printf function, the # flag can prefix any formatting code. In this case, the meaning of the format code is changed as follows:

The format code ...	for the ... platform	means ...
%#a %#A %#b %#B %#p %#X %#z %#Z %#%	Windows	that the # flag is ignored.
%#c	Windows	that long date and time representation, appropriate to the current local time-zone, is used (for example, "Thursday, March 21, 2002, 11:30:15").
%#x	Windows	that long date representation, appropriate to the current local time-zone, is used (for example, Thursday, March 21, 2002)..
%#d %#H %#I %#j %#m %#M %#S %#U %#w %#W %#y %#Y	Windows	that any leading zeros are removed.

The format code ...	for the ... platform	means ...
%C	UNIX	that long date and time representation, appropriate to the current local time-zone, is used (for example, "Thursday, March 21, 2002, 11:30:15").

Objecteering/UML management

Purpose

The *Objecteering/UML management* services provide information on the current Objecteering/UML environment, and realize certain actions on the tool.

Save

```
boolean Object:save(in boolean askUserConfirmation)
```

Saves all modifications carried out since the last save in the current UML modeling project. If "*askUserConfirmation*" is true and the interpreter is not in "*batch*" mode, a confirmation dialog box will be displayed. In all other cases, a save is triggered. The return value indicates whether or not the save has been triggered (it is only "*false*" if the user has refused the save when asked for confirmation). After the save, the history of actions is empty.

Pre-condition:No UML Profile Builder session underway.

Post-condition:Return is true => save carried out.

Clear

```
Object:clearActionHistory()
```

Clears the history of actions carried out. This obviously means that the "*undo/redo*" operations cannot be used for actions carried out before this command is run.

Pre-condition:NO UML Profile Builder session underway.

Version

```
String Object:getObjecteeringVersion()
```

Returns the Objecteering/UML version number. Example : "*5.1.1*".

File system management

Presentation

J has a set of portable primitives (UNIX/Windows) for typical file system management operations.

The J stream management services will then be able to access file contents.

File system services

```
boolean Object:removeFile(in String name)
```

Destroys the "*name*" file.

The return is "*true*" if the action has been properly carried out, and "*false*" if this is not the case.

```
boolean Object:moveFile(in String nameOrig, in String
nameDest)
```

Moves the "*nameOrig*" file to "*nameDest*".

The return is "*true*" if the action has been properly carried out, and "*false*" if this is not the case.

```
boolean Object:copyFile(in String nameOrig, in String
nameDest)
```

Copies the "*nameOrig*" file to "*nameDest*".

The return is "*true*" if the files are identical, and "*false*" if this is not the case.

If a file has not been found, "*false*" is returned.

```
boolean Object:compareFile(in String name1, in String
name2)
```

Compares "*name1*" with "*name2*".

The return is "*true*" if the files are identical, and "*false*" if this is not the case.

If a file has not been found, "*false*" is returned.

```
boolean Object:tmpFileName(out String name)
```

Calculates a temporary file name in "*name*".

The return is "*true*" if the action is properly carried out, and "*false*" if this is not the case.

```
boolean Object:getFileSize(in String name, out int
size)
```

Returns the size of the "*name*" file in "*size*".

The return is "*true*" if the action is properly carried out, and "*false*" if this is not the case.

```
boolean Object:mkDir(in String name)
```

Creates the "*name*" directory.

The return is "*true*" if the action has been properly carried out or if the directory already existed, and "*false*" if this is not the case.

```
boolean Object:mkDirRec(in String name)
```

Creates the "*name*" directory and the parent directories if necessary.

The return is "*true*" if the action has been properly carried out or if the directories already existed, and "*false*" if this is not the case.

```
boolean Object:rmDir(in String name, in boolean
removeContent)
```

Destroys the "*name*" directory. If "*removeContent*" is false, then the directory must be empty, otherwise it must only contain files which will also be deleted.

The return is"*true*" if the action has been properly carried out, and "*false*" if this is not the case.

```
boolean Object:rmDirRec(in String name)
```

Destroys the "*name*" directory and its contents recursively.

The return is "*true*" if the action has been properly carried out, and "*false*" if this is not the case.

```
String Object:getOSFamily()
```

Indicates the family of the host operating system : "*Windows*" or "*Unix*".

```
String[] Object:getFilesInDirectory (in String
directory)
```

This returns the list of files contained in the "directory" directory.

```
boolean Object:getFileTime (in String fileName, out int
date)
```

This retrieves the creation date of the "fileName" file in the "date" variable. It returns true if the operation has been successfully carried out.

```
Boolean Object:listFile (in String directory, in
boolean prependDir, in boolean recursive, in String
separator, in boolean withFile, in boolean
withDirectory, out String files)
```

This lists the contents of the "directory" directory in "files".

If "prependDir" is true, directory is concatenated before every entry.

If "recursive" is true, the list contains entries in the sub-directories. "recursive" implies "prependDir".

"separator" indicates the separator used to separate file names in "files". It returns true if the operation has been successfully carried out.

"withFile" indicates whether or not files should be returned.

"withDirectory" indicates whether or not sub-directories should be returned.

Chapter 5: Module management facilities

Module management services

Presentation

Modules are packaged in a directory. We recommend using the "*$OBJING_PATH/modules/<ModuleName>*" directory for a *<ModuleName>* module.

In this directory, there will be a module externalization file, containing the necessary J instructions, stereotype definitions, command definitions, and so on. In addition, there frequently exist additional scripts and resources, such as bitmaps, help files, binaries, etc.

During the installation and initialization of a module, configuration operations, used to prepare the scripts or binaries or to properly define an adapted directory structure, may be necessary.

A well packaged module will carry out these operations, by writing predefined J methods defined hereafter.

These methods will be automatically triggered during the reception/deletion and installation/un-installation operations (for further details on module management operations, please refer to chapter 10, "*Exchanging profiles and modules*" of the *Objecteering/UML Profile Builder module*).

Module management services

```
boolean Object:moduleSelect()
```

This service is called after a module is selected (only available from Objecteering/UML Modeler, and not from Objecteering/UML Profile Builder). It is also called for its parent modules, even if these are masked by the child module.

```
boolean Object:moduleUnselect()
```

This service is called after a module is unselected (only available from Objecteering/UML Modeler).

```
boolean Object:moduleInstall()
```

This service is called after a module is installed. This can either be:

- after a module is installed through the Objecteering/UML administration tool
- after a module is selected for the first time in Objecteering/UML Modeler, if the same version of the module is not already present in the database ("*moduleInstall*" is called first and then "*moduleSelect*").
- after a new version of a module already present in the database is being selected for the first time. Please note that the code of this operation should not assume that the module has already been selected.

If the module has already been installed through the Objecteering/UML administration tool, the "*moduleInstall*" service is not called again when the module is first selected in Objecteering/UML Modeler.

```
boolean Object:moduleUninstall()
```

This service is called after a module is uninstalled (only available from the Objecteering/UML administration tool).

Note: When a module is unselected, it is not uninstalled from the database. Module uninstallation from a database is carried out via the Objecteering/UML administration tool command "*Uninstall a module from a database...*" (for further information, please refer to the "*Detailed view of the Configuration menu*" section in chapter 3 of the *Objecteering/Administrating Objecteering Sites* user guide).

```
boolean Object:moduleInit()
```

This service is called after a module is delivered to a site.

Note: There is no equivalent "un-initialization" operation when a module is deleted from a site.

```
boolean Object:moduleStart()
```

This service is called for each selected module (and its parent modules) when a UML modeling project is loaded. It is not called when a new module is selected.

```
boolean Object:moduleStop()
```

This service is called for each selected module (and its parent modules) when a UML modeling project is unloaded. It is not called when a module is unselected.

```
Package Object:getTypesPackage(String pPackageName);
```

This service is used to obtain the "*pPackageName*" types package. This types package is first searched for in the current test project, before being searched for in the "*_predefinedTypes*" UML modeling project, where necessary.

```
Void Object:deleteTypesPackage(String pPackageName);
```

This service is used to delete the "*pPackageName*" types package from the "*_predefinedTypes*" UML modeling project.

```
Void Object:installHelp();
```

This service is used to incorporate a module help file, which is located in <Module work directory>/help/index.htm, into the main Objecteering/UML index. In general, it is best to use this service from moduleInit(), described above.

```
Void Object:uninstallHelp();
```

This service is used to remove a module entry point from the main Objecteering/UML index.

```
Void Object:internalizePackage(String pDir,
                               String pPackageName);
```

This service is used to internalize an externalized package file. The file name which has the <pPackageName>.ext pDir form contains this file.

```
Void Object:internalizeTypesPackage(String
                                      pPackageName);
```

This service is used to internalize a types package called "*pPackageName*" in the test project, or, by default, in the "*_predefinedTypes*" UML modeling project.

Example

This trivial example illustrates the installation of a module.

```
boolean
Object:default#external#Code#MyGen#moduleInstall ()
{
  StdOut.write("Installation of the MyGen module", NL);
  return(true);
}

boolean
Object:default#external#Code#MyGen#moduleUninstall ()
{
  StdOut.write("Uninstallation of the MyGen module",
NL);
  return(false);
}
```

Getting module parameter values

Inside a module, parameter values are used for adapting module behavior. This value can be accessed from J.

```
boolean Object:getCurrentModuleParameterValue
               (in String name,
                in String profile,
                out String value)

boolean Object:getParameterValue
               (in String name,
                in String profile,
                in String module,
                out String value)
```

These two services return in "*value*" the value of the "*name*" parameter defined in the "*profile*" profile (of the current module for the first service).

If the profile is incorrect, in UML Profile Builder mode, they return false, and a warning appears.

If the profile is incorrect, in UML Modeler mode, they return false.

If the parameter and the environment variable of the same name are non-existent, in UML Profile Builder mode, they return true, and a warning appears.

If the parameter and the environment variable of the same name are non-existent, in UML Modeler mode, they return true.

```
Boolean Object:moduleParameterExists
               (in String pParamName,
                in String pProfileName,
                in String pModuleName)
```

This returns false if the parameter does not exist for the given module and profile. Otherwise, it returns true.

```
boolean Object:currentModuleParameterExists(in String
pParamName, in String pProfileName)
```

This returns false if the parameter does not exist for the given profile. Otherwise, it returns true.

Managing consistency checks

Presentation

The Objecteering/UML CASE tool provides the user with over 200 powerful consistency checks, which guarantee the quality and coherence of the model produced.

However, the user may, in some modeling situations, prefer to have a certain degree of flexibility with regard to the consistency checks applied to his model, and for this reason, certain Objecteering/UML consistency checks are removable. For further information on removable consistency checks, please refer to the "*Removable consistency checks*" section in chapter 3 of the *Objecteering/UML Modeler* user guide.

Consistency check management services

`setBuildInConsistencyCheckOff()`
This call is used to deactivate optional consistency checks. It has no parameters and sends back no values. This call is only accepted during a session, since it modifies a saved value.

`isBuildInConsistencyCheckOn()`
This call indicates whether or not optional consistency checks are present.

Example

This simple example traces the installation of a module.

```
boolean Object:default#external#Code#MyGen#IsCheck ()
{
StdOut.write("test the presence of checks", NL);
return(isBuildInConsistencyCheckOn());
}

boolean Object:default#external#Code#MyGen#CheckOff ()
{
StdOut.write("Checks prevented, NL);
isBuildInConsistencyCheckOff();
return(true);
}
```

Managing work products

Overview

Work products have several predefined mechanisms, such as "*propagation*", or "*file management*", which are supported by several methods that can be redefined.

A work product groups together a generator's set of features as well as its results. Through a dialog box, a work product is used to define the attributes which are to be taken into account by the generator. It is also used to execute the J methods and to manage files generated by the work product.

All work products specialize the *MpGenProduct* metaclass.

A work product can be associated to files which will be automatically moved or deleted when changes to the associated model occur. For example, the deletion of a class in Objecteering/UML brings about the deletion of the associated files (C++ sources).

Managing work product attributes

A work product contains attributes that can be modified in a dialog box. In a J method, we sometimes need to access the attribute values of this work product for generation purposes.

```
MpGenProduct:getAttributeVal (in String Name)
```

This service is used to access the value of an attribute defined on a work product.

```
MpGenProduct:setAttributeVal (in String Name, in String
Value)
```

This service is used to assign the value of an attribute defined on a work product.

Managing the work product propagation

Very often, a work product of the same kind exists in a model tree structure. Typically, a package can have a work product, which has equivalent elements for sub-packages and sub-classes. Work product propagation is a useful facility for end users, who can carry out operations which will be propagated on the entire tree (for example, code generation).

The four following J methods must be defined on the UML profile which manages the module and on the work product.

To adapt the management of work products (consistency and propagation on a model), as well as the management of the generated files, these methods must be redefined.

The ... method on the MpGenProduct metaclass	is automatically triggered on a work product when ...
initProduct (in MpGenProduct product)	it is managed for the first time. When a work product is propagated to a model, the Product parameter corresponds to a "parent" work product. The current work product can then be initialized according to the "parent" work product.
update (in MpGenProduct product)	this work product or its associated model element is modified. The work product is therefore informed of the modifications that are carried out, provided it has been managed. This method allows the user to manage the generated files according to the modifications made to the work product (different suffix, directory, etc.)
boolean mustPropagate ()	it is necessary to define until which model elements (facade package, package, or class), the propagation must continue. Therefore, it is possible to avoid the propagating of a work product to the packages or classes depending on the model element that is related to the work product. For example, if the work product that is being processed is related to a class, this method must return False if we do not wish to create a work product on a class automatically.
boolean isPresent (in MpGenProduct product)	It is necessary to know whether a similar work product should be created during the propagation. This feature is used when a work product is already defined on a model element, but hasn't yet been managed. In this case, it is not necessary to create this work product.

Retrieving a work product

When a model element is being generated, (package or class), the user may need to know the generation work product with the same type defined on this element to recover the attributes that are specific to this work product.

A package generation work product can easily bring on the generation of the classes of this package. It is then possible to retrieve the class' generation work product.

```
ModelElement:default#external#getAnyProduct()
MpGenProduct
```

This service is used to find the generation work product of the current element which is driving the current J execution.

Managing consistency and creation

```
initObject()
```

This is called on an element before its creation.

```
boolean verify()
```

This is called when the button "*OK*" is pressed to permit the product to close, especially to increase the consistency rules.

Managing generation templates

A work product can be associated to a document template or to a generation template. The following methods are necessary for connecting these.

```
initTemplate (in String pName)
```

This relates a generation template to a product. *pName* is the Name of the template that must be accessible in the product module.

This method returns a boolean, as follows

- "*false*" if the generation template whose name has been used as a parameter is not found
- "*true*" if the generation template whose name has been used as a parameter is found

```
String generateWithTemplate()
```

This generates with the generation template related to the product.

Example

```
MyProduct:default#external#Code#MyGen#initProduct
          (in MpGenProduct Product)
{
String ProductName;
String Suffix;
String Path;

   // opening of a session
   sessionBegin ("Propagate", true);

   if (notVoid (Product))
{
      // retrieval of the parent work product values
      ProductName = Product.Name;
      Path        = Product.getAttributeVal("path");
      Suffix      = Product.getAttributeVal("suffix");

      // initialization of the current work product
      setName (ProductName);
      setAttributeVal ("path", Path);
      setAttributeVal ("suffix", Suffix);

}

   // closing of the "Propagate" session
   sessionEnd ();
}

MyProduct:default#external#Code#MyGen#update
          (in MpGenProduct Product)
{
String ProductName;
String Suffix;
String Path;
```

```
   // opening of a session
   sessionBegin ("Propagate", true);

   if (notVoid (Product))
{
      // retrieval of the values of the parent work
         product
      ProductName = Product.Name;
      Path        = Product.getAttributeVal("path");
      Suffix      = Product.getAttributeVal("suffix");

      // initialization of the current work product
      setName (ProductName);
      setAttributeVal ("path", Path);
      setAttributeVal ("suffix", Suffix);

}

   // deletion of all the files managed by the work
      product
   deleteAllFiles ();

   // closing of the "Propagate" session
   sessionEnd ();
}

MyProduct:default#external#Code#MyGen#mustPropagate ()
return boolean
{
   // propagation is carried out on any model
   // element associated to the current work product
   // A similar work product will then be built for all
   // the packages and classes
   return true;
}
```

```
MyProduct:default#external#Code#MyGen#isPresent
(Product : in MpGenProduct) return boolean
{
   // avoids having a work product with the same type
   // on a model element with the package or class type
   return Product.ClassOf==ClassOf;

}

Class:default#external#Code#MyGen#generateClass
{
MpGenProduct ClassProduct;
String Path;

   // returns the first work product found on
   // the class, with the defined "MyProduct" type
   // on the "default#Code#MyGen" module
   ClassProduct = getAnyProduct ();
   Path = ClassProduct.getAttributeName("path");

   StdOut.write("The Myen file of the class is
generated in ");
   StdOut.write(Path, NL);
}
```

Managing generated files

Overview

Files are managed by a work product. This allows the easy addition or deletion of a file, according to the changes made to the model.

For example, the files managed by a work product can be automatically deleted when the work product is deleted.

Features

On a generation work product, two methods can be used to manage the generated files. These are used to add or delete a file managed by the work product.

The ... method	is used to ...
mngFile (fileName : in String, contents : in String)	create the "fileName" file with its "contents" and to add it as a manager of the generation work product.
reverseAllFiles()	reverses all files generated by the product and modified by an editor.

Note: if mngFile(...) is called on a file that is already managed, its content is updated if there are any differences.

Example

```
JavaProduct:default#external#Code#Java#generate ()
{
String content;
String fileName;

   // method used to generate Java code
   content = generateJava ();

   fileName.strcat (getAttributeVal("path"),
                    "/",
                    Name,
                    ".",
                    getAttributeVal("suffix"));

   // create and manage the generated file
   mngFile (fileName, content);

   // update all the open editors
   updateAllEditors ();
}
```

Editing generated files

Overview

Objecteering/UML provides a service that maintains consistency between external text editors and the model.

It is possible to automatically open a dialog box on the corresponding element in text visualizers, by double-clicking on a generated file zone, providing that specific markers have been generated.

In the external editors, the same markers allow the Objecteering/UML repository to be updated according to the modifications made by the user on the generated file.

Markers

The user must define markers in order to use the edition mechanisms mentioned in the above paragraph. Without a marker, the visualizer cannot be used to edit a generated element by double-clicking, and it is impossible to retrieve the contents of the generated files that have been modified by an external editor.

Markers are used to:

- call the dialog boxes from the internal view
- retrieve notes from the generated files, after external edition or during recursive retrieval

The purpose of a marker is to keep:

- the line number concerned in the generated code
- the model element that has given rise to the part of generated code
- the "*style*" of the identifier

Marker styles

The ... style	corresponds to ...
IdGen	a text completely generated by the tool.
IdBox	a model element (Element) entered in a dialog box.
IdTxt	a text the user can modify in the generated files.
IdEnd	the end of a marker zone. There is no particular meaning attached to the corresponding type of code.

Marker primitives

The ... method	returns an identifier ...
String idGen ()	which cannot be used to modify the model element in a visualizer.
String idBox ()	which allows the model element to be edited thanks to a dialog box.
String idTxt ()	used to modify the model element in an external editor.
String idEnd ()	indicating the end of the zone containing the model element.
String marker (inString depName, inString textTypeName)	which allows the automatic creation of text on the current element during the retrieval of code after using an external editor. The "depName" parameter is the name of the dependency which points to the text we wish to create from the current element (example "Descriptor"); "textTypeName" is the name of the text type to be created (for example, "C++").

The markers inserted into the text are formatted when the "*mngFile()*" method is called. This method uses the following feature, which must be redefined according to the language targeted by the generator:

```
MpGenProduct:default#external#MpGenProduct:getIdLineCom
ment () return String
```

This method returns the string that indicates the beginning of a comment line (for example, in Java, it returns "//").

Internal visualization

```
MpGenProduct:default#external#MpGenProduct:intVisuFileN
ame (in String fileName)
```

This method is used to call an internal visualizer in the context of the generation work product and to display a file managed by this generation work product. The "fileName" parameter is the complete name of the concerned file, including the path and suffix.

External edition

```
MpGenProduct:default#external#MpGenProduct:extEditFileN
ame (in String fileName;)
```

This is used to call an external visualizer in the context of the generation work product and to thus display a file managed by this generation work product. The "fileName" parameter is the complete name of the concerned file, including the path and suffix.

The extEditFileName method uses the following method to retrieve the command used to launch the editor:

```
String
Object:default#external#Object:getEditorCommandLine ()
```

By default, this method returns the value of the ExtEditorCommandLine parameter.

For more complex editors, the user can redefine this method on the work product.

Updating the visualizers

The visualizers contain a generated file. The user may wish to update these visualizers following a new generation, in order to update the files.

```
MpGenProduct:default#external#updateAllEditors ()
```

This is used to update all the open graphic editors.

Example 1: idGen () method

To generate the comment preceeding a class declaration:

```
Class:default#Code#MyGen#generateClass ()
{
String Marker;
                 Marker=idGen () +
                 " //Class declaration" +
                 NL +
                 idEnd ();
return Marker
}
```

Thus, during the internal visualization of the file containing this class, double-clicking on the comment will remain without effect.

Example 2: idBox () method

To generate the attributes of a class:

```
Attribute:default#Code#MyGen#generateAttribute ()
{
   String Marker;
   PartAttribute
   {
   Marker=Marker + idBox ()
                 + getType()
                 + " ; "
                 + NL
                 + idEnd ();
   }
   Return Marker
}
```

Therefore, when a file containing this class is being visualized internally, it will be possible to open the edition dialog box of the corresponding attribute, by double-clicking on each declaration string of an attribute.

Example 3: idTxt ()method

To generate the method code from the context of this method:

```
Operation:default#Code#MyProd#generateMethod ()
{
   String Marker;
   PartOperation
   {
      getOneNoteOfType ("C++")
      {
         Marker=Marker + idTxt ()
                       + Content
                       + NL
                       + idEnd ();
      }
   }
      if (Marker = "") {
         Marker=marker("Descriptor" + "C++");
      }
   Return Marker;
}
```

If the C++ text...	**during the internal visualization ...**	**during the external visualization ...**
existed before generation	double-clicking on the generated code opens the edition dialog box of the "C++" text	the code appears surrounded by the markers. It is possible to modify it, and the "C++" type text will be updated at the end of the edition in the Objecteering/UML repository
did not exist before generation	only the markers appear, double-clicking is not effective	Only the markers appear. If the user inserts code between the markers, a C++ type text will be automatically created at the end of the edition.

Example 4: Internal visualization of a file

```
MyProduct:default#external#Code#MyGen#visualize ()
{
   String fileName;
   Filename=getAttributeVal("path")
            + Name,
            + getAttributeVal("suffix");
   intVisuFileName (fileName);
}
```

Example 5: External edition of a file

```
MyProduct:default#external#Code#MyGen#edit ()
{
   String fileName;
   Filename=getAttributeVal("path")
            + Name,
            + getAttributeVal("suffix");
   extEditFileName(fileName);
}
```

Chapter 6: Search services

Overview of search services

Introduction

In order to carry out searches in Objecteering/UML using the J language, a number of services belonging to the JSearch metaclass must be used. The JSearch metaclass is used not only to memorize search parameters, but also to run searches and get related results.

The following diagram (Figure 6-1) illustrates the different metaclasses used.

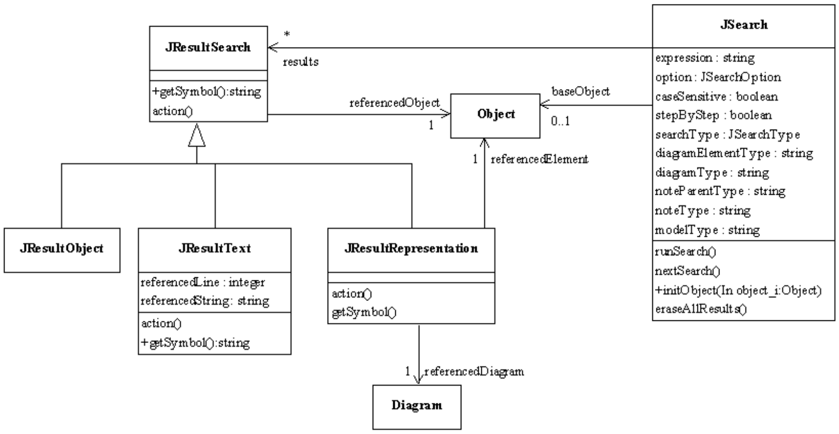


Figure 6-1. Class diagram of the different metaclasses used in search operations

The following example of code provides an illustration of code used to run the search mechanism:

```
// Initialization of the structure encapsulated in a
J session (obligatory)

JSearch objecteeringSearch ;
// In J, variable declarations are carried out at the
beginning of a session

sessionBegin ("search", true) ; // Launch of a J session

// Initialization of obligatory parameters

objecteeringSearch.initObject (this) ;
// The search is run from the element selected
(for example, Package)

objecteeringSearch.setexpression ("Class") ;
// The Class string is searched for

objecteeringSearch.runSearch () ;
// Running the search

// objecteeringSearch.nextSearch () ;
Running the search in step by step mode

// Simple display of results:
objecteeringSearch.resultsJResultSearch
{
   StdOut.write (getSymbol(), NL) ;
// The result is displayed in textual form.
}

objecteeringSearch.eraseAllResults () ;
// All results are deleted in Objecteering/UML

objecteeringSearch.delete ;
// The search is deleted

sessionEnd () ;
// End of the J session
```

Search services

Overview of the JSearch metaclass

Figure 6-2 shows the JSearch metaclass, complete with its methods and attributes.

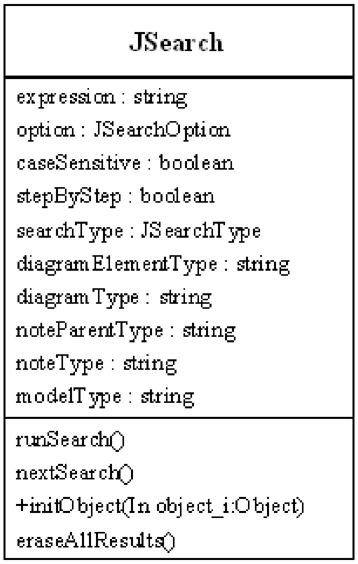


Figure 6-2. The JSearch metaclass

JSearch methods

The following methods are contained in the JSearch metaclass:

`runSearch()`

This runs or re-runs the search. This method takes as its parameters the different options positioned on the JSearch class and runs the search according to these options.

`nextSearch()`

This is used in the step by step search mode (which must be selected), and is used to continue a search or launch it for the first time. It returns only one element.

`initObject(Object)`

This method is used to initialize the structure using the default values, and takes as its parameter the object from which the search is to be run.

`eraseAllResults()`

This method is used to delete all those elements found.

Note: This method must be called before either destroying the JSearch or saving the results and destroying them later, as this operation is the responsibility of the user.

JSearch attributes

The JSearch metaclass has the following attributes for all search contexts:

- *expression*: The "*expression*" attribute indicates the expression which is to be searched for (by default, this is empty).
- *option*: The "*option*" attribute specifies the correspondence between the expression to be searched for and the string used in matches. Options are *JContains*, *JBeginsWith*, *JEndsWith* and *JMatchesExactly*. By default, this is set to *JContains*.
- *caseSensitive*: The "*caseSensitive*" attribute indicates whether or not the search should distinguish between uppercase and lowercase characters (by default, this is set to false).
- *stepByStep*: The "*stepByStep*" attribute specifies whether or not the search should be carried out in step by step mode (by default, this is set to false).
- *searchType*: The "*searchType*" attribute indicates the type of search to be run, so as to indicate what type of elements the search is to find. Options are *JModel*, *JDiagram*, *JText*, *JAll*. By default, this is set ot *JModel*.

The JSearch metaclass has the following attributes for searches in diagrams:

- *diagramElementType*: The "*diagramElementType*" attribute indicates the metaclass of the element searched for (by default, this is set to "*All*").
- *diagramType*: The "*diagramType*" attribute specifies the metaclass of the diagram in which the search is to be carried out (by default, this is set to "*All*").

The JSearch metaclass has the following attributes for searches in textual elements (notes, constraints):

- *noteParentType*: The "*noteParentType*" attribute indicates the metaclass of the note's parent (by default, this is set to "*All*").
- *noteType*: The "*noteType*" attribute specifies the type of note (by default, this is set to "*All*").

The JSearch metaclass has the following attributes for searches in the model:

- *modelType*: The "*modelType*" attribute indicates the metaclass of the model element searched for.

Overview of the JResultSearch metaclass

Figure 6-3 shows the JResultSearch metaclass, complete with its methods.

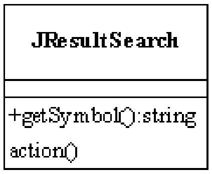


Figure 6-3. The JResultSearch metaclass

JResultSearch methods

The following methods are contained in the JResultSearch metaclass:

`getSymbol ()`

This method is redefined in the different JResults and returns a character string which corresponds to the definition of the result, and more particularly, those elements found.

`action ()`

This method is used, in the context of a simple object, to select the object in the explorer. For searches in diagrams, diagrams are selected and also opened.

Overview of the JResultText metaclass

Figure 6-4 shows the JResultText metaclass, complete with its attributes.

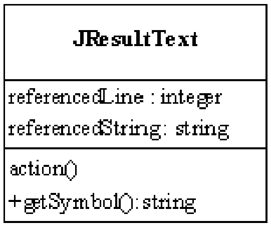


Figure 6-4. The JResultText metaclass

JResultText attributes

The JResultText metaclass has the following attributes:

- *referencedLine*: The "*referencedLine*" attribute indicates the line where the string has been found.
- *referencedString*: The "*referencedString*" attribute gives the contents of the entire line where the string has been found.

Examples of searches

Example 1: Simple search in a diagram

Example 1 below shows the search for a class named "*Object*" in a class diagram.

```
// Initializing the structure
JSearch objecteeringSearch;
sessionBegin ("search", true);

// Initializing parameters
objecteeringSearch.initObject (this);
objecteeringSearch.setexpression ("Object");
objecteeringSearch.setelementType ("Class");
objecteeringSearch.setdiagramType ("StaticClassDiagram");
objecteeringSearch.setsearchType ("JsearchTypeDiagram");

// Running the search
objecteeringSearch.runSearch ();

// Processing results (display)
objecteeringSearch.resultsJResultSearch
{
   StdOut.write (getSymbol(), NL)
}

// Destroying the structure
objecteeringSearch.eraseAllResults ();
objecteeringSearch.delete;
sessionEnd ();
```

Example 2: Step by step search in Java notes

Example 2 shows the search for the first two "*newObject (*" strings found in Java-type notes.

```
// Initializing the structure
JSearch objecteeringSearch;
sessionBegin ("search", true);

// Initializing parameters
objecteeringSearch.initObject (this);
objecteeringSearch.setexpression ("new Object (");
objecteeringSearch.setnoteType ("JavaCode;JavaResult");
objecteeringSearch.setnoteParentType ("Operation");
objecteeringSearch.setsearchType ("JsearchTypeNote");
objecteeringSearch.setstepByStep (true);

// Running the search in step by step mode
objecteeringSearch.nextSearch ();

// Displaying the result
objecteeringSearch.resultsJResultSearch
{
   StdOut.write (getSymbol(), NL)
}
Continuing the search
objecteeringSearch.nextSearch ();

// Displaying the result
objecteeringSearch.resultsJResultSearch
{
   StdOut.write (getSymbol(), NL)
}

// Destroying the structure
objecteeringSearch.eraseAllResults ();
objecteeringSearch.delete;
sessionEnd ();
```

Chapter 7: Dynamic dialog boxes

Introduction to dynamic dialog boxes

Introduction

When a module developer wishes to "dialog" with a user, he can create a dialog box, either modal or non-modal, through which communication can take place. Modal dialog boxes block other operations until the dialog box is closed, whilst non-modal dialog boxes can be left open whilst other actions are carried out. For example:

```
JBox box;
//start processing
box.show()
//end processing
```

The last line of the above code will only be executed after the dialog box has been closed.

Controls are positioned vertically by default. This positioning is handled by a cursor, which updates its vertical or horizontal position during the next control.

The following diagram illustrates the different metaclasses used (Figure 7-1).

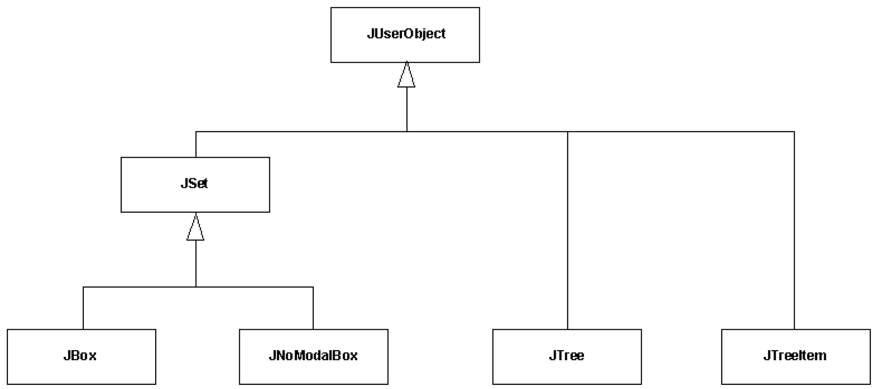


Figure 7-1. Class diagram of the different metaclasses used

Various features can be added to these dialog box, whether modal or non-modal, for example, text fields or combo boxes.

J boxes are constructed in horizontal sections and/or vertical sections.

Calls used

Calls used to create and destroy dialog boxes

```
Object:createJBox (ident: in String, title: in String,
                   direction: in JLayoutType,
                   consultMode: boolean)
  return JBox
```

This creates a JBox object which corresponds to a modal dialog box.

If the *consultMode* parameter has been set to true, then none of the dialog box elements will be modifiable, and only the "*OK*" button will remain.

```
Object:createJNoModalBox (ident: in String,
                          title: in String,
                          direction: in JLayoutType,
                          profileNameCB: in String,
                          operationNameOkCB: in String,
                          operationNameCancelCB: in
                                          String)
  return JNoModalBox
```

This creates a JNoModalBox object corresponding to a non-modal dialog box.

The last three parameters are used to position call backs on the "*OK*" and "*Cancel*" buttons. They represent respectively:

- Profilename: the complete name of the profile where the call back method is defined
- operationNameOkCB: the name of the operation called on the *OK* action
- operationNameCancelCB: the name of the operation called on the *Cancel* action

```
Jset:delete ()
```

This destroys the JSet dialog box.

Note 1: The user is responsible for destroying the dialog box, whether its creation is included in a session or not.

Note 2: An identifier must be unique within a JSet, in the same way as each dialog box must have a unique identifier.

Calls used to retrieve dialog boxes

```
Object:getJSet (identSet: in String)
  return JSet
```

This retrieves the dialog box (modal or non-modal) which has *identSet* as its identifier.

JUserObject methods

JUserObject methods

JUserObject methods provide a usage context for the box in question (please note that in Figure 7-2, the use link is present to simulate a relationship).

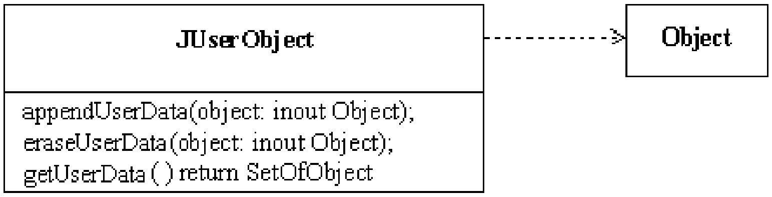

JUserObject
appendUserData(object: inout Object);
eraseUserData(object: inout Object);
getUserData ( ) return SetOfObject
Object

Figure 7-2. The JUserObject metaclass

The following methods are contained in the JUserObject class, which is an abstract class:

```
appendUserData (object : inout Object);
```

This adds to the user data list a model object, in order to provide a usage context.

```
eraseUserData(object : inout Object);
```

This removes from the user data list a model object, if it has already been added using the previous method.

```
getUserData ()
       return SetOfObject
```

This returns the user data list.

```
setUserInfo (label : in String)
```

This is used to memorize textual information.

```
getUserInfo()
       return String
```

This returns textual information.

JSet methods

JSet layout methods

```
beginLayoutSection (layout: in JLayoutType,
                    frame: in boolean,
                    frameLabel: in String)
```

According to the value defined for the JLayoutType, either a vertical (JLayoutVertical) or a horizontal (JLayoutHorizontal)section is opened, with a frame surrounding this section if *frame* has been set to true. A title is given to this frame is *frameLabel* has been defined.

```
endLayoutSection()
```

This closes the section previously opened. If no section has been previously opened, nothing happens.

```
changeColumn (spacing: in integer)
```

This creates a new column if the section is vertical, and a new line if the section is horizontal.

```
getPosition (x: out integer
             y: out integer)
```

This is used to obtain the position of the cursor.

```
changePosition (posCursor: in integer)
```

This changes the position of the cursor according to the co-ordinates provided. It is not, however, possible to move the cursor to a position inferior to the default position.

JSet addition methods

Graphic elements which include internal values are identified by a character string defined by the user upon their creation. This identifier makes it possible to modify and retrieve graphic element values.

Graphic elements are made up of a title (this is optional), placed above the gadget in question, except in the case of a toggle.

```
addLabel (label: in String)
```

This adds a remark. The carriage return character is permitted.

```
addBitmap (bitmapName: in String)
```

This adds a bitmap (either a .gif or .bmp file).

```
addToggle (ident: in String, label: in String,
           value: in boolean)
```

This adds a two-state button.

```
addField (ident: in String, label: in String,
          value: in String,
          type: in JcontrolFieldTextType,
          width: in integer)
```

This adds a text field. The following JControlTextType types are possible:

- JControlTextField: a text field which may be edited
- JControlNumberField: a numeric field
- JControlDate: a date field
- JControlPassword: a password field
- JControlFileOpen: a file open field (this is a text field linked to a button and used to open a file or directory selector, for both Windows and UNIX)
- JControlFileSave: a file save field (this is a text field linked to a button and used to open a file or directory selector, for both Windows and UNIX)
- JControlDirectory: a file directory field (this is a text field linked to a button and used to open a file or directory selector, for both Windows and UNIX)

Where the field is linked to a file or directory selector, the field is composed of the text indicating the selected path and a button used to open the selector box. In UNIX, Open, Save and Directory open the same dialog box.

```
addText (ident: in String, label: in String,
         value: in String, width: in integer,
         height: in integer);
```

This adds a multi-line text. The carriage return character separates the lines.

```
addList (ident: in String, label: in String,
         multiple: in boolean, width: in integer,
         height: in integer)
```

This adds a list. The list is a multiple selection list, where the *multiple* parameter has been set to true. If this is not the case, a simple selection list is added.

List values are updated by the *addListItem* and *removeListItem* methods.

```
addCombo (ident: in String, label: in String,
          width: in integer)
```

This adds a combo box. List values are updated by the *addListItem* and *removeListItem* methods.

```
addHelp (labelHelp: in String,  htmlFile: in String)
```

This adds a button called *labelHelp* , which is associated with the File html file. This can be a url.

```
addTree (ident : in String, label : in String,
         width : in integer, height : in integer,
         multiple : in boolean)
         return JTree
```

This adds a tree (and returns an instance of the JTree class). The multiple boolean indicates the trees selection mode (multiple/simple).

JSet value retrieval methods

```
getValue (ident: in String, value: inout String)
         return boolean
```

This retrieves a value in the form of a string corresponding to the check which has the *ident* identifier. The gadget can be text, a field (text, date, numeric), a toggle, in which case the state is converted into a string (true or false) or a list, in which case the first selection is retrieved.

```
getValueList (ident: in String,
              justSelection: in boolean,
              valueList: inout SetOfString)
  return boolean
```

This retrieves the list of those items which make up the list, if the *justSelection* has been set to false. Otherwise, the list of selected items is retrieved.

```
getTree (ident : in String)
         return JTree
```

This retrieves the instance of the JTree class named "ident".

JSet value modifying methods

```
setValue (ident: in String, value: in String)
  return boolean
```

This allocates the character string to the check which has the *ident* identifier. This check can be text, a field (text, date, etc.), a number (whose value must be numeric) or a toggle (whose value must be either true or false).

```
addListItem (identList: in String, value: in String,
             bitmapName: in String,
             selected: in boolean)
  return boolean
```

This adds a field to the list, which has the *identList* as identifier, *value* as its value and *bitmapName* (this can be left empty) as its bitmap. The bitmap is not taken into account for combo boxes. If the *selected* parameter is set to true, it will be selected in the list. This is true for list and combo type checks.

```
removeListItem (identList: in String,
                value: in String)
  return boolean
```

This removes the *value* field from the list which has *identList* as its identifier. This is true for list and combo type checks.

```
getIdent ()
  return String
```

This returns the dialog box identifier.

```
readAndAddFile (ilvFileName: in String,
                ilvScriptFileName: in String)
```

This adds an Ilog resource file to the dialog box.

```
resetList (ident : in String)
```

This is used to empty the "ident" identifying list. This method also works with combos.

JSet presentation methods

```
boolean JSet::setFont(in string ident_i ,
                      in string family,
                      in integer fontSize,
                      in boolean bold,
                      in boolean italic)
```

This changes the character font of the "*ident_i*" identifier field. It returns true if the field exists and it is possible to allocate to this field the "*family*" font name with the "*fontsize*" size, in bold if "*bold*" is true and in italics if "*italics*" is true.

Certain combinations of font size and bold/italic attributes are not possible for certain fonts. This depends on the font in question and the operating system.

```
boolean setForegroundColor(in string ident_i ,
                           in integer red ,
                           in integer green ,
                           in integer blue)
```

This changes the color of the text for the "*ident_i*" identifier field. The intensity values for red, green and blue should be situated between 0 and 255. The method returns true if the operation has been carried out correctly. The method returns false if the field has not been found, or if one of the values is incorrect.

```
boolean setBackgroundColor(in string ident_i,
                           in integer red,
                           in integer green,
                           in integer blue)
```

This changes the background color for the "*ident_i*" identifier field. The intensity values for red, green and blue should be situated between 0 and 255. The method returns true if the operation has been carried out correctly. The method returns false if the field has not been found, or if one of the values is incorrect.

```
setFocus (ident : in string)
```

This gives the focus to the graphic element identified by the ident string.

```
setFocusOkButton ()
```

This gives the focus to the "*OK*" button.

```
setFocusCancelButton ()
```

This gives the focus to the "*Cancel*" button (if this exists).

JTree and JTreeItem methods

JTree methods

The JTree class specializes the JUserObject class and defines a tree service. It contains the following operations:

```
addItem (parentItem : inout JTreeItem,
         label : in String, bitmapName : in String,
         selected : in boolean, index : in integer)
         return JTreeItem
```

This adds a "*parentItem*" parent node to the JTree, by specifying its label, the name of the associated bitmap (empty string if no bitmap is associated), its selection state and an index (starting at 0) indicating its position in the list of nodes of the same level.

```
addRootItem (label : in String, bitmapName : in String,
             selected : in boolean, index : in integer)
             return JTreeItem
```

This adds a node to the JTree, under the root, by specifying its label, the name of the associated bitmap (empty string if no bitmap is associated), its selection state and an index indicating its position in the list of nodes of the same level.

```
moveItem (item : inout JTreeItem,
          itemParent : inout JTreeItem,
          index : in integer)
```

This moves a JTreeItem (item) into another JTreeItem (itemParent). The index determines at which position the item is inserted in the parent item.

```
getRoot ()
return JTreeItem
```

This retrieves the root of the JTree. This node is not displayed in the tree, and cannot, therefore, be a ghost item (for further information, please refer to the JTreeItem method list). To obtain a list of nodes added to the root, the root can be obtained using this method, and then the list of child nodes (for further information, please refer to the "*getChildList*" method in the JTreeItem method list).

```
removeItem(itemToRemove : inout JTreeItem)
```

This deletes from the tree the branch whose root it the "*itemToRemove*" item.

```
getSelectedItems(itemList : inout SetOfJTreeItem)
```

This retrieves the set of items selected in the JTree from the "*itemList*" list.

```
getCallbackItem ()
return JTreeItem
```

This retrieves the tree item on which a call to a callback has been triggered.

JTreeItem methods

```
isSelected ()
return boolean
```

This returns the selection state of an item.

```
setSelected (selected : in boolean)
```

This changes an item's selection state.

```
getLabel ()
return String
```

This returns the label associated with the item.

```
setLabel (label : in String)
```

This modifies the label associated with the item.

```
getBitmapName ()
return String
```

This returns the name of the bitmap associated with the item.

```
setBitmap (bitmapName : in String)
```

This modifies the bitmap associated with the item.

```
getParent ()
return JTreeItem
```

This retrieves the item's parent node.

```
getChildList(itemList : inout SetOfJTreeItem)
```

This retrieves from the "*itemList*" list the set of JTreeItem's child items.

```
getChild (index : in integer)
return JTreeItem
```

This retrieves the JTreeItem child of the current "*index*" index.

```
setGhostItem (ghostItem : in boolean)
```

This is used to state whether or not an item is a ghost item. A ghost item internally manages a "*ghost*" child. The item has a "+" symbol, which is used to trigger a Callback (defined through "*setTreeExpandGhostItemCallback*"). This can be useful, for example, if the user wishes to build his tree directly. Once the ghost item has been expanded once by the user, it becomes a "*normal*" item.

```
isGhostItem ()
return boolean
```

This returns true if the current item is a ghost item.

```
searchSelectedRecursive (itemList : inout
SetOfJTreeItem)
```

This finds the list of selected JTreeItems. The search is carried out on the branch defined by the JTreeItem.

```
searchDataRecursive (object : inout Object)
return JTreeItem
```

This finds the JTreeItem to which the Object given as a parameter is associated. The search is carried out on the branch defined by the JTreeItem.

```
getIndex ()
return integer
```

This returns the index corresponding to the position of the item in its branch.

```
expand ()
```

This triggers the expansion of the branch of the tree defined by the JTreeItem.

```
isExpanded ()
return boolean
```

This returns true if the branch defined by the JTreeItem is expanded.

```
shrink()
```

This triggers the reduction of the branch defined by the JTreeItem.

JBox and JNoModalBox methods

JBox methods

The JBox class specializes the JSet class and is used to implement modal dialog boxes. It contains the following method:

```
show ()
  return boolean
```

This displays the dialog box in modal form (that is to say, all other operations are blocked whilst the dialog box is open). It returns true if the user clicks on the "*OK*" button and false if the user clicks on the "*Cancel*" button (or the cross in the top right-hand corner used to close the dialog box).

JNoModalBox methods

The JNoModalBox class specializes the JSet class and is used to implement non-modal dialog boxes. It contains the following methods:

```
setCallback(ident: in String, profileName: in String,
           operationName: in String)
```

This is used to position the *operationName* callback on the profile used as a parameter on the box object.

If the identifier is a button identifier, the "*operationName*" is called when you click on the button.

If the identifier is a list, the callback is called when you double-click on an element in the list.

Note: In this case, the "ident" refers to another model element.

```
addButton (ident: in String, label: in String,
           profileName: in String,
           operationName: in String,
           width: in integer)

addButtonBitmap (ident: in String, label: in String,
                 profileName: in String,
                 operationName: in String)
```

These methods are used to respectively add a button and a bitmap to a call back.

Note: The complete path must be provided for the bitmap.

When the callback is activated, only the JNoModalBox object is known. If the user wishes to store another usage context (for example, the object on which the command has been run), then the "*appendUserData*", "*getUserData*" and "*eraseUserData*" methods, detailed in the "*JSet complementary methods*" theme of this section, should be used. If the object is destroyed, then it is removed from the user data list.

```
show (consultMode: in boolean)
```

This displays the dialog box in non-modal form (that is to say, the dialog box does not block other operations).

If *consultMode* has been set, then Objecteering/UML changes over to consultation mode, and an egg-timer appears in the dialog boxes (except the current JNoModalBox).

```
hide ()
```

This hides the dialog box.

JNoModalBox also contains operations used to associate callbacks to a JTree.

Note: The "*setCallback*" method is used to position the callback by default on the JTree, which is then triggered by double-clicking on one of the tree's leaves.

```
setTreeSelectionCallback(ident: in String,
                         profileName : in String,
                         operationName: in String)
```

This is used to position an "*operationName*" callback. This callback is related to the "*ident*" identifying JTree, and is triggered by the selection of one of the tree's nodes.

```
setTreeExpandCallback (ident: in String,
                       profileName : in String,
                       operationName: in String)
```

This is used to set the "*operationName*" callback. This callback is related to the "*ident*" identifying JTree, and is triggered by the expansion of a node (double-click on a node which has children, or on the "+" symbol adjoining the node).

```
setTreeExpandGhostItemCallback(ident: in String,
                               profileName : in String,
                               operationName: in String)
```

This is used to set an "*operationName*" callback. This callback is related to the "*ident*" identifying JTree, and is triggered by the expansion of a particular node, namely "*GhostItem*". This node simulates the existence of a child (for further information, please refer to the "*JTree and JTreeItem methods*" section in the current chapter of this user guide).

```
setTreeShrinkCallback(ident: in String,
                      profileName : in String,
                      operationName: in String)
```

This sets an "*operationName*" callback. This callback is related to the "*ident*" identifying JTree, and is triggered by the shrinking of a branch (double-clicking on a node whose children are visible, or on the "-" symbol adjoining the node).

JNoModalBox drag and drop methods

```
Object JNoModalBox::addDropZone
                    (in String ident, // field identifier
                     in String label, // field title
                     in Object initial, // initial value
                     in int width) // width
```

This adds and returns a drag and drop field which resembles those you can see in standard windows. Contrary to other fields, it is not editable, but accepts drops of objects from the explorer.

This field as the "*ident*" identifier with the "*label*" label contains the "*initial*" object (which can be void) and is of "*width*" width.

Important: Before being usable, at least one call to the appendAllowedMetaclass method must be carried out (see below).

Example:

```
JObjectField f = box.addDropField("drarDropField",
                                "Drop field", null, 200);
 f.appendAllowedMetaclass("NameSpace") // allow drop of
                                         Class, Package,
                                         Enumerate, ...
```

To create a drop field without the standard "target" bitmap, use the following method:

```
JObjectField JNoModalBox::addDropField
             (in String ident, // field identifier
              in String label, // field title
              in Object initial, // initial value
              in int width) // width
```

```
JObjectField JNoModalBox::getDropField
             (in string ident_i )
```

This returns the droppable field called "*ident_i*" or void if no droppable field of this name exists.

```
void JObjectField::appendAllowedMetaclass(in String
                      aMetaclassName)
```

This indicates that a drag and drop from a "*aMetaclassName*" class or child class object is authorized. "*aMetaclassName*" is the name of a metaclass you wish to allow the user to drop in the field. All metaclass names except "*Object*" are valid.

Warning: If this method is never called on the drag and drop field, no drag and drop operations are authorized.

```
Object JObjectField::getValue()
```

This returns the object which has been dropped in the field.

```
JObjectField::setValue(in Object obj)
```

This changes the dropped object it "*obj*" is from a metaclass authorizing drops. If "*obj*" is void, this empties the field.

```
boolean JObjectField::acceptDropOf(in Object obj)
```

This returns true if the field accepts the object in question. It checks that the object's metaclass is either part of or inherits from one of the authorized metaclasses.

User interaction methods

The queryUser method

```
boolean queryUser (in String title, in String label)
```

This method is used to ask the user a question, in order to carry out conditional processing.

For example:

```
if (queryUser ("Confirmation", "Do you really want to
delete the files?")) {
    // File deletion
    ...
} else '
    // No file deletion
    ...
}
```

The infoUser method

```
infoUser (in String title, in String label)
```

This method is used to display an information dialog box.

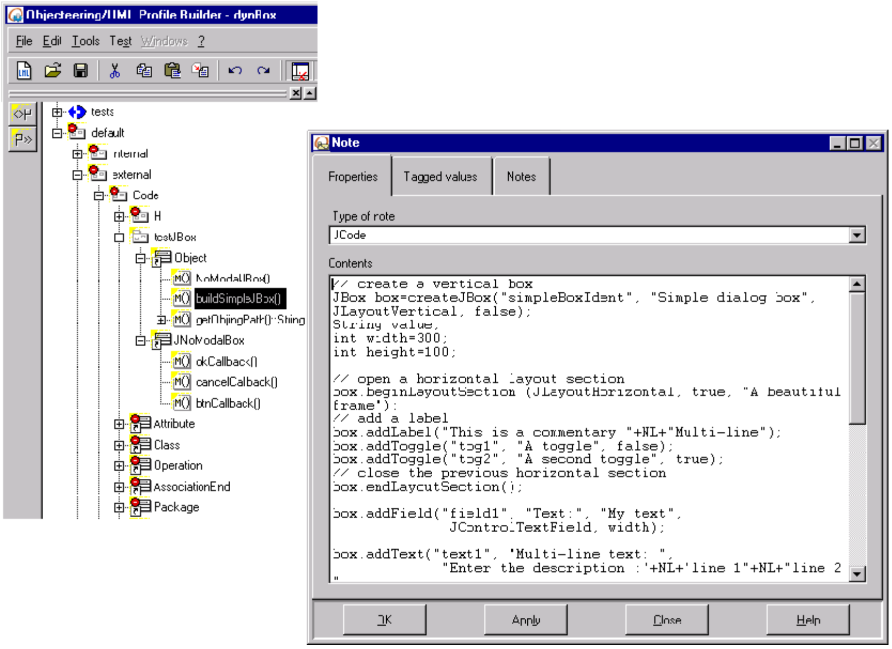
For example:

```
infoUser ("Information", "Generation complete." + NL +
"No errors during generation.")
```

Examples of J box creation

Example 1: Creating a simple modal dialog box

Figure 7-3 shows part of the J code developed in the UML profiling project for the creation of a simple modal dialog box.



Type of rote: JCode

Contents

```
// create a vertical box
JBox box=createJBox("simpleBoxIdent", "Simple dialog box",
JLayoutVertical, false);
String value;
int width=300;
int height=100;

// open a horizontal layout section
box.beginLayoutSection (JLayoutHorizontal, true, "A beautiful
frame');
// add a label
box.addLabel("This is a commentary "+NL+"Multi-line");
box.addToggle("tog1", "A toggle", false);
box.addToggle("tog2", "A second toggle", true);
// close the previous horizontal section
box.endLaycutSection();

box.addField("field1", "Text:", "My text",
               JControlTextField, width);

box.addText("text1", 'Multi-line text: ",
              "Enter the description :'+NL+'line 1"+NL+"line 2
"
```

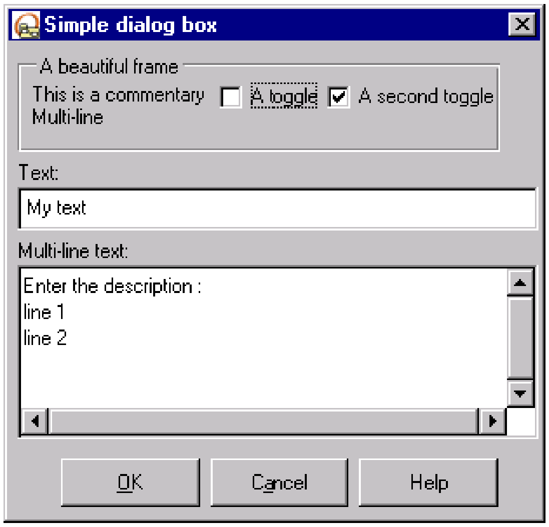
Figure 7-3. Creating a simple modal dialog box

The complete J code is as follows:

```
// create a vertical box
JBox box=createJBox("simpleBoxIdent", "Simple dialog box",
                    JLayoutVertical, false);
String value;
int width=300;
int height=100;

// open a horizontal layout section
box.beginLayoutSection (JLayoutHorizontal,
                        true, "A beautiful frame");

// add a label
box.addLabel("This is a commentary "+NL+"Multi-line");
box.addToggle("tog1", "A toggle", false);
box.addToggle("tog2", "A second toggle", true);

// close the previous horizontal layout section.
box.endLayoutSection();

box.addField("field1", "Text:", "My text",
             JControlTextField, width);

box.addText("text1", "Multi-line text: ",
            "Enter the description:"+NL+"line 1"+NL+"line
2",
            width, height);
// creation of a help button which accesses the file set as
a parameter
box.addHelp("Help",
getObjingPath()+"/help/whatsnew_us.htm");
```

```
if (box.show()==true)
{
  StdOut.write("Ok, Results :"+NL);
  box.getValue("field1", value);
  StdOut.write(value+NL);
  box.getValue("text1", value);
  StdOut.write(value+NL);
  box.getValue("tog1", value);
  StdOut.write(value+NL);
}
else
{
  StdOut.write("Cancel"+NL);
}

// Do not forget to delete the box
box.delete();
```

The result of this code (shown in Figure 7-4) can be tested in your test project, by running the related command from the context menu.



□ A toggle ☑ A second toggle

Text: My text

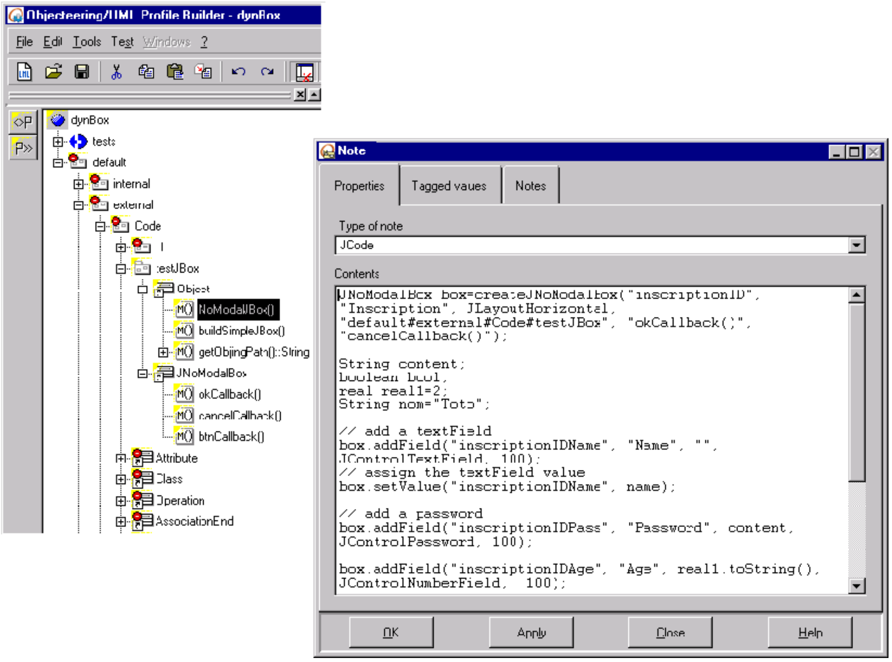
Multi-line text:

Enter the description :
line 1
line 2

Figure 7-4. The newly created simple dialog box

Example 2: Creating a non-modal dialog box

Figure 7-5 shows part of the J code developed in the UML profiling project for the creation of a non-modal dialog box.



Type of note: JCode

Contents

```
JNoModalBox box=createJNoModalBox("inscriptionID",
"Inscription", JLayoutHorizontal,
"default#external#Code#testJBox", "okCallback()",
"cancelCallback()");

String content;
boolean bool;
real real1=2;
String nom="Toto";

// add a textField
box.addField("inscriptionIDName", "Name", "",
JControlTextField, 100);
// assign the textField value
box.setValue("inscriptionIDName", name);

// add a password
box.addField("inscriptionIDPass", "Password", content,
JControlPassword, 100);

box.addField("inscriptionIDAge", "Age", real1.toString(),
JControlNumberField,  100);
```

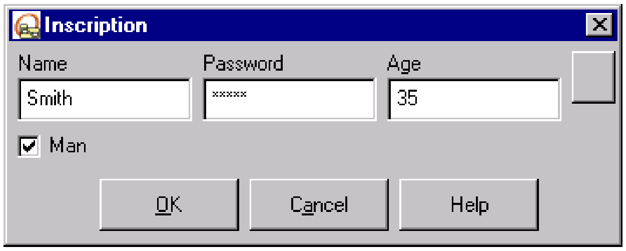
Figure 7-5. Creating a non-modal dialog box

Note: Please note that the three callbacks ("*okCallback*", "*cancelCallback*" and "*btnCallback*") which feature in the J code for the creation of a non-modal dialog box are themselves J methods defined in the "*JNoModalBox*" metaclass reference. Their associated J code can be visualized through the "*JCode*" notes available on each of the J methods in question.

The complete J code is as follows:

```
JNoModalBox box=createJNoModalBox(
                             "inscriptionID",
                             "Inscription",
                             JLayoutHorizontal,
                             "default#external#Code#testJBox"
                             ,
                             "okCallback()",
                             "cancelCallback()");

String content;
String name="Toto";
real real1=2;

// add a textField
box.addField("inscriptionIDName", "Name", "",
             JControlTextField, 100);

// assign the textField value
box.setValue("inscriptionIDName", name);

// add a password
box.addField("inscriptionIDPass", "Password", content,
             JControlPassword, 100);

box.addField("inscriptionIDAge", "Age", real1.toString(),
             JControlNumberField,  100);

// add a button related to a callback (btnCallback)
box.addButtonBitmap("button2",
                     getObjingPath()+"mybmp.bmp",
                    "default#external#Code#testJBox",
                    "btnCallback()");
box.changeColumn(10);
box.addToggle("inscriptionIDMale", "Man", true);
box.addHelp("Help",
getObjingPath()+"/help/whatsnew_us.htm");

// show the box and set consultation mode
box.show(true);
```

The result of this code (shown in Figure 7-6) can be tested in your test project, by running the related command from the context menu.



Inscription

Name: Smith

Password: *****

Age: 35

☑ Man

Figure 7-6. The newly created non-modal dialog box

Callbacks used

The following code is the code for the "*okCallback*" callback:

```
String Name;
String Password;
String Age;

String Man;
StdOut.write ("ok !", NL);

if (getIdent()=="inscriptionID")
{
     getValue("inscriptionIDName", Name);
     getValue("inscriptionIDPass", Password);
     getValue("inscriptionIDAge", Age);
     getValue("inscriptionIDMale", Man);

     if (Man=="true")
     {
          StdOut.write("Mr ");

     }
     else
     {
          StdOut.write("Ms ");
     }
     StdOut.write(Name, "You are", Age, "years old and your
     password is: ", Password,NL);
}

// Do not forget to delete the box
delete();
```

The following code is the code for the "*cancelCallback*" callback:

```
StdOut.write ("Canceled!", NL);

// Do not forget to delete the box
delete();
```

The following code is the code for the "*btnCallback*" callback:

```
String Name;
String Password;
String Age;
String Man;

// create a consultation box (with only an OK button)

JBox box=createJBox("boxInfo", "Infos",
                     JLayoutVertical, true);

getValue("inscriptionIDName",Name);
getValue("inscriptionIDPass",Password);
getValue("inscriptionIDAge",Age);
getValue("inscriptionIDMale", Man);

if (Man=="true")  box.addLabel("Man");

else

box.addLabel("Woman");
box.addLabel("Name:"+Name);
box.addLabel("Password: "+Password);
box.addLabel("Age: "+Age);box.show();

// Do not forget to delete the box
box.delete();
```

Chapter 8: DOM parser usage services

Introduction to the DOM parser

Introduction

XML is considered as being the next important stage in the evolution of the Web. A universal Web exchange format, recommended by the W3C, many standards are now based around XML, so as to facilitate data sharing at a lesser cost.

Objecteering/UML incorporates an XML parser which can be accessed through J programming via a DOM (Document Object Model) API, whose functioning will be presented in this chapter.

A model dedicated to use of the parser

In order to implement the parser from J, users must be familiar with the simplified model providing access to the DOM API of the XML parser. Figure 8-1 presents a class diagram of this model.

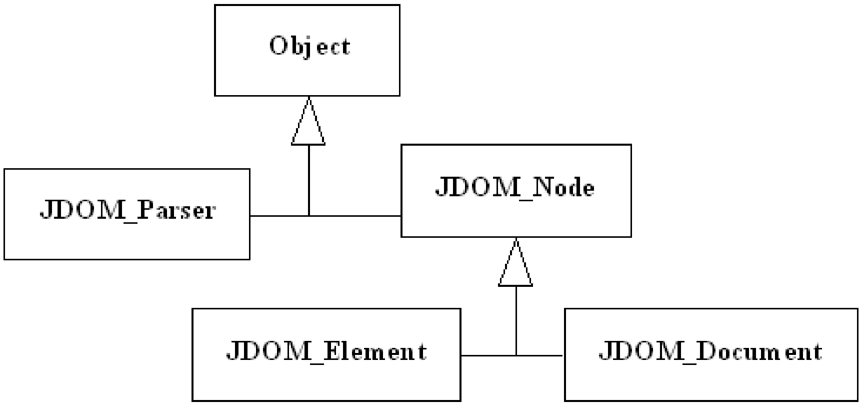


Figure 8-1. Class diagram of the DOM API providing access to the XML parser

Why a simplified model?

Most DOM objects (Attributes, Texts, Comments…) do not provide services different enough from those of a node to justify their existence as separate objects. This means that these types of element can be created from the JDOM_Document (createAttribute(), createComment()…) class, whilst retrieving JDOM_Nodes and not JDOM_Attributes or JDOM_Comments. This does not affect the DOM possibilities of our API.

To illustrate the dynamics of parser use, Figure 8-2 shows a sequence diagram of API use.

Note: To use this model, a J session must be open.

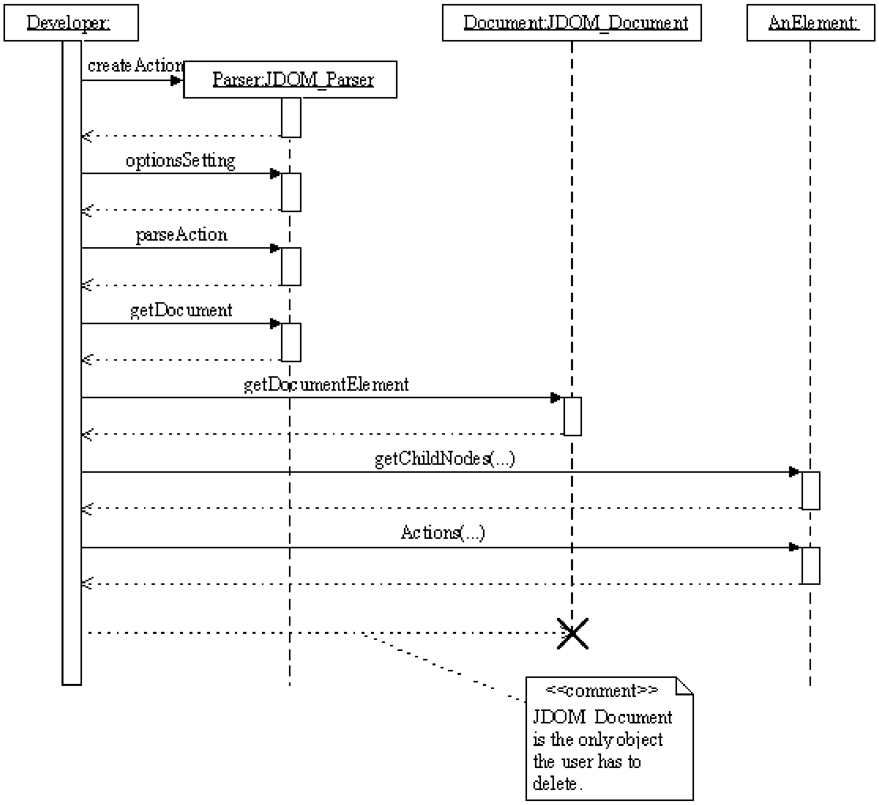


Figure 8-2. Sequence diagram illustrating the use of the API

Memory management

With the exception of JDOM_Document instances, JDOM_Node, JDOM_Parser or JDOM_Element type objects are destroyed at the end of a J session, without intervention on the part of the developer. This makes it important to indicate the reasons for this choice and their implications on the functioning of J programs.

Objects available in J are only proxies to real underlying DOM parser object. These objects, instantiated when an XML file is read or when element creation methods are called from the DOM API provided, are only destroyed when their reference counters return to zero. This means that the destruction of a JDOM_Node element in J does not destroy the associated DOM element, as long as another JDOM_[...] object references it.

To lighten Objecteering/UML's memory load, several objects are automatically destroyed at the end of a J session. This allows the JDOM tree to be destroyed without destroying the underlying DOM tree, and provides the possibility of adding the document to the user data contained in a non-modal dialog box (a window which exists after the end of a J session).

This allows the developer to access tree elements during the life of the dialog box, but implies the risk of not being able to free up the entire tree if destruction is forgotten.

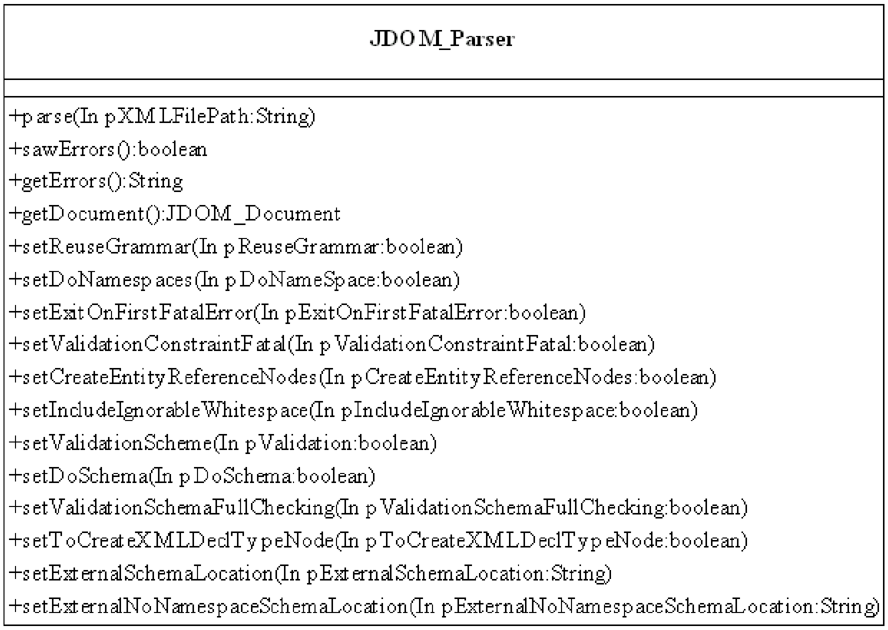
The memory management rule is, therefore, as follows:

A JDOM_Node, JDOM_Parser or JDOM_Element object must never be saved in a non-modal dialog box, as these objects are destroyed at the end of a J session.

The desired JDOM_Document(s) must be attached to the non-modal dialog boxes and destroyed when these dialog boxes are destroyed, or destroyed manually before the end of the J session, if you only wish to handle documents locally.

JDOM_Parser methods

JDOM_Parser methods



JDOM_Parser

+parse(In pXMLFilePath:String)
+sawErrors():boolean
+getErrors():String
+getDocument():JDOM_Document
+setReuseGrammar(In pReuseGrammar:boolean)
+setDoNamespaces(In pDoNameSpace:boolean)
+setExitOnFirstFatalError(In pExitOnFirstFatalError:boolean)
+setValidationConstraintFatal(In pValidationConstraintFatal:boolean)
+setCreateEntityReferenceNodes(In pCreateEntityReferenceNodes:boolean)
+setIncludeIgnorableWhitespace(In pIncludeIgnorableWhitespace:boolean)
+setValidationScheme(In pValidation:boolean)
+setDoSchema(In pDoSchema:boolean)
+setValidationSchemaFullChecking(In pValidationSchemaFullChecking:boolean)
+setToCreateXMLDeclTypeNode(In pToCreateXMLDeclTypeNode:boolean)
+setExternalSchemaLocation(In pExternalSchemaLocation:String)
+setExternalNoNamespaceSchemaLocation(In pExternalNoNamespaceSchemaLocation:String)

Figure 8-3. The JDOM_Parser metaclass

The following methods are provided by the JDOM_Parser metaclass:

```
parse (pXMLFilePath : in String);
```

This method is used to run the parsing of an XML file whose complete path is provided as an incoming parameter.

Parser options must be set before running this operation.

```
sawErrors(): boolean;
```

This method is used to check if the parser is in an incorrect state or not.

```
getErrors() : String;
```

This method retrieves the complete string of errors encountered by the parser.

```
getDocument(): JDOM_Document;
```

This method returns the JDOM_Document object representing the root of the document tree. This object provides primary access to the document's data.

```
setReuseGrammar(pReuseGrammar : in boolean);
```

This method does not exist on the DOM API. It indicates whether or not the existing grammar should be reused for the next parsing run. If true, there can be no internal subsets.

```
setDoNamespaces (pDoNameSpace : in boolean);
```

This method enables or disables the parser's namespace processing. If set to true, the parser starts enforcing all the constraints and rules specified by the NameSpace specification. By default, this is set to false.

```
setExitOnFirstFatalError (pExitOnFirstFatalError :
                          in boolean);
```

This method changes the behavior of the parser with regard to the first fatal error. If set to true, the parser will exit at the first fatal error. If set to false, then it will report the error and continue processing. By default, this is set to true.

```
setValidationConstraintFatal
(pValidationConstraintFatal : in boolean);
```

This changes the processing mode of a validation constraint. If set to true, validation constraints which are not respected are considered as being fatal errors. If set to true, the error is reported in the normal way. By default, this is set to false.

```
setCreateEntityReferenceNodes
(pCreateEntityReferenceNodes : in boolean);
```

This method allows the user to specify whether the parser should create entity reference nodes in the JDOM tree being produced. When the "create'" flag is true, the JDOM tree contains entity reference nodes. When the "create" flag is false, no entity reference nodes are included in the JDOM tree.

The replacement text of the entity is included in either case, either as a child of the Entity Reference node or in place at the location of the reference.

```
setIncludeIgnorableWhitespace
(pIncludeIgnorableWhitespace : in boolean);
```

This method allows the user to specify whether a "validating" parser should include ignorable whitespaces as text nodes. It has no effect on non-validating parsers, which always include non-markup text.

If set to true, ignorable whitespaces will be added to the JDOM tree as text nodes. The *isIgnorableWhitespace* method available on a JDOM_Node representing a text node will return true for those text nodes only.

If set to false, all ignorable whitespaces will be discarded and no text node added to the JDOM tree.

The use of this flag can clash with the use of the "*xml:space*" attribute. Similarly, this flag can clash with the use of the "*preserve*" keyword in diagrams.

By default, this is set to true.

```
setValidationScheme(pValidation : in boolean);
```

This sets the validation mode of the parser. When set to false, validation is de-activated, whilst when set to true, validation is activated. By default, this is set to true.

```
setDoSchema (pDoSchema : in boolean);
```

This method enables or disables the parser's schema processing. By default, this is set to false.

```
setValidationSchemaFullChecking
(pValidationSchemaFullChecking : in boolean);
```

This method activates or deactivates the exhaustive validation of constraints which can be applied to schemas. Exhaustive schema validation can be greedy in terms of memory or runtime.

If this option is set to false, only basic rules are checked. Currently, particle unique attribution constraint checking and derivation restriction checking are managed by this option.

This option is ignored if the parser does not validate schemas.

By default, this is set to false.

```
setToCreateXMLDeclTypeNode
(pToCreateXMLDeclTypeNode : in boolean);
```

XMLDecl-type nodes can be attached to the JDOM tree if this option has been activated. By default, this is set to false.

```
setExternalSchemaLocation
(pExternalSchemaLocation : in string);
```

This method redefines the values of *schemaLocation* attributes belonging to the document or the "import" element. The new location is given by this method.

Only the last call is recorded.

The syntax to be used to the same as for the document's *schemaLocation* attribute.

The user can specify more than one XML Schema in the list.

```
setExternalNoNamespaceSchemaLocation
(pExternalNoNamespaceSchemaLocation : in string);
```

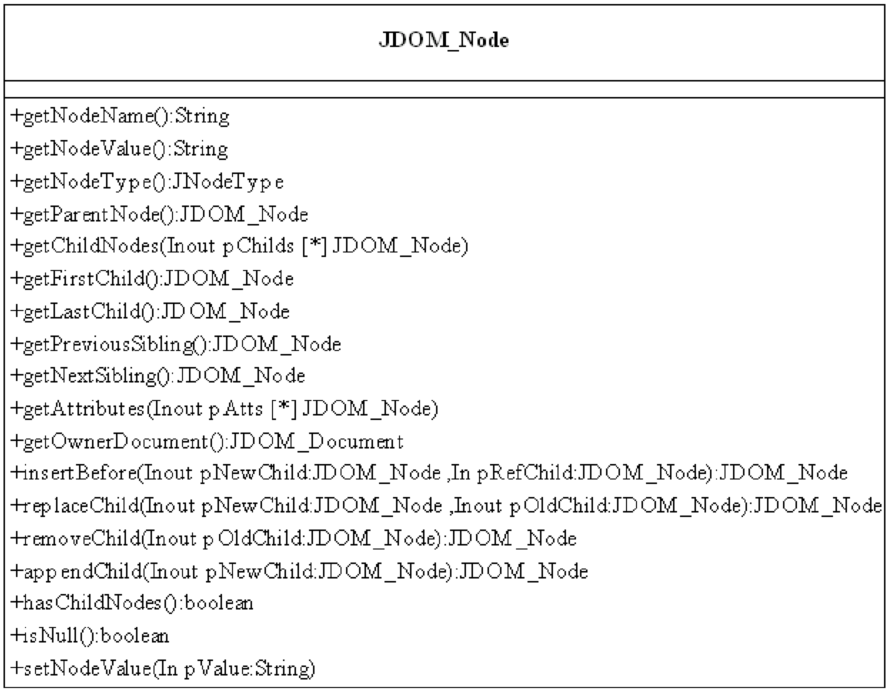
This redefines the document's *noNamespaceSchemaLocation* attribute externally.

Only the last call is recorded.

The syntax to be used is the same as for the document's *noNamespaceSchemaLocation* attribute.

JDOM_Node methods

JDOM_Node methods



```
JDOM_Node

+getNodeName():String
+getNodeValue():String
+getNodeType():JNodeType
+getParentNode():JDOM_Node
+getChildNodes(Inout pChilds [*] JDOM_Node)
+getFirstChild():JDOM_Node
+getLastChild():JDOM_Node
+getPreviousSibling():JDOM_Node
+getNextSibling():JDOM_Node
+getAttributes(Inout pAtts [*] JDOM_Node)
+getOwnerDocument():JDOM_Document
+insertBefore(Inout pNewChild:JDOM_Node ,In pRefChild:JDOM_Node):JDOM_Node
+replaceChild(Inout pNewChild:JDOM_Node ,Inout pOldChild:JDOM_Node):JDOM_Node
+removeChild(Inout pOldChild:JDOM_Node):JDOM_Node
+appendChild(Inout pNewChild:JDOM_Node):JDOM_Node
+hasChildNodes():boolean
+isNull():boolean
+setNodeValue(In pValue:String)
```

Figure 8-4. The JDOM_Node metaclass

The following methods are provided by the JDOM_Node metaclass:

`getNodeName () : String;`
This method returns the name of the node according to its type.

`getNodeValue() : String;`
This method returns the value of the node according to its type.

`getNodeType() : JNodeType;`
This method returns the type of the node. This type can be one of the following enumerate values:

- JNODE_NONE
- ELEMENT_JNODE
- ATTRIBUTE_JNODE
- TEXT_JNODE
- CDATA_SECTION_JNODE
- ENTITY_REFERENCE_JNODE
- ENTITY_JNODE
- PROCESSING_INSTRUCTION_JNODE
- COMMENT_JNODE
- DOCUMENT_JNODE
- DOCUMENT_TYPE_JNODE
- DOCUMENT_FRAGMENT_JNODE
- NOTATION_JNODE
- XML_DECL_JNODE

`getParentNode () : JDOM_Node;`
This returns the parent node if it has been defined or null if this is not the case.

`getChildNodes (Inout pChilds [*] JDOM_Node) ;`

This method returns the set of child elements of the current node in the pChilds list, and returns null if there are no child elements.

Note: For optimization reasons, the set is not returned but is instead passed as the InOut parameter. In this respect, this method differs from the standard DOM API.

`getFirstChild () : JDOM_Node ;`

This returns the first child node of the current node, and returns null if no child nodes exists.

`getLastChild () : JDOM_Node;`

This returns the last child node of the current node, and returns null if no child nodes exist.

`getPreviousSibling () JDOM_Node;`

This returns the sibling node preceding the current node, and returns empty if no sibling nodes exist.

`getNextSibling () : JDOM_Node;`

This returns the next sibling node of the current node, and returns empty if no sibling nodes exist.

`getAttributes (InOut pAtts [*] JDOM_Node) ;`

This returns the list of the current node's attributes if a JDOM_Element is concerned, and returns an empty list if this is not the case.

Note: For optimization reasons, the set is not returned but is instead passed as the InOut parameter. In this respect, this method differs from the standard DOM API.

`getOwnerDocument () : JDOM_Document ;`

This returns the node definition document. If a JDOM_Document of a JDOM_Node linked to a DOM_DocumentTree is concerned, this returns null.

```
cloneNode (In pDeep : boolean) : JDOM_Node;
```

Please refer to *importNode* on the JDOM_Document class.

```
insertBefore(Inout pNewChild: JDOM_Node, In pRefChild :
JDOM_Node):JDOM_Node;
```

This inserts the *pNewChild* node before the existing *pRefChild* node. If *pRefChild* is null, *pNewChild* is inserted after the last child of the current node.

```
replaceChild(Inout pNewChild: JDOM_Node, InOut
pOldChild : JDOM_Node):JDOM_Node;
```

This replaces *pOldChild* by *pNewChild*. If *pNewChild* is already present in the JDOM tree, it is first removed from it.

```
removeChild (Inout pOldChild : JDOM_Node) : JDOM_Node;
```

This removes *pOldChild* from the list of the current node's children and returns the removed node.

```
appendChild (Inout pNewChild : JDOM_Node) : JDOM_Node;
```

This adds *pNewChild* after the last of the current node's children and returns the added node.

If *pNewChild* is already in the JDOM tree, it is first removed from it.

```
hasChild () : boolean;
```

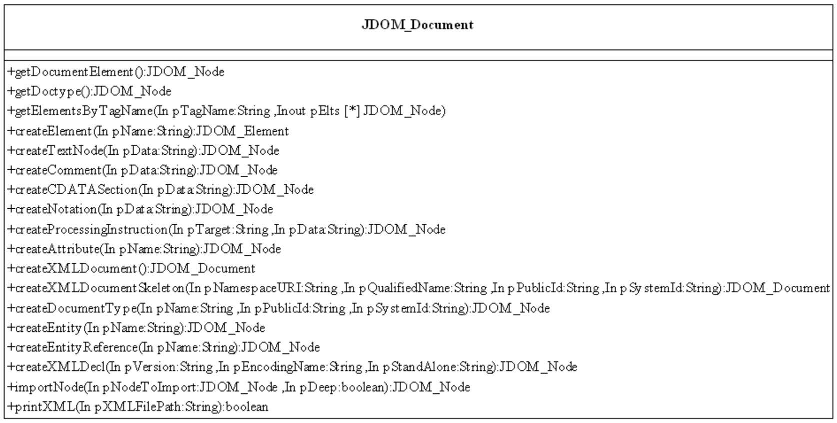
This indicates whether the current node has a child or not.

```
setNodeValue (In pValue : String);
```

This sets the value of the node for each node which accepts a value.

JDOM_Document methods

JDOM_Document methods



```
JDOM_Document

+getDocumentElement():JDOM_Node
+getDoctype():JDOM_Node
+getElementsByTagName(In pTagName:String ,Inout pElts [*] JDOM_Node)
+createElement(In pName:String):JDOM_Element
+createTextNode(In pData:String):JDOM_Node
+createComment(In pData:String):JDOM_Node
+createCDATASection(In pData:String):JDOM_Node
+createNotation(In pData:String):JDOM_Node
+createProcessingInstruction(In pTarget:String ,In pData:String):JDOM_Node
+createAttribute(In pName:String):JDOM_Node
+createXMLDocument():JDOM_Document
+createXMLDocumentSkeleton(In pNamespaceURI:String ,In pQualifiedName:String ,In pPublicId:String ,In pSystemId:String):JDOM_Document
+createDocumentType(In pName:String ,In pPublicId:String ,In pSystemId:String):JDOM_Node
+createEntity(In pName:String):JDOM_Node
+createEntityReference(In pName:String):JDOM_Node
+createXMLDecl(In pVersion:String ,In pEncodingName:String ,In pStandAlone:String):JDOM_Node
+importNode(In pNodeToImport:JDOM_Node ,In pDeep:boolean):JDOM_Node
+printXML(In pXMLFilePath:String):boolean
```

Figure 8-5. The JDOM_Document metaclass

The following methods are provided by the JDOM_Document metaclass:

```
getDocumentElement () : JDOM_Node ;
```
This returns a reference to the XML document root element.

```
getDoctype() : JDOM_Node;
```
This returns the *Doctype* of the XML document, or null if this is not present.

```
getElementsByTagName(In pTagName : String,
                     Inout pElts [*] JDOM_Node);
```
This returns elements named *pTagName* which are contained in the XML document in the *pElts* list.

Note: For optimization reasons, the set is not returned but is instead passed as the InOut parameter. In this respect, this method differs from the standard DOM API.

Unlike the DOM API, the returned list is not dynamic, but is rather a copy of the tree at a given moment in time.

```
createElement (In pName : String) : JDOM_Element;
```
This creates an element named *pName*.

```
createTextNode (In pData : String) : JDOM_Node;
```
This creates a text using the *pData* data.

```
createComment (In pData : String) : JDOM_Node;
```
This creates a comment using the *pData* data.

```
createCDATASection (In pData : String) : JDOM_Node;
```
This creates a *CDATASection* using the *pData* data.

```
createNotation (In pData : String) : JDOM_Node;
```
This creates a notation using the *pData* data.

```
createProcessingInstruction  (In pTarget : String,
                              In pData : String):
                              JDOM_Node;
```

This creates a "*Processing Instruction*" using the desired information.

```
createAttribute (In pName : String) : JDOM_Node;
```

This creates an attribute named *pName*.

```
createXMLDocument () : JDOM_Document;
```

This creates a new empty document (this replaces createDocument () of the DOM API).

```
createXMLDocumentSkeleton (In pNamespaceURI : String,
                           In  pQualifiedName : String,
                            In pPublicId : String,
                           In pSystemId : String) :
                           JDOM_Document;
```

This creates a document skeleton containing a *DocumentType* node, as well as an empty root element.

```
createDocumentType (In pName : String,
                    In pPublicId : String,
                    In pSystemId : String):
                    JDOM_Node;
```

This creates a *DocumentType* node.

```
createEntity (In pName : String) : JDOM_Node;
```

This creates an entity named *pName*.

```
createEntityReference (In pName : String) : JDOM_Node;
```

This creates an entity reference named *pName*.

```
createXMLDecl(In pVersion:String,
              In pEncoding:String,
              In pStandAlone:String) :JDOM_Node;
```

This creates an XML declaration with the transmitted data.

```
importNode(Inout pNodeToImport: JDOM_Node,
           In pDeep : JDOM_Node):JDOM_Node;
```

This method should be used instead of the *cloneNode* method, available on JDOM_Node.

The import runs a duplication of the node and attaches it to the method's application document. *pDeep* indicates that you wish to copy all the sub-elements of the node which is to be imported.

For example:

```
lElement = lDoc.createElement("AnElement") ;
lElement2= lDoc2.importNode(lElement,true);
//Copying the element named "AnElement" with its
sub-elements and affecting it to lElement2
lDoc2.getDocumentElement().appendChild(lElement2) ;
//Attaching the copy to the "root" element of the
lDoc2 document.
```

```
printXML (In pXMLFilePath : String) : Boolean;
```

This saves the XML file, giving it the complete path of the XML file.

JDOM_Element methods

JDOM_Element methods

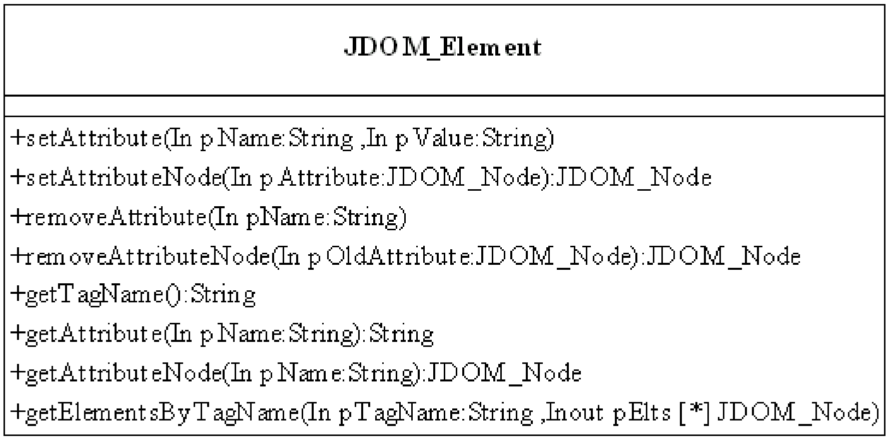

JDOM_Element
+setAttribute(In pName:String ,In pValue:String)
+setAttributeNode(In pAttribute:JDOM_Node):JDOM_Node
+removeAttribute(In pName:String)
+removeAttributeNode(In pOldAttribute:JDOM_Node):JDOM_Node
+getTagName():String
+getAttribute(In pName:String):String
+getAttributeNode(In pName:String):JDOM_Node
+getElementsByTagName(In pTagName:String ,Inout pElts [*] JDOM_Node)

Figure 8-6. The JDOM_Element metaclass

The following methods are provided by the JDOM_Element metaclass:

```
setAttribute (In pName : String, In pValue : String) ;
```

If the attribute named *pName* exists in the current element, this sets the value of the attribute to *pValue*. If this is not the case, the attribute named *pName* is created under the current element with *pValue* as its value.

Note: *pValue* must be a simple string and must not contain elements such as entity references. To create such an attribute, you must first create the attribute and then attached an entity reference to it, before adding it to the element using a *setAttributenode*.

```
setAttributeNode (Inout pAttribute : JDOM_Node)
             : JDOM_Node ;
```

This adds a new attribute to the element. If an attribute of the same name as *pAttribute->getNodeName()* already exists, it is simply replaced and returned. Otherwise, this returns null.

```
removeAttribute (In pName : String) ;
```

This removes the attribute named *pName* from the current element.

```
removeAttributeNode (InOut pOldAttribute : JDOM_Node) :
                 JDOM_Node;
```

This removes *pOldAttribute* from the element's attributes and returns this same element.

```
getTagName() : String;
```

This returns the name of the current element.

```
getAttribute (In pName : String) : String;
```

This returns the value of the attribute named *pName* of the current element. This is null if the attribute has no value defined or no default value.

```
getAttributeNode (In pName : String) : JDOM_Node;
```

This returns an attribute by its name or returns null if the attribute is not under the element.

```
getElementsByTagName(In pTagName : String,
                     Inout pElts [*] JDOM_Node);
```

This returns the elements named *pTagName* encountered from the current element onwards in the *pElts* list.

Note: For optimization reasons, the set is not returned but is instead passed as the InOut parameter. In this respect, this method differs from the standard DOM API.

Unlike the DOM API, the returned list is not dynamic, but is rather a copy of the tree at a given moment in time.

Example

Code of a method referenced by a module command

```
JDOM_Parser lParser;
JDOM_Node lRoot;
JDOM_Document lDoc;
String lName;

sessionBegin("loadXMLFile",true);

lParser = JDOM_Parser.create();
lParser.setValidationScheme(true);
lParser.setDoNamespaces(true);
lParser.setExitOnFirstFatalError(false);
lParser.setValidationConstraintFatal(false);
lParser.setIncludeIgnorableWhitespace(true);
lParser.parse("/tmp/MyFile.xml");

if (lParser.sawErrors() == true)
{
   StdOut.write("Status parser 1 = ",
   lParser.getErrors(),NL);
}

lDoc = lParser.getDocument();
if (notVoid(lDoc))
{
  StdOut.write(" --- Document successfully
                      retrieved!",NL);
  lRoot = lDoc.getDocumentElement();
  if (notVoid(lRoot))
{
  lName = lRoot.getNodeName();
  StdOut.write(" --- Retrieving the root element ~"",
  lName, "~" successful !",NL);

  lRoot.<recursiveGetChilds();

}
else
```

```
  StdOut.write(" --- Impossible to retrieve the root
                    element!",NL);
  }

  lDoc.printXML("/tmp/modules.xml");
  lDoc.delete() ;
  sessionEnd();
```

Code of the recursiveGetChilds method (defined on the JDOM_Node metaclass)

```
JDOM_Node[] lChilds;
JDOM_Node[] lAtts;

StdOut.write("----------------------------------------
------------------",NL);

getChildNodes(lChilds);
lChilds
{
  StdOut.write(" --- Retrieving the element ~"",
  getNodeName(), "~" :",NL);
  StdOut.write(" --- Retrieving the composed
  element ~"", getParentNode().getNodeName(), "~"
:",NL);
  if (getNodeType() == JNODE_NONE)
{
StdOut.write(" --- Text = ", getNodeValue(),NL);
}
  getAttributes(lAtts);
  lAtts
  {
    StdOut.write ("Attribute = ", getNodeName(),
    " Value = ", getNodeValue(), NL);
    setNodeValue(getNodeValue()+"_NewValue");
  }
  lAtts.clear();
  recursiveGetChilds();
}
```

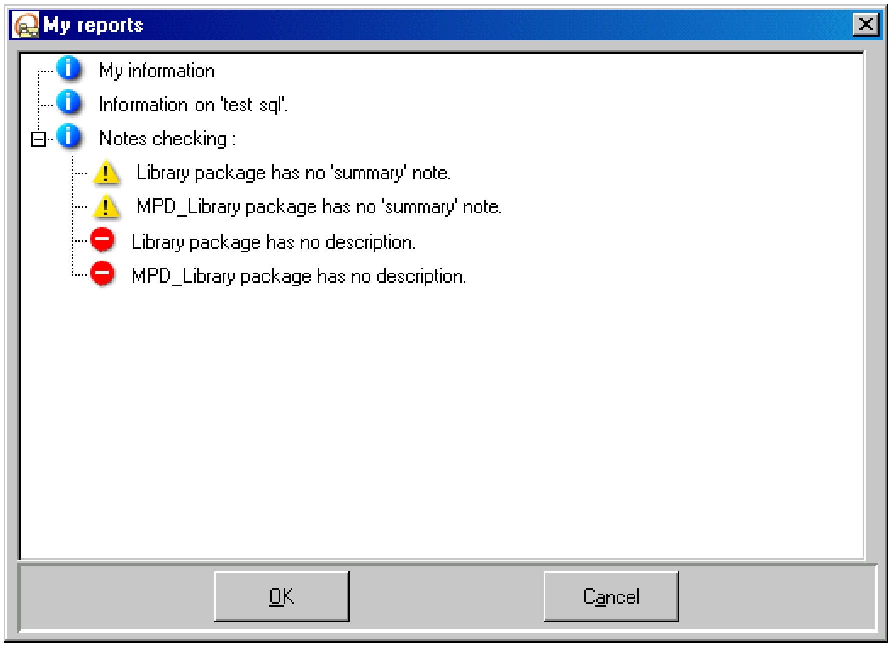
Chapter 9: J report window services

Overview of the J report window

Introduction

When a module developer wishes to "report" the results of a processing operation to a user , he can create a J report window.

A J report window is a non-modal dialog box, containing a hierarchical view of a set of report lines. Each line has an icon which indicates its type, for example information, warning or error (as shown in Figure 9-1 below).



My information

Information on 'test sql'.

Notes checking :

- Library package has no 'summary' note.
- MPD_Library package has no 'summary' note.
- Library package has no description.
- MPD_Library package has no description.

Figure 9-1. A J report window

Using the J report window

Important information on using the J report window!

All the methods used to handle J report windows are stored in the *default#Report* profile. This means that every time you call a method, you must prefix the name of the method with `#Report#`.

Creating a report window

```
JNoModalBox Object:createReportBox (in String title)
```

This method creates a J report window, and destroys the former J report window, if this already exists.

This method does not display the newly-created J report window. To display it, you should "manually" call the following standard method on the J report window:

```
show (in boolean objecteeringInConsultMode)
```

Example:

```
JNoModalBox box = #Report#createReportBox
                  ( "My report box") ;
```

```
JNoModalBox Object:createNewReportBox (in String title)
```

This creates a new J report window, even if one already exists.

This method does not display the newly-created J report window. To display it, you should "manually" call the following standard method on the J report window:

```
show (in boolean objecteeringInConsultMode)
```

```
JNoModalBox Object:createReportBoxWithIdent
                      (in String title,
                       in String pIdentifier);
```

This creates a new J report window whose identifier is used in the "*pIdentifier*" parameter. If a J report window with this identifier already exists, it is destroyed.

This method does not display the newly-created J report window. To display it, you should "manually" call the following standard method on the J report window:

```
show (in boolean objecteeringInConsultMode)
```

Creating and displaying a report window

```
JNoModalBox Object:createAndShowReportBox (in String
title)
```

This method creates a J report window, destroys the former J report window (if this already exists), and displays the newly-created J report window.

```
JNoModalBox Object:createAndShowNewReportBox (in String
title)
```

This creates a new J report window, even if one already exists, and displays it.

```
JNoModalBox Object:createAndShowReportBoxWithIdent
                      (in String title,
                       in String pIdentifier);
```

This creates and displays a new J report window whose identifier is used in the "*pIdentifier*" parameter. If a J report window with this identifier already exists, it is destroyed.

Adding report lines

Each method below enables the developer to add lines to a J report window. These methods all use the following parameters:

- *String type*: This gives a string determining which icon should be used for the report line. This can be the path to a bitmap or one of the predefined strings shown in the table below:

The ... type	has the ... bitmap
info	ℹ
warning	⚠
error	⛔
modification	✏
addition	➕
deletion	✖

- *String title*: This indicates the title of the report line.
- *Object link*: This specifies a browsable object to which the report line is linked. When the user double-clicks on the report line in question, the linked object will be selected in the explorer.

Adding a report line at the root

```
JTreeItem Object:linkToReport ( in string type ,
                                in string title )
```

This adds a report line to the root of the report window and links it to the current (*this*) object.

Example:

```
#Report#linkToReport("info", "Information on
                     '"+Name+"'.") ;
```

```
JTreeItem JTree:addReport ( in string type ,
                            in string title ,
                            in Object link )
```

This adds a report line to the root of the report window and links it to the *link* object.

Example:

```
JTreeItem aReportItem = box.#Report#addReport
                        ("warning" ,
                         "A warning" ,
                         this);
```

Adding a report line under another report line

```
JTreeItem Object:linkToReportUnder
                            (in string parentTitle ,
                             in string type ,
                             in string title  )
```

This adds a report line to the J report box and links it to the current object. The new report line is added under the "*parentTitle*" access path line. Each access path line should be separated by a "/".

Example:

To insert a new report line under the "*B*" report line, which itself is under the "*A*" report line, proceed as follows:

```
#Report#linkToReportUnder("A/B","info", "a title");
```

```
JTreeItem JTreeItem:addReport ( in string type
                              , in string title
                              , in Object link )
```

This adds a report line under another report line.

Customizing the J report window

Introduction

The J report window can be customized, by creating a child profile under the *default#Report* profile.

Through customization, it is possible to:

- add fields to the top and the bottom of the J report window
- define new types of report line and give them an icon
- redefine the behavior of the J report window when the user selects or double-clicks on one of the report lines

Adding fields to the J report window

To add fields to the J report window, the "*initBoxHeader* ()" and "*initBoxBottom*" protected abstract methods can be overloaded:

All the methods available on the JSet and JNoModalBox metaclasses can be used.

Defining new types of report line

To define new types of report line, the "*getBitmapFromType (in string type)*" method can be overloaded.

This method returns the access path to the bitmap to be used, from a report line type.

Selecting and double-clicking

The following methods can be redefined:

`JTreeItem::onSelect()`

Call back called when a line is selected.

`JTreeItem::onAction()`

Call back called when a line is double-clicked upon.

`Object::onSelect()`

Method called by "*JTreeItem::onSelect ()*" on the linked object. By default, this does nothing.

`Object::onAction()`

Method called by "*JTreeItem::onAction ()*" on the linked object. By default, this select the object in the explorer.

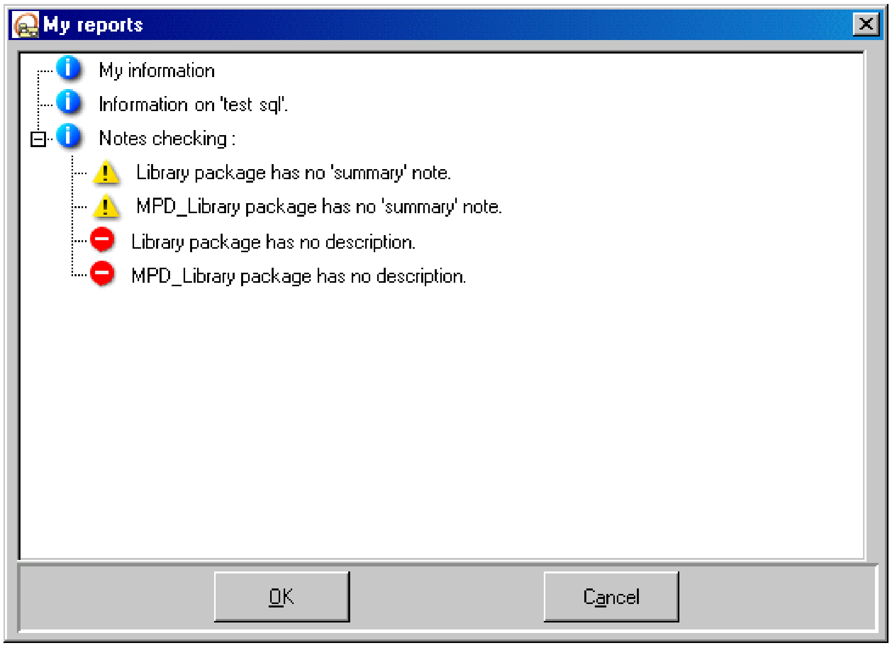
By default, "*JTreeItem::onSelect()*" calls "*Object::onSelect()*" and "*JTreeItem::onAction ()*" calls "*Object::onAction()*" on the linked object.

J report window example

You can test this example by creating a macro using the "*Macros*" module.

```
// Creating the report box
JNoModalBox box = #Report#createReportBox
                  ("My reports");

// Adding the "My information" report line to the root
JTreeItem aReportItem = box.#Report#addReport
                       ("info",
                        "My informations",
                         this);

getRootPackage()
{
   // Adding a line linked to the root package
   #Report#linkToReport("info",
                        "Information on '"+Name+"'.") ;

   // Adding "My warnings" to the root
   aReportItem = box.#Report#addReport
                ("info",
                 "Notes checking :",
                  this);
   // Finding all packages which don't have any
     "summary" notes
   OwnedElementPackage.<select
     ( void(getAllNotesOfType("summary")))
   {
      // Adding a warning line under the "Notes
         checking :" line
      #Report#linkToReportUnder("Notes checking :",
                                "warning",
                                Name+" package has no
                                'summary' note.");
   }

   // Finding all packages which don't have any
      "description" notes
   OwnedElementPackage.<select
     ( void(getAllNotesOfType("description")))
   {
      // Adding an error line under the aReportLine
         line
      aReportItem.#Report#addReport
         ("error", Name+" package has no description.",
           this);
   }
}
```

This code will display a J report window like that shown in Figure 9-2.



My reports

- My information
- Information on 'test sql'.
- Notes checking :
  - Library package has no 'summary' note.
  - MPD_Library package has no 'summary' note.
  - Library package has no description.
  - MPD_Library package has no description.

Figure 9-2. The J report window you just created

If you double-click on one of the leaf nodes, the corresponding package will be selected in the explorer.

Chapter 10: Properties editor services

Overview of properties editor services

Metaclasses used in properties editor management

The following diagram (Figure 10-1) illustrates the different metaclasses used in the management of the properties editor.

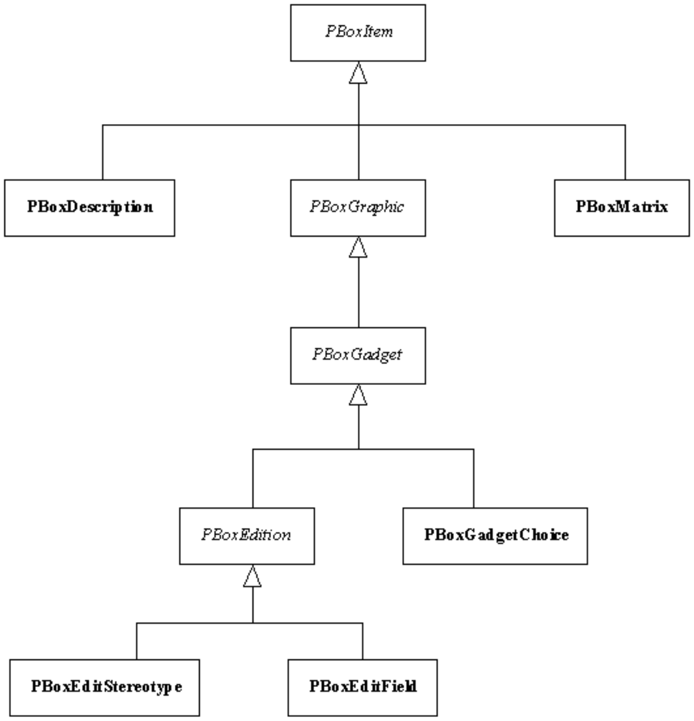


Figure 10-1. Class diagram of the different metaclasses used in the management of the properties editor

The properties editor services provided by these metaclasses are detailed further on in this chapter.

The PBoxDescription metaclass

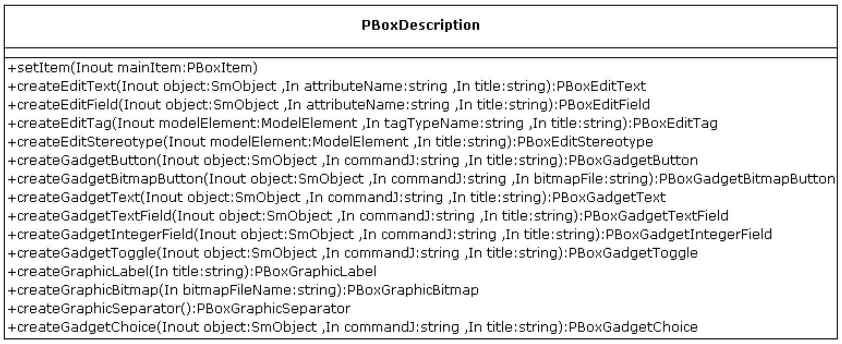

PBoxDescription
+setItem(Inout mainItem:PBoxItem)
+createEditText(Inout object:SmObject ,In attributeName:string ,In title:string):PBoxEditText
+createEditField(Inout object:SmObject ,In attributeName:string ,In title:string):PBoxEditField
+createEditTag(Inout modelElement:ModelElement ,In tagTypeName:string ,In title:string):PBoxEditTag
+createEditStereotype(Inout modelElement:ModelElement ,In title:string):PBoxEditStereotype
+createGadgetButton(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetButton
+createGadgetBitmapButton(Inout object:SmObject ,In commandJ:string ,In bitmapFile:string):PBoxGadgetBitmapButton
+createGadgetText(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetText
+createGadgetTextField(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetTextField
+createGadgetIntegerField(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetIntegerField
+createGadgetToggle(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetToggle
+createGraphicLabel(In title:string):PBoxGraphicLabel
+createGraphicBitmap(In bitmapFileName:string):PBoxGraphicBitmap
+createGraphicSeparator():PBoxGraphicSeparator
+createGadgetChoice(Inout object:SmObject ,In commandJ:string ,In title:string):PBoxGadgetChoice

Figure 10-2. The "*PBoxDescription*" metaclass

For details on the following services, please refer to the "*Creating fields*" section in the current chapter of this user guide:

- "*createEditField*"
- "*createEditStereotype*"
- "*createEditTag*"
- "*createEditText*"
- "*createGraphicLabel*"
- "*createGraphicBitmap*"
- "*createGraphicSeparator*"
- "*createGadgetButton*"
- "*createGadgetBitmapButton*"
- "*createGadgetChoice*"
- "*createGadgetToggle*"

For details on the "*setItem*" service, please refer to the "*Managing field positioning using matrices*" section in the current chapter of this user guide.

The PBoxMatrix metaclass

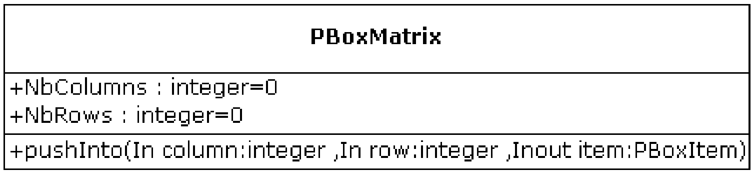

PBoxMatrix
+NbColumns : integer=0
+NbRows : integer=0
+pushInto(In column:integer ,In row:integer ,Inout item:PBoxItem)

Figure 10-3. The "*PBoxMatrix*" metaclass

For details on the "*pushInto*" service, please refer to the "*Managing field positioning using matrices*" section in the current chapter of this user guide.

Initializing the properties editor

Displaying new tabs in the properties editor

The appearance of a new tab in the properties editor is requested in:

```
Object:moduleSelect
```

and

```
Object:moduleStart
```

using the following service:

```
addPropertiesPage(String pageName, String profileName);
```

For example:

```
addPropertiesPage("My page",
"default#external#Code#myProfile#PropertiesBox");
```

Removing tabs from the properties editor

The removal of a tab from the properties editor is requested in:

```
Object:moduleStop()
```

and

```
Object:moduleUnselect()
```

in the module's installation profile. The removal of a tab is carried out using the following service:

```
removePropertiesPage(String pageName);
```

For example:

```
removePropertiesPage("My page");
```

Displaying the contents of the properties editor

The properties editor systematically runs the "*initializePropertiesBox*" method within the given profile, when "*addPropertiesPage*" service is called. This method is called on the element selected in the explorer or in a graphic editor, and has the following syntax:

```
initializePropertiesBox(inout PBoxDescription box)
```

This method's code is usually the following:

```
PBoxMatrix matrix;
// the matrix is the equivalent of beginLayoutSection
   on JBoxes

sessionBegin("My properties page initialization",
             true);
// persistent session

matrix = box.createMatrix("");
// creation of a matrix

initMatrix(box, matrix);
// initialization of the matrix, see further on for
   details

box.setItem(matrix);
// affectation of the matrix to the editor

sessionEnd();
// end of a session
```

In this case, we are going to start by creating a matrix, using the "*createMatrix*" method.

All the work is delegated to the "*initMatrix*" method, which will be defined on all the metaclasses for which we require a different display. Its syntax is usually the following:

```
void initMatrix(inout PBoxDescription box,
                inout PBoxMatrix matrix)
```

The last step is to affect the newly-created matrix to the properties editor, using the "*setItem*" method. Since the properties editor itself can only contain one single item, a matrix containing all the elements to be displayed is always affected to it.

Example of the contents of the "initMatrix" method

```
void Class:initMatrix(inout PBoxDescription box,
                      inout PBoxMatrix matrix)
{
PBoxItem item; // items to be added to the matrix

// Addition of a label
item = box.createGraphicLabel("My_label");
matrix.pushInto(0, matrix.NbRows(), item);

// Addition of a group of radio buttons
item = box.createGadgetChoice
       (this,"Persistence", "label_Persistence");
item.setRadio(true);
        // true => radio buttons, false ==> combo box
item.appendLabel("label_Transient");
item.appendLabel("label_Persistent");
item.appendLabel("label_Undefined");

// Addition of the item at the bottom of the matrix
matrix.pushInto(0, matrix.NbRows(), item);

// Addition of the management of a tagged value type
item = box.createEditTag
       (this,"sqlName","label_SQL_Name");
matrix.pushInto(0, matrix.NbRows(), item);


// Addition of a tickbox
item = box.createGadgetToggle(this,"Null","Not null");
matrix.pushInto(0, matrix.NbRows(), item);
// Addition of a simple text field
item = box.createEditField
       (this,"TypeConstraint","String_size");
matrix.pushInto(0, matrix.NbRows(), item);

// Addition of a multi-line text field
item = box.createEditText
       (this,"Check","Check_constraint");
matrix.pushInto(0, matrix.NbRows(), item);
}
```

Managing field positioning using matrices

A matrix is a zone broken down into lines and columns, and functions a little like an Excel spreadsheet. Items are positioned and aligned in one of the matrix cells.

A matrix is created using the following method:

```
PBoxMatrix PBoxDescription:createMatrix
          (in String title);
```

This method creates a matrix zone, "*PBoxMatrix*", in which graphic gadgets are positioned.

The following service is used to position an item in a cell:

```
PBoxMatrix:pushInto(in integer column,
                    in integer row,
                    inout PBoxItem item);
```

Column and line numbers start at zero.

A cell can only contain one item.

The "*NbRows()*" and "*NbColumns()*" methods are used to find out the current number of lines and columns in the matrix. If you run:

```
matrix.pushInto(0, matrix.NbRows(), item);
```

the item is automatically placed on the first free line of the matrix, in the first column.

The main matrix must be affected to the properties editor, using the following method:

```
PBoxDescription:setItem(inout PBoxItem mainItem);
```

This method is used to associate a specific item to the properties editor. There can be only one item, which is why a matrix is always affected to the properties editor.

Note: For an overview of the "*PBoxDescription*" and "*PBoxMatrix*" metaclasses, please refer to the "*Overview of properties editor services*" section in the current chapter of this user guide.

Creating fields

The "*PBoxDescription*" class provides basic properties editor item creation services. These services are methods, all of which send back an object whose basic class is "*PBoxItem*".

The "*PBoxDescription*" class has two relationships:

- "*MainItem*", which is the element to be put in the properties editor. A matrix created with the element is generally affected to it.
- "*Object*" is the edited model element. This relationship is not needed by the user, as "*this*" is already directed to it.

Note: For an overview of the "*PBoxDescription*" metaclass, please refer to the "*Overview of properties editor services*" section in the current chapter of this user guide.

Creating a simple text entry field

To create a simply text entry field, the following service should be used.

```
createEditField (inout Object object,
                 in String attributeName,
                 String title)
return PBoxEditField;
```

For example:

```
item = box.createEditField(this,"TypeConstraint",
                           "label_String_size");
```

This method is used to create a new editing field for an attribute named "*attributeName*" which belongs to an "*object*" object. The "*title*" field is used to define the label preceding the text entry zone.

Note: It should be noted that for all field titles, the properties editor will look for the translation of the "*title*" message in the *<ModuleName>Ihmlabel.us* resource file. This is the case for all fields in the properties editor.

The graphic associated depends on the nature of the attribute:

- for a String type attribute, the field is a simple text entry field
- for an enumerate attribute, the field is a combobox which automatically contains all possible values for the enumerate

If a non-existent attribute name is entered, the following J methods are used:

```
String get<attributeName>;
```

to initialize the graphic, and

```
boolean set<attributeName> (content:in string);
```

to save information at user model level.

Note: The return value must indicate whether or not processing has been correctly carried out.

The associated graphic is then a simple text zone.

For example:

```
void Class:initMatrix(inout PBoxDescription box,
                      inout PBoxMatrix matrix)
{
  PBoxItem item;

  item =
box.createEditField(this,"theName","label_theName");
  matrix.pushInto(0, matrix.NbRows(), item);
}

String Class:gettheName()
{
  return Name;
}

boolean Class:settheName (content : in string)
{
  if (content.findLast(« «)== -1)
  {
    // names should not contain spaces
    return false ;
  }
  else
  {
    sessionBegin(«Change name»,true);
    // a session must be opened
    setName(content);
    return sessionEnd();
    // the best method is to send back if the session
       is valid
  }
}
```

If the attribute is an enumerate, the following service is used:

```
setRadio(mode : in boolean) ;
```

When the "*mode*" parameter is true, this service is used to graphically represent the attribute in the form of radio buttons. By default, or if the "*mode*" parameter is false, the attribute is represented by a combobox.

Note: This is only true where the attribute represented is an enumerate. In other cases, this method has no effect on the graphic object (a text field, for example).

Creating a single-line gadget field to enter an integer

```
PBoxGadgetIntegerField createGadgetIntegerField
                       (inout SmObject object,
                        in string commandJ ,
                        in string title);
```

This method allows the definition of an editing zone for integers.

Please refer to the "*createGadgetTextField*" and "*createEditField*" methods for further information.

Creating a stereotype selection field

```
createEditStereotype(inout ModelElement modelElement,
                     in string  title )
  return PBoxEditStereotype;
```

This method is used to create a selection field, associated with a "*modelElement*" stereotype. The "*title*" field allows the label preceding the field to be defined.

The associated field can be a combobox or radio buttons. By default, the field is a combobox, but this can be changed using the following method:

```
setRadio(mode : in boolean) ;
```

When the "*mode*" parameter is true, this method is used to graphically represent the attribute in the form of radio buttons. By default, or if the "*mode*" parameter is false, the attribute is represented by a combobox.

```
append(stereotypeName : in string);
```

This service is used to add the "*stereotypeName*" stereotype to the field stereotype list.

```
appendAll();
```

This service is used to add all the possible values to the list of stereotypes.

For example:

```
void Class :initMatrix(inout PBoxDescription box,
                       inout PBoxMatrix matrix)
{
  PBoxItem item;

  item = box.createEditStereotype
         (this,"label_Stereotype");
  item.setRadio(true);
  if (isInPhysicalModel())
  // a method specific to the module
  {
item.append("table");
  }
  item.append("sqlView");
  item.append("procedureClass");
  matrix.pushInto(0, matrix.NbRows(), item);
}
```

Creating an editing field for a tagged value

```
PBoxEditTag createEditTag (inout ModelElement
                              modelElement,
                           in string tagTypeName,
                           in string  title );
```

This method is used to create an editing field for a tagged value named "*tagTypeName*" which belongs to a *"modelElement*" element, and its parameters. The "*title*" field defines the label preceding the text entry zone.

Graphically, the field is presented in the form of a tickbox, indicating the name of the tagged value ("*tagTypeName*") and the name of a text field containing the parameter(s) of the tagged value.

Note: Objecteering/UML does not handle mistakes in the name of the tagged value type!

For example:

```
// Addition of the management of a tagged value type
item = box.createEditTag
       (this,"sqlName","label_SQL_Name");
matrix.pushInto(0, matrix.NbRows(), item);
```

Creating a multi-line editing field

```
PBoxEditText createEditText (inout SmObject object,
                             in string attributeName,
                             in string  title);
```

This method is used to create a new editing field for an attribute named "*attributeName*" belonging to an "*object*" object. The "*title*" field is used to define the label preceding the text entry zone.

The associated graphic is automatically a multi-line zone.

Creating a label

```
createGraphicLabel (in string  title )
  return [@*] class PBoxGraphicLabel;
```

This method is used to create a label. This element will be translated in the properties editor by a non-modifiable text string.

Adding a bitmap

```
createGraphicBitmap
       (bitmapFileName : in string () := /@
CR_string::empty() @/)
  return [@*] class PBoxGraphicBitmap;
```

This method is used to create a bitmap.

Adding a separation line

```
createGraphicSeparator ()
  return [@*] class PBoxGraphicSeparator;
```

This method is used to create a horizontal separation line.

Adding a button

```
createGadgetButton
     object [@*] : inout class SmObject,
     commandJ : in string (),
     title : in string () := /@ CR_string::empty() @/)
  return [@*] class PBoxGadgetButton;
```

This method is used to create a button. The J callback associated with the button's action is defined in the "*commandJ*" field. This method is run on the "*object*" object, and *"title*" is the button's label.

```
createGadgetBitmapButton
         (object [@*] : inout class SmObject,
          commandJ : in string (),
          bitmapFileName : in string () :=
                           /@ CR_string::empty() @/)
  return [@*] class PBoxGadgetBitmapButton;
```

This method provides exactly the same service as the previous method. However, instead of having a label on the button, a bitmap whose file is indicated in "*bitmapFileName*" appears.

Note 1: In the J callback, the current module is not known! As a consequence, the "*getCurrentModuleParameterValue*" and "*getMulMessage*" services do not function.

Note 2: "*bitmapFileName*" must be the name of a file located in the *$(OBJING_PATH)/res/bmp* directory.

For example:

```
void Class :initMatrix(inout PBoxDescription box,
                       inout PBoxMatrix matrix)
{
PBoxItem item;
PBoxMatrix m3;

// The properties editor only accepts bitmaps which are
// located in $objing/res/bmp/.
// Providing a path serves no purpose.

item = box.createGadgetBitmapButton
              (this, "selectInBrowser",
               "loupe.gif");
               m3.pushInto(0, m3.NbRows(), item);

//
// Label for physical model on the same line
//
item=box.createGraphicLabel
              (ihmMsg1("Physical_model_x1",Name));
// Here, the second item is placed on the last occupied
// line of the matrix in the second column
m3.pushInto(1, m3.NbRows()-1, item);

// My m3 matrix is positioned in the initial matrix.

matrix.pushInto(0, m3.NbRows(), item);
}
```

Adding a combobox or radio buttons

```
PBoxGadgetChoice createGadgetChoice
           (inout SmObject object,
            in string commandJ ,
            in string title );
```

This method is used to create a selected field associated with an "*object*" object. "*title*" is used to define the label preceding the field.

"*commandJ*" is used to define the basis of function names which will be called to initialize the graphic gadget and save information.

To initialize the graphic, the following service is used:

```
string  get<commandJ> ();
```

To save information at user model level, the following service is used:

```
boolean set<commandJ> (content : in string);
```

The return value must indicate whether or not processing has been correctly carried out.

These methods are run on the "*object*" object. The associated graphic can be a combobox or radio buttons.

Services provided by the "PBoxGadgetChoice" class

```
setRadio(mode : in boolean) ;
```

When the "*mode*" parameter is true, this service is used to graphically represent the attribute in the form of radio buttons. By default, or if the "*mode*" parameter is false, the attribute is represented by a combobox.

```
appendLabel(label [@*] : in string ());
```

This service is used to add the "*label*" string to the list of attribute values.

For example:

```
// Addition of a set of radio buttons
item = box.createGadgetChoice(this,"Persistence",
                              "label_Persistence" );
item.setRadio(true);
// true => radio buttons, false ==> combobox
item.appendLabel("label_Transient");
item.appendLabel("label_Persistent");
item.appendLabel("label_Undefined");
// Addition of the item at the bottom of the matrix
matrix.pushInto(0, matrix.NbRows(), item);
```

Adding a tickbox

```
PBoxGadgetToggle createGadgetToggle
                  (inout SmObject object,
                   in string commandJ ,
                   in string title );
```

This method is used to define an editing zone for tickboxes.

"*title*" is used to define the label preceding the field.

"*commandJ*" is used to define the basis for function names which will be called to initialize the graphic field and save information.

To initialize the field, the following service is used:

```
boolean  get<commandJ> ();
```

To save information at user model, the following service is used:

```
boolean set<commandJ> (in boolean content );
```

The return value must indicate whether or not processing has been correctly carried out.

These methods are run on the "*object*" object.

The associated field can be a combobox or radio buttons. By default, the field is a combobox, but this can be changed using the following method:

```
setRadio(mode : in boolean) ;
```

When the "*mode*" parameter is true, this is used to graphically represent the attribute in the form of radio buttons. By default, or if the "*mode*" parameter is false, the attribute is represented by a combobox.

Chapter 11: Miscellaneous

Scan services

Purpose

It is always possible and not discouraged to navigate through the metamodel meta-associations by using the usual J navigation mechanism. Scan services provide a uniform way to browse a model, by using the most commonly used paths.

Scan services skip other uninteresting intermediate metaclasses, such as metaclasses representing links ("*Use*", "*Communication*", "*Realization*").

Usage

Most metaclasses have scan services. Any time you wish to browse, please consult the origin metaclass, and see if the required service is present.

All scan services return sets, even if there can exist only one single related element.

This is practical for diffusing messages to the result, even if nothing is returned.

Example

We want to know the *UseCases* that cooperate with an *Actor*. We first have to look at the "*Actor*" documentation. We can get this result in the usual way, by navigating the associated "*Communication*" occurrences, and then the *UseCases* associated with these occurrences.

Scan services will provide the result through the following simple instruction:

```
cooperatingUseCase = myActor.getCooperatingUseCases();
```

Stream exchange services

Presentation

Objecteering/UML provides two services for stream exchange facilities and two formats:

- the "*externalization format*": This format is specific to Objecteering/UML. It has the advantage of providing a complete stream storage of the Objecteering/UML data, and of using the universal identification mechanism. The teamwork facility and the module externalization service use this format.
- the XMI format: This format corresponds to the OMG UML XMI standard. It is an open format which allows the exchange of models between different tools. It contains neither Objecteering/UML-specific data nor diagrams, nor does it use identifiers.

"Externalization" format services

```
boolean Object:externalizeIntoDir(in String dirName, in
boolean isRecursive)
```

This either recursively externalizes or not, according to "*isRecursive*", the current object in the "*dirName*" directory in one or several files. It returns "*true*" if this has been correctly carried out.

```
boolean Object:internalizeFile(in String rootFile, in
String dirName, in boolean recursive)
```

This internalizes the "*rootFile*" file contained in "*dirName*". If "*recursive*" is true, then all components of the object described in "*rootFile*" are also internalized.

If "*recursive*" if false, then only components which are not described in their own file are internalized.

In this mode, if the object described in "*rootFile*" exists in the database before internalization and there are non-internalized components described in "*rootFile*", these remain linked as components.

Those which are not described in "*rootFile*" are deleted.

Finally, those which are described in "*rootFile*" and which do not exist in the database are created in simplified form.

This returns "*true*" if the internalization has been properly carried out. The internalization is always carried out in the context of a UML modeling project.

```
boolean Object:internalizeSetOfFiles(in String
rootFiles[], in String dirName, in boolean recursive)
```

The same is true for "*internalizeFile*" but a set of files which are internalized are taken into account all at once.

```
String Object:getExternalFileNameFromObject(in String
pRootDir, in boolean pWithExt)
```

This returns the name of the externalization file associated with the current object.

If "*pWithExt*" is true, the externalization files extension is added. "*pRootDir*" can be an empty string.

```
String Object:getExternalFileNameFromString(in String
pRootDir, in String pObjectLogicalId, in String
pObjectName, in boolean pWithExt)
```

This returns the name of the externalization file associated with the object identified by "*pObjectLogicalId*" or "*pObjectName*" name, preceded by "*pRootDir*". If "*pWithExt*" is true, the externalization files extension is added. "*pRootDir*" can be an empty string.

The pObjectLogicalId and pObjectName parameters are exclusive. In the case of conflict, "*pObjectLogicalId*" is selected. Use "*pObjectName*" for a "*Project*" or a "*SoModule*".

XMI services

The *Objecteering/XMI* module is used to generate and re-read XMI files. The file format is the standard UML format for model exchange between different kinds of tools. XMI exchange is based on the OMG UML 1.3 and 1.4 standards.

Note: *Objecteering/XMI module import* can also be used to re-read XMI files, but cannot be used to generate them.

XMI functions

The *Objecteering/XMI* module has three main functions:

- The generation of XMI files from an Objecteering/UML model
- The creation of an Objecteering/UML model from an XMI file
- The update of an Objecteering/UML model from an XMI file

For further information on XMI, please refer to the *Objecteering/XMI* user guide.

"Object" class

Overview

In the J language, every element is an instance of *Object*. Primitive types such as "*int*" or "*String*" are sub-classes of *Object*, but every *Class* metamodel, such as "*Element*" or "*Class*", is also a sub-class of *Object*.

The "*Object*" class is also described in the J reference manual. Its interest here is to implement several general methods available in the *J library*.

Project management services

Overview

In the J language, services are provided for project management. For example, J provides services for the creation or deletion of UML modeling projects, as well as for the selections of modules, and so on.

Administration mode - Creating a UML modeling project

```
Project Object:createProject (in String projectName)
```

This service creates a UML modeling project named *projectName*. The name must be unique. The UML modeling project is created with selection of default modules. This service must be called only in administration mode. The new UML modeling project is returned if the creation is a success, otherwise the interpreter is stopped.

This API run on *Object* necessitates a particular context which can be retrieved in three ways:

1 - in the context of module installation (the *installModule* method)

2 - in the initial context of launching J on-line (administration mode)

3 - after retrieving the J on-line initialization object (administration mode) using the *getStartUpObject()* method

Example: *MyProject = getStartUpObject() .createProject("MyProject");*

Opening a UML modeling project

```
Project Object:openProject(in String projectName)
```

This opens the "*projectName*" project and returns the *Project* open. An error is sent if the UML modeling project does not exist. Opening consists of positioning the internalization context on the realized UML modeling project.

Adding a module used by a UML modeling project

```
boolean Project:addUsedModule (in String moduleName)
```

This service adds the *moduleName* module as a module used by the UML modeling project to which the message is sent. The module must be installed. The service returns *true* if the module has been added or if it was already present, and *false* otherwise. This service must be called in the context of a session (for further information, please refer to the "*Managing the Session*" section of chapter 2 of the *Objecteering/J Libraries User Guide*).

Selecting a module

```
Project:updateSelectedModules (in String[]
pModuleNamesToSelect, in String[]
pModuleNamesToDeselect)
```

In the current UML modeling project, this selects the modules whose names are contained in "*pModuleNamesToSelect*", and deselects those in "*pModuleNamesToDeselect*". For each module selected, a license token is taken if the module is subject to a license. When the module is unselected, the license token is released.

Getting parameter values

```
boolean Object:getParameterValue (in String name, in
String profile, in String module, out String value)
```

This returns in "*value*" the value of the parameter "*name*" defined in the "*profile*" profile of the "*module*" module. This returns false if the parameter does not exist.

Finding predefined types

```
Object Object:findPredefinedType (in String TypeName)
```

This returns the "*TypeName*"predefined type or returns empty if this is not found. The type is searched for in the "*_predefinedTypes*" UML modeling project.

Finding an object from its identifiers

```
Element Object:projectFindFromIds (in String
PhysicalId, in String SiteId)
```

This looks for an object in the current UML modeling project from its identifiers. "*PhysicalId*" is the base identifier and "*SiteId*" is the site identifier. The object is first looked for with regard to its "*PhysicalId*" and then regarding its "*SiteId*", if the first search is not successful.

If the object is not found or if there is no current UML modeling project, an empty object is returned.

Finding the two identifiers of an element

```
Object:getBothIds (out String PhysicalId, out String
SiteId)
```

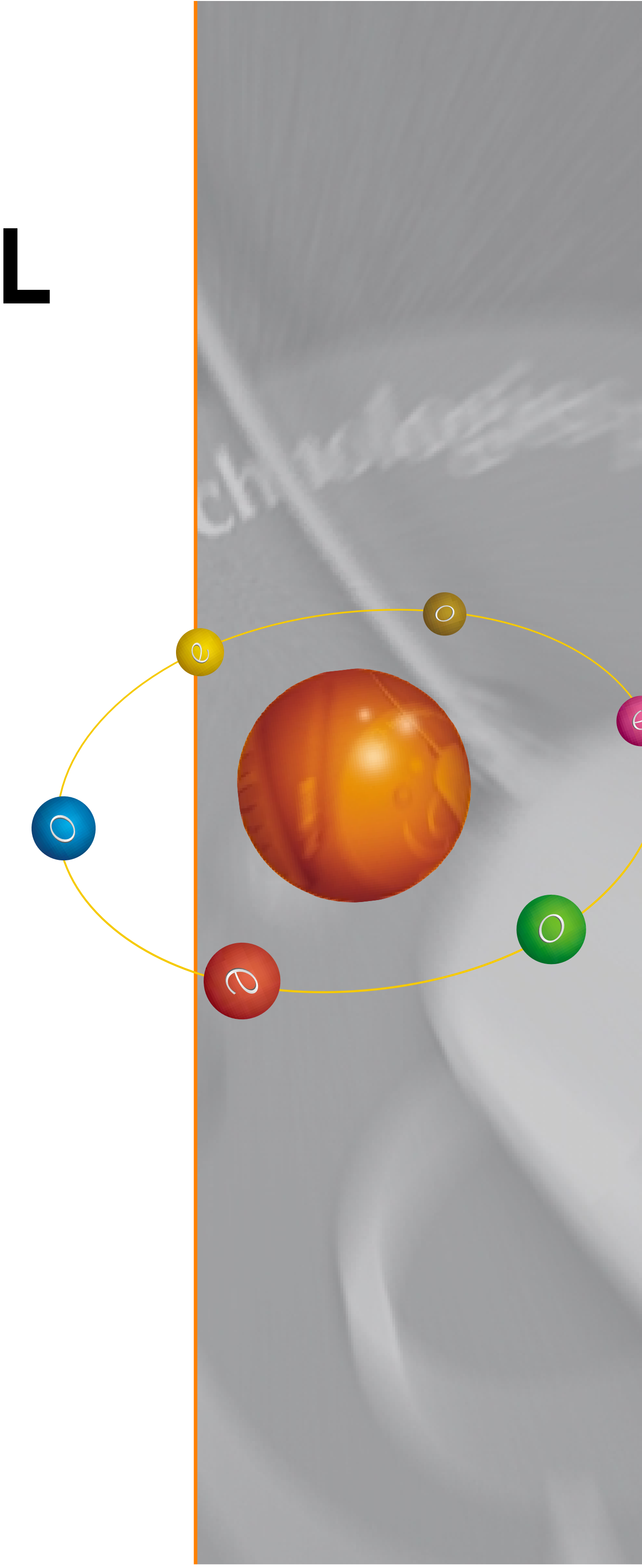
This returns both the physical and logical identifiers of the current element. An empty string is returned if no identifier exists.

Index

Objecteering/UML

Objecteering/Metamodel User Guide

Version 5.2.2



Objecteering

Software

www.objecteering.com

Taking object development one step further

Contents

Chapter 5: Static Model

Chapter 6: Use Case Model

Chapter 7: State Machine Model

Chapter 8: Activity Model

Chapter 9: Physical Model (Components and Nodes)

Chapter 10: Collaboration, Roles and Instances (Collaboration and Instances) Model

Chapter 11: Implementation of the OMG 1.4 UML metamodel in Objecteering/UML

Index

Chapter 1: Overview

Presentation

Introduction

Welcome to the *Objecteering/Metamodel* user guide!

The Objecteering/UML metamodel contains the most accurate description of the information managed by the Objecteering/UML CASE tool, as well as its definition and links.

This user guide constitutes the working base for all those who wish to implement new services which use the metamodel, such as model transformation, code or documentation generation, metrics calculation, requests to the model, and so on. It is a programmer's guide to users using the *Objecteering/UML Profile Builder* tool. *J language* programmers will find in this user guide all the predefined J classes (Objecteering/UML metaclasses) which can be handled.

What is a metamodel?

A metamodel is the model of a model. The Objecteering/UML metamodel provides a detailed description of the model supported by Objecteering/UML. The Objecteering/UML CASE tool is built using automated model transformation and code generation techniques, based on the presented metamodel. All its elements (model dialog boxes, graphic editors, model manager, etc.) are deduced from the metamodel. The user, therefore, has here the model of the Objecteering/UML CASE tool itself.

Using the metamodel

Each metaclass is documented, with a description of its attributes, relationships and relationships in the opposite direction. The names provided are precisely those which must be used from the *J language*, in order to work with the metamodel (attribute names, names of the roles which are concatenated with the ones of the opposite classes, names of the classes). Services and consistency rules are also presented. In addition, element composition graphs are explained, in order to present the way in which new instances are created.

Glossary

Metamodel: model of a model. The UML metamodel, as implemented by Objecteering/UML, is defined using this metamodel.

Metaclass: a class which represents an element of a metamodel. For example, "*Component*" is a metaclass. Every component in a UML model is an instance of this metaclass. Metaclasses are defined using meta-attributes, meta-associations, etc.

Chapter 2: First Steps

Presentation

Preliminary steps

Welcome to the *Objecteering/Metamodel* First Steps!

Objecteering/UML Profile Builder is the tool through which the metamodel can be edited and J can be executed. We recommend that you carry out the *UML Profile Builder* First Steps (chapters 2 and 3 of the *Objecteering/UML Profile Builder* user guide) before trying out these metamodel First Steps.

J is structured and interpreted and provides additional utilities, such as the possibility of creating UML extensions, or defining new kinds of work products or generation templates. J can also be dynamically executed.

To be able to use the *Objecteering/UML Profile Builder* tool, you must have the correct license.

Getting familiar with metamodel handling

Conventions exist which allow a J programmer to navigate within a metamodel, and to access information. This is illustrated in Example 1.

Creating models automatically: Model transformation

Objecteering/UML provides a session management mechanism, which makes it possible to change and transform a model using J. A small transformation example will be provided in Example 2.

Preparing the development environment

Editing a UML profiling project

The *Objecteering/UML Profile Builder* user guide explains how to run the *Objecteering/UML Profile Builder* tool.

- start Objecteering/UML
- create or open a UML profiling project
- develop the "*default#external#J_examples*" profile, consult the code for the "*Class*" metaclass methods and consult the code of the metaclass in the meta-explorer
- try to run related commands on example UML modeling projects

Example 1: Writing J methods on the Class metaclass

J principle

Once a J method has been created on a specific metaclass, its J code has direct access to the metaclass property. By simply naming the attributes, J gets the values of a metaclass occurrence. Associations are handled through role names. Adding a role name to an associated metaclass name designates the set of occurrences related to the current occurrence.

For example, the "*Class*" metaclass is related by the "*Behavior*" association to the "*StateMachine*" metaclass, which plays the "*Behavior*" role. From a "*Class*" context, the name "*BehaviorStateMachine*" designates all *StateMachine* occurrences related to the current class.

Example

The "*Class*" metaclass has the following attributes: "*isMain*" and "*isActive*". It specializes "*GeneralClass*", which has the "*isElementary*" attribute. "*GeneralClass*" specializes "*Classifier*", which in turn specializes "*NameSpace*" (whose attributes are "*IsAbstract*", "*IsRoot*", "*IsLeaf*" and "*Visibility*"), which specializes "*ModelElement*", which has the "*Name*" attribute.

The following "*printClassProperties*" J method prints the values of all values of a class. This method can be run on a test project using the "*printClassProperties*" command.

```
Class: printClassProperties ()

{
StdOut.write (IsMain, IsActive, IsElementary, IsAbstract,
IsLeaf, IsRoot, Visibility,  Name, NL);
}
```

Result example:

```
falsetruefalsefalsetruefalsePublicC1
```

Example 2: Creating models automatically: Model transformation

Session mechanism

The following example will create an accessor on a class for every defined attribute. For each one, a J session model transformation operation (for further information, please see chapter 2, "*Modifying a UML Model with J*", of the *Objecteering/J Libraries User Guide*) should be created. The "*SessionBegin*" and "*SessionEnd*" instructions are used for this.

Example

The addAccessor J method below creates "*get*" and "*set*" operations for each attribute of the class. Words extracted from the metamodel (roles, attribute and metaclass names) are in **bold**. "*create*", "*set*" and "*append*" are low level instructions for transforming a model, and can be run on a model using the "*addAccessors*" command.

```
Class:addAccessors()
{
Operation M;
Parameter P;
Class C;
sessionBegin ("addAccessor", true);
C = this;
PartAttribute
{
   M = Operation.new;
   M.setName("set-" + Name);
   C.appendPart(M);
   P=Parameter.new;
   P.setName("AttValue");
   M.appendIO(P);
   M = Operation.new;
   M.setName("get-" + Name);
   C.appendPart(M);
   P = Parameter.new;
   P.setName("R");
   M.appendReturn(P);
}
sessionEnd();}
```

Chapter 3: Metamodel Overview

Metaclasses defined in the metamodel

Generalization graph

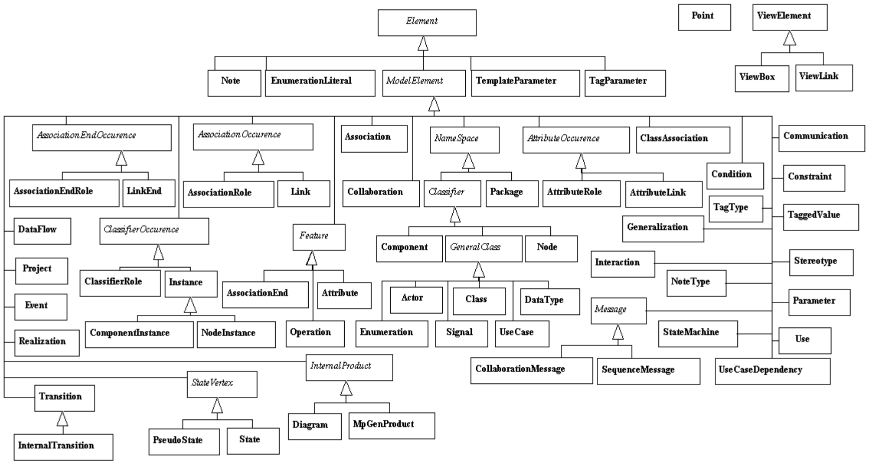


Figure 3-1. Generalization graph of the metamodel

This class generalization graph presents all the accessible metaclasses of the metamodel. *Element* and *ModelElement* are the root classes, *ModelElement* being the most important one. A short definition of every class is provided in the "*Class hierarchy*" section of the current chapter of this user guide.

Major elements of the metamodel

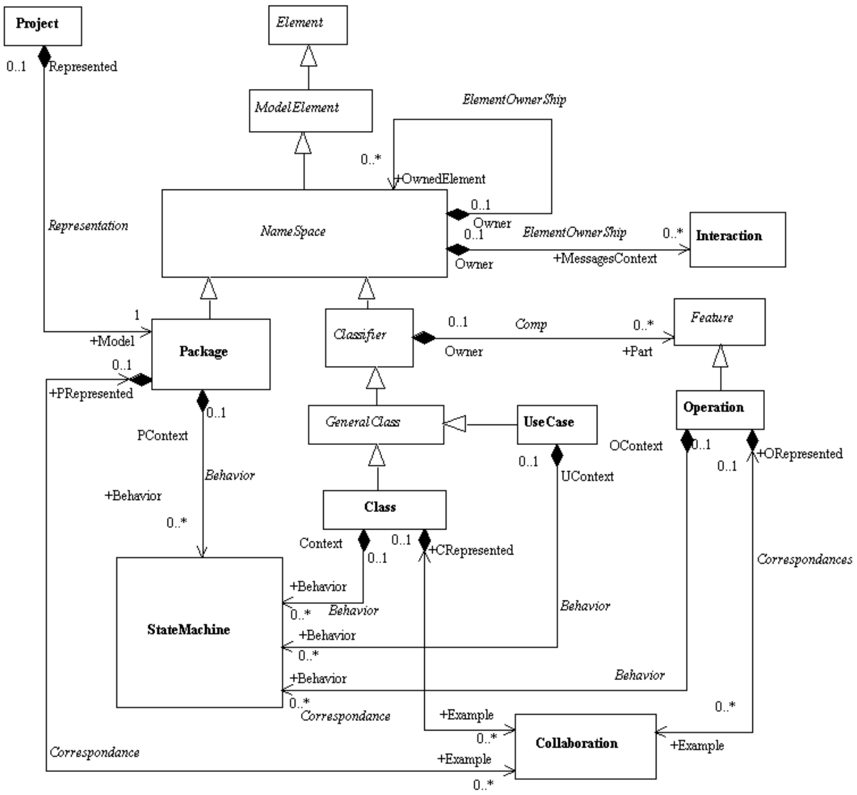


Figure 3-2. The principal elements of the metamodel

The diagram shown in Figure 3-2 summarizes the major elements which structure the entire metamodel, such as *NameSpace*, and its subsidiary sub-classes. The root of a model is the *Project*, related to one *ModelPackage*. This *Package* will then be broken down into into a *Package organization* tree, down to more detailed *NameSpaces* such as *Classes* or *UseCases*. These elements will then have other related models structured by *StateMachines*, *Collaborations* and *Interactions*.

Metamodel packages

Presentation

Figure 3-3 presents the packages contained in the metamodel. Every package is dedicated to a specific UML modeling area. The two exceptions are *CoreModel*, which presents the most abstract classes, and *DiagramsAndViewElements,* which presents the graphical part of the model.

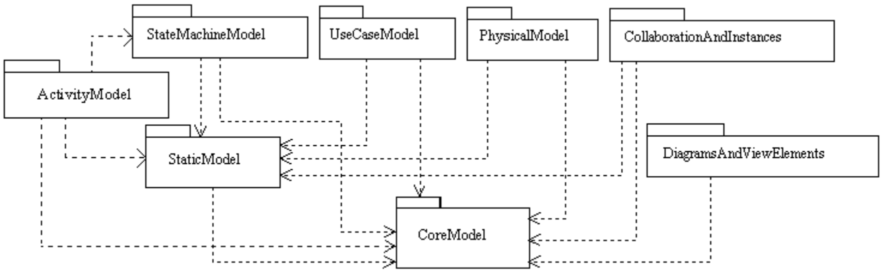


Figure 3-3. Metamodel packages

CoreModel:Abstract base classes of the model, and UML extensibility mechanisms.

DiagramsAndViewElements: Model of diagrams supported by Objecteering/UML.

StaticModel: Classes and packages, with their various properties and links.

UseCaseModel: Use cases and actors with their properties and links.

PhysicalModel: Nodes and components, which make up implementation and component diagrams.

StateMachineModel: State machines, states and transitions, which describe the dynamics of a system.

ActivityModel: Activity graphs, action states, object flow states and partitions, representing a process or a workflow.

CollaborationAndInstances: Roles and instances are described at this level. Object model and sequence diagrams are presented here.

Class hierarchy

Element: Atomic constituent of a model

..........*Note*: Textual part, attached to a *ModelElement*.

..........*EnumerationLiteral*: Defines an atom (i.e., with no relevant substructure), represents one in the list of values that an enumeration may have.

..........*TemplateParameter*: Parameter for *Templated* elements.

..........*TagParameter*: Parameter for *TaggedValues*.

..........*ModelElement*: Named entity in a model.

ModelElement: Named entity in a model

..........*Association*: Definition of links which may exist between objects.

..........*AssociationEndOccurence*: End point of a link.

.....................*AssociationEndRole*: Specifies an endpoint of an association as used in a collaboration.

.....................*LinkEnd*: End point of a link.

..........*AssociationOccurence*: Occurrence of an association. Presented as a link in a collaboration diagram.

.....................*AssociationRole*: Specific usage of an association needed in a collaboration.

.....................*Link*: Connection between instances.

..........*AttributeOccurence*: Named slot in an instance or role, which has the value of an attribute.

.....................*AttributeLink*: Named slot in an instance, which has the value of an attribute.

.....................*AttributeRole*: Named slot in a *ClassifierRole*, which has the value of an attribute.

..........*ClassAssociation*: Class relating other classes. It is both a class and an association.

..........*ClassifierOccurence*: Instance or Role which is an occurrence of a classifier

....................*ClassifierRole*: A classifier role is a specific role played by a participant in a *collaboration*.

....................*Instance*: Entity to which a set of operations can be applied and which has a state that stores the results of the operations.

..............................*ComponentInstance*: Instance of a component.

..............................*NodeInstance*: Instance of a *node*.

..........*Collaboration*: Describes how Instances or roles can cooperate to implement a *modelElement* such as an operation or a *UseCase*.

..........*Communication*: Communication link between actors and use cases.

..........*Condition*: Boolean expression for making a choice.

..........*Constraint*: Semantic restriction expressed as an expression in text form.

..........*DataFlow*: Circulation of information between model elements.

Dependency: Shows that the implementation or functioning of one or more elements requires the presence of one or more other elements.

..........*Event*: Specification of a significant occurrence which has a location in time and space.

..........*Feature*: Property, like operation or attribute, which is encapsulated within another entity, such as an interface, a class, or a data type.

....................*AssociationEnd*: Connection of an association to one of its related classes.

....................*Attribute*: Property of a class.

....................*Operation*: Individual pieces of invokable behavior.

..........*Generalization*: Taxonomic relationship between a more general element and a more specific element.

..........*Interaction*: Message sequencing context.

..........*InternalProduct*: Additional elements attached to *ModelElements*.

....................*Diagram*: Graphic representation of a model.

.....................*MpGenProduct*: Work product produced automatically by Objecteering/UML.

ModelTree: Root class for each each element organizing *ModelElements*.

Item: Can be specialized to add new elements to the metamodel.

..........*NameSpace*: Part of a model in which each name has a unique meaning.

.....................*Classifier*: Element which describes behavioral and structural features.

.............................*Component*: Physical unit of implementation with well-defined interfaces, which is intended to be used as a replaceable part of a system.

.............................*GeneralClass*: General definition of a class

....*Actor*: Active element external to the system, and which cooperates with it.

....*Class*: Description of a set of objects which share the same attributes, operations, methods, relationships, and semantics.

....*DataType*: A descriptor of a set of primitive values which lack identity

....*Enumeration*: Special kind of *DataType* whose range is a list of predefined values, called *EnumerationLiterals*.

....*Signal*: Specification of an asynchronous stimulus communicated between instances.

....*UseCase*: Unit of externally visible functionality provided by part of a system.

.............................*Node*: Run-time physical object which represents a computational resource.

.....................*Package*: Decomposition unit of a model.

..........*Message*: Occurrence of an operation, processed by instances.

.....................*CollaborationMessage*: Message used in the *collaboration* or object diagrams.

.....................*SequenceMessage*: Messages used for sequence diagrams.

..........*NoteType*: Defines a specific kind of *note*.

..........*Parameter*: Information received as input or returned as output by an *operation*.

..........*Partition*: Mechanism for dividing the states of an *ActivityGraph* into groups.

..........*Project*: Working space for a model.

..........*Realization*: Implementation link between a class and its interface, or between a component and its interface.

..........*StateMachine*: Graph of states and transitions which describes the dynamic behavior of objects.

.....................*ActivityGraph*: Special case of a *StateMachine* that defines a computational process in terms of the control-flow and object-flow among its constituent actions.

..........*StateVertex*: Abstraction of a node in a *statechart* graph.

.....................*PseudoState*: Abstraction of different types of nodes in the state machine graph.

.....................*State*: Notable situation or condition during the life of an object.

.............................*ActivityState*: A state representing a specific activity at a given point in time.

....*ActionState*: Description of an action that cannot be broken down further.

....*SubActivityState*: Activities that are decomposed into sub activities.

.............................*ObjectFlowState*: Defines an object flow between actions in an ActivityGraph.

..........*Stereotype*: Specific adaptation of *ModelElement* semantics.

..........*TaggedValue*: Attachment of a piece of information to a *ModelElement*.

..........*TagType*: Definition of *TaggedValues* allowed for a defined *metaclass*.

..........*Transition*: Path from one state to another.

.....................*InternalTransition*: Transition which is internal to a state.

..........*UseCaseDependency*: Inheritance or dependency link (Uses) between *UseCases*.

..........*Use*: Usage dependency between elements.

Point: Graphic point in a diagram.

ViewElement: Graphic representation of a ModelElement.

..........*ViewLink*: Graphical link.

..........*ViewBox*: Graphical boxes.

Enumerate types used in the metamodel

Presentation

As in the UML 1.4 OMG standards, many values are expressed by predefined literals. For example, the visibility of a *Feature* is expressed by values such as "*Public*". All predefined literals are summarized here.

Description

- *KindOfAccess* (Read, Write, ReadWrite, AccessNone): Different sorts of access that an *Attribute* may have.
- *VisibilityMode* (Public, Protected, Private, VisibilityUndefined): Visibility that a *Feature* or a *Component* of a *NameSpace* may have.
- *MethodPassingMode* (MethodIn, MethodOut): Defines whether or not the receiver object will be modified by the *Operation*.
- *PassingMode* (In, Out, Inout): Passing mode of a *Parameter* in an *Operation*.
- *StateKind* (InitialState, DeepHistoryState, ShallowHistoryState, JoinState, ForkState, BranchState, FinalState, SignalReceiptState, SignalSendingState, SynchronizationState): Defines every kind of pseudo state, each of them having specific notation and semantics.
- *ActionKind* (CallAction, ReturnAction, CreateAction, DestroyAction, TerminateAction): Useful for sequence diagrams, by defining the specific nature of every message. Defines the different kinds of Action that exist in UML.
- *KindOfStateMachine* (Dynamic, Protocol): Specifies whether a State machine describes the dynamics of a *ModelElement* (*Class*, *UseCase*, *Package*), or is a protocol state machine.
- *EventType* (SignalEvent, CallEvent, TimeEvent, ChangeEvent): Defines every possible kind of *Event*, which may be "*Receiving a signal*", receiving or sending a message, time event (clock), change defined by a boolean expression.
- *PredefinedEventType* (EntryEvent, DoEvent, ExitEvent): *InternalTransitions* may be of one of these predefined types.
- *AggregationKind* (KindIsAssociation, KindIsAggregation, KindIsComposition): Establishes whether an *Association* is not an Aggregation, or is a shared aggregation or a true composition.

Chapter 4: Extensibility mechanism and general elements (CoreModel)

Overview

Presentation

Element and *ModelElement* are the most abstract metaclasses in the metamodel. Every element of the metamodel is derived from these metaclasses. *ModelElements* represent the most important UML element. They have a *Name*, can be described by *Notes* and can be annotated with *Constraints*.

In addition, the extensibility mechanism provided by UML can be applied to these elements through *TaggedValues* and *Stereotypes*. Objecteering/UML is advanced with regard to UML 1.4, inasmuch as it implements the *Profile* concept, which structures the definition of *TaggedValues*, *Stereotypes* and *Notes*.

CoreModel metaclasses

- *Constraint*: Semantic restriction expressed as an expression in text form.
- *Element*: Atomic constituent of a model.
- *InternalProduct*: Additional elements attached to *ModelElements*.
- *ModelElement*: Named entity in a Model.
- *ModelTree*: The root for each element organizing *ModelElements* using an "*OwnerShip*" association.
- *Item*: Used to add new elements.
- *Dependency*: Used to state that the implementation or functioning of one or more elements requires the presence of one or more other elements.
- *MpGenProduct*: Work product produced automatically by Objecteering/UML.
- *Note*: Textual part, attached to a *ModelElement.*
- *NoteType*: Defines a specific kind of *Note*.
- *Stereotype*: Specific adaptation of *ModelElement* semantics.
- *TagParameter*: Parameter for *TaggedValues*.
- *TaggedValue*: Attachment of a piece of information to a *ModelElement*.
- *TagType*: Definition of *TaggedValues* allowed for a defined *metaclass*.

Metamodel synthesis

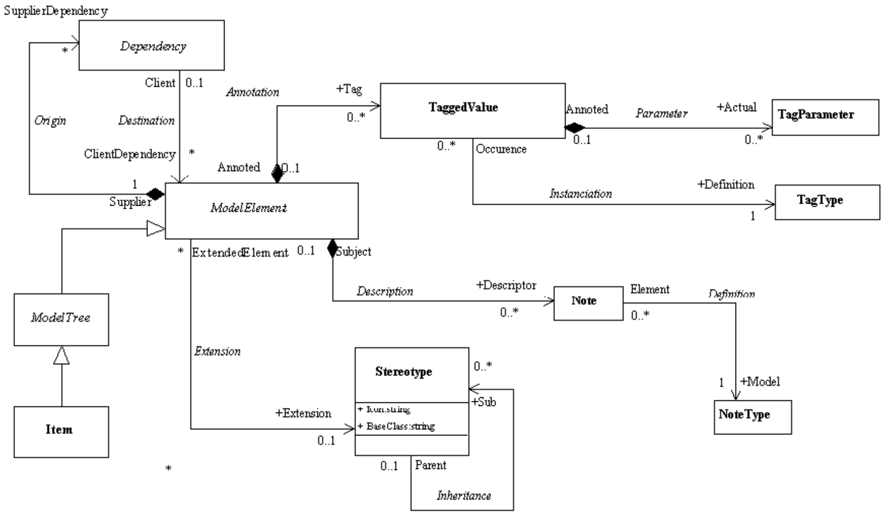


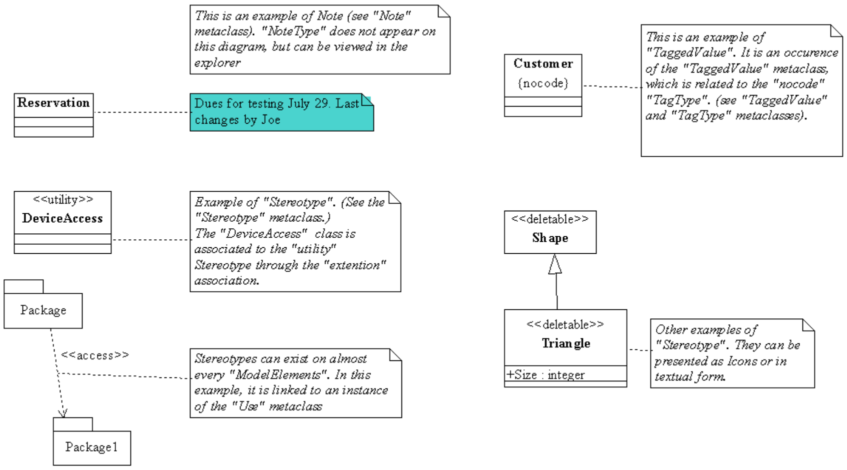
Figure 4-1. Diagram of the core model

UML defines *TaggedValues* and *Stereotypes* as extensibility mechanisms. In addition, Objecteering/UML provides the "*Note*" concept, which defines the annotated descriptions, and defines *NoteType* and *TagType*, which define which *Notes* and *TaggedValues* are allowed in a given context.

Objecteering/UML provides the notion of *Profile*, supported by the *Objecteering/UML Profile Builder* module, which organizes the specification of *Stereotypes*, *Tagged Values*, and *Notes*. At *Profile Specification* level are defined instances of *TagTypes*, *Stereotypes* and *NoteTypes*. At model level (using *Objecteering/UML Modeler*), *TaggedValues*, *Notes* and references to *Stereotypes* are realized, in accordance with the Profile specification.

ModelTree and *Item* are extensions to UML, used for extensibility purposes.

Example 1: Extensibility mechanism



This is an example of Note (see "Note" metaclass). "NoteType" does not appear on this diagram, but can be viewed in the explorer

Dues for testing July 29. Last changes by Joe

Example of "Stereotype". (See the "Stereotype" metaclass.) The "DeviceAccess" class is associated to the "utility" Stereotype through the "extention" association.

Stereotypes can exist on almost every "ModelElements". In this example, it is linked to an instance of the "Use" metaclass

This is an example of "TaggedValue". It is an occurence of the "TaggedValue" metaclass, which is related to the "nocode" "TagType". (see "TaggedValue" and "TagType" metaclasses).

Other examples of "Stereotype". They can be presented as Icons or in textual form.

Figure 4-2. Examples of *Notes*, *Stereotype* and *Tagged Values* are provided here

In this diagram, are presented representation of instances of the following metaclasses: *Notes* related to *NoteTypes* (not visible), *TaggedValues* related to *TagType* and *TaggedValues*.

Example 2: Constraints

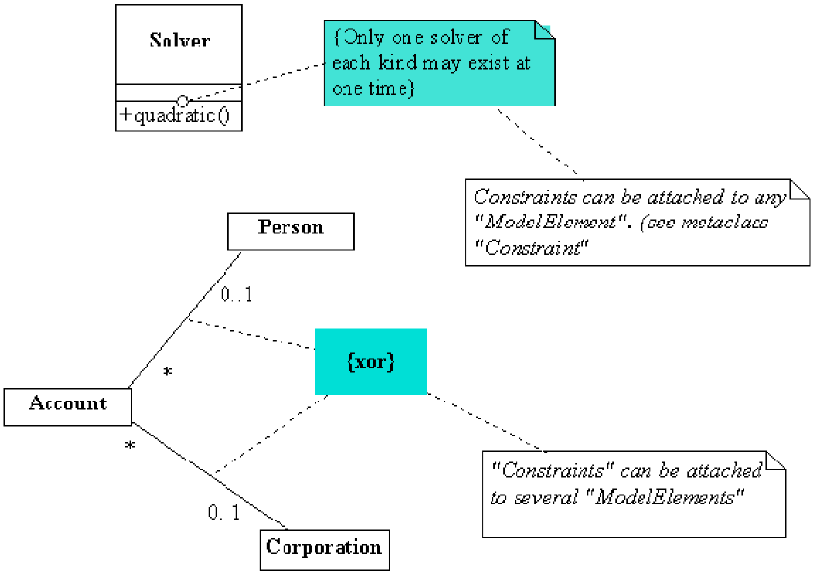

Solver
+quadratic()
{Only one solver of each kind may exist at one time}
Constraints can be attached to any "ModelElement". (see metaclass "Constraint"
Person
0..1
*
{xor}
Account
*
0. 1
Corporation
"Constraints" can be attached to several "ModelElements"

Figure 4-3. Examples of *Constraints*

Several representations of *Constraints* are presented here.

"Element" class

Element overview

```
abstract class Element;
```

Atomic constituent of a model.

In the metamodel an *Element* is the top metaclass of the metaclass hierarchy.

Every element of a model is an *Element*. *Elements* are structured into *Projects*, which constitute their definition space. Every element is "*universally identified*" in Objecteering/UML. An identifier has the following form: "*Site id* + *Base id* + *Project id* + *element id*". The identifier is assigned upon creation, and never changes. *Elements* in different projects can have the same identifier. This means that they are identical.

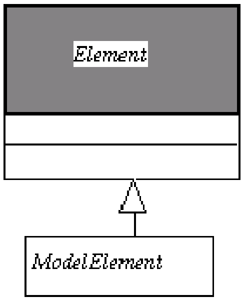


Figure 4-4. Detailed class diagram for *Element*

<u>See also</u>: *Note*, *EnumerationLiteral*, *ModelElement*, *TemplateParameter*, *TagParameter*.

Element methods

```
String getIdentifier()
```

Return the identifier of the object, which is the universal identifier of the element and which is allocated by Objecteering/UML. This value cannot be set by J.

```
object:findFromSiteId (in String className, in String
siteId)
```

Finds an *Element* in the current database from its id. The *Object* is void if no object is founded.

Example:

```
C?=findFromSiteIdentifier ("Class", "1285576512");
```

"ModelElement" class

ModelElement overview

```
abstract class ModelElement extends Element;
```

Named entity in a model. Describes every element that can exist in a model. Only low level elements are not *ModelElements*. *ModelElements* can be annotated by *TaggedValues*, by *Constraints*, by *Stereotypes* and by *Notes*.

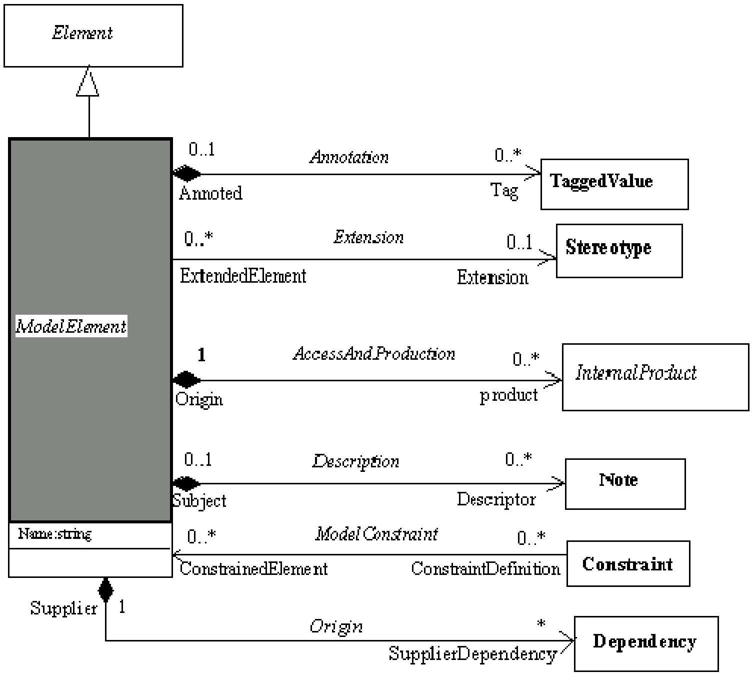


Figure 4-5. Detailed class diagram for *ModelElement*

See also: *Tagged values*, *Stereotypes*, *Constraints*, *Notes*

ModelElement properties

The class has the following associations:

- *Product:InternalProduct*: *Link* to a work product deduced (generated) from the *ModelElement*
- *Descriptor:Note*: *Notes* (documentation, sections of code, etc.) describing the *ModelElement*
- *Tag:TaggedValue*: *TaggedValues* which annotate the *ModelElement*
- *Extension:Stereotype*: *Stereotype* which metaclassifies the *ModelElement*
- *ConstraintDefinition:Constraint*: *Constraints* which express restrictions on the *ModelElement*
- *SupplierDependency:Dependency*: Designates a dependency that relates to a supplier *ModelElement*

The class owns the following attribute:

Name: *Name* of the element. Frequently A *ModelElement* belongs to a *NameSpace*, and the Name must follow unicity rules.

ModelElement consistency rules

- A *ModelElement* accesses *Stereotypes*, *NoteTypes* and *TagTypes* provided by the *Profiles* selected by the project.
- Classes designated by the "*BaseClass*" attributes of *Stereotypes* or *NoteTypes* or *TagTypes* associated with the elements are compatible with the element class.

ModelElement methods

Scan methods*:*

`Note[] getNotes()`
Returns the *Notes* which describe the element.

`MpGenProduct[] getProducts()`
Returns the work product (C++, Java, Documentation, ...) associated to the element (see *MpGenProduct* metaclass).

`TaggedValue[] getTaggedValues()`
Returns the *TaggedValues* which annotate the element.

`boolean isTaggedValue (in String ofName)`
Determines if the element is annotated with the tagged value whose name is given.

`TaggedValue[] getAllTaggedValues (in String ofName)`
Returns the *TaggedValues*, whose name is given, which annotate the elements.

"ModelTree" class

ModelTree overview

```
abstract class ModelTree extends ModelElement;
```

ModelTree is the root for each element organizing *ModelElements* using an "*OwnerShip*" association. The "*ElementOwnerShip*" association provides a hierarchy of model elements that can be managed by the model explorer or by the teamwork facility.

This metaclass is not part of the UML standard. "*NameSpace*" is a typical subclass, taking advantage of the containment facility provided by *ModelTree*.

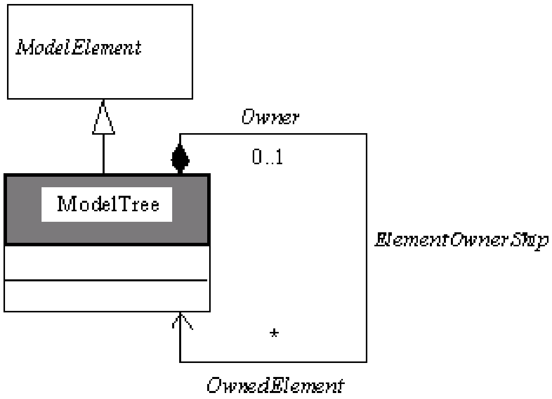


Figure 4-6. Detailed class diagram for *ModelTree*

ModelTree properties

The class has the following association:

♦ *OwnedElement:ModelTree*: Defines the composition tree of a model. For example, the decomposition of packages into sub-packages and classifiers is based on this association.

"Item" class

Item overview

```
Item extends ModelTree;
```

Item is a concrete class, that can be specialized for adding new elements to UML. Item, like "*ModelTree*", can be decomposed hierarchically, and browsed by the explorer. As a typical example, dictionary terms, or requirements have been modeled as stereotypes of *Item*. This metaclass is an extension to UML.

A tree of *Item* must have a root *Item* belonging directly to a *Project* instance.

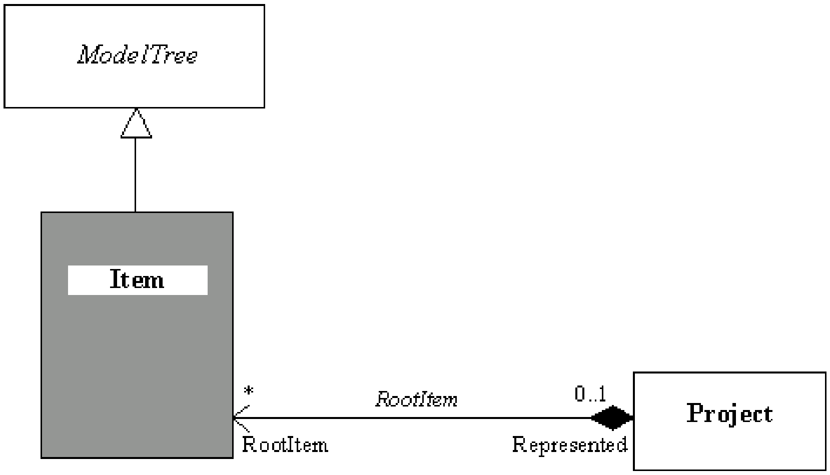


Figure 4-7. Class diagram for *Item*

"Dependency" class

Dependency overview

```
Dependency extends ModelElement;
```

A *Dependency* states that the implementation or functioning of one or more elements requires the presence of one or more other elements.

In the metamodel, a *Dependency* is a directed relationship from a client (or clients) to a supplier (or suppliers), stating that the client is dependent on the supplier (in other words, the client element requires the presence and knowledge of the supplier element). In the metamodel, an *Abstraction* is a *Dependency* in which there is mapping between the supplier and the client. Depending on the specific stereotype of *Abstraction*, mapping may be formal or informal, and may be unidirectional or bidirectional. If an *Abstraction* element has more than one client element, the supplier element maps into the set of client elements as a group. For example, an analysis-level class might be split into several design-level classes. The situation is similar if there is more than one supplier element.

Dependencies are widely used inside Objecteering/UML to manage traceability. Any model element can be traced to another model element using this association.

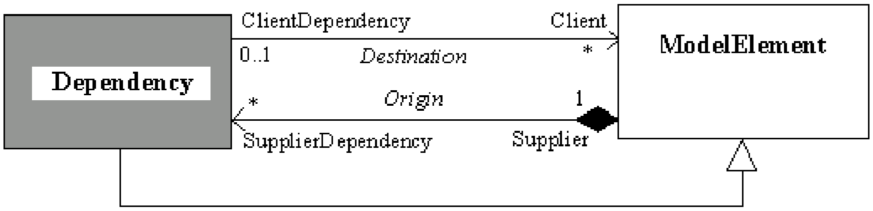


Figure 4-8. Class diagram for *Dependency*

Dependency properties

The class has the following association:

- *ClientDependency*: designates a dependency that relates to a client ModelElement.

"Stereotype" class

Stereotype overview

```
Stereotype extends ModelElement;
```

Specific adaptation of *ModelElement* semantics. Through *Stereotypes*, the end user can create new icons, and new adaptations to *ModelElements*. *Stereotypes* are defined in dedicated Profiles.

Example 1 presents a set of Stereotypes applied to various *ModelElements*.

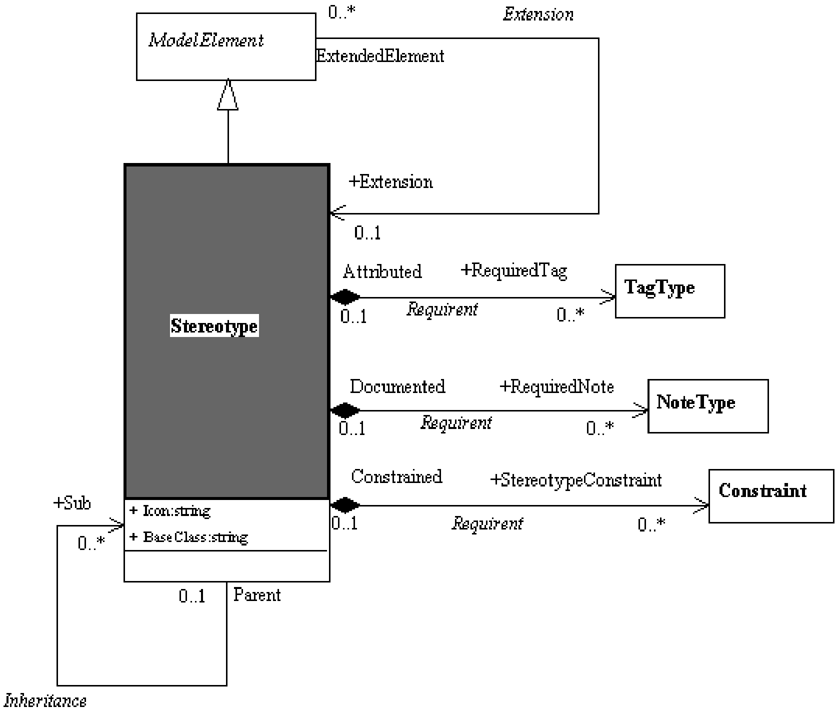


Figure 4-9. Detailed class diagram for *Stereotype*

Stereotype properties

The class has the following associations:

- *Sub:Stereotype*: Stereotypes may be inherited through this *Association*.
- *RequiredTag:TagType*: *Tagged values* allowed by the current *Stereotype*. The *Stereotype* acts as a new metaclass whose attributes are these *TaggedValues*.
- *RequiredNote:NoteType*: *Notes* that are specifically defined for *ModelElement* annotated by this *Stereotype*.
- *StereotypeConstraint:Constraint*: *Constraint* that will apply to every element annotated by this *Stereotype*.

The class owns the following attributes:

- *Icon*: Icon that can represent the stereotyped *ModelElement*.
- *BaseClass*: Metaclass whose instances can be annotated by the current *Stereotype*.

Stereotype consistency rules

- A modeled element can only be associated with, at most, one *Stereotype*.
- There can be no loops in the generalization relationship (*Inheritance*) between *Stereotypes*.

Stereotype constructor

Stereotypes cannot be created with the *J language*. They are defined at the meta-level (in *Profiles*), and do not belong to a *ModelElement* at the model level. Services are provided for getting the available *Stereotypes*.

```
Stereotype Object: findStereotypeInProject (in String
stereotypeName, in String metaClassName)
```

Finds the *Stereotype* defined in the current project, which has the indicated *Name*, and for the indicated *metaClass.*

"TaggedValue" class

TaggedValue overview

```
TaggedValue extends ModelElement;
```

Attachment of a piece of information to a *ModelElement*. In Objecteering/UML, *TaggedValues* can have parameters, and have to comply with *TagTypes* which define what kind of *TaggedValues* may exist.

Example 1 presents various *TaggedValues*.

TaggedValues belong to their annotated *ModelElement*, or to their annotated *Note*.

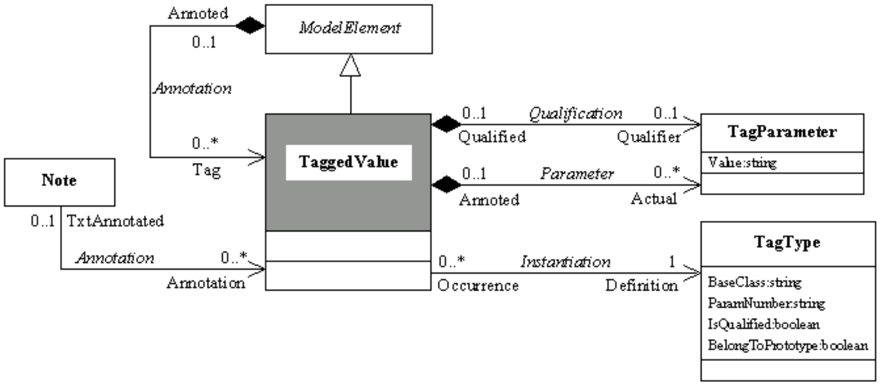


Figure 4-10. Detailed class diagram for *TaggedValue*

See also: *TagType*, *TagParameter*.

TaggedValue properties

The class has the following associations:

- *Definition*:*TagType*: Determines the *TagType* which is the model of the current *TaggedValue*.
- *Qualifier*:*TagParameter*: Current qualifier of the *TaggedValue*. A qualifier is a parameter that is placed in first place *Tag*:*Qualifier* (Parameters)
- *Actual*:*TagParameter*: Parameters of the *TaggedValue*.

TaggedValue consistency rules

- The *BaseClass* attribute of the *TagType* associated with the *TaggedValue* (*Instantiation* association) must be consistent with the modeling element which contains the *TaggedValue*.
- The number of values entered for a *TagValue* must be consistent with the number of parameters defined in the corresponding *TagType*.
- The *Annotation* associations are exclusives.

TaggedValue constructor

```
TaggedValueElement : createTaggedValue (in String pName, in
String pTagType, in String pMetaClass)
```

This operation creates a *TaggedValue* whose type and metaclass on which it is defined are given.

Example:

```
TaggedValue MyTag = createTaggedValue ("", "persistent",
"Class");
```

```
TaggedValue Element : createTaggedValueWithParams (in String
pName, in String pTagType, in Object[] pParameterList, in
String pQualifier, in String pMetaClass)
```

This operation creates a *TaggedValue* whose *Type*, *Parameters*, *Qualifier* and *metaClass* on which it is defined are given.

Example:

```
TaggedValue MyTag = createTaggedValueWithParams ("",
"Heval", Params, "aQualifier", "Note");
```

```
ModelElement : addTaggedValue (in TaggedValue pTaggedValue)
```

This operation adds a *TaggedValue* to the current *ModelElement*.

```
TaggedValue Element:createAndAddTaggedValue (in String
pName, in String pTageType)
```

This operation creates, and adds to the element it is applied to, a *TaggedValue* whose type (*TagType*) is given.

Example:

```
TaggedValue MyTag = MyClass.createAndAddTaggedValue ("",
"nocode");
```

```
TaggedValue Element:createAndAddTaggedValueWithParams (in
String pName, in String pTagType, in Object []
pParameterList, in String pQualifier)
```

This operation creates, and adds to the element it is applied to, a *TaggedValue* whose type, parameters and qualifier are given.

Example:

```
TaggedValue MyTag =
MyPackage.createAndAddTaggedValueWithParams ("", "hygen",
Params, "Aqualifier");
```

TaggedValue methods

```
TagParameter[] getParameters()
```
Returns the *TagParameters* of the *TaggedValue*

```
TagType[] getTagType()
```
Returns the *TagType* of the *TaggedValue*.

```
TagParameter TaggedValue::createAndAddTagParameter
                          (in String value)
```
Creates and adds a *TagParameter* to a *TaggedValue*.

Example:

```
TagParameter MyParameter =
ATaggedValue.<createAndAddTagParameter("Val");
```

```
void TaggedValue::addTagParameter
                  (in TagParameter parameter)
```
Adds a *TagParameter* to a *TaggedValue*.

Example:

```
TagParameter parameter = createTagParameter ("Val");
if (parameter != null) {
    MyTaggedValue.addTagParameter (parameter);
}
```

"TagParameter" class

TagParameter overview

```
TagParameter extends Element;
```

Parameter for *TaggedValues*. *TaggedValues* are somewhat more powerful in Objecteering/UML. They can have parameters which must conform to the *TagType* structure.

TagParameters belong to their *TaggedValue*. (See Example 1).

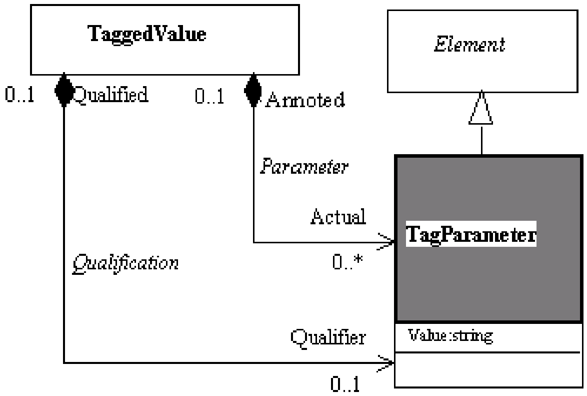


Figure 4-11. Detailed class diagram for *TagParameter*

See also: *TaggedValue*, *TagType*.

TagParameter properties

The class owns the following attribute:

- *Value*: Parameter value.

TagParameter consistency rules

- ♦ The number of *TagParameters* contained in a *TaggedValue* must correspond to the number of parameters defined in its *TagType*.

TagParameter constructor

```
TagParameter Object::createTagParameter
                      (in String value)
```

Creates a *TagParameter*.

Example:

```
TagParameter parameter = createTagParameter ("Val");
```

Please refer to *TaggedValues*, which have services to create *TagParameters* with the owner *TaggedValue.*

"TagType" class

TagType overview

```
TagType extends ModelElement;
```

Definition of *TaggedValues* authorized for a defined metaclass.

These *TagTypes* are defined in UML profiling projects and structured by Profiles.

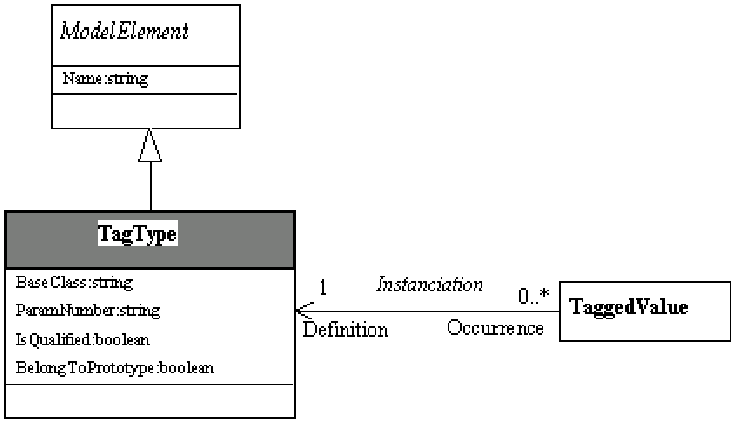


Figure 4-12. Detailed class diagram for *TagType*

See also: *TaggedValue*.

TagType properties

The class owns the following attributes:

- *BaseClass*: *Metaclass* that can be annotated by tagged values which are occurrences of the current *TagType*.
- *ParamNumber*: Number of parameters an occurrence may have.
- *IsQualified*: Determines whether or not an occurrence (tagged value) has a qualifier.
- *BelongToPrototype*: Determines whether or not a *TagType* occurrence belongs to the signature. For example, the *TagType* "*" which defines a pointer declaration in C++, belongs to the prototype of operations.

TagType constructor

TagTypes cannot be created with the J language. They belong to Profiles at metalevel, and are not structured at model level.

```
TagType Object: findTagTypeInProject (in String tagTypeName,
in String metaClassName)
```

Returns the *TagType* defined in the project, corresponding to the name and metaclass specified as parameters.

"Constraint" class

Constraint overview

```
Constraint extends ModelElement;
```

Semantic restriction expressed as an expression in text form.

Constraints can express restrictions and relationships which cannot be expressed using UML notation. They are particularly useful for stating global conditions or conditions that affect a number of elements.

Constraints can have predefined names, and can also represent pre-conditions, post-conditions and invariants (pre-defined stereotypes). Language-specific modules (C++, Java) will add a specific stereotype for the pre and post conditions and invariants expressed in these languages, such as, for example, *C++Invariant*, or *JavaPreCondition*.

In Objecteering/UML, a *Constraint* is not composed of anything. It is simply managed by specific copy/transfer rules.

Example 2 shows different cases of *Constraints*.

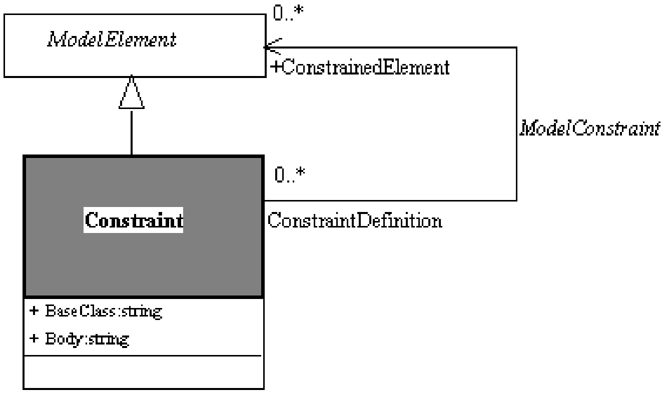


Figure 4-13. Detailed class diagram for *Constraint*

Constraint properties

The class has the following association:

- *ConstrainedElement:ModelElement*: Defines which elements are concerned by the constraint.

The class owns the following attributes:

- *BaseClass*: Metaclass whose instances can be constrained by the current *Constraint*.
- *Body*: If the constraint is not predefined (e.g., ordered), then it is expressed in the body. Objecteering/UML supports natural language. For each generator (C++, Java), certain constraints have a dedicated stereotype (*JavaPrecondition*, *C++Invariant*) and are taken into account during code generation.

Constraint consistency rules

The *Name* and *Body* attributes are exclusive.

Constraint constructor

```
Constraint ModelElement: createConstraint (in String pName,
in String pType, in String pBody)
```

This operation creates a *Constraint* whose name, type and body are given.

```
ModelElement:addConstraint (in Constraint pConstraint)
```

This operation adds a *Constraint* to the current element.

```
Constraint ModelElement:createAndAddConstraint (in String
pName, in String pType, in String pBody)
```

This operation creates and adds to the current element a *Constraint* whose name, type and body are given.

"Note" class

Note overview

```
Note extends Element;
```

Textual part, attached to a *ModelElement*. *Notes* correspond to UML Notes which appear in diagrams, but also to every textual part associated to *ModelElements*. They include implementation code, documentation, or every possible kind of textual information. *Notes* are related to *NoteTypes*, which declare the permitted *Notes* in a model.

In Objecteering/UML, *Notes* belong to their annotated *ModelElement*. (See Example 1).

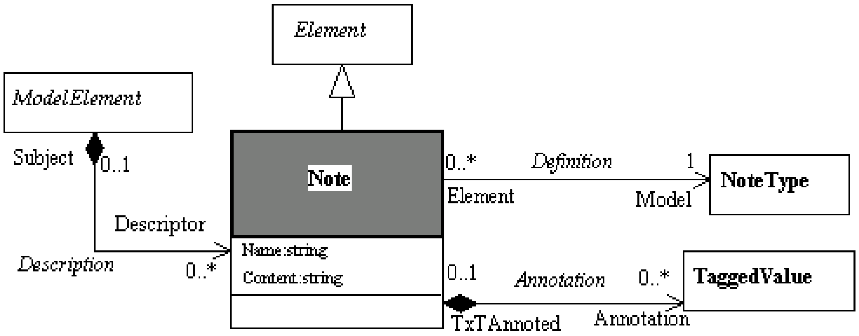


Figure 4-14. Detailed class diagram for *Note*

<u>See also</u>: *NoteType*.

Note properties

The class has the following associations:

- *Model:NoteType*: The *NoteType* defines authorized kinds of *Notes* in a particular context (in specific modules, *Profiles*).
- *Annotation:TaggedValue*: For recursion reasons, a *Note* is not a *ModelElement*. Therefore, the current specific association makes it possible to annotate a *Note* by a *TaggedValue*.

The class owns the following attributes:

- *Name*: Name of the *Note*. This defines the purpose of the *Note*, and has to conform to the *NoteType* Name, when it is defined.
- *Content*: Textual content of the *Note*. This text can be a description or any syntax used for a target language.

Note consistency rules

The *BaseClass* attribute of the *NoteType* associated with a *Note* must be consistent with the modeling element which contains the *Note*.

Note constructor

```
Note ModelElement:createNote (in String pName, in String
pNoteType, in String pContent, in String pMetaClass);
```

This operation creates a *Note* whose type and content are given.

<u>Example</u>:

```
Note MyNote = createNote ("", "comment", "This is a
comment!", "Package");
```

```
ModelElement:addNote (in Note pNote)
```

This operation adds a *Note* to the current element

<u>Example</u>:

```
MyPackage.<addNote (MyNote);
```

```
Note ModelElement:createAndAddNote (in String pName, in
String pNoteType, in String pContent)
```

This operation creates and adds to the current element a *Note* whose type and content are given.

<u>Example</u>:

```
Note MyNote = MyClass.<createAndAddNote ("", "comment",
"This is a comment!");
```

```
Note ModelElement:createAndAddNoteWithTaggedValues (in
String pName, in String pNoteType, in String pContent, in
Object[] pTaggedValues)
```

This operation creates and adds to the current element a *Note* whose *TaggedValues* are given.

<u>Example</u>:

```
Note MyNote = MyPackage.<createAndAddNoteWithTaggedValues
("", "comment", "This is a comment!", TaggedValues);
```

```
Note ModelElement:createNoteWithTaggedValues (in String
pName, in String pNoteType, in String pContent, in Object[]
pTaggedValues, in String pMetaClass)
```

This operation creates a *Note* whose *TaggedValues* are given.

Example:

```
Note MyNote =createNoteWithTaggedValue ("", "comment", "This
is a comment!", TaggedValues, "Package");
```

```
Note clone ()
```

This operation makes a complete copy of the current *Note*.

Example:

```
Note MyCopy =MyNote.<clone ();
```

```
addTaggedValue (in TaggedValue pTaggedValue)
```

This operation adds a *TaggedValue* to the current *Note*.

Example:

```
MyNote.<addTaggedValue (ATaggedValue);
```

Note methods

Scan methods*:*

`TaggedValues[] getTaggedValues()`
Returns the *TaggedValues* which annotate the *Note*.

`NoteType[] getNoteType()`
Returns the types (*NoteType*) of the *Note*.

`boolean isTaggedValue (in boolean ofName)`
Determines if the *Note* is annotated with the *TaggedValue* whose name is given.

`TaggedValue[] getAllTaggedValues (in String ofName)`
Returns the *TaggedValues,* whose name is given, which annotate the *Note*.

`String pathName ()`
This operation returns the complete "path" of a Note.

Example:

```
String MyPath = MyNote.<pathName ();
```

"NoteType" class

NoteType overview

```
NoteType extends ModelElement;
```

Defines a specific kind of *Note*. *NoteTypes* are defined in UML profiling projects within a particular *Profile*. They are conditioned by Objecteering/UML annex tools, editors and generators.

For example, document templates are defined before associated text. This text is determined by textual headings which must be entered for use by generated documents. The same applies to generated code, which determines the textual zones to be entered.

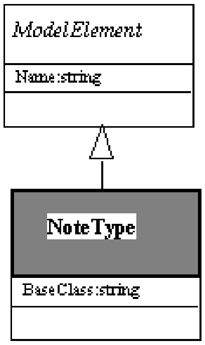


Figure 4-15. Detailed class diagram for *NoteType*

<u>See also</u>: *Note*.

NoteType properties

The class owns the following attribute:

- *BaseClass*: Defines the metaclass to be documented by the occurrences of the *NoteType*.

NoteType constructor

NoteTypes cannot be created using the J language in Objecteering/UML. They belong to the meta-level, and should only be referred to. Services are provided in order to find the available *NotesTypes* given a current Project.

```
NoteType Object : findNoteTypeInProject (in String note
TypeName, in String metaClassName)
```

Returns the *NoteType* defined in the project, which conform to the parameters.

"InternalProduct" class

InternalProduct overview

```
abstract class InternalProduct extends ModelElement;
```

Additional elements attached to *ModelElements*. Represents additional elements attached to *ModelElements* such as work products or diagrams.

An *InternalProduct* belongs to its *ModelElement*.

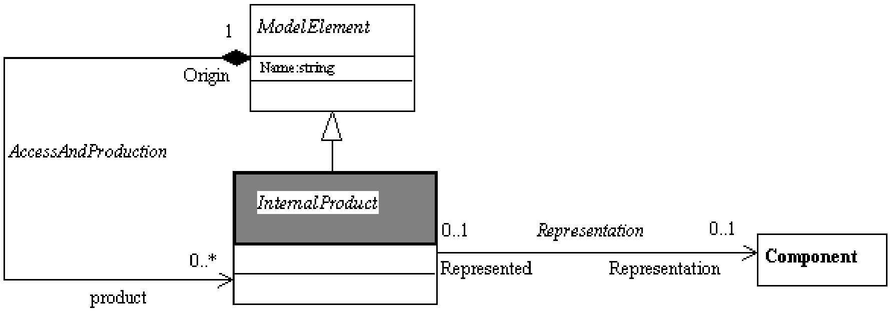


Figure 4-16. Detailed class diagram for *InternalProduct*

See also: *MpGenProduct*

InternalProduct properties

The class has the following associations:

- *Representation:Component*: *Component* which models the *InternalProduct*.

"MpGenProduct" class

MpGenProduct overview

```
MpGenProduct extends InternalProduct;
```

Work products produced automatically by Objecteering/UML. This metaclass provides a logical view on physical parts generated by Objecteering/UML. For example, an HTML document is a set of files, that are represented by this *MpGenProduct* in Objecteering/UML.

Objecteering/UML manages the consistency between the model, this *WorkProduct* and the corresponding physical elements (Files).

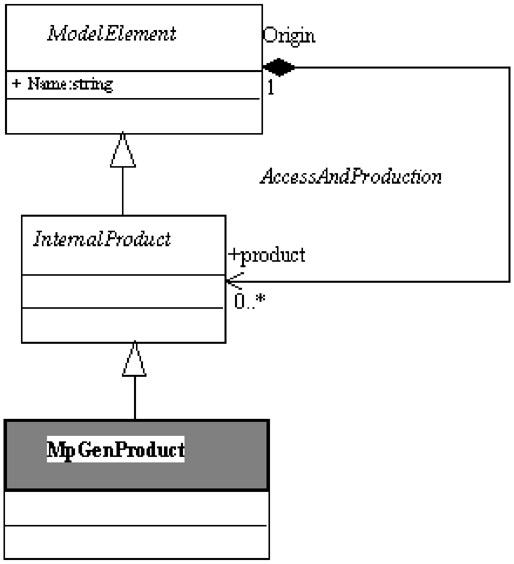


Figure 4-17. Detailed class diagram for *MpGenProduct*

MpGenProduct methods

See the "*Managing work products*" section of chapter 5 ("*Module Management Facilities*" of the *Objecteering/J Libraries* user guide. Work products have several mechanisms that can be used through a set of methods.

Chapter 5: Static Model

Overview

Presentation

The static model is organized by the *NameSpace* concept. The two major notions are *Classifier* and *Package*, with all their properties and dependencies. The *DataFlow* notion is an extension of UML.

Metamodel synthesis

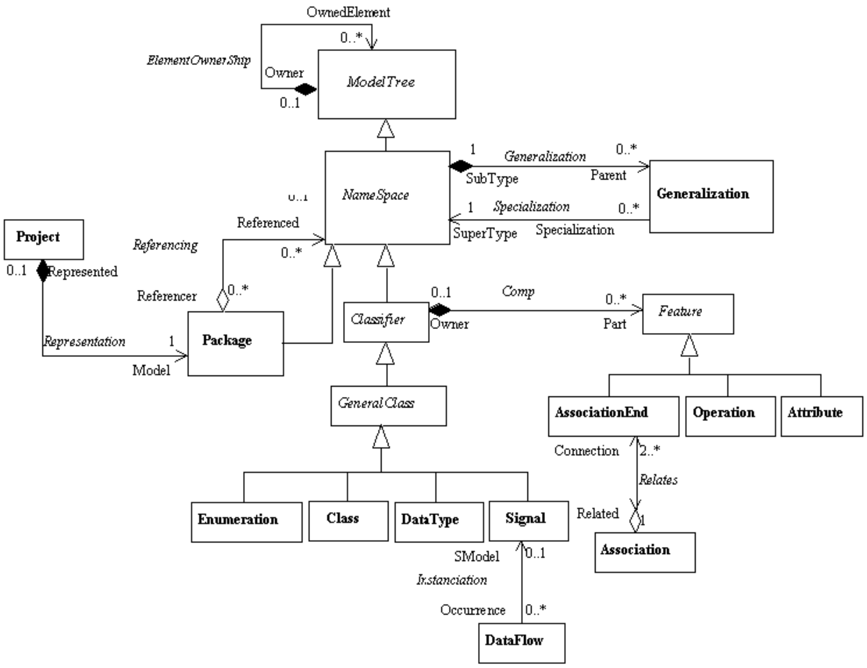
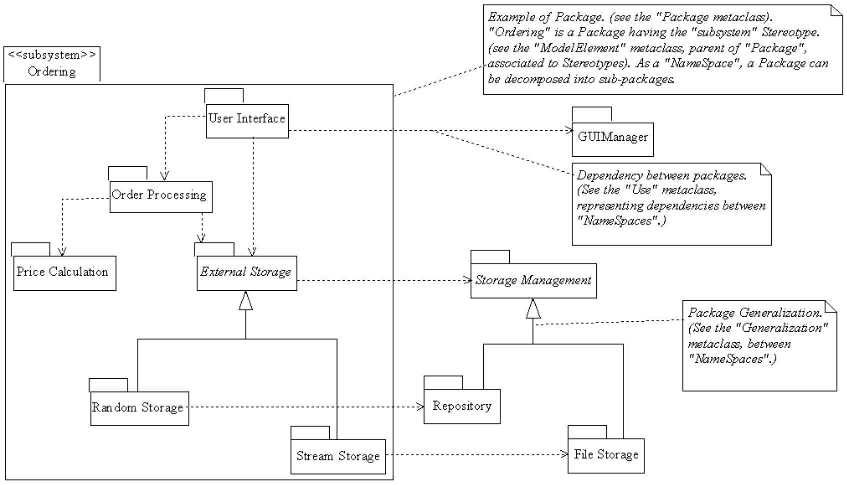


Figure 5-1. The *StaticModel*

StaticModel metaclasses

- *Project*: Working space for a model.
- *ModelTree*: Used to create hierarchical links.
- *NameSpace*: Part of a model in which each name has a unique meaning.
- *Package*: Decomposition unit of a model.
- *Classifier*: Element which describes behavioral and structural features.
- *GeneralClass*: General definition of a class.
- *Class*: Description of a set of objects which share the same attributes, operations, methods, relationships, and semantics.
- *DataType*: A descriptor of a set of primitive values which lack identity.
- *Feature*: Property, like operation or attribute, which is encapsulated within another entity, such as an interface, a class, or a data type.
- *Attribute*: Property of a class.
- *Operation*: Individual pieces of invokable behavior.
- *Parameter*: Information received as input or returned as output by an operation.
- *Association*: Definition of links which may exist between objects.
- *AssociationEnd*: Connection of an association to one of its related classes.
- *ClassAssociation*: Class relating other classes. It is both a class and an association.
- *Enumeration*: Special kind of DataType whose range is a list of predefined values, called EnumerationLiterals.
- *EnumerationLiteral*: Defines an atom (i.e., with no relevant substructure), represents one in the list of values that an enumeration may have.
- *Signal*: Specification of an asynchronous stimulus communicated between instances.
- *DataFlow*: Circulation of information between model elements.
- *Generalization*: Taxonomic relationship between a more general element and a more specific element.
- *Use*: Usage dependency between elements.
- *Realization*: Implementation link between a class and its interface, or between a component and its interface.
- *TemplateParameter*: Parameter for templated elements.

Example 3: Package diagram



Example of Package. (see the "Package metaclass). "Ordering" is a Package having the "subsystem" Stereotype. (see the "ModelElement" metaclass, parent of "Package", associated to Stereotypes). As a "NameSpace", a Package can be decomposed into sub-packages.

Dependency between packages. (See the "Use" metaclass, representing dependencies between "NameSpaces".)

Package Generalization. (See the "Generalization" metaclass, between "NameSpaces".)

Figure 5-2. Subsystem package made up of packages of various functionality areas

Packages, composed packages, use links and Generalization metaclasses feature in this diagram.

Example 4: Class diagram which has various combinations of classes

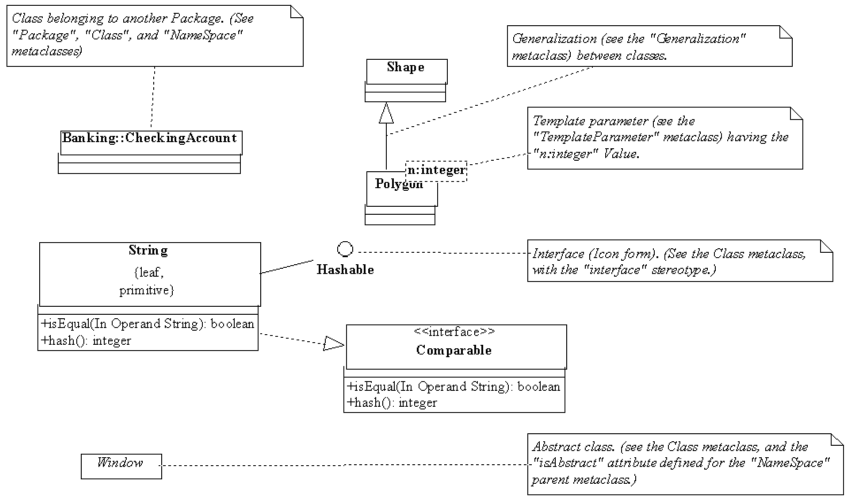
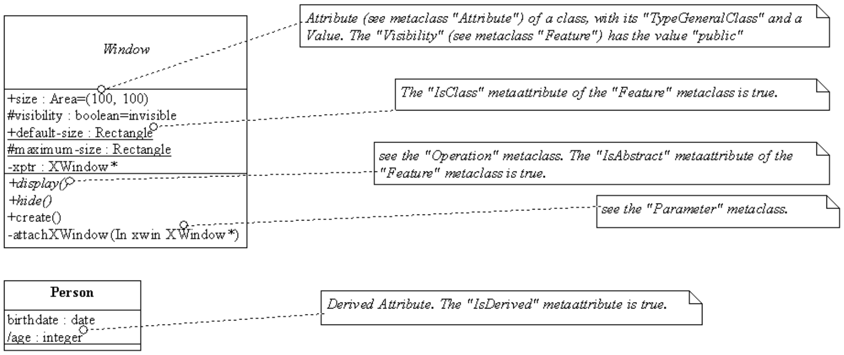

Class belonging to another Package. (See "Package", "Class", and "NameSpace" metaclasses)
Banking::CheckingAccount
Shape
Generalization (see the "Generalization" metaclass) between classes.
Template parameter (see the "TemplateParameter" metaclass) having the "n:integer" Value.
n:integer
Polygon
String
{leaf, primitive}
+isEqual(In Operand String): boolean
+hash(): integer
Hashable
Interface (Icon form). (See the Class metaclass, with the "interface" stereotype.)
<<interface>>
Comparable
+isEqual(In Operand String): boolean
+hash(): integer
Window
Abstract class. (see the Class metaclass, and the "isAbstract" attribute defined for the "NameSpace" parent metaclass.)

Figure 5-3. Different examples of classes

Example 5: Various kinds of class members


Window
+size : Area=(100, 100)
#visibility : boolean=invisible
+default-size : Rectangle
#maximum-size : Rectangle
-xptr : XWindow*
+display()
+hide()
+create()
-attachXWindow(In xwin XWindow*)
Person
birthdate : date
/age : integer

Attribute (see metaclass "Attribute") of a class, with its "TypeGeneralClass" and a Value. The "Visibility" (see metaclass "Feature") has the value "public"

The "IsClass" metaattribute of the "Feature" metaclass is true.

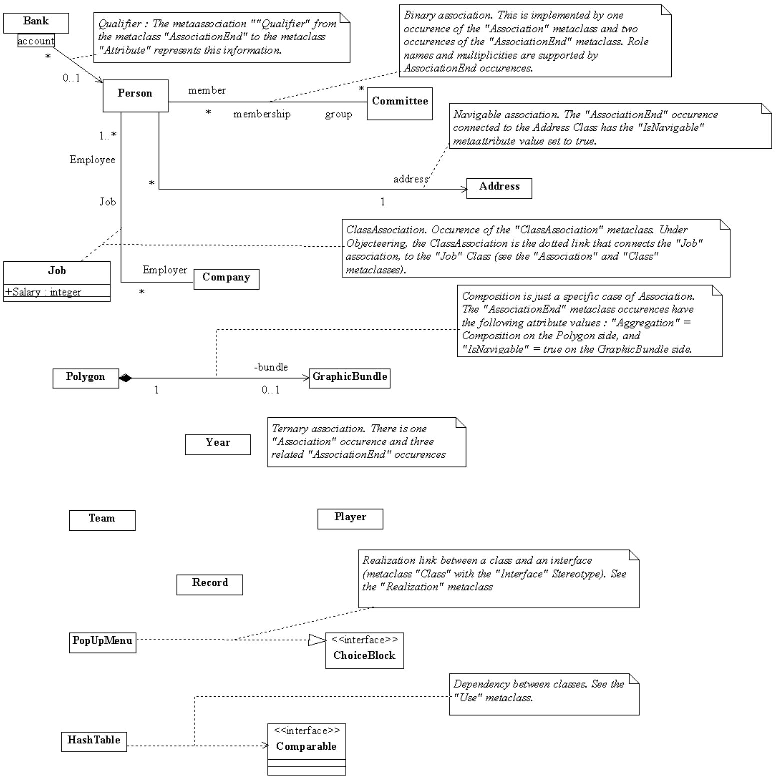
see the "Operation" metaclass. The "IsAbstract" metaattribute of the "Feature" metaclass is true.

see the "Parameter" metaclass.

Derived Attribute. The "IsDerived" metaattribute is true.

Figure 5-4. Various examples of class features (*Feature*, *Attribute*, *Operation* metaclasses)

Example 6: Various dependencies between classes (Associations and ClassAssociations, Realization, dependencies)



Qualifier : The metaassociation ""Qualifier" from the metaclass "AssociationEnd" to the metaclass "Attribute" represents this information.

Binary association. This is implemented by one occurence of the "Association" metaclass and two occurences of the "AssociationEnd" metaclass. Role names and multiplicities are supported by AssociationEnd occurences.

Navigable association. The "AssociationEnd" occurence connected to the Address Class has the "IsNavigable" metaattribute value set to true.

ClassAssociation. Occurence of the "ClassAssociation" metaclass. Under Objecteering, the ClassAssociation is the dotted link that connects the "Job" association, to the "Job" Class (see the "Association" and "Class" metaclasses).

Composition is just a specific case of Association. The "AssociationEnd" metaclass occurences have the following attribute values : "Aggregation" = Composition on the Polygon side, and "IsNavigable" = true on the GraphicBundle side.

Ternary association. There is one "Association" occurence and three related "AssociationEnd" occurences

Realization link between a class and an interface (metaclass "Class" with the "Interface" Stereotype). See the "Realization" metaclass

Dependency between classes. See the "Use" metaclass.

Figure 5-5. Class dependencies

For further information, please see the "*Association*", "*AssociationLink*", "*Qualifiers*", "*Class*", "*ClassAssociation*", "*Realization*" and "*Use*" metaclasses

"Project" class

Project overview

```
Project extends ModelElement;
```

Working space for a model. A *Project* in Objecteering/UML corresponds to the UML "*Model*" concept. A *Project* in Objecteering/UML has a root *Package*, and in addition a configuration, expressing the loaded modules (and their *Profiles*), the value of the *ModuleParameters*, etc.

A *Project* does not belong to any other element.

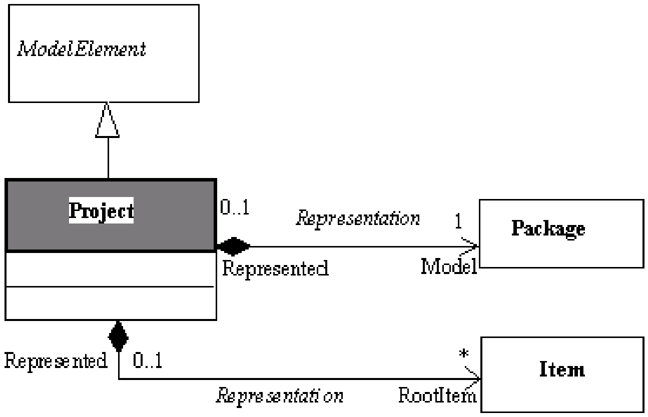


Figure 5-6. Detailed class diagram for *Project*

See also: *Package*.

Project properties

The class has the following associations:

- *Model:Package*: Defines the *Package* associated to the *Project* (Equivalent to the UML "*Model*" notion), that is the root of the *Project's Package* organization.
- *RootItem:Item*: Specifies an *Item* which may be the root of an *Item* tree.

Please refer to the "*Project management services*" section in chapter 7 of the *Objecteering/J Libraries User Guide* user guide.

"NameSpace" class

NameSpace overview

```
abstract class NameSpace extends ModelTree ;
```

Part of a model in which each name has a unique meaning. A *NameSpace* encompasses both the UML notion of *NameSpace* and of *GeneralizableElement*. *NamesSpaces* are *Packages* and *Classifiers*.

In Objecteering/UML, a *NameSpace* belongs to another *NameSpace*, with the single exception of the root *NameSpace*, which is associated to the *Project*.

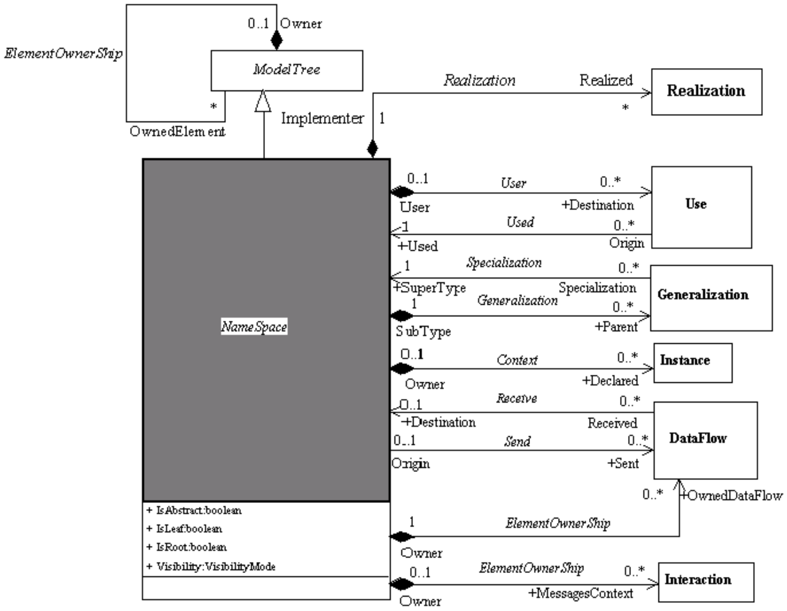


Figure 5-7. Metamodel of the *NameSpace* metaclass

NameSpace properties

The class has the following associations:

- *Sent:DataFlow*: *DataFlows* sent by the *NameSpace*
- *Parent:Generalization*: Association to the parent *NameSpace* through the intermediate *Generalization* Class.
- *Declared:Instance*: *Instances* declared in the context of the current *NameSpace*.
- *Destination:Use: NameSpaces* "*used*" (having an <<access>> or an <<import>> dependency) by the current *NameSpace*.
- *OwnedDataFlow:DataFlow*: *DataFlows* belonging to a *NameSpace*. If they have an origin *NameSpace*, then it is the owner. Otherwise, the *NameSpace* which constitute the definition context of the *DataFlow* (typically designates the *Package* in which the diagram has been defined) will be the owner.
- *MessagesContext:Interaction*: Defines various interactions that may exist between instances belonging to the current *NameSpace*.
- *Realized:Realization*: Link to the *Realization* intermediate metaclass, in order to define the realization link between an *Interface* and a *Class*.

The class owns the following attributes:

- *IsAbstract*: An abstract *NameSpace* is defined on a very general level and does not have direct *instances*.
- *IsLeaf*: Determines if the *NameSpace* is an inheritance tree leaf. This prohibits future inheritance.
- *IsRoot*: Determines that the current *NameSpace* is the root of a *Generalization* tree.
- *Visibility*: Defines the visibility of the *NameSpace*, in its owning *NameSpace*. (visibility of a *Class* in a *Package*, for example)

NameSpace consistency rules

For an element *E* contained in a *NameSpace N*, *E* will be able to access element *E'*, if *E'* :

- belongs to *N*, or belongs to a *NameSpace N1*, with "*public*" visibility, and where *N1* checks the following properties:
    - *N1* belongs to *N*, or
    - *N1* is used by *N*, or
    - *N1* is referenced by *N*, or
    - these dependencies are run through on a single level (non transitivity)
- or belongs to a parent *NameSpace N2* of *N*, with public or protected visibility, or belongs to a *NameSpace N3* which checks one of the properties above (*N1* to *N*) with regard to *N2'* (*N3'* belongs to, is referenced by or is used by *N2'*).

A *NameSpace* has a unique name in its naming space, which itself is a *NameSpace*. The uniqueness check concerns the components and the elements referenced by a *NameSpace*. Furthermore, predefined types (integer, etc.) which cannot have homonyms are also used.

The "uniqueness of names" rule also concerns instances defined in the *NameSpace*.

A maximum of one generalization link can exist between two *NameSpaces*.

A maximum of one use link can exist between two *NameSpaces*.

An abstract *NameSpace* can only specialize an abstract *NameSpace*.(Warning)

A Root *NameSpace* cannot have parents.

A *NameSpace* cannot specialize a "Leaf" *NameSpace*.

A Leaf *NameSpace* cannot be abstract. This exclusion is only managed by the entry dialog box.

Cycles between *Generalization* links cannot exist.

NameSpace methods

```
DataType findType (in String typeName)
```

This operation returns the *DataType* whose name is given if present in the current element.

Example:

```
DataType MyColor = MyPackage.<findType ("Color");
```

```
Enumeration findEnumerate (in String enumName)
```

This operation returns the *Enumeration* whose name is given if present in the current element.

Example:

```
Enumeration MyColor = MyPackage.<findEnumerate ("Color");
```

```
Class getClassByName (in String pName)
```

This operation returns the first *Class* owned by the element whose name is given.

Example:

```
Class MyClass =MyPackage.getClassByName ("MyClass");
```

```
Package getPackageByName (in String pName)
```

This operation returns the first *Package* owned by the element whose name is given.

Example:

```
Package MyPackage = AParentPackage.getPackageByName
("MyPackage");
```

```
NameSpace containsThisNameSpace (in NameSpace pNameSpace)
```

This operation returns the *Parameter* if it is owned by the element and has the same name.

Example:

```
Class MyClass =MyPackage.containsThisNameSpace (ThisClass);
```

```
NameSpace: addClass (inout Class pClass)
```

This operation adds a *Class* to the current *NameSpace*.

Example:

```
MyPackage.<addClass (Aclass);
```

```
NameSpace:addEnumeration (inout Enumeration pEnumeration)
```

This operation adds an *Enumeration* to the current element.

Example:

```
MyPackage.<addEnumeration (AEnumeration);
```

```
NameSpace:addDataType (inout DataType pDataType)
```

This operation adds a *DataType* to the current element.

Example:

```
MyPackage.<addDataType (ADataType);
```

```
NameSpace:addUseCase (inout UseCase pUseCase)
```

This operation adds a *UseCase* to the current element.

```
NameSpace:addActor (inout Actor pActor)
```

This operation adds an *Actor* to the current element.

Example:

```
MyPackage.<addActor (MyActor).
```

```
NameSpace:setNameSpaceParams (in VisibilityMode pVisibility,
in boolean pIsAbstract, in boolean pIsLeaf, in boolean
pIsRoot)
```

This operation sets attributes to a *NameSpace*.

Example:

```
MyClass.<setNameSpaceParams (Public, false, false, false);
```

```
NameSpace [] getUsedNameSpaces
```

This operation returns the *NameSpaces* used by the *NameSpace*.

"Package" class

Package overview

```
Package extends NameSpace;
```

Decomposition unit of a model. The *Package* is the main structuring unit of models. It defines a hierarchy that decomposes a Model. *Packages* can contain *Packages*, *Classifiers*, etc.

Example 3 presents examples of *Packages*, package embedding, and dependencies between *Packages*.

A *Package* belongs to its parent *Package*, represented as a *NameSpace* in the metamodel, except for the root *Package*, which belongs to a *Project*.

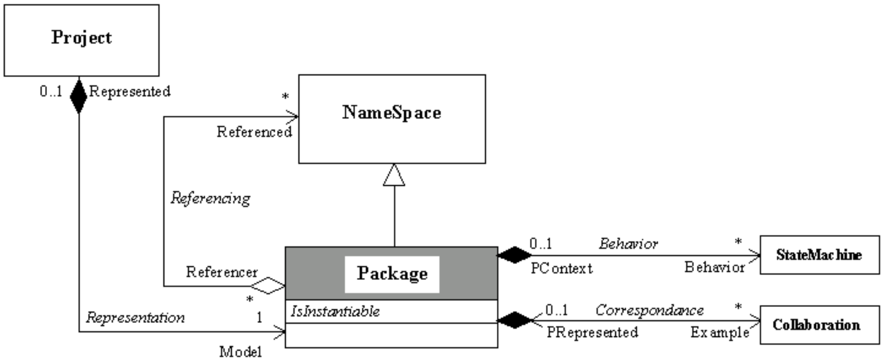


Figure 5-8. Detailed class diagram for *Package*

Package properties

The class has the following associations:

- *Referenced:NameSpace*: Referenced *NameSpaces*. Referencing is a specific UML mechanism, that provides accessibility to the referenced element, without having the hierarchical ownership constraints.
- *Behavior:StateMachine*: *StateMachines* that may express behavior at the *Package* level.
- *Example:Collaboration*: *Collaborations* expressing the dynamics of the current *Package*.

The class has the following attribute:

- *IsInstantiable*: Indicates that the *Package* may be instantiated.

Package consistency rules

- A *Package* belongs to a *Package* or to a *Project*.
- A *Package* can only specialize a *Package*.
- Generalization links between *Packages* are acyclic.
- Use links between *Packages* are acyclic.
- A *Package* uses only *Packages*.
- A *Package* cannot specialize and use a *Package* at the same time.
- The combination of generalization links and use links cannot form a cycle.
- A *StateMachine* is unique in the *Package*.
- A *Collaboration* is unique in the *Package*.
- A *NameSpace* is unique in the *Package*.
- An *Instance* is unique in the package.
- A *Package* can reference any *NameSpace* (*NameSpace* visibility checks do not apply).
- A *Package* stereotyped <<sub-system>> can implement or use interfaces.
- According to element accessibility rules for a *Package*, *P1*, embedded *Packages* can also access interfaces accessed by their owner *Package*.

Package constructor

```
Package ModelElement:createPackage (in String pName)
```

This operation creates a *Package* whose name is given.

Example:

```
Package MyPackage = createPackage ("MyPackage");
```

```
Package createAndAddPackage (in String pName)
```

This operation creates and adds to the current *Package* a *sub-Package*.

Example:

```
Package MyPackage =Apackage.<createAndAddPackage
("MyPackage");
```

```
Package:addPackage (inoutPackage pPackage)
```

This operation makes a *Package* owns another one.

Example:

```
MyOwnerPackage.<addPackage (AnOwnedPackage);
```

```
addReference (in NameSpace pReferenced)
```

This operation makes a *Package* reference another one.

Example:

```
MyReferencerPackage.<addReference (AReferencedPackage);
```

Package methods

`Actor[] getActors ()`
Returns the *Actors* of the *Package*.

`Actor[] getReferencedActors()`
Returns the *Actors* referenced by the *Package*.

`StateMachine[] getStateMachines()`
Returns the *StateMachines* expressing the behavior at the package level.

`UseCase[] getUseCases()`
Returns the *UseCases* of the *Package*.

`UseCase[] getReferencedUseCases ()`
Returns the *UseCases* referenced by the *Package*.

`Class[] getAllClasses()`
Returns the *Classes* of the *Package*.

`Class[] getReferencedClasses()`
Returns the *Classes* referenced by the *Package*.

`Collaboration[] getCollaborations()`
Returns the *Collaborations* which express the dynamic of the *Package*.

`Diagram[] getDiagrams()`
Returns the *Class*, *Deployment*, *Deployment instance*, *Object*, *Sequence* and *use case Diagrams* of the *Package*.

`Class NameSpace[] getUsedNameSpaces()`
Returns the *Elements* used by the *Package*.

`NameSpace[] getReferencedNameSpaces()`
Returns the *Elements* referenced by the *Package*.

`Enumeration[] getEnumerations()`
Returns the *Enumerations* of the *Package.*

`DataFlow[] getSentDataFlows()`
Returns the *DataFlows* sent by the *Package*.

`DataFlow[] getReceivedDataFlows()`
Returns the *DataFlows* received by the *Package*.

`Instance[] getInstances()`
Returns the *Instances* declared in the context of the *Package*.

`Package[] getParentPackages()`
Returns the parent *Package* which is specialized by the current *Package*.

`Package[] getUsedPackages()`
Returns the used *Packages*.

`Package[] getReferencedPackages()`
Returns the *Packages* referenced by the *Packages().*

`DataType[] getDataTypes()`
Returns the *DataTypes* of the *Package*.

`Package[] getPackages()`
Returns the *sub-packages* of the *Package*.

`Component[] getComponents()`
Returns the *Components* of the *Package*.

`Node[] getNodes()`
Returns the *Nodes* of the *Package*.

`NodeInstance[] getNodeInstances()`
Returns the *NodeInstances* of the *Package*.

`ComponentInstance[] getComponentInstances()`
Returns the *ComponentInstances* of the *Package*.

`Component[] getReferencedComponents()`
Returns the *Components* referenced by the *Package*.

`Node[] getReferencedNodes()`
Returns the *Nodes* referenced by the *Package*.

"Classifier" class

Classifier overview

```
abstract class Classifier extends NameSpace;
```

Element that describes behavioral and structural features. A *Classifier* is an abstract view of the most important metaclasses such as *Class*, *UseCase*, *Actor*, *Component* and *Node*. A *Classifier* notably factorizes the aggregation to features.

A *Classifier* is owned by a *NameSpace* (see the concrete subclasses).

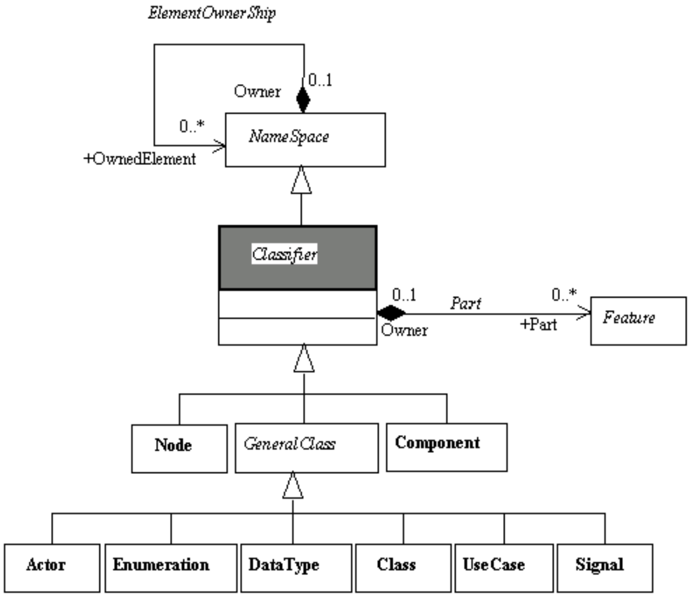


Figure 5-9. Detailed class diagram for *Classifier*

See also: *Class*, *GeneralClass*, *Component*, *Node*, *UseCase*, *Actor*.

Classifier properties

The class has the following association:

- *Part:Feature*: Relates a *Classifier* to its *Attributes*, *Operations*, and also navigable *AssociationEnds*.

Classifier consistency rules

A *Classifier* accesses its composed *NameSpace*, public *NameSpaces* contained in or referenced by the composed *NameSpace*, *NameSpaces* used by its compound and public *NameSpaces* contained in or referenced by them. It recursively accesses the parent *NameSpaces* of its compound, and the public *NameSpaces* contained in or referenced by them.

The names of the current *Classifier*'s *AssociationEnds*, *Operations* and *Attributes* must be unique. This uniqueness also applies to the parent *Classifiers* of the current *Classifier* (Warning).

The signature of an operation intervenes in its control of uniqueness, with regard to other operations. It must be different from the signatures of the current *Classifier*'s operations, and also from the signatures of the compound's operations (Warning), if the operation is not redefined.

For members of parents, only public and protected visibility members are checked.

There is no check on repetition in parents for the "*create*" and "*delete*" operations.

There is no check on repetition if the member name is "*undefined*".

A *Classifier* can implement an interface once only, but may implement several.

A *Classifier* may only implement interface classes. An interface class cannot implement another interface class.

A non-abstract *Classifier* cannot have abstract methods (Warning).

A protected *Classifier* is accessible by public and protected *Classifiers*, its *NameSpace* , only through private relationships. (Warning)

A private *Classifier* is accessible by public and protected *Classifiers*, its *NameSpace*, only through private relationships. (Warning)

Classifier methods

```
Classifier:addAttribute (inout Attribute pAttribute)
```

This operation adds an *Attribute* to the current element.

Example:

```
MyClass.<addAttribute (AnAttribute);
```

```
Classifier:addAssociationEnd (inout AssociationEnd
pAssociationEnd)
```

This operation adds an *AssociationEnd* to the current element.

Example:

```
MyClass.<addAssociationEnd (AnAssociationEnd);
```

```
Classifier:addOperation (inout Operation pOperation)
```

This operation adds an *Operation* to the current element.

Example:

```
MyClass.<addOperation (AnOperation);
```

```
FeatureClassifier containsThisFeature (in Feature pFeature)
```

This operation returns the *Feature* if it is owned by the element.

Example:

```
Operation MyOperation = MyClass.<containsThisFeature
(ThisOperation);
```

```
Attribute[] getAttributes()
```

Returns the *Attributes* of the *Classifier*.

```
Operation[] getOperations()
```

Returns the *Operations* of the *Classifier*.

"GeneralClass" class

GeneralClass overview

```
abstract class GeneralClass extends Classifier;
```

A *GeneralClass* is an elaborated *Classifier*.

A *GeneralClass* belongs to its *NameSpace*.

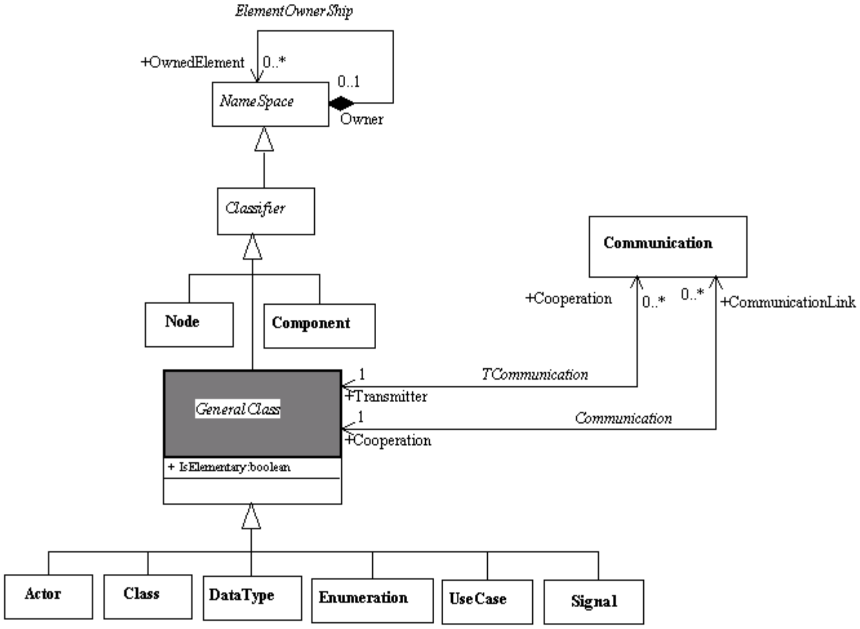


Figure 5-10. Detailed class diagram for *GeneralClass*

See also: *Class*, *Actor*, *DataType*, *Enumeration*, *UseCase*, *Signal*.

GeneralClass properties

The class has the following associations:

- *Cooperation:Communication*: Manages the *Actor/UseCase* communication link. This link is symmetrical.
- *CommunicationLink:Communication*: Manages the *Actor/UseCase* communication link. This link is symmetrical.

The class owns the following attribute:

- *IsElementary* : Determines whether a *Class* is elementary or primitive. A *Class* is primitive if its value cannot be decomposed and its instances are not handled by the application. For example, "*integer*" and "*boolean*" are elementary *Classes*, whereas "*Human*" or "*Device*" are generally not.

GeneralClass consistency rules

No *Associations* can exist on a primitive *Class*.

A non primitive *Class* cannot type an attribute.

"Class" class

Class overview

```
Class extends GeneralClass;
```

Description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. The *Class* is the major concept for the object-oriented model. It specifies which *Instances* can exist in an application.

In Objecteering/UML, an *Interface* is a specific kind of *Class*, which is represented by a predefined *Stereotype*. (See Example 4 and 6 for Class diagrams).

In Objecteering/UML, a *Class* is owned by a *NameSpace*, which can be a *Package* or a *Class*.

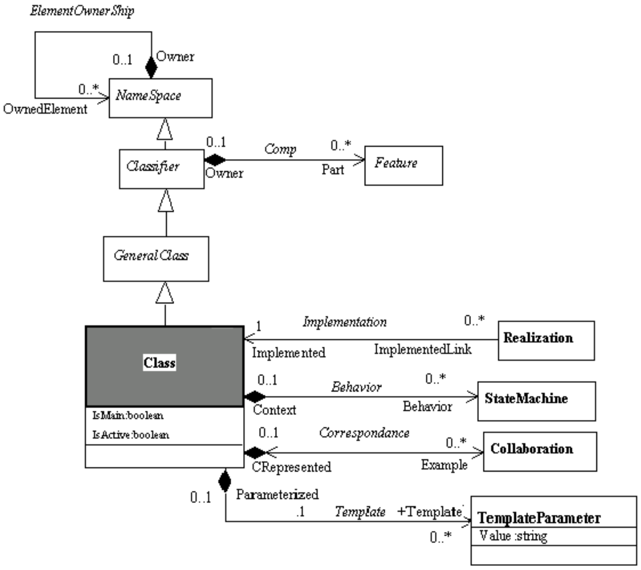


Figure 5-11. Detailed class diagram for *Class*

<u>See also</u>: *Classifier*, *NameSpace*, *Feature*, *Attribute*, *Operation*.

Class properties

A *Class* has the following associations:

- *Template:TemplateParameter*: In case of template *Classes*, this association defines its template parameters.
- *Behavior:StateMachine*: Defines the state machines that specify the behavior of the *Class*.
- *Example:Collaboration*: Collaborations expressing the dynamics of the current *NameSpace*.

The class owns the following attributes:

- *IsMain*: A main class is a *Class* whose unique instance represents the application.
- *IsActive*: Specifies whether an *Object* of the *Class* maintains its own thread of control. If true, then an *Object* has its own thread of control and runs concurrently with other active *Objects*. If false, then *Operations* run in the address space and under the control of the active *Object* that controls the caller.

Class consistency rules

- A *Class* can belong to a *Package*, or to a *Class*.
- A *Class* has no communication links.
- A *Class* can only contain *Classes*, *DataTypes* and *Enumerations*.
- A *Class* can only specialize (*Generalization*) *Class*. Between two *Classes*, a maximum of one *generalization* link may exist.
- An *Interface* class can only specialize an *Interface* class.
- An *Active* class can only specialize an *Active* class.
- A *Class* can only use (*Use*) *Classes*. Between two *Classes*, a maximum of one *Use* link may exist.
- An active *Class* cannot be *Interface*.
- A primitive *Class* can have neither *StateMachines* nor *Collaborations*.

Class constructor

```
Class ModelElement:createClass (in String pName)
```

This operation creates a *Class* whose name is given.

<u>Example</u>:

```
Class MyClass = createClass ("MyClass");
```

```
Class NameSpace:createAndAddClass (in String pName)
```

This operation creates and adds to the current *NameSpace* a *Class* whose name is given.

<u>Example</u>:

```
Class MyClass = MyPackage.<createAndAddClass ("MyClass");
```

Class methods

```
StateMachine[] getStateMachines()
```

Returns the *StateMachine* describing the behavior of the *Class*.

```
Class[] getClasses()
```

Returns the sub-classes of the *Class*.

```
Class[] getUsedClasses()
```

Returns used *Classes*.

```
Class[] getParentClasses()
```

Returns parent *Classes*.

```
Collaboration[] getCollaborations()
```

Returns *Collaborations* expressing the dynamic of the *Class*.

```
Diagram[] getDiagrams()
```

Returns the *Class*, *Object*, or *Sequence* diagrams of the *Class*.

```
Enumeration[] getEnumerations()
```

Returns the *Enumerations* defined locally in the *Class*.

```
DataFlow[] getSentDataFlows()
```

Returns the *DataFlows* sent by the *Class*.

```
DataFlow[] getReceivedDataFlows()
```

Returns the *DataFlows* received by the *Class*.

```
Instance[] getInstances()
```

Returns the *Instances* declared in the context of the *Class*.

```
Class[] getImplementedInterfaces()
```

Returns implemented *Interfaces*.

```
AssociationEnd[] getAssociationEnds()
```

Returns the navigable links starting from the *Class*.

```
AssociationEnd[] getNoNavigableAssociationEnds()
```

Returns the non-navigable links starting from the *Class*.

```
DataType[] getDataTypes()
```

Returns the *Datatypes* declared in the *Class*.

```
boolean isAbstract()
```

Determines if the *Class* in an abstract *Class*.

```
boolean isMain()
```

Determines if the *Class* is a main *Class*.

```
String pathName()
```

This operation returns the complete "*path*" of a *Class*. The path is the composition link of the *Packages* and *Classes* containing this *Class*, such as "*P_1 : P_2 : Class*".

Example:

```
String MyPath =MyClass.<pathName();
```

```
Class:setClassParams (in boolean pIsElementary, in boolean
pIsMain)
```

This operation sets parameters of a *Class*.

Example:

```
MyClass.<setClassParams (false, false);
```

```
Class Class:clone()
```

This operation makes a complete copy of the current *Class*.

Example:

```
Class MyCopy = Class MyClass.<clone();
```

"DataType" class

DataType overview

```
DataType extends GeneralClass;
```

A descriptor of a set of primitive values which lacks identity.

DataTypes include numbers, Strings, and enumerated values. *DataTypes* are passed by value and are immutable entities. *DataTypes* can be used as primitive classes.

In Objecteering/UML, *DataTypes* belong to a *NameSpace*.

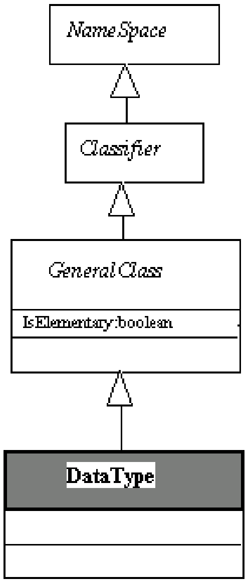


Figure 5-12. Detailed class diagram for *DataType*

DataType consistency rules

A *DataType* cannot contain *NameSpaces*.

A *DataType* neither sends nor receives *DataFlows*.

A *DataType* does not implement Interface classes.

A *DataType* has no communication links.

A *DataType* can only belong to a *Class*, a *Package*, or a *Signal*.

A *DataType* may only specialize (*Generalization*) a *DataType*.

A *DataType* may only use (*Use*) a *DataType*.

DataType constructor

```
DataType NameSpace:createAndAddDataType (in String pName)
```

This operation creates and adds to the current *NameSpace* a *DataType* whose name is given.

Example:

```
DataType MyDataType = MyClass.<createAndAddDataType
("MyDataType");
```

```
DataType ModelElement:createDataType (in String pName)
```

This operation creates a *DataType* whose name is given.

Example:

```
DataType MyNodeType = createDataType ("Node");
```

DataType methods

```
Object Object:findPredefinedType (in String TypeName)
```

Returns the "*TypeName*" predefined name type or empty if not found. The type is searched for in the "*_predefinedTypes*" UML modeling project. Returned elements are generally of "*DataType*" type. This service is very practical in finding representatives of predefined types, like, for example, "*int*".

```
AssociationEnd[] getAssociationEnds()
```

Returns the navigable links starting from the *DataType*.

```
DataType[] getParentDataTypes()
```

Returns the parent *DataTypes*.

```
DataType[] getUsedDataTypes()
```

Returns the *DataTypes* used by the *DataType*.

"Feature" class

Feature overview

```
abstract class Feature extends ModelElement;
```

Property, like operation or attribute, which is encapsulated within another entity, such as an *Interface*, a *Class*, or a *DataType*.

In the metamodel, a *Feature* declares a behavioral or structural characteristic of an *Instance* of a *Classifier* or of the *Classifier* itself.

Properties of a *Class* that can be handled in an abstract way. In Objecteering/UML, an *AssociationEnd* is also a *Feature*. The *Name* of a *Feature* corresponds to the name of the role of the opposite related *Class*.

In Objecteering/UML, a *Feature* belongs to its *Classifier*.

See Example 5 where several examples of *Feature* exist.

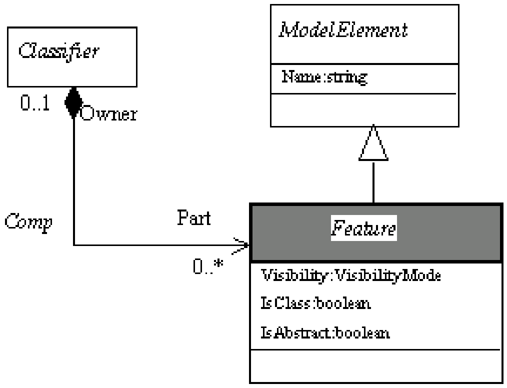


Figure 5-13. Detailed class diagram for *Feature*

<u>See also</u>: *Classifier*, *Enumerated types*.

Feature properties

The class owns the following attributes:

- *Visibility*: Member visibility (Public, Protected or Private).
- *IsClass*: Specifies a class member, that is shared by all instances of the class.
- *IsAbstract*: Determines abstract *Features*, i.e. those not implemented at this level.

Feature consistency rules

A *Feature* belongs to a *Classifier*.

It has a unique name in its *Classifier*. The uniqueness of the name is not checked if the name is *undefined*.

Feature methods

```
Classifier[] getComposedClass()
```

Provides the *Classifier* which contains the *Feature*.

```
Feature:SetFeatureParams (in VisibilityMode pVisibility, in
boolean pIsClass, in boolean pIsAbstract)
```

This operation sets parameters of a *Feature*.

Example:

```
MyOperation.<setFeatureParams (Public, false, false);
```

"Attribute" class

Attribute overview

```
Attribute extends Feature;
```

Property of a *Class*. An *Attribute* is a named slot within a *Classifier* that describes a range of values that instances of the said *Classifier* may hold. (See Example 5)

In Objecteering/UML, an *Attribute* belongs to a *GeneralClass* as a *Feature*, or to an *AssociationEnd* as a Qualifier.

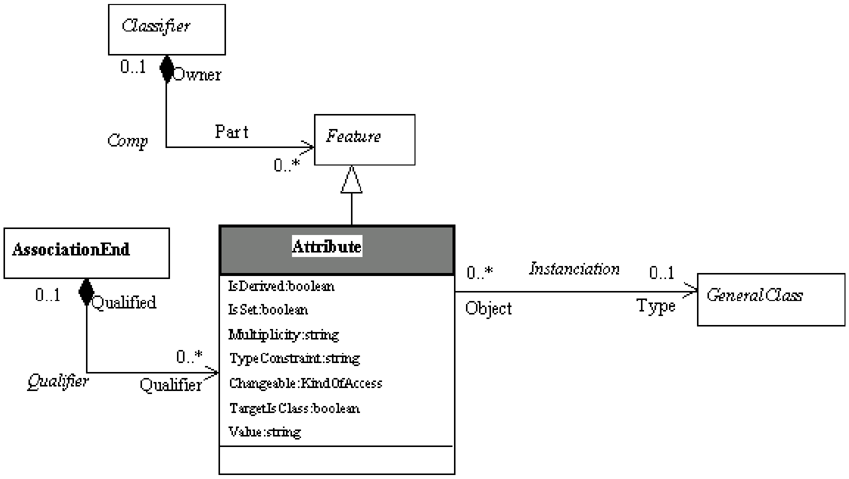


Figure 5-14. Detailed class diagram for *Attribute*

See also: *GeneralClass, Classifier*, *Enumerate type*.

Attribute properties

The class has the following association:

- *Type:GeneralClass*: Determines which class is the *Attribute*'s type

The class has the following attributes:

- *IsDerived*: Determines if the *Attribute* is a dynamic dependency, i.e. its value is calculated dynamically, via an expression.
- *IsSet*: Determines whether the *Attribute* is a set.
- *Multiplicity*: Provides, if necessary, the size of the set.
- *TypeConstraint*: Provides an indication of the instantiation of the *Attribute*'s elementary class. For example, in the case of an *Attribute String*, *TypeConstraint* determines the size of the String (*, 10, etc.).
- *Changeable*: Determines the access mode of the *Attribute* (read, write, read/write, neither).
- *TargetIsClass*: Determines that the target itself is a metaclass.
- *Value*: Default value of the *Attribute*. This value is assigned to the Instances upon creation, unless a specific value is defined.

Attribute consistency rules

- An *Attribute* can belong to a *Class*, an *Actor*, a *UseCase*, a *Component*, a *Node* or an association link.
- An *Attribute* can be typed by a *Class*, a *DataType* or a primitive *Enumeration*.
- The name of an *Attribute* must be different from those of its *AssociationEnds*, the *Operations* and the *Attribute* of its *owner*. The rule also applies to the parents of its *owner* (Warning).
- An *Attribute* has a *Classifier* as type, which is accessible from its owner, in accordance with the visibility rules on *Classifier*.

Attribute constructor

```
Attribute ModelElement:createAttribute (in String pName, in
GeneralClass pType, in String pInitValue, in String
pMultiplicity)
```

This operation creates an *Attribute* whose type is a *GeneralClass*.

Example:

```
Attribute MyColor = createAttribute ("MyColor", ColorEnum,
"Blue", "1");
```

```
Attribute Modelelement:createPredefinedAttribute (in String
pName, in String pType, in String pInitValue, in String
pMultiplicity)
```

This operation creates an *Attribute* whose type is *int*, *real*, *char*, *string*, *boolean* or *undefined*. It avoids having to get a reference to these predefined types before getting the *Attribute*.

Example:

```
Attribute MyRate = createPredefinedAttribute ("MyRate",
"real", "", "1");
```

```
Attribute Classifier:createAndAddPredefinedAttribute (in
String pName, in String pType, in String pInitValue, in
String pMultiplicity)
```

This operation creates and adds to the current element an *Attribute* whose type is either *int*, *char*, *real*, *string*, *boolean* or *undefined*.

Example:

```
Attribute MyAttribute
=Aclass.<createAndAddPredefinedAttribute ("name", "String",
"Lucas", "1");
```

```
Attribute Classifier:createAndAddAttribute (in String pName,
in GeneralClass pType, in String pInitValue, in String
pMultiplicity)
```

This operation creates and adds to the current element an *Attribute* whose type is a *GeneralClass*.

Example:

```
Attribute MyAttribute =Aclass.<createAndAddAttribute
("MyAttribute", ColorEnum, "Blue", "l");
```

Attribute methods

```
GeneralClass[] getType()
```

Returns the type of the *Attribute*.

```
boolean isClass()
```

Returns "*true*" if the *Attribute* is a class member.

```
boolean isSet()
```

Returns "*true*" if the *Attribute* is a set of elements.

```
String pathName()
```

This operation returns the complete "*path*" of an *Attribute*, that is the composition list of *Packages*, and classes for the *Attribute*, such as "*P_1 : P_2 : C_1 : Attribute*".

Example:

```
String MyPath =MyAttribute.<pathName();
```

```
Attribute:setAttributeParams (in KindOfAccess pChangeable,
in String pTypeConstraint, in boolean pIsSet, in boolean
pTargetIsClass, in boolean pIsDerived)
```

This operation sets parameters of an *Attribute*. It has to be excecuted in a J session.

Example:

```
MyAttribute.<setAttributeParams (Read, "", false, false,
false);
```

"Operation" class

Operation overview

```
Operation extends Feature;
```

Individual pieces of invocable behavior. In Objecteering/UML, this metaclass defines both the *Operation*, and the method which implements it. Example 5 presents various kind of *Operations*.

An *Operation* belongs to its *Classifier*.

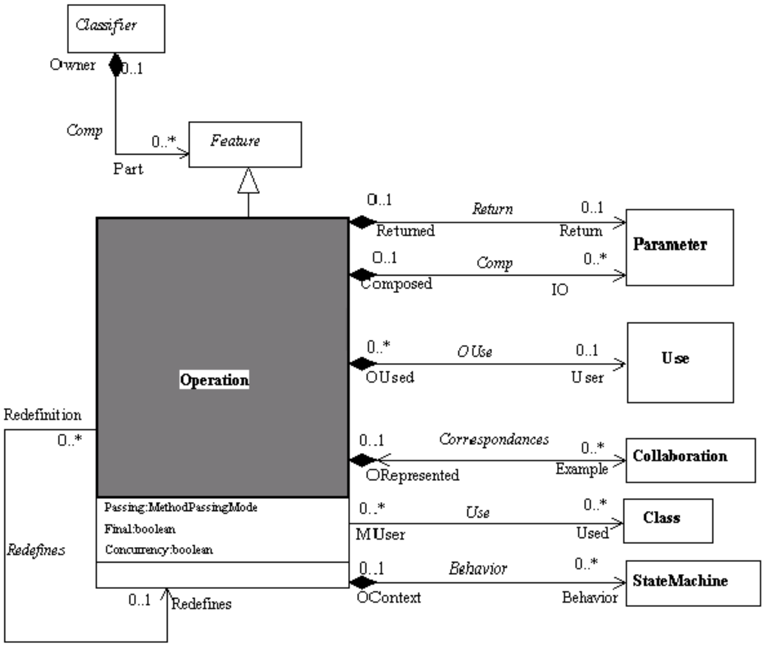


Figure 5-15. Detailed class diagram for *Operation*

See also: *Parameter*, *Classifier*, *Enumerated types*.

Operation properties

The class has the following associations:

- *Redefines:Operation*: Redefinition link between *Operations* of inherited classes. Redefining an *Operation* preserves the same names and parameters, but may specify the pre and post conditions, and the method's internal behavior
- *IO:Parameter*: Defines the parameters making up the method.
- *Return:Parameter*: Link to the eventual return parameter. The return parameter is only distinguished by this association, from the *IOParameter*.
- *Behavior:StateMachine*: Defines the *States diagram* associated to the *Operation*.
- *Used:Class*: Determines the classes specifically used in the implementation of the current *Operation*.
- *Example:Collaboration*: *Collaborations* that illustrate the dynamic of this *Operation*.
- *OUsed:Use*: *Signals* that may be raised as exceptions by the *Operation*.

The class owns the following attributes:

- *Passing*: Operation mode (in or inout). By default, this is "*inout*". This mode determines whether the message receiver object is updated (inout) or not (in) when the method is invoked.
- *Final*: Final *Operations* cannot be redefined. Some OO languages, such as *Java*, optimize final operations.
- *Concurrency*: Distinguishes different invocation mode of an operation. It typically specifies concurrent modes.

Operation consistency rules

- The name of an *Operation* must be different from the names of the *AssociationEnds* and *Attributes* of its owner. The same is true for the parent *Classifiers* of the owner. (Warning).
- The signature of an *Operation* must be different from the signatures of the operations of its owner. The same is true for the parents of the owner (Warning). In the latter case, the check only concerns the *Features* of public and protected visibility.
- There is no repetition check in parent *owners* for the "*create*" and "*delete*" *Operations*.
- The signature is made up of the name, mode, directives and parameters of the operation, as well as its return type. *TaggedValues*, for whom the *BelongToPrototype TagType* is true, are taken into account in the calculation of the signature. For example, in C++ the "*" or "&" tagged values exist on the parameters which intervene in the overload for the C++ language.
- A parameter name must be unique for an *Operation*.
- A redefined *Operation* must belong to a parent of the owner of the *Operation*.
- The *Operation* must have the same signature as the redefined *Operation*, if the case arises.
- A private *Operation* cannot be redefined.
- The visibility of an *Operation* cannot be wider (public > protected > private) than that of the redefined *Operation*.
- A class *Operation* cannot be redefined.
- If the *Operation* is abstract, then the owner must be abstract (Warning).
- A non-abstract *Operation* cannot be redefined in an abstract *Operation*.(Warning)
- A constructor has no return parameters.
- A destroyer has neither *Parameters* nor return parameters.

Operation constructor

```
Operation ModelElement:createOperation (in String pName)
```

This operation creates an *Operation* whose name is given only.

Example:

```
Operation MyOperation =createOperation ("MyOperation");
```

```
Operation createCompleteOperation (in String pName, in
Parameter[] pParamSet, in Parameter pParamReturn, in Note
pBody, in Note pReturnExpr, in VisibilityMode pVisibility,
in boolean pIsClass, in boolean pIsAbstract,
inMethodPassingMode pPassing)
```

This operation creates a complete *Operation* with its name, parameters, return parameter, body and several others attributes.

Example:

```
Operation MyOperation =createCompleteOperation
("OperationName", Parameters, ReturnParameter, BodyExpr,
ReturnExpr, Public, true, false, MethodOut);
```

```
Operation Classifier : createAndAddOperation (in String
pName)
```

This operation creates and adds to the current *Classifier* an *Operation* whose name is given.

Example:

```
Operation MyOperation =Aclass.<createAndAddOperation
("print");
```

Operation methods

`Parameter[] getReturnParameter()`
Returns the "*return Parameter*".

`StateMachine[] getStateMachines()`
Returns the *StateMachines* specifying the behavior of the *Operation*.

`Class[] getUsedClasses()`
Returns the *Classes* specifically used in the implementation of the *Operation*.

`Collaboration[] getCollaborations()`
Returns the *Collaborations* illustrating the dynamic of the *Operation*.

`Operation[] getRedefinesOperation()`
Returns the parent *Operation*.

`Parameter[] getIOParameters()`
Returns the *Parameters* of the *Operation*.

`Signal[] getSignals()`
Returns the *Signals* that may be raised as exceptions by the *Operation*.

`boolean isAbstract()`
Determines if the *Operation* is an abstract *Operation*.

`boolean isClass()`
Determines if the *Operation* is a class *Operation*.

`String pathName()`
This operation returns the complete "*path*" of an *Operation*.

Example:

```
String MyPath =MyOperation.<pathName();
```

```
boolean Operation:sameSignature (in Operation method)
```

This operation determines if both *Operations* have the same interface. It does not compare the names of the methods.

Example:

```
boolean Same =MyOperation.<sameSignature (AnotherOperation);
```

```
boolean Operation:sameSignatureWithName (in Operation
method)
```

This operation determines if both *Operations* have the same interface. The names of the methods are compared. If two operations have identical names, this operations returns false.

Example:

```
boolean Same =MyOperation.<sameSignatureWithName
(AnotherOperation);
```

```
Operation:addParameter (in Parameter pParameter)
```

This operation adds a *Parameter* to the current *Operation*.

Example:

```
MyOperation.<addParameter (MyParameter);
```

```
Operation:addReturnParameter (in Parameter pParameter)
```

This operation adds a return *Parameter* to the current *Operation*.

Example:

```
MyOperation.<addReturnParameter (MyParameter);
```

```
Operation:addUsedClass (in Class pClass)
```

This operation makes the current *Operation* use a *Class*.

Example:

```
MyOperation.<addUsedClass (AUsedClass);
```

```
Operation:addRedefine (in Operation pOperation)
```

This operation adds a Redefine link to the current operation.

Example:

```
RedefinedOperation.<addRedefine (AnOperation);
```

```
Operation Operation:clone()
```

This operation makes a complete copy of the current *Operation*.

Example:

```
Operation MyCopy =MyOperation.<clone();
```

```
Operation:setOperationParams (in MethodPassingMode pPassing,
in boolean pFinal)
```

This operation sets *Parameters* of an *Operation*.

Example:

```
MyOperation.<setOperationParams (MethodOut, false);
```

```
Operation:clearParameter()
```

This operation deletes all the *Parameters* of the current *Operation*.

Example:

```
MyOperation.<clearParameter();
```

"Parameter" class

Parameter overview

```
Parameter extends ModelElement;
```

Information received as input or returned as output by an *Operation*.

Their main characteristics are their name, passing mode and type.

The *return value* of an *Operation* is a specific case managed by a specific association between *Operation* and *Parameter*. *Parameters* can be seen in Example 5.

In Objecteering/UML, a *Parameter* belongs to its *Operation*.

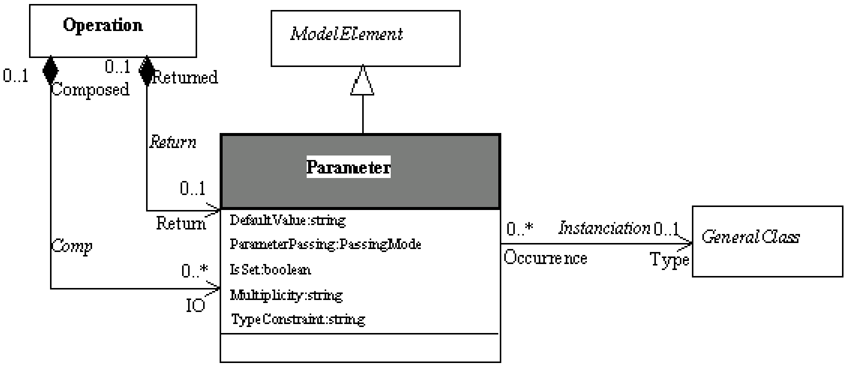


Figure 5-16. Detailed class diagram for *Parameter*

<u>See also</u>: *Enumerated types, Operation*, *GeneralClass*.

Parameter properties

The class has the following association:

- *Type:GeneralClass*: Defines the class to which the *Parameter* belongs.

The class owns the following attributes:

- *DefaultValue*: Default value of the *Parameter*. When the caller does not specify any value, then the default value is applied.
- *ParameterPassing*: Defines the passing mode (in, out or inout) of the *Parameter*.
- *IsSet*: Determines whether it is a set of elements (set of ...).
- *Multiplicity*: Defines the size of the Set (* if unlimited, constant, or whole number), if *IsSet* is true.
- *TypeConstraint*: Construction *Parameter* of the *Parameter*'s class (for example, the size of a characters String).

Parameter consistency rules

- A *Parameter* belongs to an *Operation*.
- A *Parameter* has as a type a *Class*, a *DataType* or an *Enumeration*.
- A *Parameter Out* cannot have a default value,
- The "*Returned*" and "*Composed*" relationships are exclusive. One of them must be defined.
- The *Parameter* type must be accessible from the *Classifier* of its *Operation*.

Parameter constructor

```
Parameter ModelElement:createReturnPredefinedParameter (in
String pType, in boolean pIsSet, in String pMultiplicity, in
String pTypeConstraint)
```

This operation creates a return *Parameter* whose type is *int*, *real*, *char*, *string, boolean* or *undefined*.

Example:

```
Parameter MyParameter =createReturnPredefinedParameter
("float", false, "l", "");
```

```
Parameter ModelElement:createReturnParameter (in
GeneralClass pType, in boolean pIsSet, in String
pMultiplicity, in String pTypeConstraint)
```

This operation creates a return *Parameter* whose type is a *GeneralClass*.

Example:

```
Parameter MyParameter =createReturnParameter (Var__Graphic,
false, "l", "");
```

```
Parameter ModelElement:createParameter (in String pName, in
GeneralClass pType, in PassingMode ParameterPassing, in
boolean pIsSet, in String pMultiplicity, in String
pTypeConstraint, in String pDefaultValue)
```

This operation creates a *Parameter* whose type is a *GeneralClass*.

```
Parameter ModelElement:createPredefinedParameter (in String
pName, in String pType, in PassingMode pParameterPassing, in
booleaan pIsSet, in String pMultiplicity, in String
pTypeConstraint, in String pDefaultValue)
```

This operation creates a *Parameter* whose type is *integer*, *real*, *undefined*, *char*, *boolean* or *String*.

Example:

```
Parameter MyParameter =createPredefinedParameter ("width",
"real", In, false, "l", "", "0");
```

Parameter methods

```
GeneralClass[] getParameterType()
```

Returns the *GeneralClass* which types the *Parameter*.

```
boolean isSet()
```

Determines if the *Parameter* is a set of elements.

```
String pathName()
```

This operation returns the complete "*path"* of a *Parameter*.

Example:

```
String MyPath =MyParameter.<pathName();
```

```
Parameter:setParameterParams (in PassingMode
pParameterPassing, in boolean pIsSet, in String
pMultiplicity, in String pTypeConstraint, in String
pDefaultValue)
```

This operation sets the *Attributes* of a *Parameter.*

Example:

```
MyParameter.<setParameterParams (Out, false, "1", "", "");
```

```
Parameter clone()
```

This operation makes a complete copy of the current *Parameter*.

Example:

```
Parameter MyCopy =MyParameter.<clone();
```

```
Parameter cloneWithoutTag()
```

This operation makes a copy of the current *Parameter* without its *TaggedValues*.

Example:

```
Parameter MyCopy =MyParameter.<cloneWithoutTag();
```

"Association" class

Association overview

```
Association extends ModelElement;
```

Definition of links that may exist between objects. An *Association* describes discrete connections among objects or other instances in a system.

An *Association* is often established between two *Classes* (binary associations), but can be established between several classes (n-ary associations). An *Association* can be related to a *ClassAssociation*, which may, for example, provide attributes and operations.

The connections to the associated *Classes* are specified through the *AssociationEnd* metaclass. The *AssociationEnd* metaclass will provide the properties of an association, such as multiplicities, navigability, etc. Aggregation is a specific case of *Association*.

See Example 6 for the different cases of *Associations*.

An *Association* in Objecteering/UML physically belongs to no other elements. It behaves in a specific way during transfer and copy/paste operations, depending on whether the connected classes are transferred in conjunction or not.

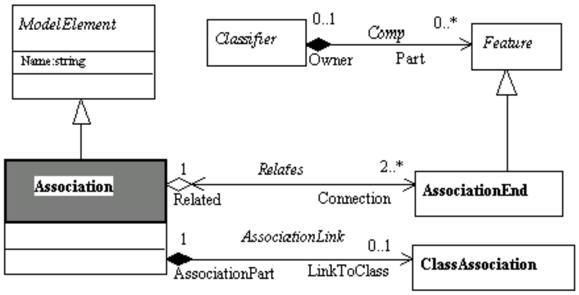


Figure 5-17. Association metamodel

See also: *AssociationEnd*, *ClassAssociation*.

Association properties

The class has the following associations:

- *Connection*:*AssociationEnd*: Defines the links of an *Association* to the *Classes*.
- *LinkToClass*:*ClassAssociation*: Specifies a *ClassAssociation* which may be related to the *Association*.

Association consistency rules

- There can be no composition on an n-ary *Association*.
- There can only be one single aggregation on an *Association*'s *AssociationEnds*.

Association constructor

```
Association ModelElement:createAssociation (in String pName,
in AssociationEnd pOrigin, in AssociationEnd pDestination)
```

This operation creates a binary *Association* whose name and two *AssociationEnds* are given.

Example:

```
Association MyAssociation =createAssociation
("MyAssociation", Origin, Destination);
```

Association methods

```
Class[] getAssociatedClass()
```

Returns the *Classes* related to the *Association*.

```
AssociationEnd[] getAssociationEnds()
```

Returns the links of the *Association* to the *Classes*.

"AssociationEnd" class

AssociationEnd overview

```
AssociationEnd extends Feature;
```

Connection of an *Association* to one of its related *Classes*. An *AssociationEnd* is an endpoint of an *Association*, which connects the *Association* to a *Classifier*. Each *AssociationEnd* is part of one *Association*.

When the *Association* is navigable, this link is considered to be a property of the connected *Class*. It is thus a *Feature* in the same way as *Attributes* or *Operations*. An *Association* is linked to several *Classes* via *AssociationEnds* which determine roles, multiplicities and navigabilities.

The connected *Classifier* is defined by the usual composition link from *Classifier* to *Feature*.

(See Example 6)

For Objecteering, an *AssociationEnd* is a *Component* of a *Class*.

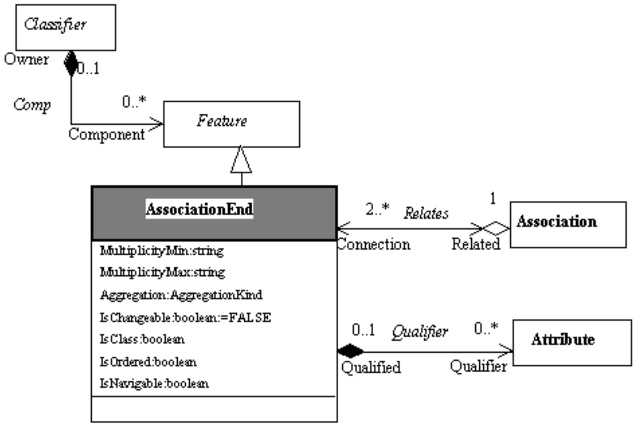


Figure 5-18. Detailed diagram for *AssociationEnd*

<u>See also</u>: *Association, Classifier, Enumerated types*.

AssociationEnd properties

The class has the following associations:

- *Related*:*Association*: Defines the links of an *Association* with the related *Classes*.
- *Qualifier*:*Attribute*: Defines a Qualifier on the *AssociationEnd*. Qualified *Associations* are presented in Example 6.

The class owns the following attributes:

- *MultiplicityMin*: Minimum value of the *Association*'s multiplicity. When placed on a target end, the multiplicity specifies the number of target instances that may be associated with a single source instance across the given Association.
- *MultiplicityMax*: Maximum value of the multiplicity.
- *Aggregation*: This attribute is used to distinguish between the usual associations (*KindIsAssociation*), shared aggregation (*KindIsAggregation*), and strong aggregations (*KindIsComposition*)
- *IsChangeable*: When placed on a target end, specifies whether an instance of the *Association* may be modified from the source end.
- *IsClass*: If this boolean is true, then the link is shared by all the target class instances.
- *IsOrdered*: When placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered.
- *IsNavigable*: Specifies that the association (which must be binary) can be browsed from the opposite class to the class attached to the current *AssociationEnd*.

AssociationEnd consistency rules

- An *AssociationEnd* can belong to a *Class*, an *Actor*, a *DataType*, a *Node* or a *Signal*.
- In the case of a binary *Association*, the *Classifier* accessed by an *AssociationEnd* must be accessible by the *Classifier owner* of the *Association*.
- An *AssociationEnd* can link a *Class* to a *Class*, a *Datatype* or a *Signal*.
- An *AssociationEnd* can link a *DataType* to a *Class* or a *DataType*.
- An *AssociationEnd* can link a *Signal* to a *Class* or a *DataType*.
- An *AssociationEnd* can link a *Actor* to a *Class*, an *Actor* or a *DataType*.
- An *AssociationEnd* can link a *Node* to a *Node*.
- There are no controlled visibility rules on n-ary *Associations*.
- An *AssociationEnd*'s name must be different from that of the *AssociationEnds*, *Operations* and *Attributes* of its owner. This is also the case for the “*owner*'s" parents (Warning).
- An *AssociationEnd* can only involve a *Classifier*.
- Multiplicities (also called cardinalities) must be consistent: *CardinalityMax* > *CardinalityMin*. In particular, *CardinalityMin* must not equal "*".
- The *IsAbstract* and *IsClass* attributes are exclusive.
- If the association is an aggregation, then the maximum multiplicity is 1.
- A public *Association* oriented from a public *Classifier* cannot be linked to a private or protected *Classifier*.
- An *AssociationEnd* can be made up of *Attributes*. These *Attributes* have a unique name in the *AssociationEnd*.

AssociationEnd constructor

```
AssociationEnd Classifier:createAndAddAssociationEnd (in
String pRole, in String pCardMin, in String pCardMax, in
boolean pIsNavigable)
```

This operation creates and adds to the current *Classifier* an *AssociationEnd* whose role name and multiplicities are given.

Example:

```
AssociationEnd MyAssociationEnd
=Aclass.<createAndAddAssociationEnd ("ClientRole", "0", "1",
true);
```

```
AssociationEnd
Classifier:createAndAddMonoOrientedAssociation (in
Classifier pToClassifier, in String pName, in String pRole,
in String pDestMin, in String pDestMax, in String pOrigMin,
in String pOrigMax, in VisibilityMode pVisibility, in
boolean pIsClass, in boolean pIsAbstract)
```

This operation creates and adds to the current *Classifier* an *Association* which is mono-oriented. The *AssociationEnd* returned is the one linked to the current (origin) *Classifier*. This operation does not specify the origin role name, because it is a usual shorthand.

Example:

```
AssociationEnd MyAssociationEnd
=Aclass.<createAndAddMonoOrientedAssociation (ClientClass,
"Cclient", "ClientRole", "0", "1", "1", "*", Private, true,
false);
```

```
AssociationEnd ModelElement:createAssociationEnd (in String
pRole, in String pCardMin, in String pCardMax)
```

This operation creates an *AssociationEnd* whose role name and multiplicities are given.

Example:

```
AssociationEnd MyAssociationEnd = createAssociationEnd
("MyRole", "0", "*");
```

AssociationEnd methods

```
Association[] getAssociation()
```

Returns the *Association* related to the link.

```
Attribute[] getQualifiers()
```

Returns the qualifiers on the link.

```
boolean isClass()
```

Returns "*true*" if the link is a class member.

```
AssociationSetAssociationEndParams (in AggregationKind
pAggregatioon, in boolean pIsChangeable, in boolean
pIsOrdered)
```

This operation sets parameters of an *AssociationEnd*. It has to be done in a J session.

Example:

```
MyAssociationEnd.<setAssociationEndParams
(KindIsAssociation, true, false);
```

"ClassAssociation" class

ClassAssociation overview

```
ClassAssociation extends ModelElement;
```

Class relating other classes. It is both a *Class* and an *Association*. A *ClassAssociation* is represented in UML as a class that plays the role of an *Association*. (See Example 6)

In Objecteering/UML, a *ClassAssociation* is a component of an *Association*.

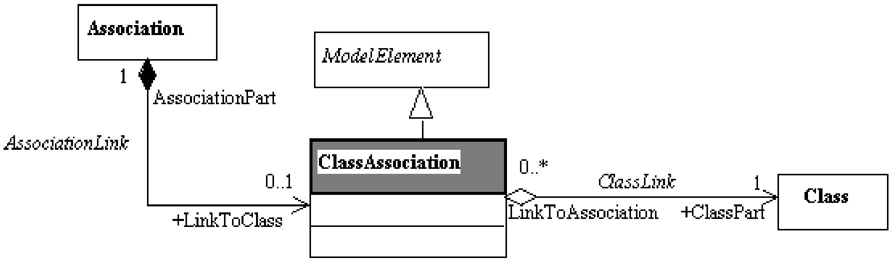


Figure 5-19. Detailed class diagram for *AssociationClass*

See also: *Class*, *Association*.

ClassAssociation properties

The class has the following association:

- *ClassPart*:*Class*: Link to the *Class* that composes the *ClassAssociation*.

"Enumeration" class

Enumeration overview

```
Enumeration extends GeneralClass;
```

Special kind of *DataType* whose range is a list of predefined values, called *EnumerationLiterals*. This notion corresponds to C++ or Java enum, or equivalent types in Pascal, Ada, or any other language.

In Objecteering/UML, an enumeration belongs to its *NameSpace*.

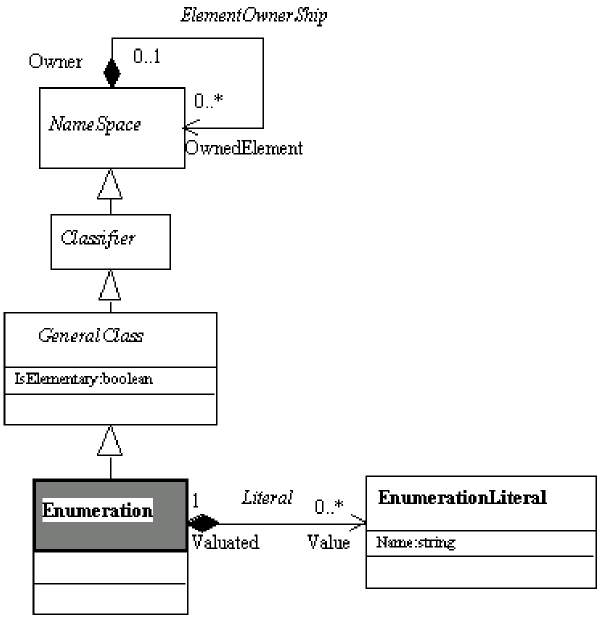


Figure 5-20. Detailed class diagram for *Enumeration*

See also: *EnumerationLiteral*.

Enumeration properties

The class has the following association:

- *Value:EnumerationLiteral*: Link with the *Literal* which represents the possible values of the type representatives.

Enumeration consistency rules

- An *Enumeration* belongs to a *Package*, a *Class* or a *Signal*.
- An *Enumeration* contains no *NameSpaces*.
- An *Enumeration* has no members.
- An *Enumeration* has no use links (USE), generalization links (*Generalization*), realization links or communication links.
- An *Enumeration* contains no *DataFlows*.
- An *Enumeration* does not implement interfaces.
- An *Enumeration* contains no *Instances*.
- An *Enumeration* cannot be instantiated.
- The *IsElementary* value is always true. The *IsAbstract* value is always false.
- *EnumerationLiterals* defined by an *Enumeration* each have a different name.

Enumeration constructor

```
Enumeration NameSpace:createAndAddEnumeration (in String
pName, in Object[] pValues)
```

This operation creates and adds to the current *NameSpace* an *Enumeratioon* whose name and values are given.

Example:

```
Enumeration MyColorType =MyClass.<createAndAddEnumeration
("color", createSetOf2("Blue", "Red"));
```

```
Enumeration ModelElement:createEnumeration (in String pName,
in String[] pValues)
```

This operation creates an *Enumeration* whose name and values are given.

Example:

```
Enumeration MyColors =createEnumeration ("MyColors",
createSetOf2("Blue", "Red"));
```

Enumeration methods

```
EnumerationLiteral[] getEnumerationLiterals()
```

Provides the possible values of the type representatives.

"EnumerationLiteral" class

EnumerationLiteral overview

```
EnumerationLiteral extends Element;
```

Defines an atom (i.e., with no relevant substructure), represents one in the list of values that an enumerated may have. The *Name* of the *EnumerationLiteral* represents its symbolic value.

An *EnumerationLiteral* belongs to its *Enumeration*.

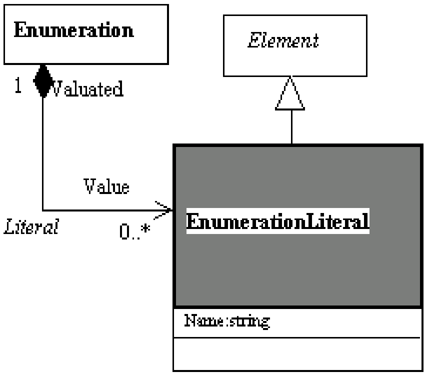


Figure 5-21. Detailed class diagram for *EnumerationLtteral*

EnumerationLiteral properties

The class owns the following attribute:

Name: Name of the *EnumerationLiteral*. It represents the symbolic value of the associated *Enumeration*.

EnumerationLiteral consistency rules

An *EnumerationLiteral* has a unique name in the *Enumeration*.

DataFlows and Signals

Metamodel synthesis

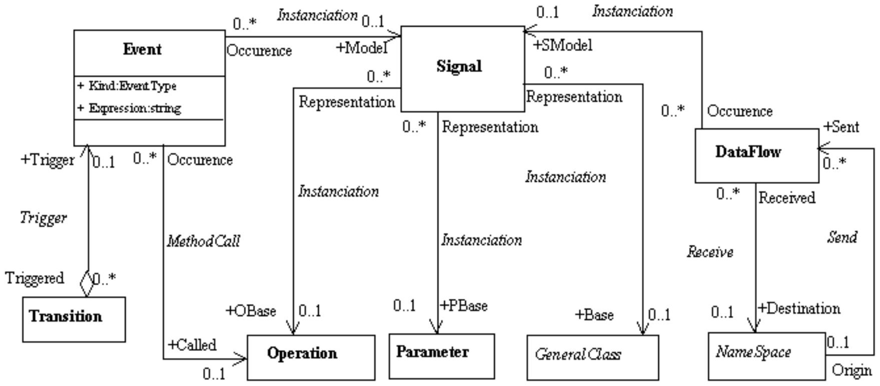


Figure 5-22. Internal package diagram for *SignalDataFlowEvent*

DataFlows are an extension of UML which provide the means of presenting the information which navigates between model elements. *Signals* define what kinds of *DataFlow* may exist. *Signals* can represent the circulation of objects, parameters, or message calls.

Example 7: Signals

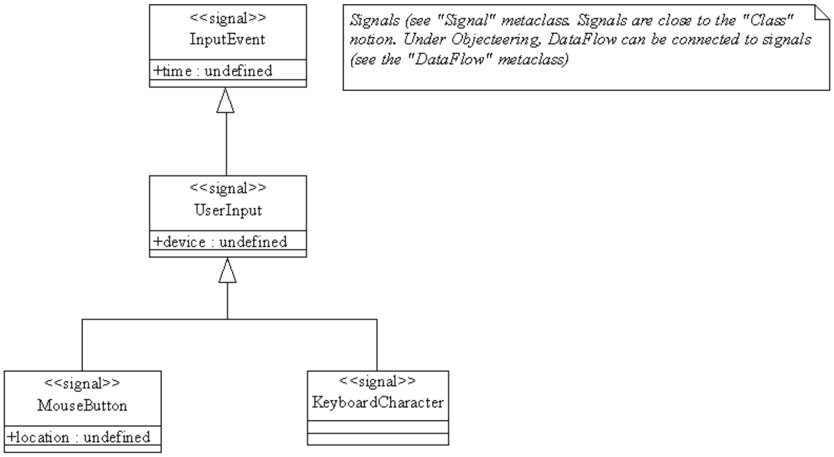

<<signal>>
InputEvent
+time : undefined
<<signal>>
UserInput
+device : undefined
<<signal>>
MouseButton
+location : undefined
<<signal>>
KeyboardCharacter
Signals (see "Signal" metaclass. Signals are close to the "Class" notion. Under Objecteering, DataFlow can be connected to signals (see the "DataFlow" metaclass)

Figure 5-23. Signals

Example 8: DataFlow

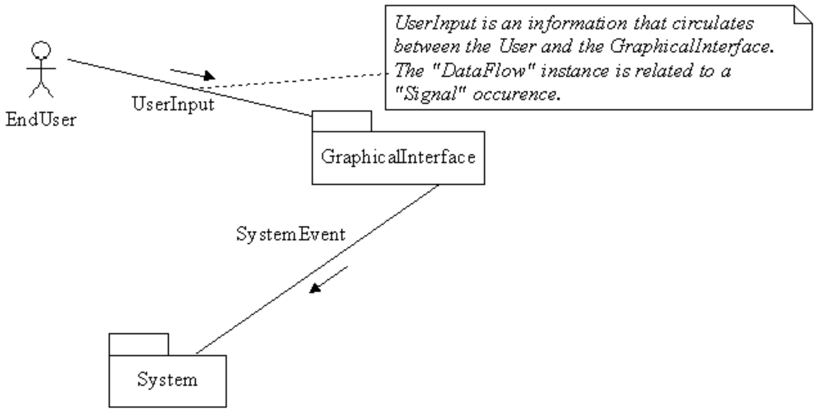

UserInput is an information that circulates between the User and the GraphicalInterface. The "DataFlow" instance is related to a "Signal" occurence.
EndUser
UserInput
GraphicalInterface
SystemEvent
System

Figure 5-24. DataFlow is an extension specific to Objecteering/UML. It is represented by the *DataFlow* metaclass

"Signal" class

Signal overview

```
Signal extends GeneralClass;
```

Specification of an asynchronous stimulus communicated between *Instances*. *Signals* are processed by *StateMachines*, which represent how *SignalEvents* are taken into account. Objecteering/UML provides the *DataFlow* extension to UML. Using this extension, a *Signal* can be declared as representing a *ModelElement* (*GeneralClass*, *Operation*, or *Parameter*). A *DataFlow* associated to the *Signal* will then be able to express that *Data*, represented by the *Signal*, may circulate between different *NameSpaces*.

Examples of *Signals* can be seen in Example 7.

In Objecteering/UML, a *Signal* belongs to a *NameSpace*, notably its *Package*.

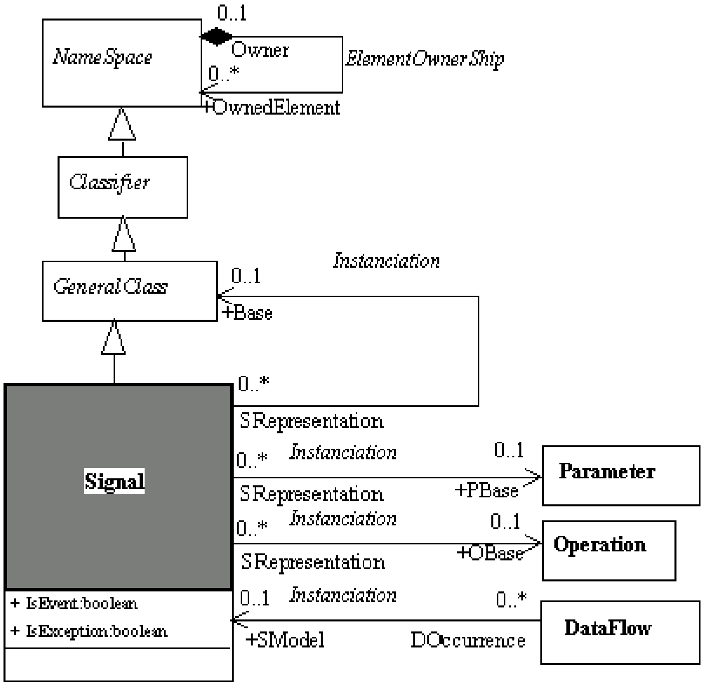


Figure 5-25. Detailed class diagram for *Signal*

See also: *DataFlow*, *Event*.

Signal properties

The class has the following associations:

- *Base:GeneralClass*: *Class* that the *Signal* may represent.
- *OBase:Operation*: *Operation* that the *Signal* may represent.
- *PBase:Parameter*: *Parameter* that the *Signal* may represent.

The class owns the following attributes:

- *IsEvent*: Establishes if it is an event in the sense of event-based systems : (CORBA, Java, XWindow's, RDBMS).
- *IsException*: Defines if it is an exception, as they exist in Java, C++, etc.

Signal consistency rules

Dependencies arising from a *Signal* respect accessibility rules between *NameSpaces*.

A *Signal* can only belong to a *Package* or to a *Class*.

There are no *DataFlows* on a *Signal*.

There are no *Communication* links on a *Signal*.

A maximum of one *dependency* can exist between a *Signal* and another element (*Generalization* or *Use*).

"DataFlow" class

DataFlow overview

```
DataFlow extends ModelElement;
```

Circulation of information between model elements. Representation of all types of information that can be transmitted between elements. For example, they can be objects or requests.

A *DataFlow* between elements expresses that the kind of information that it represents (defined through its *ModelSignal*) can circulate between the connected elements.

This is an extension to the UML model, and can provide high level (system level) information exchange diagrams.

Example 8 presents different *DataFlows*.

In Objecteering/UML, a *DataFlow* belongs to a *Package*.

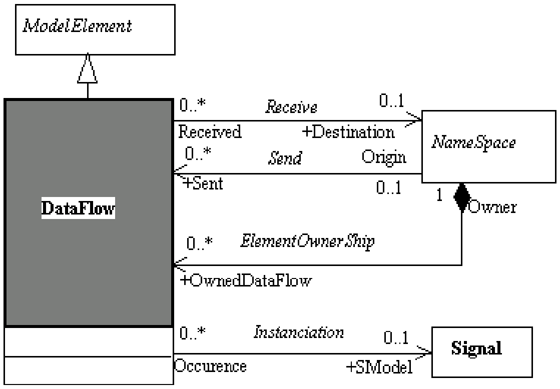


Figure 5-26. Detailed class diagram for *DataFlow*

<u>See also</u>: *Signal*, *NameSpace*.

DataFlow properties

The class has the following associations:

- *Destination:NameSpace*: Designates the *NameSpaces* (*Packages*, *Classes*, etc.) that are targeted by the *DataFlow*.
- *SModel:Signal*: Defines the *DataFlow* as being an instance of the associated *Signal*.

DataFlow consistency rules

At least one of the *Receive* or *Send* relationships must be defined.

The element designated by the *Instantiation* relationship must be accessible to the Signal's *NameSpace* owner.

"Generalization" class

Generalization overview

```
Generalization extends ModelElement;
```

Taxonomic relationship between a more general element and a more specific element.

Generalization corresponds to the usual *Inheritance* concept. In Objecteering/UML, it covers fewer elements than UML. Every element that may have this kind of link is a *GeneralizableElement* in UML, whilst this is restricted to the *NameSpace* concept in Objecteering/UML. (See Examples 3 and 4)

In Objecteering/UML, a *Generalization* belongs to its *SubTypeNameSpace*.

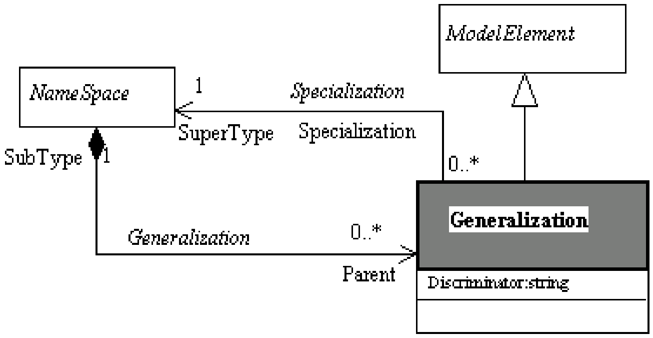


Figure 5-27. Detailed class diagram for *Generalization*

See also: *NameSpace*.

Generalization properties

The class has the following association:

- *SuperType:NameSpace* : Defines the parent element.

The class owns the following attribute:

- *Discriminator*: Designates a family of sub-classes with the same parent class. The name appears graphically, by linking the *Generalizations* belonging to this family.

Generalization consistency rules

A *Generalization* link between *NameSpaces* must conform to the accessibility rules of these *NameSpaces*.

There is no *Generalization* on *Enumeration* and *Component*.

A *Generalization* must concern two *NameSpaces* of the same *MetaClass*, with the exception of *Signal* which can inherit from a *Signal* or from a *Class*.

Generalization constructor

```
Generalization NameSpace:createAndAddGeneralization (in
NameSpace pParent)
```

This operation creates and adds to the current *NameSpace* a *Generalization* of the parameter.

Example:

```
Generalization MyGeneralization
=MyClass.<createAndAddGeneralization (AParentClass);
```

```
NameSpace:addGeneralization (inout NameSpace pParent)
```

This operation adds a *Generalization* link to the current *NameSpace*.

Example:

```
MyDerivedPackage.<addGeneralization (AParentPackage);
```

Generalization methods

```
String pathName()
```

This operation returns the complete "path" of a *Generalization*.

Example:

```
String MyPath =MyGeneralization.<pathName()
```

"Use" class

Use overview

```
Use extends ModelElement;
```

Usage dependency between elements. The *Use* metaclass represents the specific kind of <<access>> or <<import>> dependency between *Packages* or *Classes*. *Use* dependencies are presented in Examples 6 and 3.

Use dependencies belong to their user *NameSpace*.

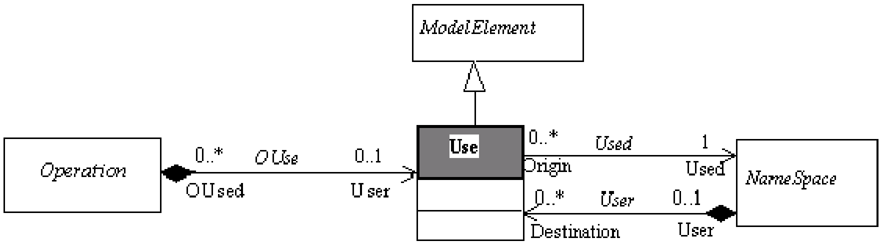


Figure 5-28. Detailed class diagram for *Use*

See also: *NameSpace*.

Use properties

The class has the following associations:

- *Used:NameSpace*: Used *NameSpace*. Its properties are accessible from the user *NameSpace*.

Use consistency rules

The *NameSpaces* which link the *Use* class must have the same terminal Metaclass, with the exception of the following :

- an *Actor* can use a *Package*, a *Class*, a *Component* or a *Signal*.
- a *Component* can use a class *Interface* or a *Component*.
- a *Signal* can use a *Signal* or a *Class*.
- an *Operation* can use a *Signal* or a *Class.*

Use constructor

```
NameSpace:addUse (in NameSpace pUsed)
```

This operation adds a *Use* to the current *NameSpace*.

Example:

```
MyUserPackage.<addUse (AUsedPackage);
```

"Realization" class

Realization overview

```
Realization extends ModelElement;
```

Implementation link between a *Class* and its *Interface*, or between a *Component* and its *Interface*. *Realization* links can be seen in Example 6.

In Objecteering/UML, a *Realization* belongs to its *NameSpace*..

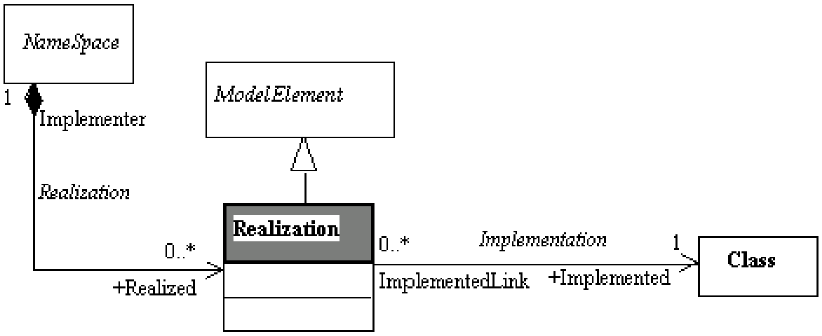


Figure 5-29. Detailed class diagram for *Realization*

<u>See also</u>: *NameSpace*, *Class*, *Component*.

Realization properties

The class has the following association:

- *Implemented:Class* : End of the implementation link towards an *Interface*.

Realization consistency rules

A *Realization* can only exist between a *Class* and a *Class Interface* or between a *Component* and a *Class Interface*.

The "*implemented*" *Interface* class must be accessible from the *NameSpace*.

"TemplateParameter" class

TemplateParameter overview

```
TemplateParameter extends Element;
```

Parameter for *Templated* elements. *Templated* Classes can be seen in Example 4.

In Objecteering/UML, *TemplateParameters* belong to their *GeneralClass*. Binding an implementation *Class* is provided in Objecteering/UML through dedicated *TaggedValues*.

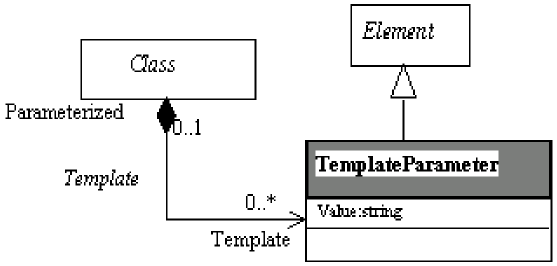


Figure 5-30. Detailed class diagram for *TemplateParameter*

See also: *GeneralClass*.

TemplateParameter properties

The class owns the following attribute:

- *Value*: Parameter value.

Chapter 6: Use Case Model

Overview

Presentation

The *UseCase Model* is based on two central notions, *Actor* and *UseCase*. It includes the various dependencies that can exist between these elements. *UseCases* are then broken down into scenarios represented by *Collaborations*, or *StateMachine* diagrams, which are defined in separate diagrams.

Metamodel synthesis

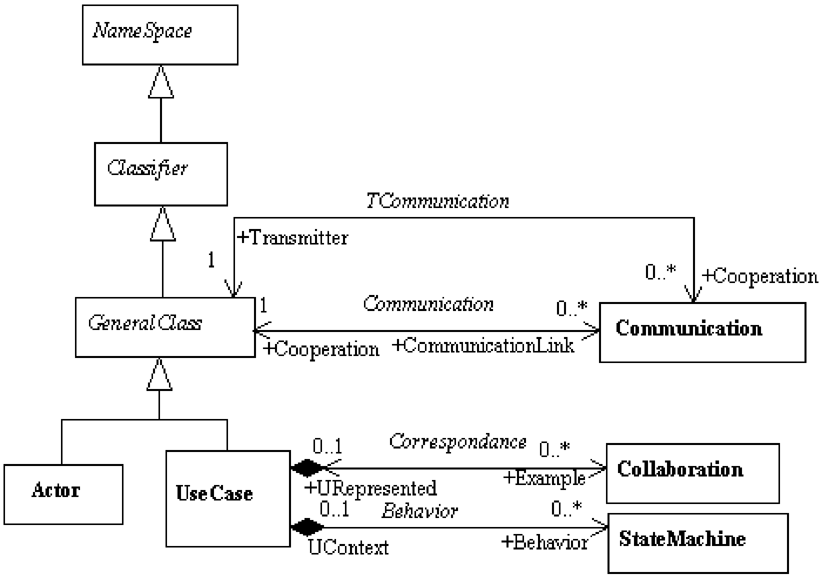


Figure 6-1. Internal package diagram for *UseCase*

UseCase and *Actor* are two main concepts, with different links that may exist (*Collaboration*, *Use* and *Inheritance*)

UseCase model metaclasses

- *UseCase*: Unit of externally visible functionality provided by part of a system.
- *Actor*: Active element external to the system, and which cooperates with it.
- *Communication*: Communication link between *Actors* and *UseCases*
- *UseCaseDependency*: Inheritance or dependency link (Uses) between *UseCases.*

Example 9: Actors and UseCases

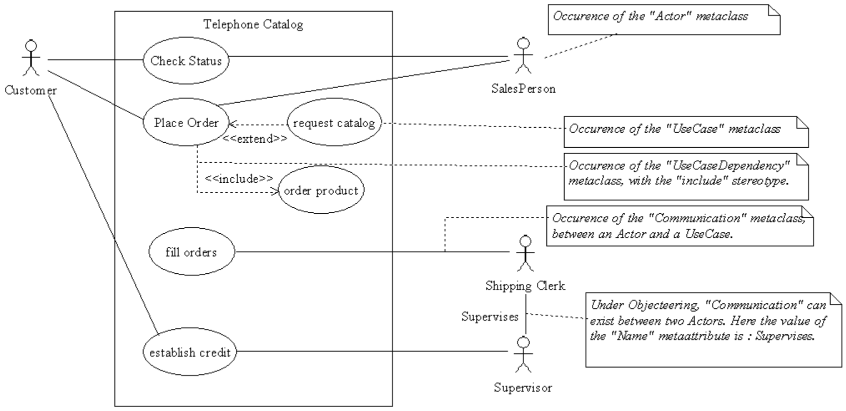

Telephone Catalog
Customer
Check Status
Place Order
request catalog
<<extend>>
<<include>>
order product
fill orders
establish credit
SalesPerson
Shipping Clerk
Supervises
Supervisor
Occurence of the "Actor" metaclass
Occurence of the "UseCase" metaclass
Occurence of the "UseCaseDependency" metaclass, with the "include" stereotype.
Occurence of the "Communication" metaclass, between an Actor and a UseCase.
Under Objecteering, "Communication" can exist between two Actors. Here the value of the "Name" metaattribute is : Supervises.

Figure 6-2. UseCase diagrams. See the *UseCase* and *Actor* metaclasses

Example 10: Actors cooperating with objects in a sequence diagram

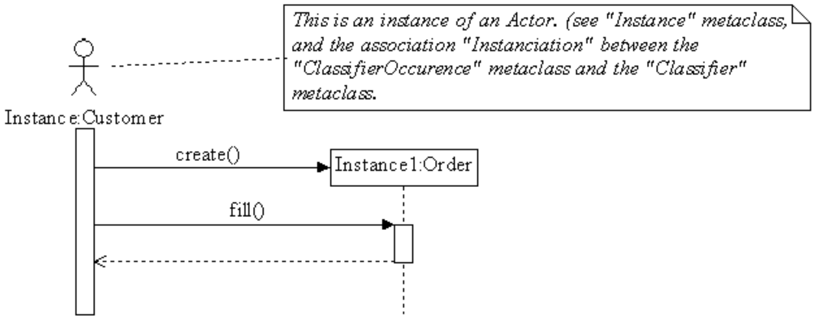

This is an instance of an Actor. (see "Instance" metaclass, and the association "Instanciation" between the "ClassifierOccurence" metaclass and the "Classifier" metaclass.
Instance:Customer
create()
Instance1:Order
fill()

Figure 6-3. "*Customer*" represents an *Actor*

"UseCase" class

UseCase overview

```
UseCase extends GeneralClass;
```

Unit of externally visible functionality provided by part of a system. A *UseCase* is expressed by sequences of messages exchanged by system units and one or more *Actors* of the system.

The definition of a *UseCase* includes all of the behavior that it entails. This behavior can be expressed by *Sequence diagrams*, *Activity diagrams*, *Object diagrams*, etc.

UseCases are structured by *Packages*, and have cooperation links with *Actors*.

Example 9 presents a *UseCase* diagram.

UseCases belong to a *NameSpace*.

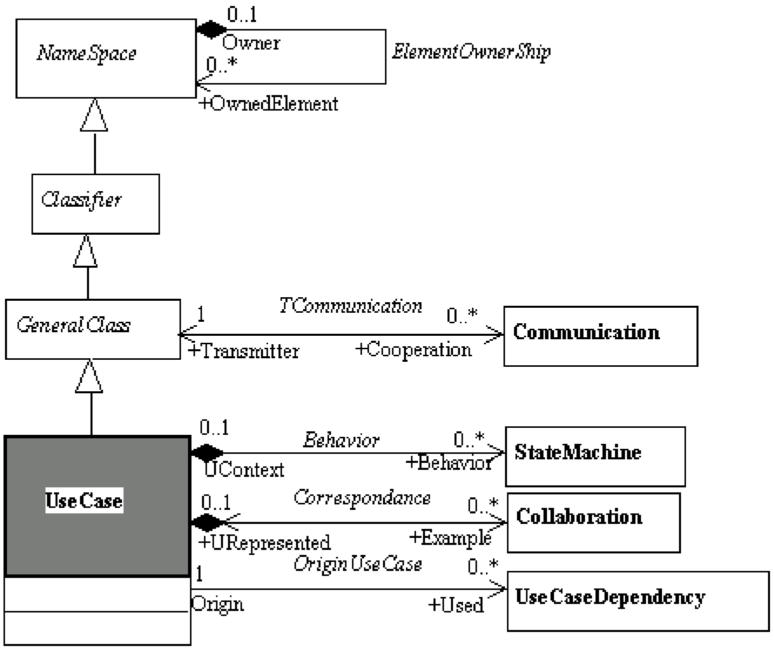


Figure 6-4. Detailed class diagram for *UseCase*

See also: *Actor*, *Collaboration*, *UseCaseDependency*.

UseCase properties

The class has the following associations:

- *Behavior:StateMachine*: *StateMachines* which describe the behavior of the *UseCase*. They can define, for example, activity diagrams, that express another view than Sequence diagrams.
- *Example:Collaboration*: *Collaborations* expressing the dynamics of the current *NameSpace*.
- *Used:UseCaseDependency*: In the dependencies between *UseCases*, this defines the link to the *UseCaseDependency* association.

UseCase consistency rules

- A *Generalization* link can exist between any *UseCases* of the current *Project* (*Model*). Although it is not obligatory, we recommend that you follow the accessibility rules of the other *Classifiers.*
- A *UseCase* follows the visibility rules of a *NameSpace*.
- A *UseCase* can only have *Generalization* links with other *UseCases*.
- There can be no communication links between *UseCases.*
- A communication on a base *UseCase* cannot be redefined on a derived *UseCase*.
- A use on a base *UseCase* can be redefined on a derived *UseCase*, if the use is stereotyped by <<*extend*>>.
- There can be no loops in a *Generalization* graph between *UseCases*.
- There can be no loops in a use graph (*UseCaseDependency*) stereotyped by <<*include*>> or <<*extend*>> only.
- A *UseCase* can only use another *UseCase* once, except in the case of the <<*extend*>> stereotype.

UseCase constructors

```
UseCase ModelElement:createUseCase (in String pname)
```

This operation creates a *UseCase* whose name is given.

Example:

```
UseCase MyUseCase =createUseCase ("MyUseCase");
```

```
UseCase NameSpace:createAndAddUseCase (inString pName)
```

This operation creates and adds to the current *NameSpace* a *UseCase* whose name is given.

Example:

```
UseCase MyUseCase = MyPackage.<createAndAddUseCase
("MyUseCase");
```

UseCase methods

```
Actor[] getTransmitterActors()
```

Returns the *Actors* which communicate with the *UseCase*.

```
StateMachine[] getStateMachines()
```

Returns the *StateMachines* of the *UseCase*.

```
UseCase[] getParentUseCases()
```

Returns the parent *UseCases*.

```
Collaboration[] getCollaborations()
```

Returns the *Collaborations* of the *UseCases.*

```
Diagram[] getDiagrams()
```

Returns the *Diagrams* of the *UseCases*.

```
UseCase[] getExtendedUseCases()
```

Returns the *UseCases* that are extended.

```
UseCase[] getIncludedUseCases()
```

Returns the *UseCases* that are included.

```
addUseCaseDependency (in UseCaseDependency
pUseCaseDeppendency)
```

This operation adds an *UseCaseDependency* to the current *UseCase*.

Example:

```
MyUseCase.<addUseCaseDependency (AUseCaseDependency);
```

"Actor" class

Actor overview

```
Actor extends GeneralClass;
```

Active element external to the system, and cooperating with it. An *Actor* is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An *Actor* characterizes the interactions that outside users may have with the system. An *Actor* may be a human, an external software component, a device, that cooperates with the system.

Actors can have communication links with *UseCases* (see Example 9). They can also communicate between *Actors* (extension of Objecteering/UML to UML).

Actors can appear in *Sequence diagrams* (see Example 10), or *Collaboration diagrams*, where the accurate communication with system's objects is shown.

Actor is a specific kind of *Classifiers*. An *Actor* can have generalization link*s* with other *Actors*, can have *Attributes* and *Operations*.

An *Actor* in Objecteering/UML physically belongs to a *NameSpace*, that must be a *Package*.

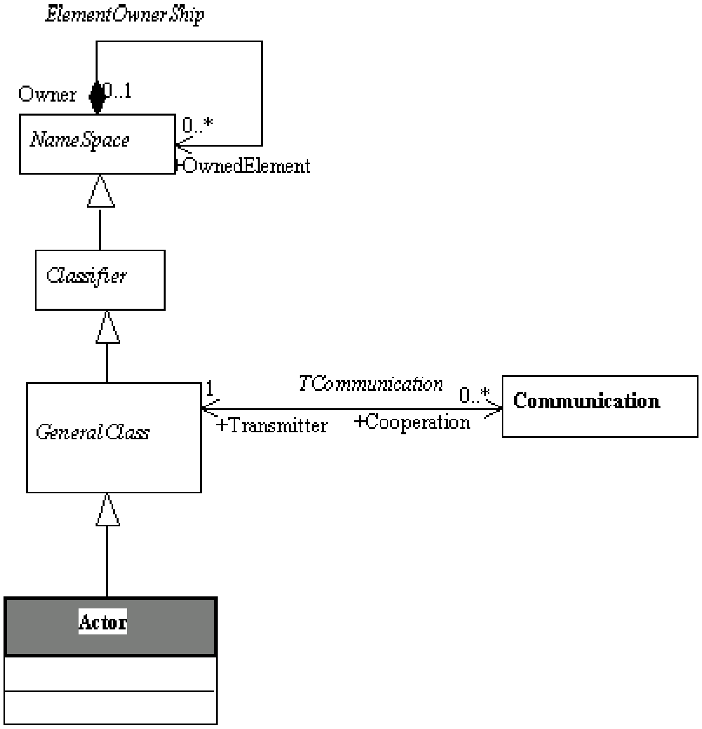


Figure 6-5. Actor *Metamodel*

See also: *Communication*, *UseCase*.

Actor consistency rules

- A *Generalization* link between *Actors* can be created between any of the *Actors* in a *Project*. Although it is not obligatory, we recommend that you follow the accessibility rules of *Classifiers*.
- An *Actor* can only specialize (*Generalization*) another *Actor*. A maximum of one generalization link may exist between two *Actors*.
- A communication link cannot be established from an *Actor* to itself.
- A communication link towards an *Actor* cannot be repeated towards the *Actors* which specialize it.
- There can be no loops in a *Generalization* graph between *Actors*.

Actor constructor

```
Actor ModelElement:createActor (in String pName)
```

This operation creates an *Actor* whose name is given.

Example:

```
Actor MyActor =createActor ("MyActor");
```

```
Actor NameSpace:createAndAddActor (in String pName)
```

This operation creates and adds to the current *NameSpace* an Actor whose name is given.

Example:

```
Actor MyActor =MyPackage.<createAndAddActor ("MyActor");
```

Actor methods

`Actor[] getCooperatingActors()`
Returns *Actors* which communicate with the current *Actor*.

`Actor[] getParentActors()`
Returns parent *Actors*.

`UseCase[] getCooperatingUseCases()`
Returns *UseCases* which communicate with the *Actor*.

"Communication" class

Communication overview

```
Communication extends ModelElement;
```

Communication link between *Actors* and *UseCases*. This kind of link might exist between an *Actor* and a *UseCase*, but also between *Actors*.

In Objecteering/UML, a *Communication* belongs to its origin *Classifier* (Transmitter).

Example 9 shows communication link examples.

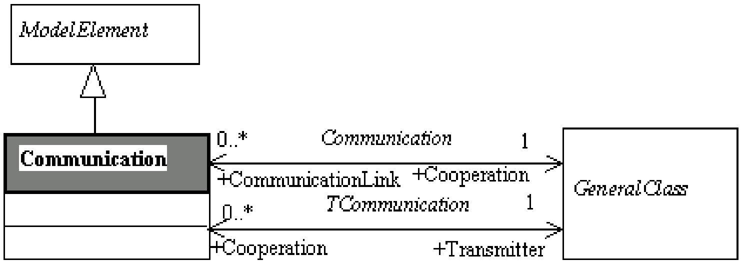


Figure 6-6. Detailed class diagram for *Communication*

Communication properties

The class has the following associations:

- *Cooperation:GeneralClass*: Destination *GeneralClass* (*UseCase* or *Actor*) for the cooperation.
- *Transmitter:GeneralClass*: Destination *GeneralClass* (*UseCase* or *Actor*) for the cooperation.

Communication consistency rules

A *Communication* must link two *Actors* or an *Actor* and a *UseCase*.

Communication constructors

```
Communication ModelElement:createCommunication (in String
pName, in GeneralClass pTarget)
```

This operation creates a *Communication* whose name and target (*UseCase* or *Actor*) are given.

Example:

```
communication Mycommunication =createCommunication
("Mycommunication", aClass);
```

```
GeneralClass:addCommunication (in Communication
pCommunication)
```

This operation adds a *Communication* to the current *GeneralClass* (*UseCase* or *Actor*).

Example:

```
MyActor.<addCommunication (Mycommunication);
```

```
Communication GeneralClass:createAndAddCommunication (in
String pName, in GeneralClass pTarget)
```

This operation creates and adds to the current *GeneralClass* a *Communication* whose name and target are given.

Example:

```
Communication MyCommunication
=Actor.<createAndAddCommunication ("", AnotherClass);
```

"UseCaseDependency" class

UseCaseDependency overview

```
UseCaseDependency extends ModelElement;
```

Inheritance or dependency link (*Uses*) between *UseCases* in Objecteering/UML. This specific metaclass has been created for defining these links.

Two predefined *Stereotypes* are defined for this link: <<*extends*>>, and <<*includes*>>.

UseCaseDependency belongs to its origin *UseCase*.

Examples of *UseCaseDependency* are shown in Example 9.

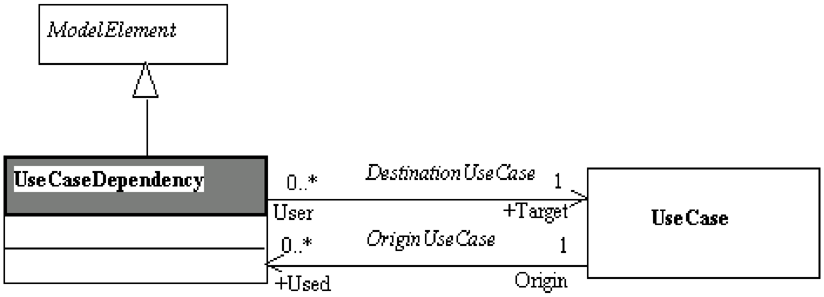


Figure 6-7. Detailed class diagram for *UseCaseDependency*

<u>See also</u>: *UseCase*.

UseCaseDependency properties

The class has the following association:

- *Target:UseCase*: In a dependency between *UseCases*, defines the link to the target *UseCase*

UseCaseDependency consistency rules

A *UseCaseDependency* must link two different *UseCases* and support one of the predefined *Stereotypes*: <<*extend*>> or <<*include*>>.

UseCaseDependency constructors

```
UseCaseDependency ModelElement:createUseCaseDependency (in
String pName, in UseCase pTarget)
```

This operation creates a *UseCaseDependency* whose name and target are given.

Example:

```
UseCaseDependency MyUseCaseDependency
=createUseCaseDependency ("MyUseCaseDependency", aUseCase);
```

```
UseCaseDependency UseCase:createAndAddUseCaseDependency (in
String pName, in UseCase pUseCase)
```

This operation creates and adds to the current *UseCase* a *UseCaseDependency* whose name is given.

Example:

```
UseCaseDependency MyUseCaseDependency
=AUseCase.<createAndAddUseCaseDependency
("MyUseCaseDependency", AnotherUseCase);
```

Chapter 7: State Machine Model

Overview

Presentation

StateMachines describe the dynamic behavior of a system by describing how *Transitions* are triggered between different *States*.

Metamodel synthesis

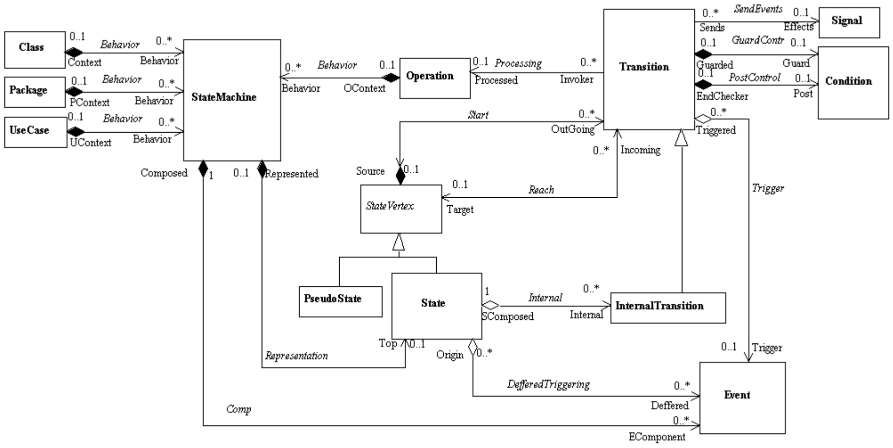


Figure 7-1. *StateMachine model*: overview of the metamodel

The State model is organized around *StateMachines* and *States*. *Transitions* and *Events* are the two other important concepts. *StateMachines* are mainly defined at *Class* level, but can also exist at *Package*, *UseCase*, and *Operation* level, where they mainly correspond to activity diagrams.

StateMachine model metaclasses

- *StateMachine*: Graph of *States* and *Transitions* which describes the dynamic behavior of objects.
- *StateVertex*: Abstraction of a *Node* in a *statechart* graph.
- *State*: Notable situation or condition during the life of an object.
- *PseudoState*: Abstraction of different types of nodes in the *StateMachine* graph.
- *Transition*: Path from one *State* to another.
- *InternalTransition*: Transition which is internal to a *State*.
- *Condition*: Boolean expression for making a choice.
- *Event*: Specification of a significant occurrence which has a location in time and space.

Example 11: State diagram - Condition and Event

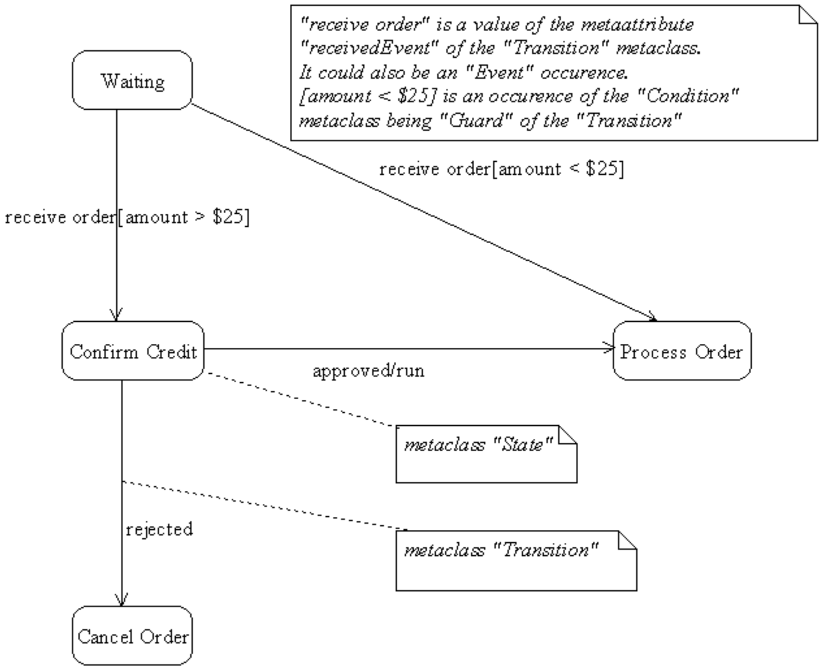

"receive order" is a value of the metaattribute
"receivedEvent" of the "Transition" metaclass.
It could also be an "Event" occurence.
[amount < $25] is an occurence of the "Condition"
metaclass being "Guard" of the "Transition"
Waiting
receive order[amount < $25]
receive order[amount > $25]
Confirm Credit
Process Order
approved/run
metaclass "State"
rejected
metaclass "Transition"
Cancel Order

Figure 7-2. Events and Conditions

Example 12: State diagram - InternalTransition Pseudo states, States and Transitions

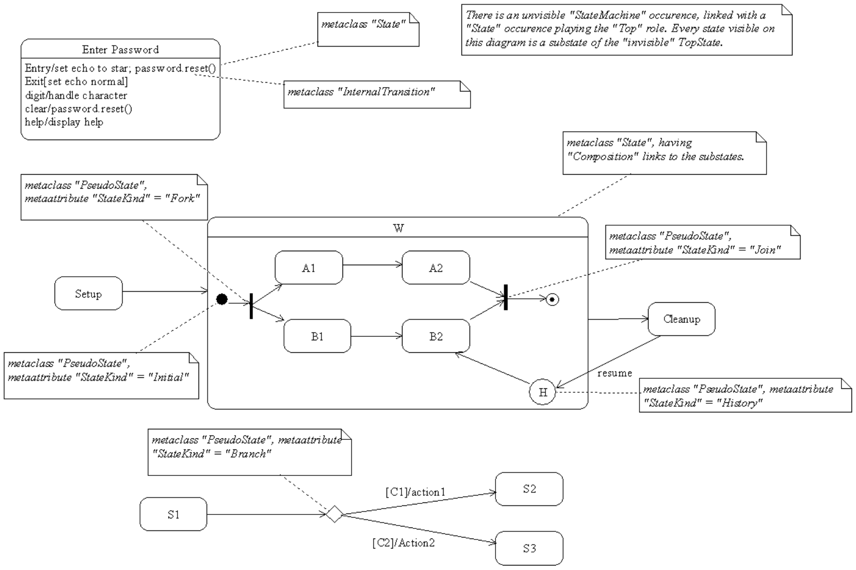

metaclass "State"
There is an unvisible "StateMachine" occurence, linked with a "State" occurence playing the "Top" role. Every state visible on this diagram is a substate of the "invisible" TopState.
Enter Password
Entry/set echo to star; password.reset()
Exit[set echo normal]
digit/handle character
clear/password.reset()
help/display help
metaclass "InternalTransition"
metaclass "State", having "Composition" links to the substates.
metaclass "PseudoState", metaattribute "StateKind" = "Fork"
W
metaclass "PseudoState", metaattribute "StateKind" = "Join"
A1
A2
Setup
B1
B2
Cleanup
resume
H
metaclass "PseudoState", metaattribute "StateKind" = "Initial"
metaclass "PseudoState", metaattribute "StateKind" = "History"
metaclass "PseudoState", metaattribute "StateKind" = "Branch"
[C1]/action1
S2
S1
[C2]/Action2
S3

Figure 7-3. Different kinds of *States* and *Transitions*

"StateMachine" class

StateMachine overview

```
StateMachine extends ModelElement;
```

Graph of *States* and *Transitions* that describes the dynamic behavior of objects. *StateDiagams* can also define the usage protocol for *Classes*.

Example 12 presents a *State* diagram.

In Objecteering/UML, a *StateMachine* belongs to a *Package*, an *Operation*, a *UseCase* or a *Class*. Its natural position is to belong to a *Class*.

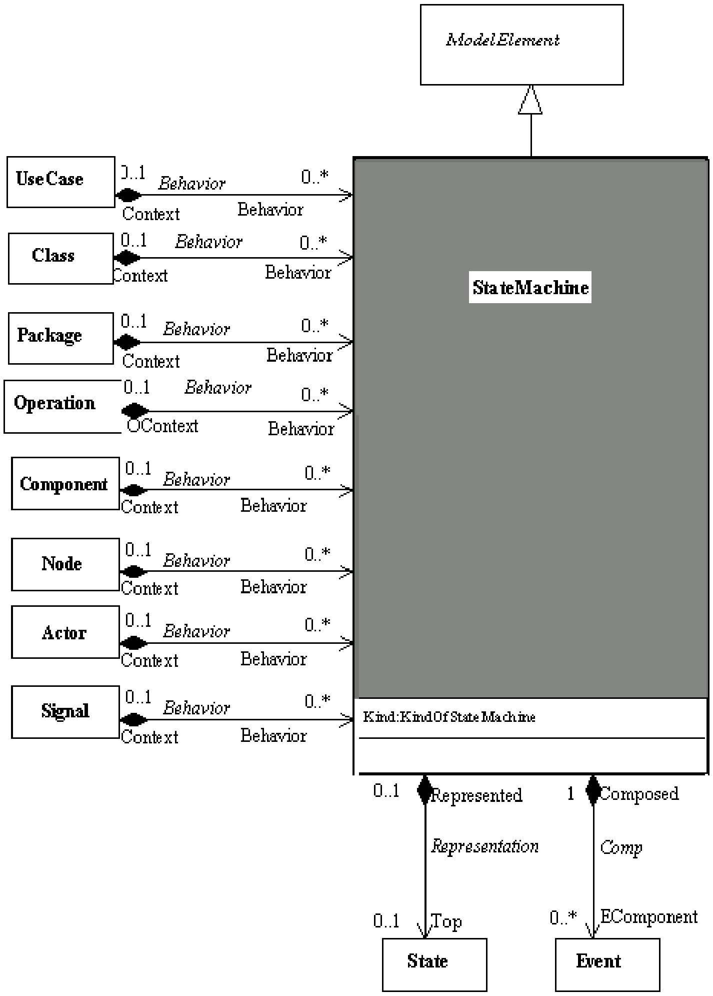


Figure 7-.4 Detailed class diagram for *StateMachine*

See also: *State*, *Transition*, *Enumerated type*.

StateMachine properties

The class has the following associations:

- *Top:State*: Defines the root state for the current *StateMachine*. All other *States* will be sub-states of the *TopStat*e.
- *EComponent:Event*: *Events* are defined in the context of a *StateMachine*.

The class owns the following attribute:

- *Kind*: A *StateMachine* can be a dynamic *StateMachine*, as usually defined in UML (Harel *State* diagrams), or protocol *StateMachine*. Protocol *StateMachines* represent the usage protocol of the class's *Operations*. It defines in which order and for which conditions and *States* an *Operation* can be invoked.

StateMachine consistency rules

A *StateMachine* can only redefine another *StateMachine* if it is a class component.

Consistency between the two *StateMachines* is not maintained.

Events defined in the *StateMachine* have a unique name.

A *StateMachine* always has a *Top* state.

StateMachine methods

`State[] getRootState()`
Returns the root *State* of the *StateMachine.*

`Diagram[] getDiagrams()`
Returns the state *Diagrams* of the *StateMachine*.

`State[] getStates()`
Returns the *States* defined in the *StateMachine*.

`Transition[] getTransitions()`
Returns the *Transitions* defined in the *StateMachine.*

"StateVertex" class

StateVertex overview

```
StateVertex extends ModelElement;
```

Abstraction of a node in a *Statechart* graph. A *StateVertex* can be either a *State*, or a *PseudoState* that is only a graphical convention. Different kinds of *StateVertex* can be seen in Example 12.

StateVertex belong to a *State*, or to the *StateMachine* if they are a root *State*.

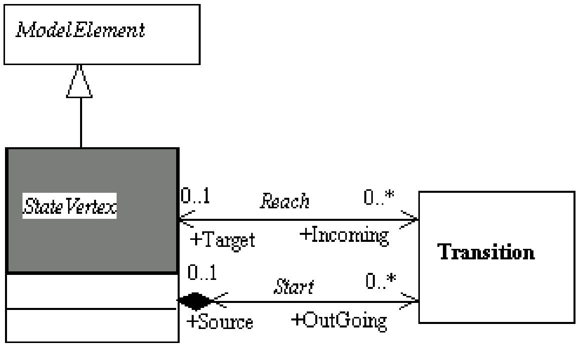


Figure 7-5. Detailed class diagram for *StateVertex*

StateVertex properties

The class has the following associations:

- *Incoming:Transition*: Specifies the *Transitions* entering the *Vertex*.
- *OutGoing:Transition*: Specifies the *Transitions* departing from the *Vertex*.

"State" class

State overview

```
State extends StateVertex;
```

Notable situation or condition during the life of an object.

A *State* represents a period of time during which an object waits for some *Event* to occur; or a period of time during which an object performs some ongoing activity. *States* are interconnected by *Transitions*.

Example 12 shows different kind of *States*.

In Objecteering/UML, *States* belong either to another *State*, or to a *StateMachine* if they are the root (*Top*).

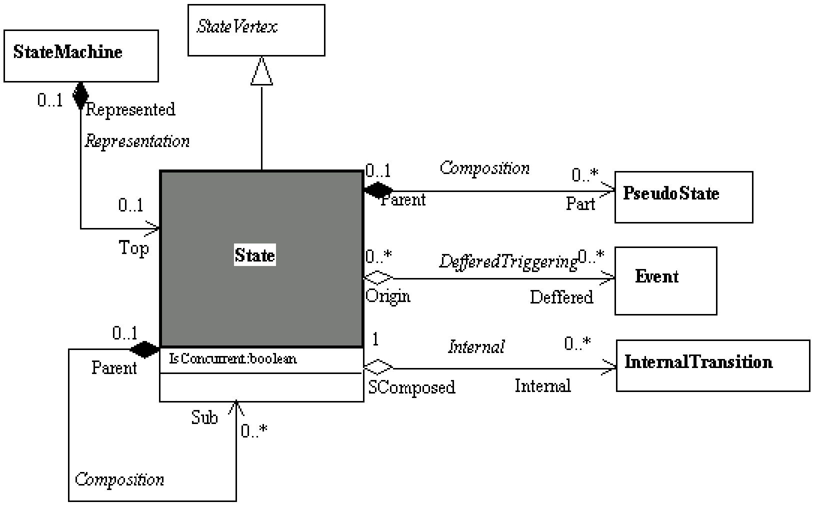


Figure 7-6. Detailed class diagram for *State*

<u>See also</u>: *StateMachine*, *Event*, *Transition.*

State properties

The class has the following associations:

- *Deferred:Event*: A list of *Events* the effect of whose occurrence during the *State* is postponed until the owner enters a *State* in which they are not deferred, at which time they may trigger *Transitions* as if they had just occurred.
- *Internal:InternalTransition*: *Transitions* that occur entirely within the *State*. If one of their triggers is satisfied then the action is performed without changing *State*. This means that the entry or exit condition of the *State* will not be invoked. These *Transitions* apply even if the *StateMachine* is in a nested region and they leave it in the same *State*.
- *Sub:State*: Hierarchical decomposition of *States*. In Objecteering/UML, a *State* is a composite *State* if there are *substates*.
- *Part:PseudoState*: *PseudoStates* which decompose the current *State*. *PseudoStates* are managed by a specific association, in order to clearly separate them from real *States*.

The class owns the following attribute:

- *IsConcurrent*: Defines a concurrent *State*. A concurrent *State* is a composite *State*, whose *substates* are concurrent areas. If a *State* is concurrent, it is made up of threads, which are states that run simulataneously.

State consistency rules

- A composite *state* must not contain more than one initial *state*, one final *state*, one deep history *state* and one shallow history *state* (which are *PseudoStates*).
- *States* within a single *state* have a unique name.
- The *Representation::StateMachine* and *Composition::State* associations are exclusive.

State methods

`State[] getParentState()`
Returns the parent *State*.

`Event[] getEvents()`
Returns the *Events* defined on the *State*.

`Transition[] getIncomingTransitions()`
Returns the *Transition* entering into the *State*.

`Transition[] getOutGoingTransitions()`
Returns the *Transitions* departing from the *State*.

`State[] getParentStates()`
Returns all parent states, except root *State*.

`State[] getSubStates()`
Returns the sub-states of the *State*.

"PseudoState" class

PseudoState overview

```
PseudoState extends StateVertex;
```

Abstraction of different types of nodes in the *StateMachine* graph. *PseudoStates* represent every kind of graphical node in a *StateMachine*, except *States*, such as a branch, a fork, etc.

Example 12 presents various kinds of *PseudoStates*.

In Objecteering/UML, a *PseudoState* belongs to a *State.*

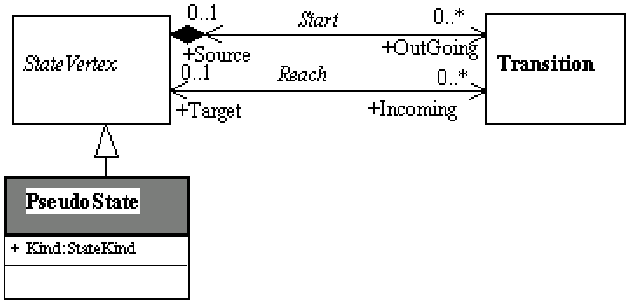


Figure 7-7. Detailed class diagram for *PseudoState*

See also: *Transition*, *Enumerated type*.

PseudoState properties

The class owns the following attribute:

- *Kind*: Nature of the *PseudoState* (InitialState, DeepHistoryState, etc.)

"Transition" class

Transition overview

```
Transition extends ModelElement;
```

Path from one *State* to another. *Transitions* represent the reaction of an object in a certain *State*, to a particular *Event*. For protocol *State diagrams*, *Transitions* represent the possible paths between *States*.

Examples 11 and 12 present *Transitions*.

In Objecteering/UML, a *Transition* belongs to its *SourceStateVertex*.

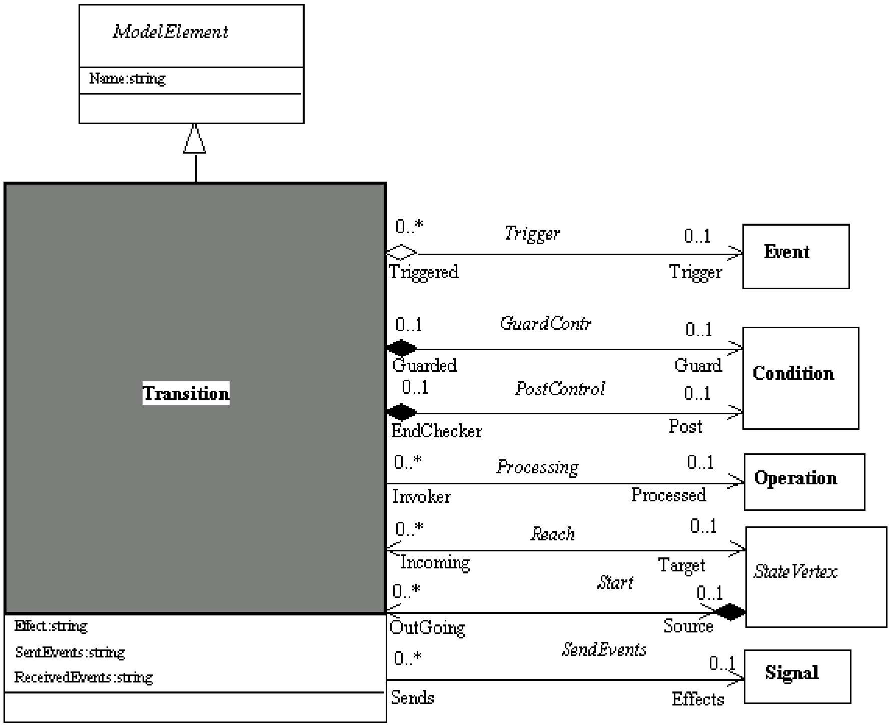


Figure 7-8. Detailed class diagram for *Transition*

See also: *StateMachine*.

Transition properties

The class has the following associations:

- *Post:Condition*: *Condition* that must be fulfilled once the transition has occurred. Useful only for protocol *State* diagrams.
- *Processed:Operation*: *Operation* processed once the *Transition* is triggered. This is a shorthand for a call *Event*, and is at the same time useful for defining the *Operation* carried by a transition in protocol *State* diagrams.
- *Guard:Condition*: *Condition* under which a *Transition* may be triggered.
- *Target:StateVertex*: Specifies the *Transitions* entering the *Vertex*.
- *Source:StateVertex*: Specifies the *Transitions* departing from the *Vertex*.
- *Trigger:Event*: *Events* that may trigger the *Transition* (under initial *State* and initial *Transitions*). This association is exclusive from the *ReceivedEvents* string.
- *Effects:Signal*: When the *Transition* is accomplished, an occurrence of this *Signal* will be sent.

The class owns the following attributes:

- *Effect*: Defines the actions triggered by the *Transition*. This field excludes the association *ProcessedOperation* that is a shorthand for defining a call action.
- *SentEvents*: *Events* sent by the *Transition* once it is triggered.
- *ReceivedEvents*: Received *Events* that trigger the *Transition*.

Transition consistency rules

- The *Processed*, *Trigger*, *ReceivedEvent* and *PredefinedEvent!=otherEvent* properties are exclusive.
- The *SendEvent* and *SentEvents* properties are exclusive.

Transition methods

`Condition[] getPostCondition()`
Returns the *Condition* that must be fulfilled once the *Transition* has occurred.

`State[] getTargetState()`
Returns the target *State* of the *Transition*.

`Condition[] getGuardCondition()`
Returns the *Condition* in which a *Transition* may be triggered.

`State[] getSourceState()`
Returns the source *State* of the *Transition*.

`Event[] getTriggerEvent()`
Returns the *Event* that may trigger the *Transition*.

`Operation[] getProcessedOperation()`
Returns the *Operation* processed once the *Transition* is triggered

`Signal[] getSignal()`
Returns the *Signal* sent when the *Transition* is accomplished.

"InternalTransition" class

InternalTransition overview

```
InternalTransition extends Transition;
```

Transition that is internal to a *State*. It can be triggered on entry, or exit to the *State*, or can describe an activity that is performed while being in the *State* (do *Transitions*). (See Example 7).

A *Transition* belongs to its origin *StateVertex*.

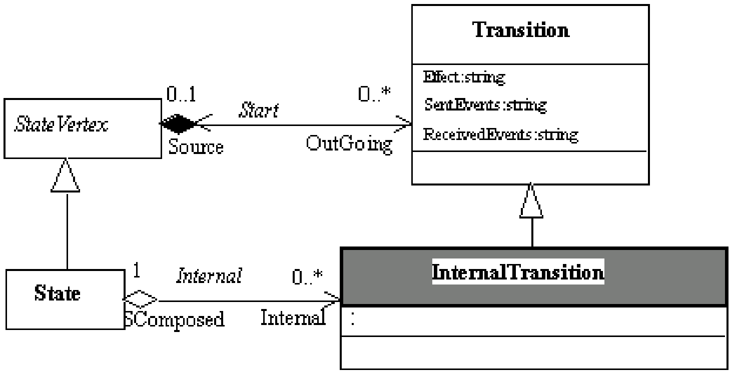


Figure 7-9. Detailed class diagram for*InternalTransition*

See also: *State*, *Enumerated type*.

InternalTransition consistency rules

The *Start* and *Reach* associations must be empty on an *InternalTransition*.

"Condition" class

Condition overview

```
Condition extends ModelElement;
```

Boolean expression for making a choice.

Conditions are mainly used in *State diagrams*, in order to guard the *Transitions*. The *Name* of a *Condition* contains an expression (string) which produces its valuation in any programming language.

Examples of *Conditions* can be seen in Example 11 (State Diagrams).

In Objecteering/UML, a *Condition* belongs to *Messages* or *Transitions*.

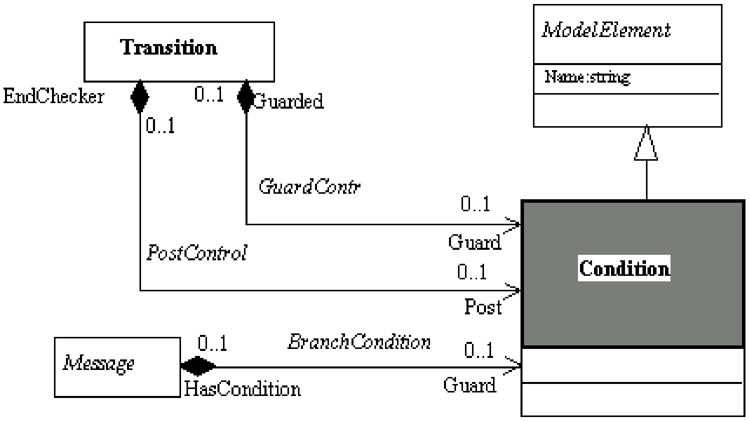


Figure 7-10. Detailed class diagram for *Condition*

<u>See also</u>: *Transition*, *Message*.

"Event" class

Event overview

```
Event extends ModelElement;
```

Specification of a significant occurrence that has a location in time and space. An instance of an *Event* can lead to the activation of a behavioral feature in an object.

An *Event* can be either an occurrence of a *Signal*, a message occurrence, a time or a change expression occurrence. (See Example 11)

In Objecteering/UML, an *Event* belongs to a *StateMachine*.

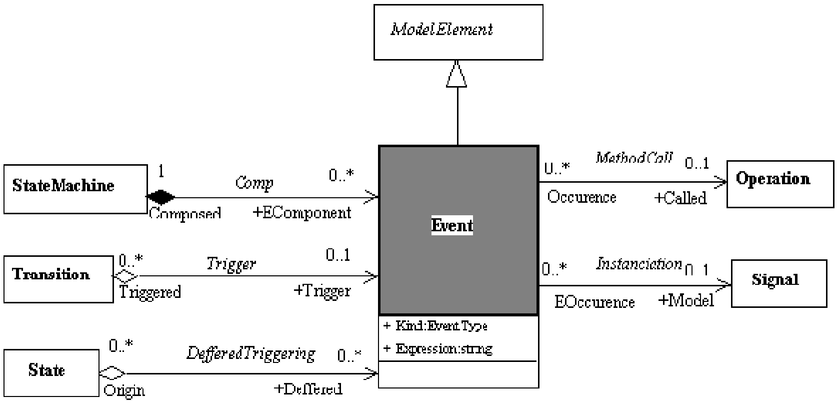


Figure 7-11. Detailed class diagram for *Event*

<u>See also</u>: *Signal*, *State*, *Transition*, *StateMachine*, *Operation*, *Enumerated type*.

Event properties

The class has the following associations:

- *Called:Operation*: Direct link to an *Operation* in case of a call event.
- *Model:Signal*: *Signal* from which the *Event* is an occurrence.

The class owns the following attributes:

- *Kind*: Defines the nature of the event (Time, Signal occurrence, etc.)
- *Expression*: Expression initiating the *Event*. This can be a time expression, or a triggering condition, it may contain parameters values in case of *Operation* call *Event*, etc.

Event consistency rules

Event names are unique within their *State Diagram* "*owner*".

The associations towards *Signal* and *Operation* are exclusive between them and with the "*Expression*" attribute. They depend on the value of the "*Kind*" attribute:

- If *Kind* == *SignalEvent*, there is a relation towards *Signal*
- If *Kind* == *CallEvent*, there is a relation towards *Operation*
- If *Kind* == *TimeEvent*, the *Expression* attribute must not be empty
- If *Kind* == *ChangeEvent*, the *Expression* attribute must not be empty

Chapter 8: Activity Model

Overview

Presentation

An activity diagram graphically illustrates an *ActivityGraph*, which shows a procedure or a workflow. An ActivityGraph is a special instance of a state machine in which all or most of the states are activity states or action states, and in which all or most of the transitions are triggered by completion of activity in the source states.

Metamodel synthesis

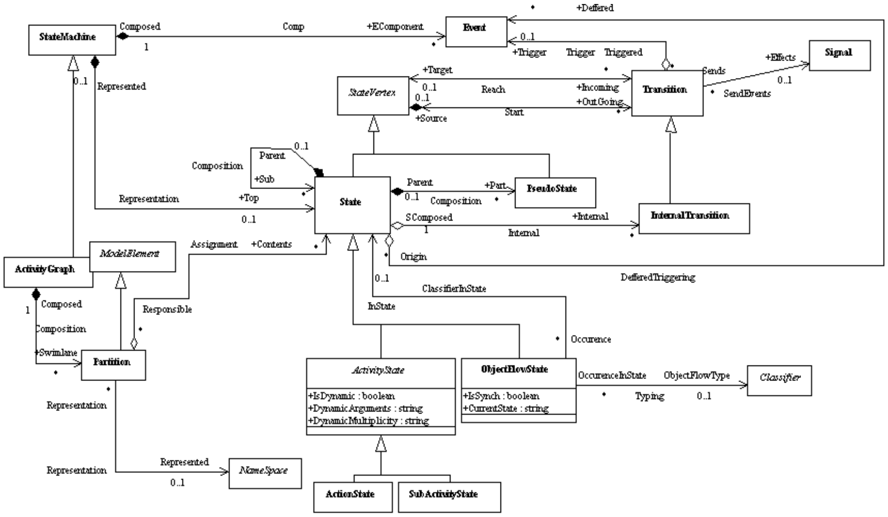


Figure 8-1. *ActivityModel*: overview of the metamodel

The Activity model is organized around *ActivityGraphs*, *ActionStates* and *ObjectFlowStates*. *Partitions* are another important concept. To create an activity diagram, an *ActivityGraph*, for which an activity diagram is subsequently created, must first be created. ActivityGraphs can be created on *Class*, *Component*, *Node*, *Signal*, *UseCase* and *Actor* classifiers, and on *Operations* and *Packages*.

Activity model metaclasses

- *ActivityGraph*: Graph of activity states, action states and object flow states, which can be organized into partitions, and which shows a procedure or workflow. Activity diagrams are created from *ActivityGraphs*.
- *ActivityState*: A state which describes a specific activity at a given point in time.
- *ActionState*: A state which may not be broken down, and which describes an action. The name of this state is, be default, the name of the action triggered.
- *SubActivityState*: A state which breaks down into another activity graph.
- *ObjectFlowState*: Special state, associated to the class it represents.
- *Partition*: The division of an activity graph. *ActionStates* can be allocated to partitions graphically, through the "*Assignment*" association. They are also sometimes called *swimlanes*.
- *SignalSending*: A pseudo state which has no actions, but whose outgoing transition sends a signal.
- *SignalReceipt*: A pseudo state which has no actions, but whose incoming transition receives a signal.

Example 13: Activity diagram without partitions

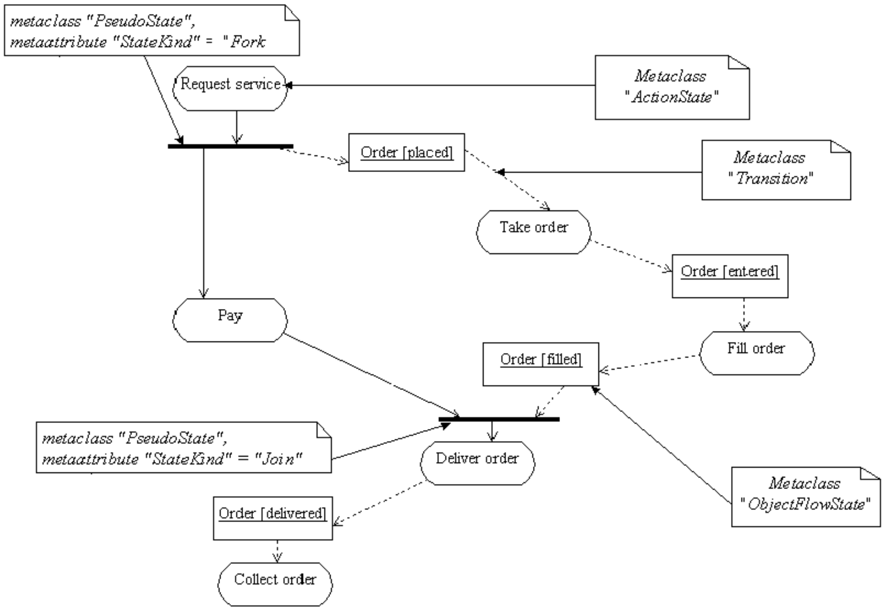


Figure 8-2. Activity diagram without partitions

Example 14: Activity diagram with partitions

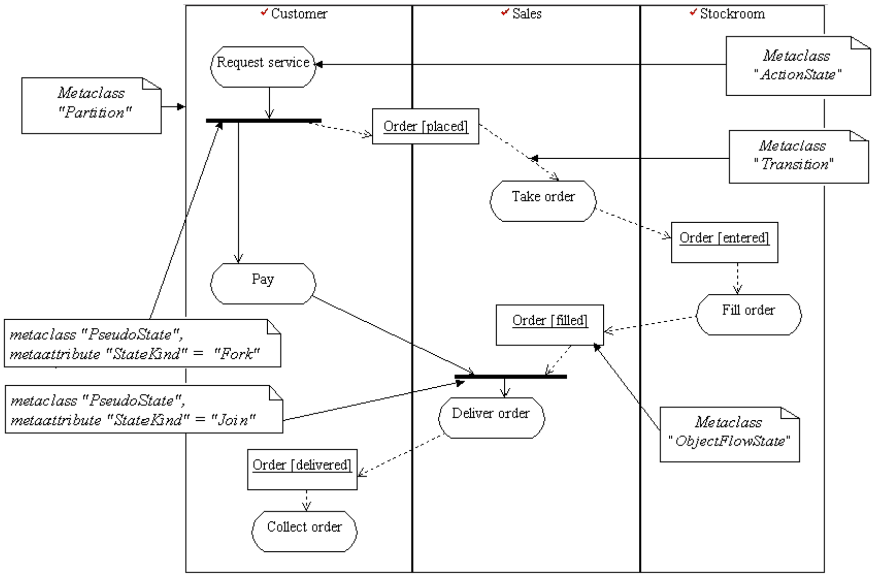


Figure 8-3. The same activity diagram as shown in Figure 8-2, but featuring partitions

ActivityGraph class

ActivityGraph overview

```
ActivityGraph extends StateMachine;
```

Special case of a *StateMachine* that defines a computational process in terms of the control-flow and object-flow among its constituent actions.

Activity diagrams define shorthand forms that are convenient for modeling control-flow and object-flow in computational and organizational processes.

The primary basis for activity graphs is to describe the states of an activity or process involving one or more classifiers. *Activity graphs* can be attached to a *Package*, a *Classifier* (including *UseCases*) or an Operation .

Activity diagrams are associated to ActivityGraphs and to *ActionStates* that are decomposed into *SubActivityStates*.

An *ActivityGraph* is the container of all elements of an activity diagram, through its partition and its top state decomposition.

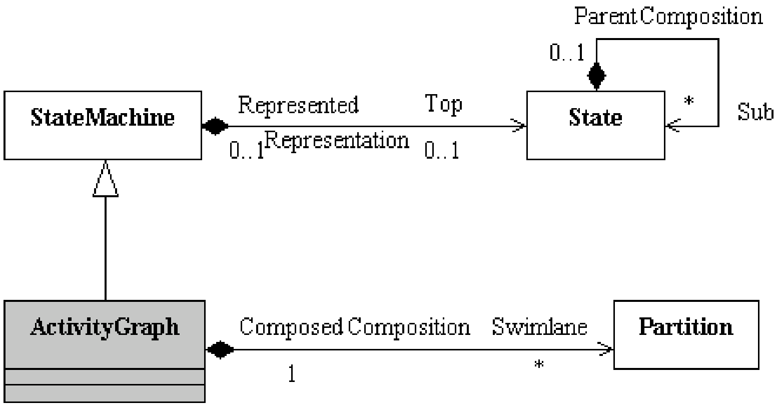


Figure 8-4. Detailed class diagram for *ActivityGraph*

See also: *StateMachine*, *State*, *Partition*

ActivityGraph properties

The *ActivityGraph* class contains the following association:

- *Swimlane:Partition*: *Partitions* belonging to the *ActivityState*, each of which contains some of the model elements.

ActivityGraph consistency rules

An *ActivityGraph* must belong to an *Operation* or a *NameSpace*, except a *DataType*, or an *Enumeration*.

An *ActivityGraph* is always made up of a root state (*SubActivityState*).

ActivityState class

ActivityState overview

```
ActivityState extends State;
```

Execution of an atomic action, typically the invocation of an *Operation*. An activity state is a simple *State* with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action. The *State* therefore corresponds to the execution of the entry action itself and the outgoing *Transition* is activated as soon as the action has completed its execution.

An ActivityState may perform more than one action as part of its entry action. An ActivityState may not have an exit action, a do activity or *InternalTransitions*.

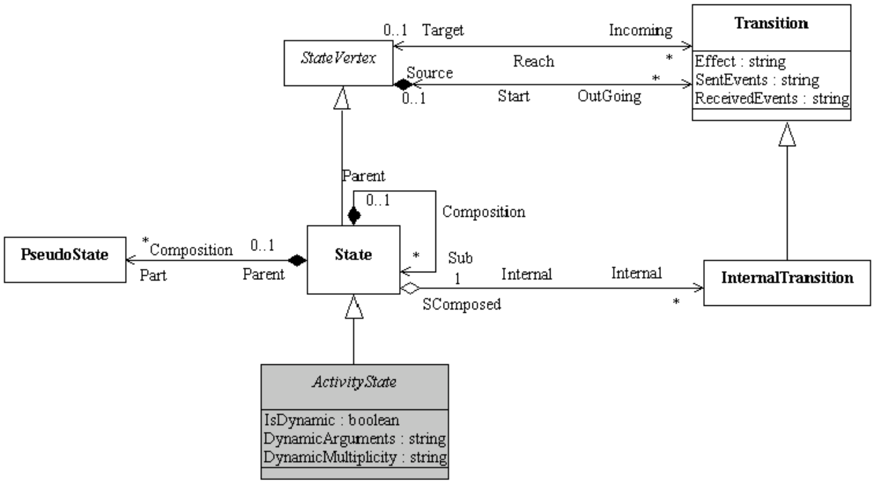


Figure 8-5. Detailed class diagram for *ActivityState*

<u>See also</u>: *State*, *ActionState*, *SubActivityState*

ActivityState properties

The *ActivityState* class contains the following attributes:

- *IsDynamic*: Specifies whether the state's actions might be executed concurrently. It is used in conjunction with the *dynamicArguments* attribute.
- *DynamicArguments*: ArgListsExpression that determines at runtime the number of parallel executions of the actions of the state. The value must be a set of lists of objects, each list serving as arguments for one execution. This attribute is ignored if the isDynamic attribute is false.
- *DynamicMultiplicity*: Number of parallel executions of the actions of state. This attribute is ignored if the isDynamic attribute is false.

ActionState class

ActionState overview

```
ActionState extends ActivityState;
```

Description of an action that cannot be broken down further. An *ActionState* has no SubStates.

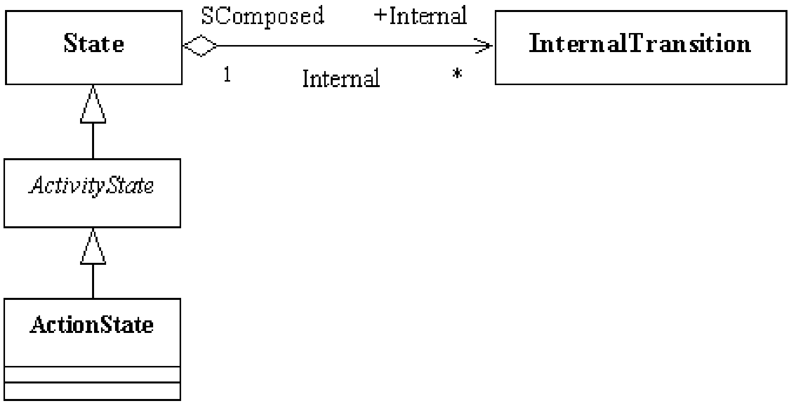


Figure 8-6. Detailed class diagram for *ActionState*

ActionState consistency rules

An *ActionState* never contains *States*.

An *ActionState* always has an *InternalTransition*.

An *ActionState* belongs to a *SubActivityState*.

SubActivityState class

SubActivityState overview

```
SubActivityState extends ActivityState;
```

Activities that are decomposed into sub activities. SubActivityStates are decomposed into SubStates through the "*Composition*" association, defined on the parent *State* metaclass.

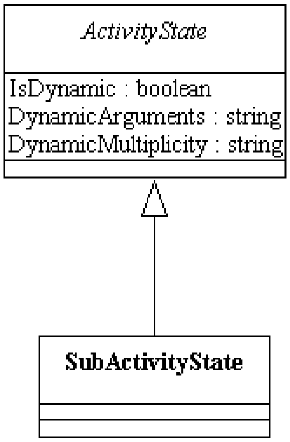


Figure 8-7. Detailed class diagram for *SubActivityState*

See also: *ActivityState*

SubActivityState consistency rules

A *SubActivityState* has no *InternalTransitions*.

A *SubActivityState* belongs to another *SubActivityState* or to an *ActivityGraph*.

ObjectFlowState class

ObjectFlowState overview

```
ObjectFlowState extends State;
```

Defines an object flow between actions in an *ActivityGraph*. Signifies the availability of an instance of a *Classifier*, possibly in a particular state or condition, usually as the result of an *Operation*. An instance of a particular class, possibly in a particular state, is available when an *ObjectFlowState* is occupied.

The generation of an object by an action in an *ActionState* may be modeled by an object flow state that is triggered by the completion of the action state. The use of the object in a subsequent *ActionState* may be modeled by connecting the output transition of the object flow state as an input transition to the action state. Generally, each action places the object in a different state that is modeled as a distinct object flow state.

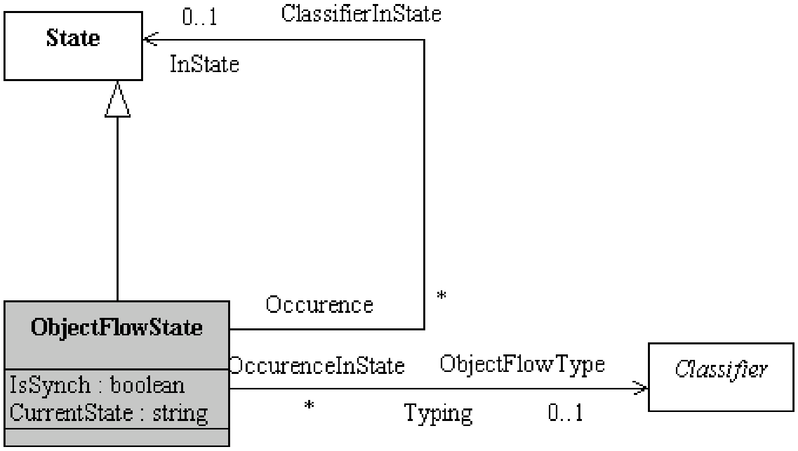


Figure 8-8. Detailed class diagram for *ObjectFlowState*

See also: *State*, *Classifier*

ObjectFlowState properties

The *ObjectFlowState* class contains the following associations:

- *ObjectFlowType:Classifier*: Classifier that defines the base class of the current ObjectFlowState. The CurrentObjectFlow is an instance of the associated Classifier.
- *InState:State*: Association to the state representing the current state of the ObjectFlowState.

The *ObjectFlowState* class contains the following attributes:

- *IsSynch*: Indicates whether an object flow state is used as a synch state.
- *CurrentState*: Value of the current state. If the association "ClassifierInState" is set, then this value has no meaning. By extension, this value can be a boolean expression, or any text expressing a current situation. This is a more flexible way of representing "ClassifierInState" elements in an activity diagram.

ObjectFlowState consistency rules

An *ObjectFlowState* never contains *States.*

An *ObjectFlowState* has no *InternalTransitions*.

An *ObjectFlowState* always belongs to a *SubActivityState*.

Partition class

Partition overview

```
Partition extends ModelElement;
```

Mechanism for dividing the states of an *ActivityGraph* into groups. Partitions often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the *States* of an activity graph.

An activity can be referenced by several partitions. This means that different responsibilities can be presented in different diagrams. For example, a department (package) can be responsible for an activity in a diagram, whereas in a more detailed diagram, an *Actor* is presented as being responsible for the activity in question. *PseudoStates* and *ObjectFlowStates* can be presented straddling several partitions. This presentation is purely graphic, and does not necessarily correspond to their affectation to a partition.

In Objecteering/UML, a partition can be associated to a namespace that it represents (extension to UML). Very often, a partition represents a *Classifier* or a *Package* (organizational unit).

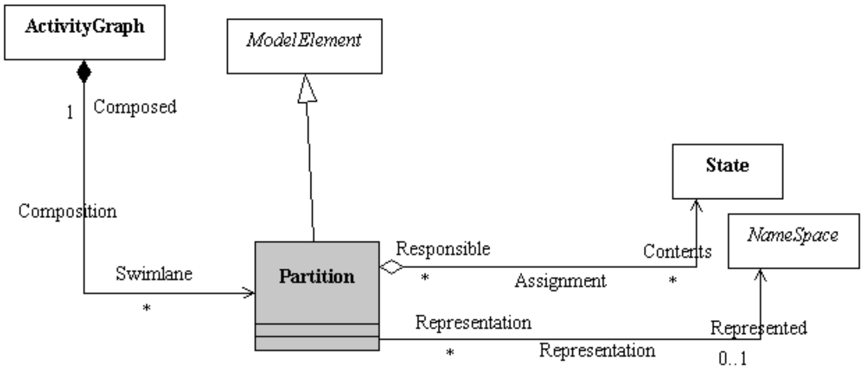


Figure 8-9. Detailed class diagram for *Partition*

See also: *ActivityGraph*, *ModelElement*, *State*, *NameSpace*

Partition properties

The *Partition* class contains the following associations:

- *Contents:State*: Specifies the states that belong to the partition. They need not constitute a nested region.
- *Represented:NameSpace*: In Objecteering/UML, partitions can represent *NameSpaces*. They very often represent *Classifiers* as active elements or *Packages* as organizational units.

SignalSending and SignalReceipt pseudo states

SignalSending overview

```
SignalSending is a particular case of PseudoState;
```

SignalSending is a state which has no actions, but whose outgoing transition sends a signal. The dialog box reflects this fact, inasmuch as it contains a "*name*" field and a "*sent signal*" field. The signal entered corresponds to the outgoing transition's "send events".

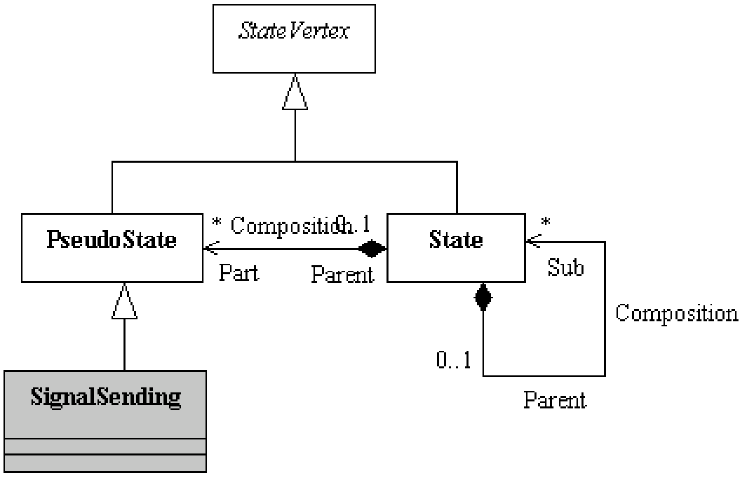


Figure 8-10. Detailed class diagram for *SignalSending*

See also: *PseudoState*, *StateVertex*

SignalReceipt overview

```
SignalReceipt is a particular case of PseudoState;
```

SignalReceipt is a state which has no actions, but whose incoming transition receives a signal. The dialog box reflects this fact, inasmuch as it contains two fields, "*Name*" and "*Received event*". "*Received event*" corresponds to the "*Trigger event*" of the incoming transition.

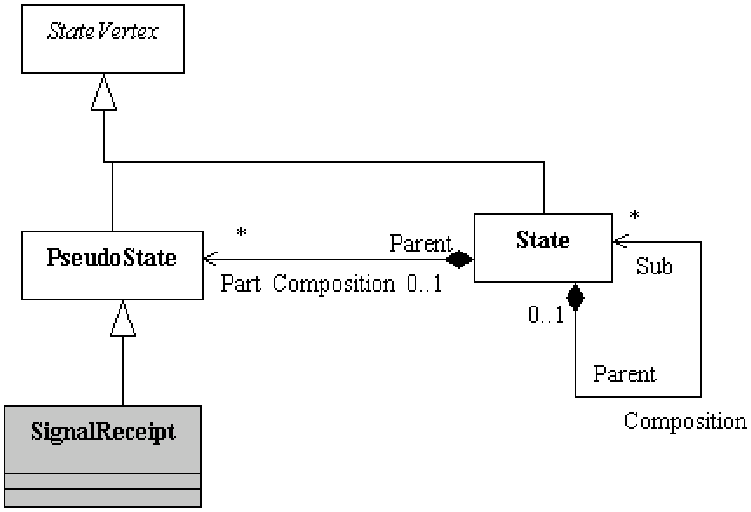


Figure 8-11. Detailed class diagram for *SignalReceipt*

See also: *PseudoState*, *StateVertex*

Chapter 9: Physical Model (Components and Nodes)

Overview

Presentation

The *Implementation diagram* and the *Deployment diagram* are supported by this package, the two major notions being *Component* and *Node*.

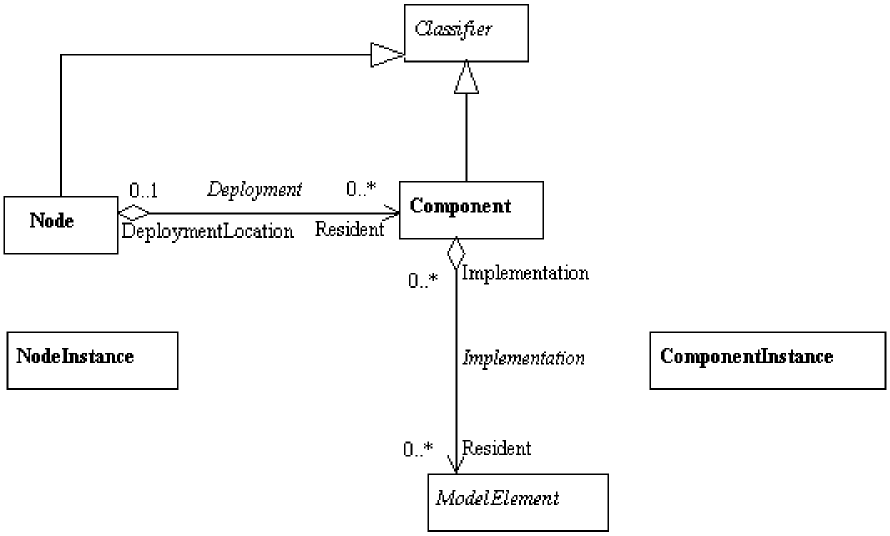


Figure 9-1. Physical model

Node and *Component* extend the *Classifier* metaclass. Therefore, they benefit from the previously defined notions of association between *Nodes*, dependencies for *Components*, *Interface* implementation, etc.

The *Deployment* association declares which *Components* are deployed on which *Nodes*. The *Implementation* association defines the correspondence between the logical and the physical models.

Physical model metaclasses

- *Node*: Run-time physical object which represents a computational resource.
- *Component*: Physical unit of implementation with well-defined interfaces, which is intended to be used as a replaceable part of a system.
- *ComponentInstance*: Instance of a *Component*.

Example 15: Implementation diagram

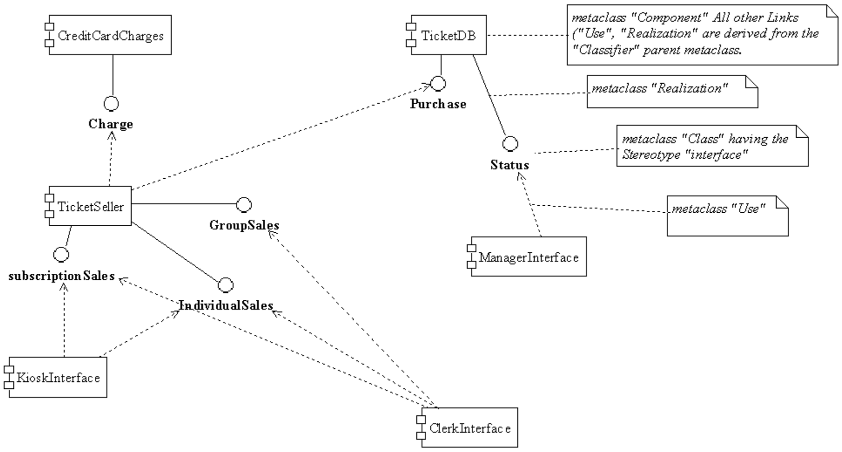


Figure 9-2. See the "*Component*" metaclass

Example 16: Deployment diagram

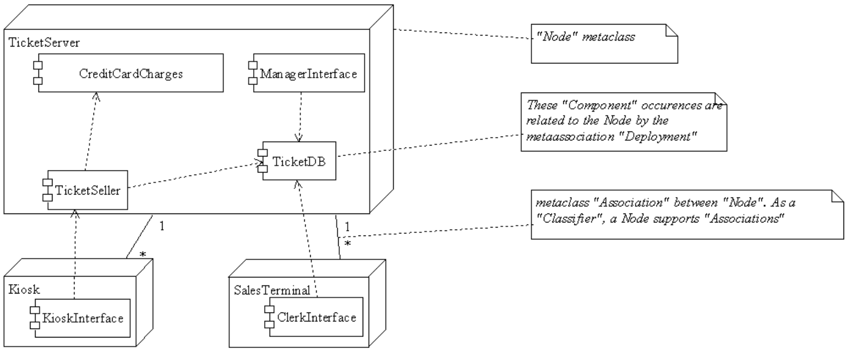

TicketServer
CreditCardCharges
ManagerInterface
TicketDB
TicketSeller
1
*
1
*
Kiosk
KioskInterface
SalesTerminal
ClerkInterface
"Node" metaclass
These "Component" occurences are related to the Node by the metaassociation "Deployment"
metaclass "Association" between "Node". As a "Classifier", a Node supports "Associations"

Figure 9-3. The "*Node*" metaclass is developed here

Example 17: Instance of Nodes and Components

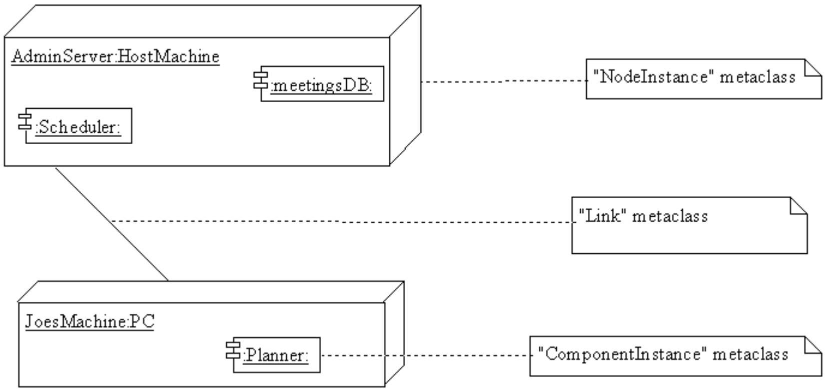


Figure 9-4. Instance of *Nodes* and *Components*

"Node" class

Node overview

```
Node extends Classifier;
```

Run-time physical object that represents computational resource, *Nodes* generally have at least a memory and often also processing capacity. *Associations* between *Nodes* represent communications paths.

As presented in Example 16, *Nodes* represent the deployment of Components.

Nodes belong to their owner *NameSpace*, which must be a *Package*.

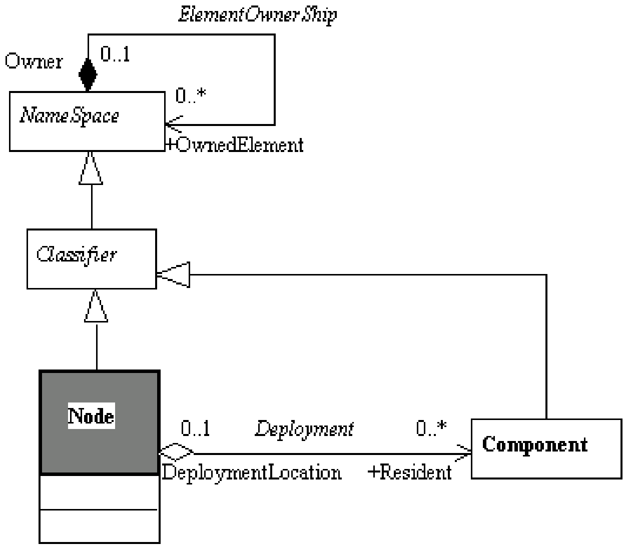


Figure 9-5. Detailed class diagram for *Node*

Node properties

The class has the following association:

- *Resident:Component*: Defines the *Components* that reside in the *Node*.

Node consistency rules

- A *Node* can specialize a *Node* (relationship *NameSpace* towards *Generalization*) which belongs to the visibility domain of the *Package* in which it is contained.
- The *Instantiation* association of *ClassifierOccurrence* towards *Classifier*, for the case of a *Node* exclusively represents *NodeInstances*.
- The *Components* accessible for the *Deployment* relationship are the *Components* accessible from the *Package* containing the *Node*.
- The name of a *Node* is unique in the *Package* in which it is contained.

Node methods

```
AssociationEnd[] getAssociationEnds()
```

Returns the *AssociationEnds* starting from the *Node*.

```
Node[] getParentNodes()
```

Returns the parent *Nodes*.

"Component" class

Component overview

```
Component extends Classifier;
```

Physical unit of implementation with well-defined *Interfaces*, which is intended to be used as a replaceable part of a system. Each *Component* embodies the implementation of certain classes from the system design. Any physical element in a software development can be represented by a *Component*. A library, a binary or a bean are examples of *Components*. By extension, any work product (a C++ source, documentation, etc.) can be a specific kind of *Component*.

In Objecteering/UML, a *Component* belongs to a *NameSpace*.

Example 15 shows representations of *Components*.

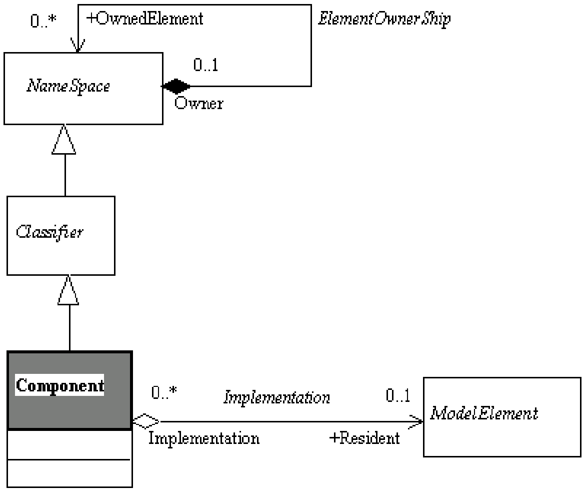


Figure 9-6. Detailed class diagram for *Component*

See also: *Node*, *MpGenProduct*.

Component properties

The class has the following association:

- *Resident:ModelElement*: Defines all the *ModelElements* implemented by the *Component*. It generally concerns classes and packages, but there is no restriction to the kind of element that a *Component* can implement.

Component consistency rules

- There can be no generalization relationship on a *Component*.
- There can be no redefinition of *Operations*, since there is no generalization.
- There can be no associations between *Components*.
- A *Component* can use (*Use* link) all the *Components*, *Actors* and *Classes* visible from the *NameSpace* in which it is contained.
- A *Component* can implement (*Realization* link, towards the class of the same name) all the interface *Classes* (containing the *interface* stereotype) visible from the *NameSpace* in which it is contained.
- A *Component* can implement all the *ModelElements* of the project (*Implementation:Component* link).
- A *Component* has a unique name in its compound.
- If it implements a *Class*, this class has the *Interface* stereotype.
- A *Component* is never linked to an *AssociationEnd*.

Component methods

```
Component[] getComponents()
```

Returns sub-components of the *Component*.

```
ModelElement[] getImplementedElements()
```

Returns the *ModelElements* implemented by the *Component*.

```
NameSpace[] getUsedNameSpaces()
```

Returns the *Elements* used by the *Component*.

```
DataFlow[] getReceivedDataFlows()
```

Returns the *DataFlows* received by the *Component*.

```
DataFlow[] getSentDataFlows()
```

Returns the *DataFlows* sent by the *Component*.

```
Operation[] getOperations()
```

Returns the *Operations* of the *Component*.

"NodeInstance" class

NodeInstance overview

```
NodeInstance extends Instance;
```

Instance of a *Node*. *Nodes* represent a kind of executable unit, whereas *NodeInstances* represent examples of communicating *Nodes*. Through a clustering association, *NodeInstances* can present the instances that they contain, such as objects or components.

Being an Instance, a *NodeInstance* also belongs to its *NameSpace*.

Example 17 presents *NodeInstance* representations.

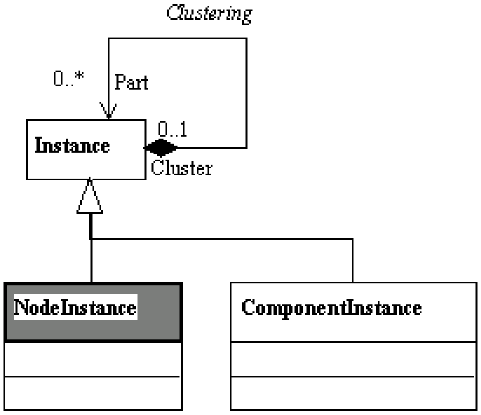


Figure 9-7. Detailed class diagram for *NodeInstance*

See also: *ComponentInstance*, *Node*, *Component*.

NodeInstance consistency rules

- A *NodeInstance* can only represent (*instantiation* link) a *Node*.
- The *Nodes* accessible are those which may be accessed by the *NameSpace* which contains the *NodeInstance*.
- A *NodeInstance* has a unique name in its *NameSpace*.

"ComponentInstance" class

ComponentInstance overview

```
ComponentInstance extends Instance;
```

Instance of a *Component*. The semantic range of components starts from C++ source code, documentation or an executable library, from the *Component* model dedicated to business objects, such as Java ejb, for example. In the "*business object Component*" examples, *ComponentInstances* are necessary. These *Instances* can be deployed in specific *NodeInstances*, and their behavior is specific.

Example 17 presents an instance of the implementation and deployment diagram.

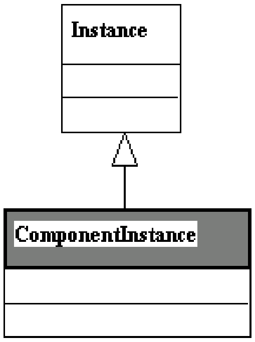


Figure 9-8. Detailed class diagram for *ComponentInstance*

See also: *NodeInstance*, *Component*.

ComponentInstance consistency rules

- A *ComponentInstance* is only created in a *NodeInstance* or in a *ComponentInstance*.
- A *ComponentInstance* can only represent (*Instantiation* association) a *Component*.
- The name of a *ComponentInstance* is unique in the instance in which it is contained.

Chapter 10: Collaboration, Roles and Instances (Collaboration and Instances) Model

Overview

Presentation

Collaboration is a structure unit for modeling roles. Object diagrams and sequence diagrams present either instances or roles, depending on whether they are in a *Collaboration* or in a *NameSpace*.

Metamodel synthesis: The "*Occurence*" metamodel

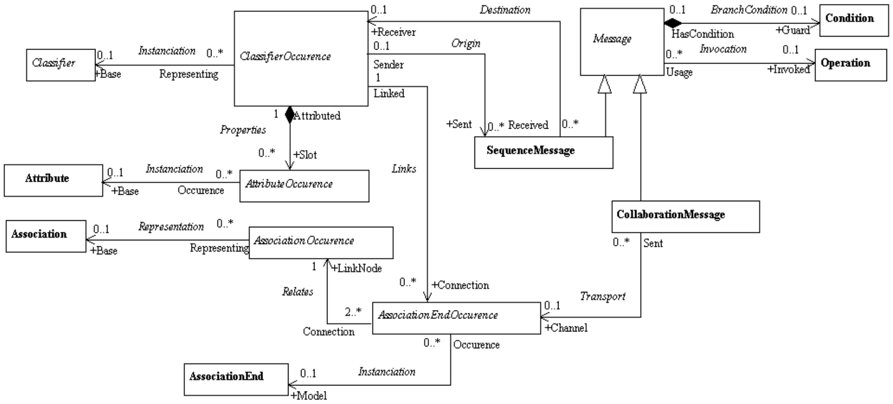


Figure 10-1. *Occurences*

Occurences define an abstract level for the instance model or the role model. They give a general view of how objects are related to their links, model, etc. *Instance model* and *Role model* then add specific features, such as *Collaboration* or *NameSpace* structuring, or the representation of *Instances* in roles.

Messages have a different structure, depending on where they appear. In sequence diagrams, they have an accurate sequencing model, defined by specific associations between *SequenceMessages*.

In *Collaboration Diagrams,* they do not have such accurate information on sequencing or focus of control, but have a sequence expression string and are related to links.

Metamodel synthesis: The "*Instance*" Metamodel

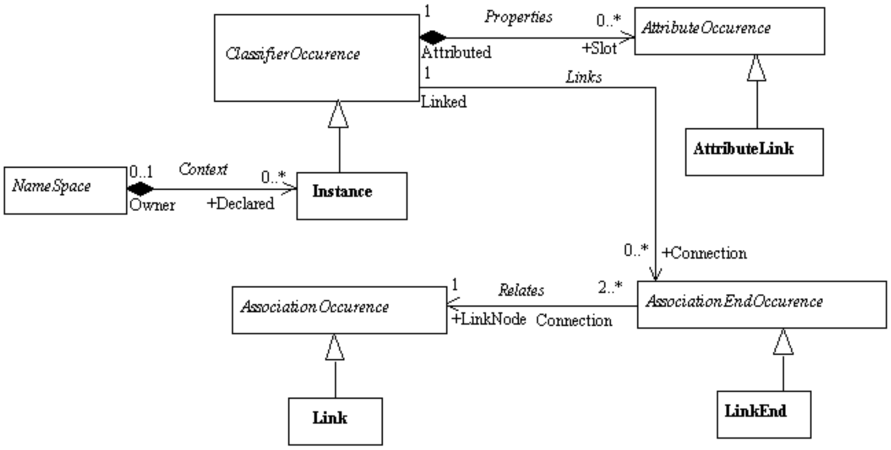


Figure 10-2. Internal package diagram for *Instances*

Instances are defined at *NameSpace* level. *Link*, *LinkEnd* and *AttributeLink* define the elements presented at *Instance* Level. *Messages* are already defined at *Occurence* abstract level.

Classes of the package

- *Collaboration*: Describes how Instances or roles can cooperate to implement a *ModelElement* such as an *Operation* or a *UseCase*.
- *Interaction* : *Message* sequencing context.
- *ClassifierOccurence*: *Instance* or *Role*, which is an occurrence of a *Classifier*.
- *Instance*: Entity to which a set of *Operations* can be applied and which has a *State* that stores the effects of the *Operations*.
- *ClassifierRole*: A *ClassifierRole* is a specific role played by a participant in a *Collaboration*.
- *AttributeOccurence*: Named slot in an *Instance* or *Role*, which has the value of an *Attribute*.
- *AttributeLink*: Named slot in an instance, which has the value of an attribute.
- *AttributeRole*: Named slot in a *ClassifierRole*, which has the value of an *Attribute*.
- *AssociationOccurence*: *Occurrence* of an *Association*. Presented as a *Link* in an *Object diagram*.
- *Link*: Connection between *Instances*.
- *AssociationRole*: Specific usage of an association needed in a collaboration.
- *AssociationEndOccurence*: End point of a *Link*.
- *LinkEnd*: End point of a *Link*.
- *AssociationEndRole*: Specifies an endpoint of an *Association* as used in a *Collaboration*.
- *Message*: Occurrence of an *Operation*, processed by *Instances*.
- *CollaborationMessage*: *Message* used in *Collaboration* or *Object diagrams*.
- *SequenceMessage*: *Messages* used for *Sequence diagrams*.

Example 18: AttributeLink, AttributeOccurrence

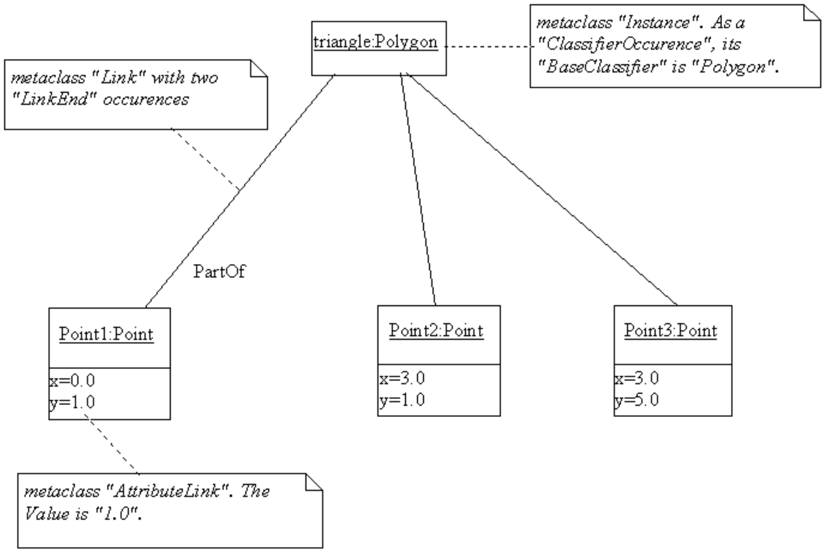

triangle:Polygon
metaclass "Instance". As a "ClassifierOccurence", its "BaseClassifier" is "Polygon".
metaclass "Link" with two "LinkEnd" occurences
PartOf
Point1:Point
x=0.0
y=1.0
Point2:Point
x=3.0
y=1.0
Point3:Point
x=3.0
y=5.0
metaclass "AttributeLink". The Value is "1.0".

Figure 10-3. Object diagram for the "*Link*", "*Instance*", "*AttributeLink*" metaclasses

Example 19: Sequence diagram for a Collaboration

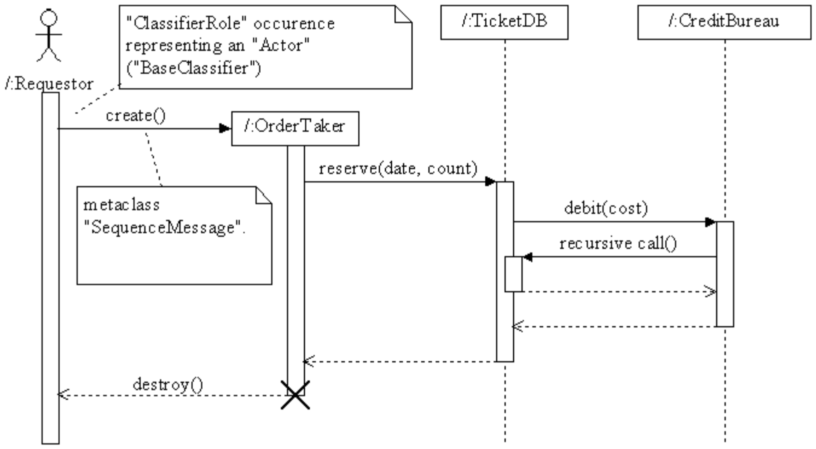


Figure 10-4. In this kind of diagram, there always exists an "*Interaction*" occurrence

Example 20: Collaboration diagram

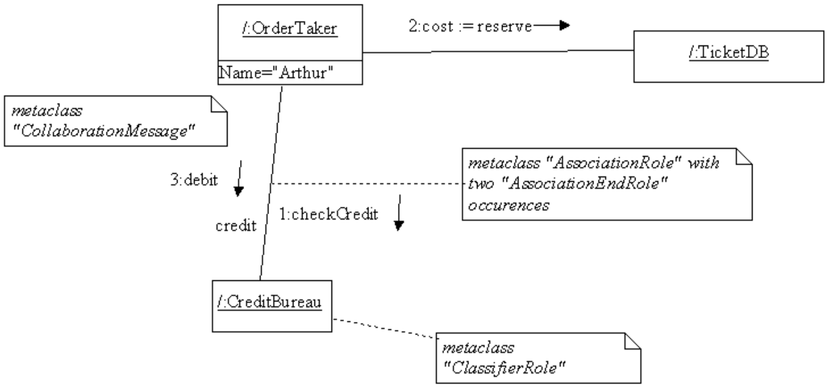

/:OrderTaker
Name="Arthur"
2:cost := reserve
/:TicketDB
metaclass "CollaborationMesage"
3:debit
credit
1:checkCredit
metaclass "AssociationRole" with two "AssociationEndRole" occurences
/:CreditBureau
metaclass "ClassifierRole"

Figure 10-5. Behind this kind of diagram, there is a "*Collaboration*" occurrence, and an "*Interaction*" occurrence

Metamodel synthesis: The "*Collaboration*" metamodel

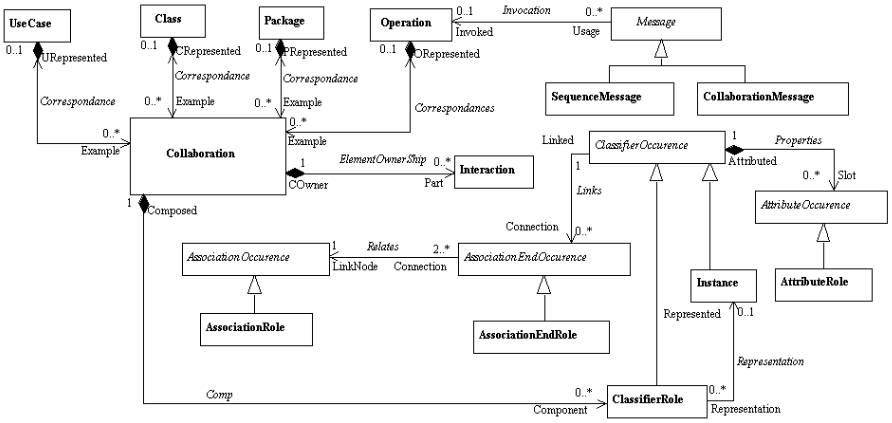


Figure 10-6. Internal package diagram for *CollaborationModel*

Collaborations present a dynamic view for *NameSpaces*, typically for *UseCases* and *Packages*. They structure models where *ClassifierRole* is the major notion, accompanied by *Messages*, *AssociationRoles*, etc.

"Collaboration" class

Collaboration overview

```
Collaboration extends ModelElement;
```

Describes how *Instances* or *Roles* can co-operate with a *ModelElement* such as an *Operation* or a *UseCase*. A *Collaboration* describes how an *Operation* or a *Classifier*, like a *UseCase*, is realized by a set of *Classifiers* and *Associations*, used in a specific way. The *Collaboration* defines a context for performing tasks defined by *Interactions*. In the metamodel, a *Collaboration* contains a set of *ClassifierRoles* and *AssociationRoles*, which represent the *Classifiers* and *Associations* that take part in the realization of the associated *Classifier* or *Operation*. Example 20 presents a Collaboration diagram.

In Objecteering/UML, a collaboration belongs to a *UseCase*, a *Class*, a *Package* or an *Operation*.

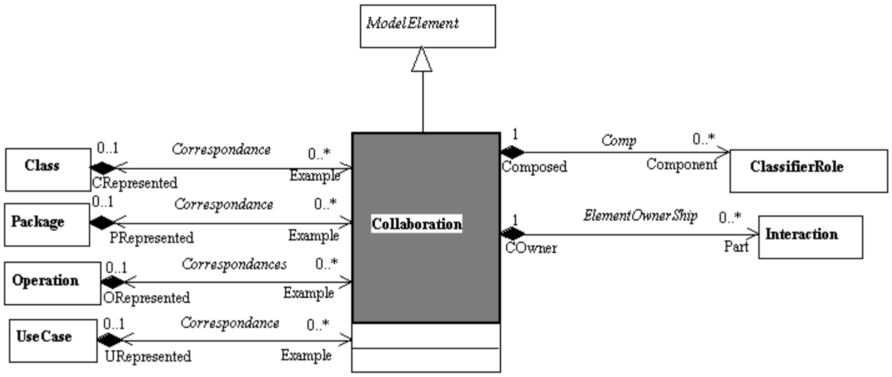


Figure 10-7. Detailed class diagram for *Collaboration*

Collaboration properties

The class has the following associations:

- *Component:ClassifierRole*: Determines the *ClassifierRoles* that belong to the *Collaboration*.
- *Part:Interaction*: Defines various *Interactions* that may exist between roles belonging to the current *NameSpace*.
- *CRepresented:Class*: *Class* being the context of the *Collaboration*
- *PRepresented:Package*: *Package* being the context of the *Collaboration*
- *ORepresented:Operation*: *Operation* being the context of the *Collaboration*
- *URepresented:UseCase*: *UseCase* being the context of the *Collaboration*

Collaboration consistency rules

- *Associations* towards *Class*, *Package, Operation* and *UseCase* are exclusive; one of them must be defined.
- The name of the *Collaboration* must be unique in its "*owner*".

Collaboration methods

```
Diagram[] getDiagrams()
```

Returns the *Collaboration* and Sequence diagrams of the *Collaboration*.

```
ClassifierRole[] getRoles()
```

Returns the *Roles* of the *Collaboration*.

```
SequenceMessage[] getSequenceMessages()
```

Returns the *Sequence Messages* of the *Collaboration*.

```
CollaborationMessage[] getCollaborationMessages()
```

Returns the *Collaboration Messages* of the *Collaboration*.

"Interaction" class

Interaction overview

```
Interaction extends ModelElement;
```

Message sequencing context. A *Message* has dependencies to successor or predecessor *Messages* and activators. It does not exist only in the context of its owner instance, and needs to be defined in an execution context called *Interaction*.

An *Interaction* belongs to a *Collaboration*, or to a *NameSpace*. In Objecteering/UML, an *Interaction* is closely linked to its representation diagram (*Sequence* or *Collaboration*). It is "*hidden*" to the end user, who sees only the diagram. The *Messages* of a diagram all belong to the same *Interaction*.

An *Interaction* belongs to a *Collaboration* or a *NameSpace*. (See Examples 19 and 20)

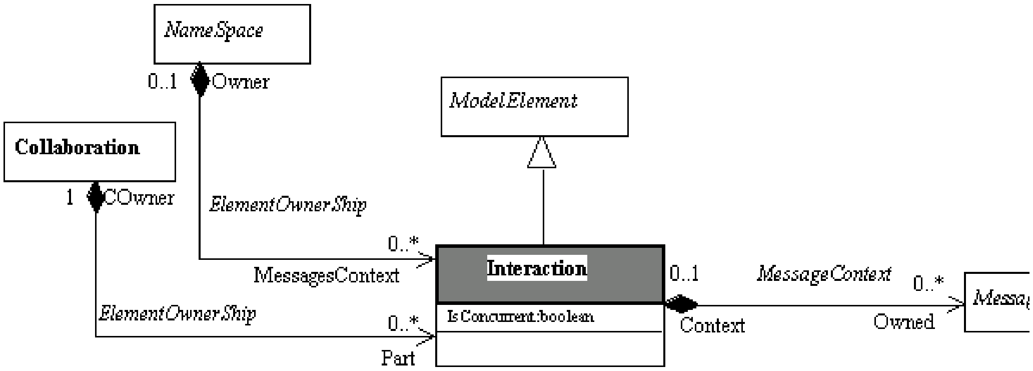


Figure 10-8. Detailed class diagram for *Interaction*

See also: *NameSpace, Collaboration*, *Message*, *Diagram*.

Interaction properties

The class has the following association:

- *Owned:Message*: *Messages* defined in the context of this *Interaction*.

The class owns the following attribute:

- *IsConcurrent*: Determines if the *Interaction* will be sequential or between concurrent objects.

Interaction consistency rules

- All the messages which make up the interaction must be of the same type, *CollaborationMessage* or *SequenceMessage*.
- *Associations* towards *Collaboration* and *NameSpace* are exclusive; one of these associations must be defined.

"ClassifierOccurence" class

ClassifierOccurence overview

```
ClassifierOccurence extends ModelElement;
```

Instance or *Role* which is an *Occurrence* of a *Classifier*. A *ClassifierOccurence* is an abstract class that factorizes the similarities between an *Instance* and a *Role*.

In Objecteering/UML, a *ClassifierOccurence* belongs to a *Collaboration* if it is a *ClassifierRole*, or to a *NameSpace* if it is an *Instance* (see Example 18 or 19).

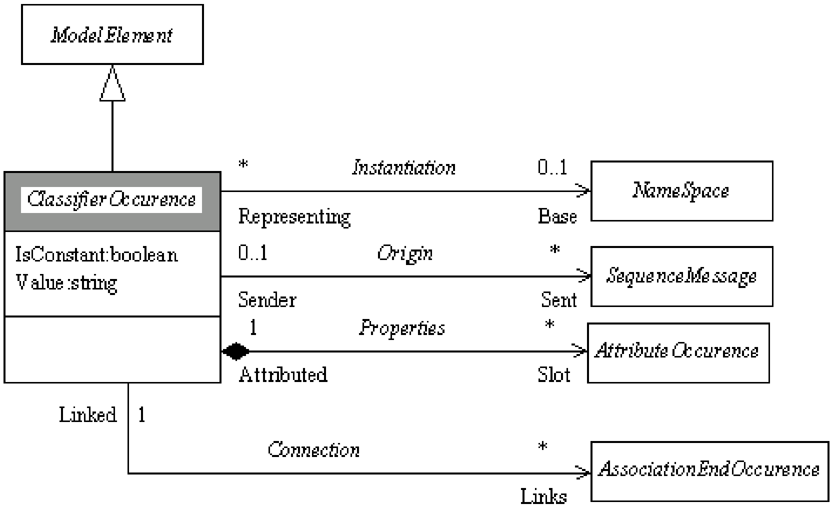


Figure 10-9. Detailed class diagram for *ClassifierOccurence*

<u>See also</u>: *Instance*, *ClassifierRole*.

ClassifierOccurence properties

The class has the following associations:

- *Base:NameSpace*: Defines the *NameSpace* as the model of the *Instance* or *Role*
- *Sent:SequenceMessage*: *Messages* sent by the *Occurrence*.
- *Slot:AttributeOccurence*: *Occurrences* of *Attributes* for the current object (*Instance* or *Role*)
- *Connection:AssociationEndOccurence*: *Links* connected to the current *ClassifierOccurence*.

The class owns the following attributes:

- *IsConstant*: Determines whether it is a constant.
- *Value*: Current value of the *Instance*.

ClassifierOccurence consistency rules

- Repetition of names is forbidden for all the *AttributeOccurences*.
- The *NameSpace* referenced by the *Instantiation* relationship must be accessible by the "*owner*" element of the *ClassifierOccurence*.
- A *ClassifierOccurence* must have a name, or the *instantiation* association must be defined.
- A *ClassifierOccurrence* can instantiate a package which is stereotyped <<sub-system>> and which is instantiable (in other words, a package which owns the *IsInstantiable* attribute).

"Instance" class

Instance overview

```
Instance extends ClassifierOccurence;
```

Entity to which a set of *Operations* can be applied and which has a state that stores the effects of *Operations*.

An *Instance* is connected to zero or one *Classifier*, which declares its structure and behavior. It has a set of attribute values and is connected to a set of *Links*, both sets matching the definitions of its *Classifier* (if there is one). The two sets implement the current state of the *Instance*. (See Example 18)

In Objecteering/UML, an *Instance* belongs to its *NameSpace*.

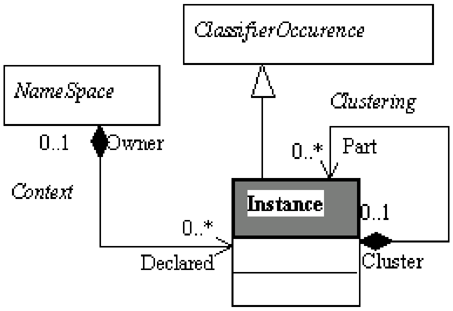


Figure 10-10. Detailed class diagram for *Instance*

<u>See also</u>: *Classifier*, *Link*, *LinkEnd*, *AttributeLink*, *Message*.

Instance properties

The class has the following association:

- *Part:Instance*: *Instances* can be embedded. This can represent cluster. In Objecteering/UML, this is used to represent *ComponentInstances* supported by *NodeInstances*, or *Objects* in *ComponentInstances* or *NodeInstances*.

Instance consistency rules

- The *ClassifierOccurence::Links* association must designate an instance of *LinkEnd*.
- The *ClassifierOccurence::Properties* association must designate an instance of *AttributeLink*.
- The *clustering* and *context* relationships are exclusive.

Instance methods

```
AttributeLink[] getAttributes()
```

Returns the occurrences of *Attribute* for the *Instance*

```
LinkEnd[] getLinkEnds()
```

Returns the *LinkEnds* connected to the *Instance*.

```
Classifier[] getType()
```

Returns the *Type* of the *Instance*.

"ClassifierRole" class

ClassifierRole overview

```
ClassifierRole extends ClassifierOccurence;
```

A *ClassifierRole* is a specific role played by a participant in a *Collaboration*. A *ClassifierRole* specifies a restricted view of a *Classifier* (a projection), defined by what is required in the *Collaboration*. (See Example 20 for Collaboration diagrams).

In Objecteering/UML, a *ClassifierRole* belongs to its *Collaboration*.

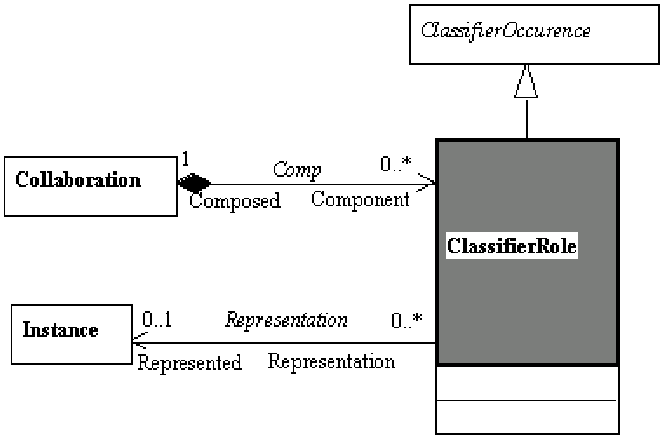


Figure 10-11. Detailed class diagram for *ClassifierRole*

<u>See also</u>: *Collaboration*

ClassifierRole properties

The class has the following association:

- *Represented:Instance*: *Instance* represented by the *Role*.

ClassifierRole consistency rules

The instance designated by the *Representation Association* must, if the case arises, respect the following accessibility rules:

- If the current *ClassifierRole* is defined in a *Collaboration* of *Class*, *Package* or *UseCase*, the accessibility rule is that of the *NameSpace* in which the *ClassifierRole*s "*owner*" *Collaboration* is defined.
- If the *ClassifierRole* is defined in a Collaboration *Operation* collaboration, the instances accessible are those of the *Operation's* “*owner*” *NameSpace*, and those of the classes used (use or parameters) by the *Operation*.
- The *ClassifierOccurence::Link* association must designate an instance of the *AssociationEndRole* class.
- The *ClassifierOccurence::Properties* association must designate an instance of the *AttributeRole* class.

ClassifierRole methods

```
Instance[] getInstance
```

Returns the *Instance* represented by the *Role*.

```
SequenceMessage[] getSentSequenceMessages()
```

Returns the *Messages* sent by the *Role*.

```
SequenceMessage[] getReceivedSequenceMessages()
```

Returns the *SequenceMessages* received by the *Role*.

```
Classifier[] getClassifierType()
```

Returns the *Classifier* which is the *Type* of the *Role*.

"AttributeOccurence" class

AttributeOccurence overview

```
abstract class AttributeOccurence extends ModelElement;
```

Named slot in an instance or role, which has the value of an attribute. Defines an attribute's value at instance level. This element appears in Object diagrams. (See Example 18 and 19)

In Objecteering/UML, an *AttributeOccurence* belongs to a *ClassifierOccurence*.

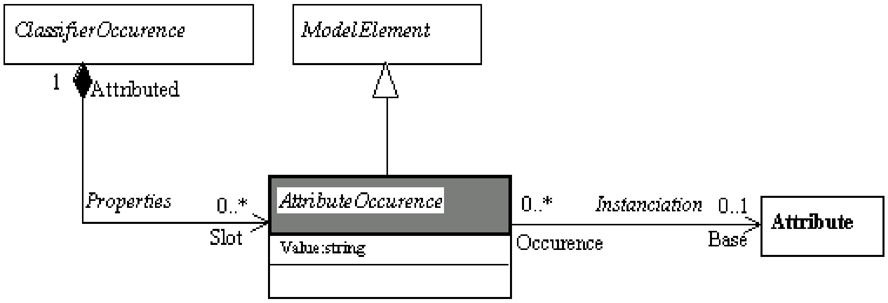


Figure 10-12. Detailed class diagram for *AttributeOccurence*

See also: *Attribute*.

AttributeOccurence properties

The class has the following association:

- *Base:Attribute*: Defines the optional *Attribute* that specifies the *AttributeLink*.

The class owns the following attribute:

- *Value*: Current value of the attribute's Slot for the *Instance*.

AttributeOccurence consistency rules

- If the *ClassifierOccurence* associated with the A*ttributeOccurence* is itself associated with a *Classifier*, the *Attribute* indicated by the *Instantiation* relationship must be an *Attribute* of *Classifier*.
- If the *Instantiation* association is defined, the name of the object must be the name of the attribute designated by the relationship.
- The name of the object must not be defined and must be unique in its *ClassifierOccurence*.

"AttributeLink" class

AttributeLink overview

```
AttributeLink extends AttributeOccurence;
```

Named slot in an *Instance*, which has the value of an *Attribute*. Defines an attribute's value at the *Instance* level. This element appears in Object diagrams. (See Example 18)

In Objecteering/UML, an *AttributeLink* belongs to an *Instance*.

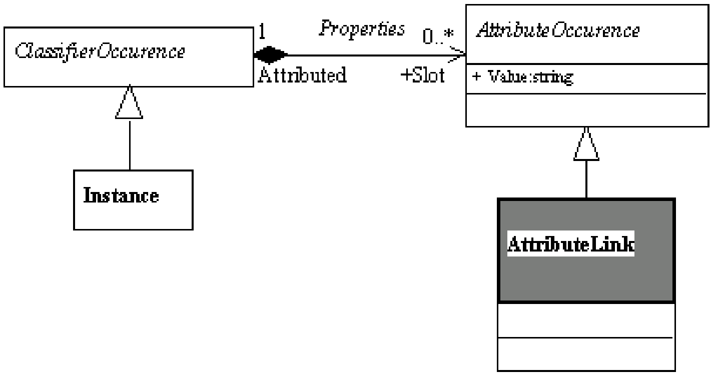


Figure 10-13. Detailed class diagram for *AttributeLink*

See also: *Instance*, *Attribute*.

AttributeLink consistency rules

- The *AttributeOccurence::Properties* association must only designate *Instances* of the *Instance* class.

"AttributeRole" class

AttributeRole overview

```
AttributeRole extends AttributeOccurence;
```

Named slot in a *ClassifierRole*, which has the value of an *Attribute*. Defines an attribute's value at *Collaboration* level. This element appears in *Collaboration* diagrams. (See Example 20)

In Objecteering/UML, an *AttributeRole* belongs to a *ClassifierRole*.

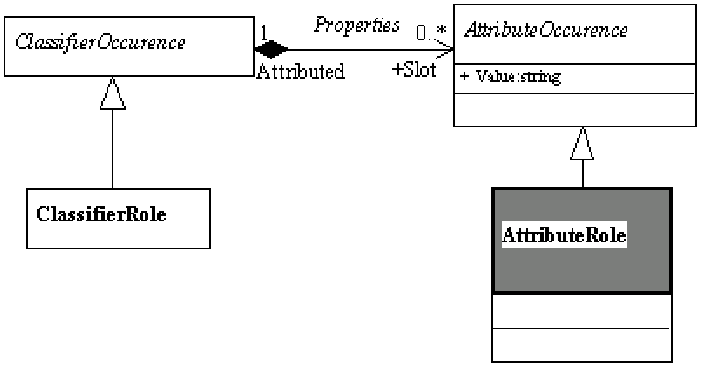


Figure 10-14. Detailed class diagram for *AttributeRole*

See also: *Attribute*, *ClassifierRole*.

AttributeRole consistency rules

The *AttributeOccurence::Properties* association must only designate instances of the *ClassifierRole* class.

"AssociationOccurence" class

AssociationOccurence overview

```
AssociationOccurrence extends ModelElement;
```

Occurence of an *Association*. Presented in a Collaboration diagram. In Collaboration diagrams, this element specifies links between objects. (See Example 18 and 19)

In Objecteering/UML, *AssociationOccurences* belong to no element.

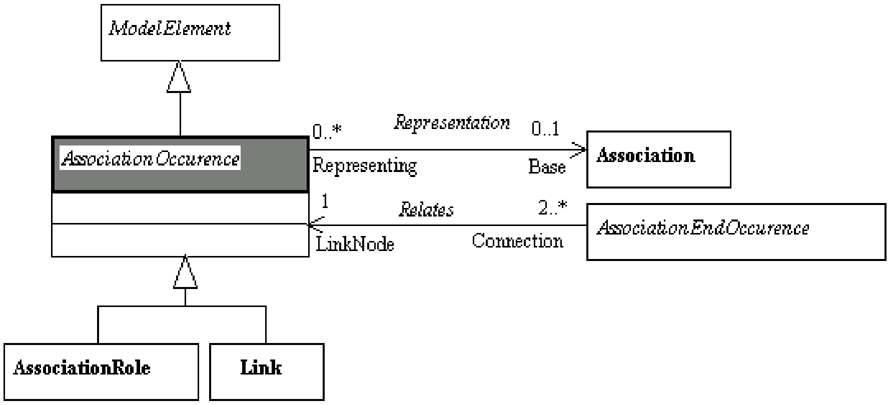


Figure 10-15. Detailed class diagram for *AssociationOccurence*

See also: *Association*, *AssociationEnd*, *AssociationRole*, *Link*.

AssociationOccurence properties

The class has the following association:

- *Base:Association*: Defines the *Association* that specifies this *AssociationOccurence*.

AssociationOccurence consistency rules

- If the *Representation* association is defined, then there must be consistency between the *AssociationEndOccurences* of the *AssociationOccurence* and the *AssociationEnds* of the *Association* represented.
- If the *Representation* association is defined, the name of the *AssociationOccurence* must be the name of the *Association*.

"Link" class

Link overview

```
Link extends AssociationOccurence;
```

Connection between *Instances*. A *Link* is an *Instance* of an *Association*. It has a set of *LinkEnds* that matches the set of *AssociationEnds* of the *Association*.

Example 18 presents *Links* in *Object diagrams*.

In Objecteering/UML, a *Link* belongs to no element.

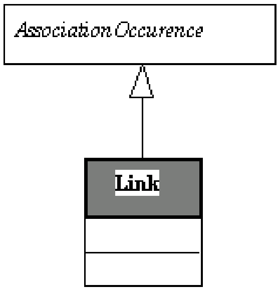


Figure 10-16. Detailed class diagram for *Link*

See also: *LinkEnd*, *Association*, *AssociationEnd*.

"AssociationRole" class

AssociationRole overview

```
AssociationRole extends AssociationOccurence;
```

Specific usage of an *Association* needed in a *Collaboration*. In Collaboration diagrams, this element specifies links between objects playing roles. (See Example 19)

In Objecteering/UML, an *AssociationRole* belongs to no element.

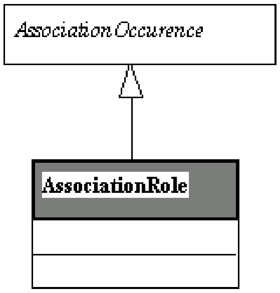


Figure 10-17. AssociationRole *Metamodel*

See also: *Association*, *AssociationEnd*, *AssociationEndRole*.

"AssociationEndOccurence" class

AssociationEndOccurence overview

```
AssociationEndOccurence extends ModelElement;
```

End point of a link. An *AssociationEndOccurence* is the part of a *Link* that connects to an object. It corresponds to an *AssociationEnd* of the Link's *Association*. (See Examples 18 and 19)

An *AssociationEndOccurence* belongs to an *AssociationOccurence*.

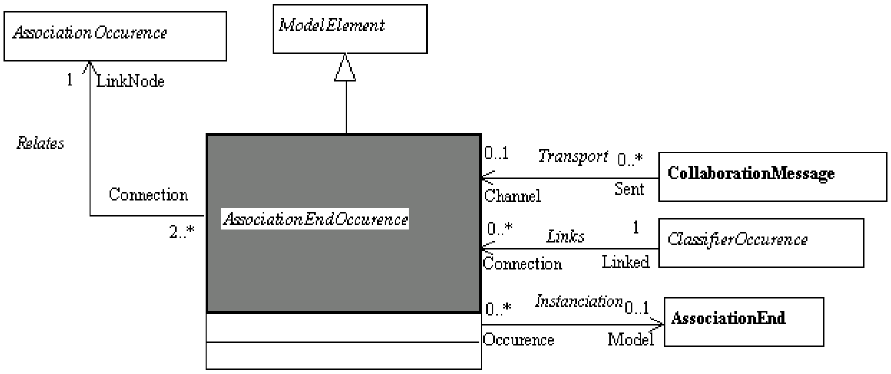


Figure 10-18. Detailed class diagram for *AssociationEndOccurence*

See also: *Association*, *AssociationEnd*, *LinkEnd*, *AssociationEndRole*.

AssociationEndOccurence properties

The class has the following associations:

- *Model:AssociationEnd*: The *AssociationEndOccurence* is an *Occurence* of this *AssociationEnd*.
- *LinkNode:AssociationOccurence*: *AssociationOccurence* ended by the current *AssociationEndOccurence*

AssociationEndOccurence consistency rules

- If the *Instantiation* association is defined, then the linked *AssociationOccurence* (*LinkNode*) must instantiate the association linked to the *AssociationEnd* which plays the role of *Model*.
- If the *Instantiation* association of the *ClassifierOccurence* referenced by the *Links* relationship is defined, and the *AssociationEndOccurence* references an *AssociationEnd*, then the model and its occurrences must correspond completely.

"LinkEnd" class

LinkEnd overview

```
LinkEnd extends AssociationEndOccurence;
```

End point of a *Link*. A *LinkEnd* is the part of a *Link* that connects to an *Instance*. It corresponds to an *AssociationEnd* of the *Link's Association*. (See Example 18)

A *LinkEnd* belongs to a *Link*.

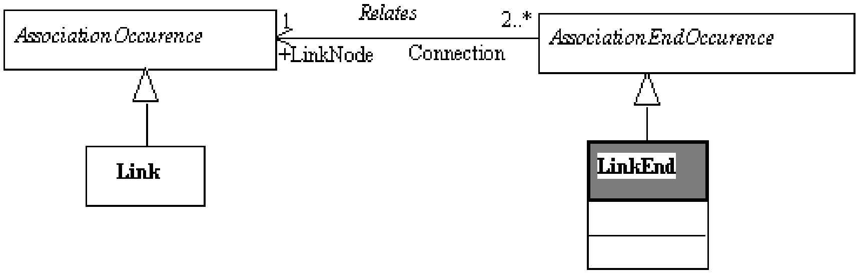


Figure 10-19. Detailed class diagram for *LinkEnd*

See also: *Link*, *AssociationEnd*, *Association*.

"AssociationEndRole" class

AssociationEndRole overview

```
AssociationEndRole extends AssociationEndOccurence;
```

Specifies an endpoint of an *Association* as used in a *Collaboration*. It is related to the target *ClassifierRole*, and to the instantiated *AssociationEnd*. (See Example 19)

In Objecteering/UML, an *AssociationEndRole* belongs to its *AssociationRole*.

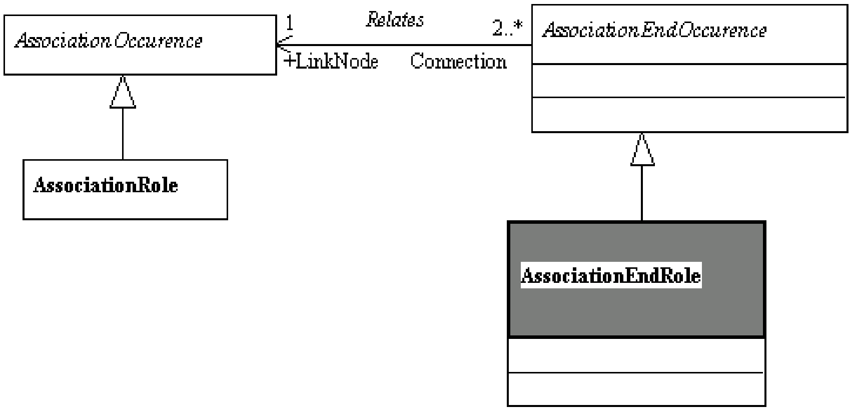


Figure 10-20. Detailed class diagram for *AssociationEndRole*

<u>See also</u>: *AssociationRole*, *AssociationEnd*, *Association*.

"Message" class

Message overview

```
Message extends ModelElement;
```

Occurrences of an *Operation*, processed by Instances, *Messages* are used in Object diagrams, Collaboration diagrams and Sequence diagrams. If the *Message* has no *InvokedOperations*, then its description is in its *Name*. A *Message* is defined in the context of an *Interaction*. The sequencing information has a meaning only in this context.

These different examples are presented in Example 19 and 20.

In Objecteering/UML, a *Message* belongs to an *Interaction*.

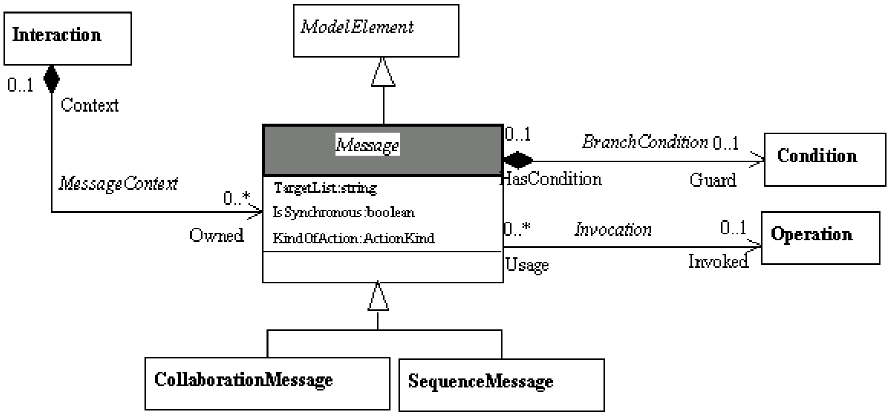


Figure 10-21. Detailed class diagram for *Message*

<u>See also</u>: *CollaborationMessage*, *SequenceMessage*, *Interaction*, *EnumeratedType*.

Message properties

The class has the following associations:

- *Invoked:Operation*: *Operation* that is invoked by the *Message*.
- *Guard:Condition*: *Condition* allowing branches to be defined between *Messages*.

The class owns the following attributes:

- *TargetList*: List of parameters passed, *Events* sent, etc.
- *IsSynchronous*: Determines if the *Message* is synchronous or not.
- *KindOfAction*: Useful for distinguishing create, destroy or return messages.

Message consistency rules

- If the *Invocation* association is defined, the name of the *Message* must be the name of the *Operation*.
- The *Invocation* association can only be defined if the type of the *Message* is *ActionCall*.

Message methods

```
Condition[] getGuardCondition()
```

Returns the *Condition* which allows the definition of branches between messages.

```
Operation[] getInvokedOperation()
```

Returns the *Operation* invoked by the message.

"CollaborationMessage" class

CollaborationMessage overview

```
CollaborationMessage extends Message;
```

Message used in *Collaboration* or Object diagrams. For the purposes of an *Object* or *Collaboration* diagram, *Messages* do not have precise sequencing information, as in Sequence diagrams, but instead have a direct connection to the transportation link. Therefore, a dedicated *Metaclass* has been defined.

CollaborationMessages belong to an interaction, just as *Messages* do (see Example 19)

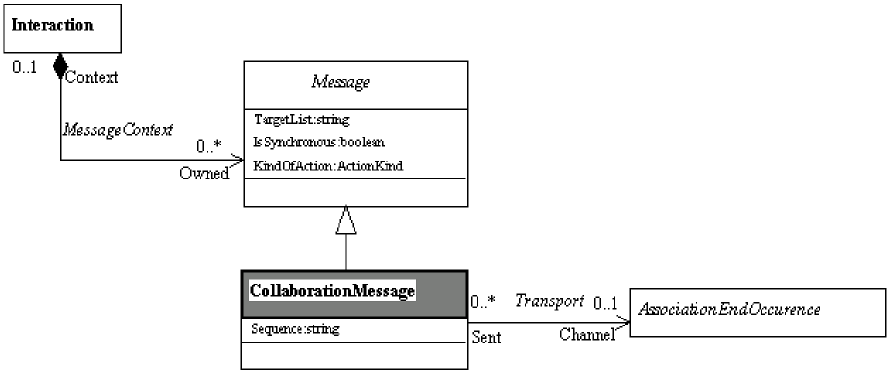


Figure 10-22. Detailed class diagram for *CollaborationMessage*

See also: *Interaction*.

CollaborationMessage properties

The class has the following association:

- *Channel:AssociationEndOccurence*: Defines the *Messages* that are carried by this link, in Collaboration diagrams. The *AssociationEndOccurence* provides the direction of the *Message* and designates the object destination of the *Message*. The opposite object is the origin of the *Message*.

The class owns the following attribute:

- *Sequence*: Sequence information used to explain parallelism.

"SequenceMessage" class

SequenceMessage overview

```
SequenceMessage extends Message;
```

Messages used for Sequence diagrams. *Messages* are specific to Sequence diagrams. There is a detailed description of the sequencing between *Messages* (see Example 19).

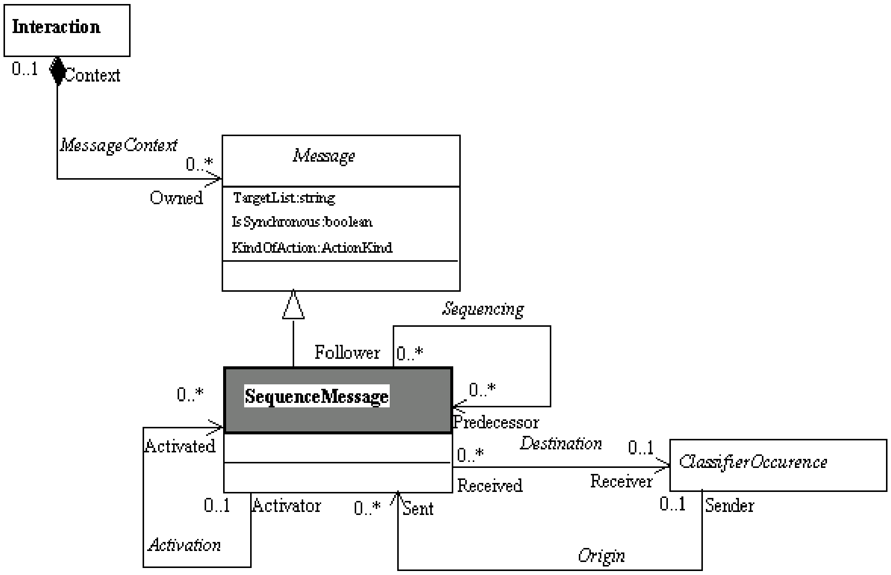


Figure 10-23. Detailed class diagram for *SequenceMessage*

SequenceMessage properties

The class has the following associations:

- *Receiver:ClassifierOccurence*: *Instance* receiving the *Message*.
- *Predecessor:SequenceMessage*: Provides the sequencing between the Messages.
- *Activated:SequenceMessage*: Determines the flow of control of messages. This means that the *ActivatorMessage* is activated by the current *Message*.

SequenceMessage consistency rules

- If the "*owner*" of a *SequenceMessage* is an *Interaction* of *NameSpace*, the *Destination* and *Origin* associations must designate *Instances*.
- If the "*owner*" of a *SequenceMessage* is an *Interaction* of *Collaboration*, the *Destination* and *Origin* associations must designate *ClassifierRoles*.

SequenceMessage methods

`ClassifierOccurence[] getReceiverClassifierOccurence()`
Returns the instance (*ClassifierOccurence*) which receives the *Message*.

`ClassifierOccurence[] getSentClassifierOccurence()`
Returns the instance (*ClassifierOccurence)* which sends the *Message*.

`SequenceMessage[] getActivatedMessages()`
Returns the activated *Messages*.

`SequenceMessage[] getPredecessorMessages()`
Returns the predecessor *Messages*.

Chapter 11: Implementation of the OMG 1.4 UML metamodel in Objecteering/UML

Why do differences exist

Table 1 expresses how the UML metaclasses defined by the OMG are implemented. Comments explain both why there are differences and expected evolutions.

The main reasons why there are differences are the following:

1 - There are implementation constraints, in particular as there is no multiple generalization and there are no class associations in the Objecteering/UML metamodel.

2 - There exist strict naming rules for a metamodel to be implemented by Objecteering/UML. For example, a role name must never be the same as a class name. There are frequent cases where UML names have to be adapted for that reason.

3 - There are “operability” constraints, in order to obtain models that can effectively be edited by the users. These constraints may for example relax some association cardinalities. Some associations need to be less generic, than those defined by the OMG metamodel, in order to provide a real type checking by the case tool, and accurate help during modeling.

4 - Some concepts are not implemented to the same level of detail, in order to avoid unnecessary information, that can be too burdensome to manage for the end user. This is typically the case of the “Action” metaclass.

5 - Some concepts are judged too new to be really stable, and are postponed for future release.

6 - Some notions are implemented differently.

However, the internal Objecteering/UML metamodel is very close to the OMG metamodel.

Implementation of the OMG UML metaclasses in Objecteering/UML

The ... OMG metaclass	is implemented as ...	comments
Abstraction	NO	Some subclasses are implemented more specifically, some are not.
Action	Action	Action is not developed in terms of the many OMG subclasses in Objecteering
ActionSequence	Action	There is no distinction in terms of classes, as well as the many other action subclasses
ActionState	ActionState	Same concept
ActivityGraph	ActivityGraph	Same concept
Actor	Actor	Same concept
AggregationKind	"AggregationKind"	Enumerated type
Argument	NO	The action notion is expressed through Objecteering/UML attributes
Association	Association	Same concept. Beware of the “*navigability*" management, that is differently managed in Objecteering/UML
AssociationClass	AssociationClass	Same concept. Associations are used instead of multiple inheritance
AssociationEnd	AssociationEnd	Same concept. Navigability is differently managed.
AssociationEndRole	AssociationEndRole	Same concept
AssociationRole	AssociationRole	Same concept
Attribute	Attribute	Same concept
AttributeLink	AttributeLink	Same concept
BehavioralFeature	See Operation, Feature	The properties are spread over Operation and Feature
Binding	NO	Tagged values on Use dependency provide predefined Objecteering/UML types.
Boolean	boolean	J type

The ... OMG metaclass	is implemented as ...	comments
CallEvent	Event, attribute “ Kind ”	Event is not decomposed into subclasses. However, it has a discriminator attribute.
CallState	ActionState (?)	No distinction has been reified by a sub-class.
ChangeableKind	“ KindOfAccess ” enumeration	Same concept
ChangeEvent	Event	Event is not decomposed into subclasses, but has a “ kind ” attribute for that purpose.
Class	Class	Same concept
Classifier	Classifier	Same concept
ClassifierInState	ClassifierInState	Same concept
ClassifierRole	ClassifierRole	Same concept
Collaboration	Collaboration	Same concept. Collaboration is merged with Interaction in Objecteering/UML
Comment	NO see Note	There is no use of this metaclass. The Objecteering/UML “ Note ” concept supports this notion.
Component	Component	Same concept
ComponentInstance	ComponentInstance	Same concept
CompositeState	State	Depending on the presence of sub-states, a state is composite or not
Constraint	Constraint	Same concept
DataType	DataType	Same concept
DataValue	String	This part is almost never used
Dependency	Use	More accurate and less generic than the Dependency class, the Use class provides close semantics. The *UseCase Dependency* class is another example.
Element	Element	Same concept. Objecteering/UML provides the *Identifier* notion (Universal identification mechanism) in addition.

The ... OMG metaclass	**is implemented as ...**	**comments**
ElementOwnership	ElementOwnership association	The association class is translated into an association. The attribute “ Visibility ” is reported on the target element.
ElementImport	Referencing association	This class is translated by an association. The consequence is that Objecteering/UML does not manage the name and visibility mechanism.
Enumeration	Enumeration	Same concept
EnumerationLiteral	EnumerationLiteral	Same concept
Event	Event	Same concept
Exception		
Expression	String attribute	Expressions, that have semantics which depend on the target language, are simply strings in Objecteering/UML.
Extend	Extend stereotype	Applied to *UseCaseDependency*
ExtensionPoint	NO	Same restriction.
Feature	Feature	Same concept
FinalState	State	Composite or final state or managed at the state level
Flow	No	No
GeneralizableElement	Merged with the “ NameSpace ” concept	NameSpace has the “ GeneralizableElement ” properties. Thus, single inheritance is preserved, and semantics are not disturbed.
Generalization	Generalization	Same concept
Guard	Condition	Same concept
Include	Include stereotype	Applied to *UseCaseDependency*
Instance	Instance	Same concept
Integer	integer	Applied to *UseCaseDependency*
Interaction	Interaction	Same concept

The ... OMG metaclass	**is implemented as ...**	**comments**
Interface	Class with the *Interface* stereotype	Having not defined a different metaclass makes it possible to the User to change the status "*Class*" vs "*Interface*" of an element during its existence.
Link	Link	Same concept
LinkEnd	LinkEnd	Same concept
LinkObject	No	Not handled
Location Reference	No	
Mapping	No	Never used
Message	Message	Same concept
MessageInstance	NO	Messages can be modeled in the collaboration diagrams
Method	Operation	Methods and operations are merged in the Objecteering/UML metamodel
Model	Project	A project is a model, plus its "*context*" made of the loaded modules, and the values of the module's parameters
ModelElement	ModelElement	Same concept
Multiplicity	two " string " attributes	the range is expressed by two strings that can be interpreted.
Name	string	name attributes are managed by the consistency mechanism of Objecteering/UML
Namespace	NameSpace	Same concept
Node	Node	Same concept
NodeInstance	NodeInstance	Same concept
Object	Instance	"*Instance*" has no subclasses. It encompasses the Object notion.
ObjectFlowState	ObjectFlowState	Same concept
Operation	Operation	Merged with the "*Method*" concept with Objecteering/UML

The ... OMG metaclass	is implemented as ...	comments
Package	Package	Same concept
Parameter	Parameter	Same concept
ParameterDirectionKind	PassingMode	These notions are very close "return" is in Objecteering/UML a specific association.
Partition	Partition	Same concept
Permission	Use	Correspond precisely to the *Use* semantics.
PresentationElement	ViewElement	Same concept. UML does not further define the graphical metamodel.
Primitive	Attribute “ IsPrimitive ” on classes	The basic types inside Objecteering/UML are represented as “*terminal*” types, not as metaclasses
ProgrammingLanguage Type	NO	Never used
Pseudostate	PseudoState	Same semantics
PseudostateKind	StateKind	Enumerated type
Reception	No	Merged with the Operation/Feature concepts
Relationship	No	Subclasses are implemented in a more concrete way.
Request	No	No identified interest
Signal	Signal	Same concept
SignalEvent	Event ; Kind attribute	The Objecteering/UML Event concept encompasses the Event subclass. The “*Kind*” attribute categorizes the different UML Event subclasses.
SimpleState	State	The State class
State	State	Same concept
StateMachine	StateMachine	Same concept
StateVertex	StateVertex	Same concept
Stereotype	Stereotype	Same concept. Objecteering/UML uses the profile concept as a structuring mechanism.

The ... OMG metaclass	is implemented as ...	comments
Stimulus	Message	0
String	string	H type
StructuralFeature	Attribute	StructuralFeature is only justified by some MOF (Meta Object Facilities - OMG standard) considerations. This intermediate abstract class has no interest for UML
Structure	NO	never used
StubState	PseudoState	This is just a specific kind of state.
SubmachineState	No	Embedded state is the only mechanism.
SubActivityState	SubActivityState	Same concept
Subsystem	Package	Package stereotyped <<sub-system>>.
SynchState	PseudoState	This is just a specific kind of state.
TaggedValue	TaggedValues	There are extensions to the UML "*TaggedValue*" concept. (Profile definition)
TemplateParameter	TemplateParameter	Same concept. Binding is realized through specific tagged values.
Transition	Transition	Same concept
UseCase	UseCase	Same concept.
UseCaseInstance	NO	Rarely used
VisibilityKind	VisibilityMode	Enumerated type

Correspondence between Objecteering/UML metaclasses and OMG UML metaclasses

The ... Objecteering/UML metaclass	**Represents in UML ...**	**comments ...**
ActionState	ActionState	Same concept
ActivityGraph	ActivityGraph	Same concept
ActivityState	Does not exist in UML	ActivityState is an abstract parent class of the ActionState and SubActivityState metaclasses.
Actor	Actor	Same concept
Association	Association	Same concept. In Objecteering/UML, associations may not be generalized.
AssociationEnd	AssociationEnd	Same concept. May be treated as a feature in Objecteering/UML, with a specific representation of navigability.
AssociationEndOccurence	AssociationEndRole LinkEnd	Abstraction of these more concrete concepts
AssociationEndRole	AssociationEndRole	Same concept.
AssociationOccurence	Link AssociationRole	Abstraction of these elements
AssociationRole	AssociationRole	Same concept. For reasons of heaviness, AssociationRole does not inherit Association, but is related to it.
Attribute	Attribute, Structural Feature	Same concept. “ StructuralFeature ” is not used
AttributeLink	AttributeLink	Same concept
AttributeOccurence	AttributeLink AttributeRole	Abstraction of these elements
AttributeRole	AttributeRole	Same concept
AttributeRole	AttributeRole	Same concept
Class	Class, Interface	Same concept

The ... Objecteering/UML metaclass	Represents in UML ...	comments ...
ClassAssociation	ClassAssociation	Same concept. Objecteering/UML represents it as being related to Class and Association, whereas the OMG metamodel represents it as specializing both metaclasses.
Classifier	Classifier	Same concept. Some properties are delegated to the “ GeneralClass ” metaclass in Objecteering/UML.
ClassifierOccurence	ClassifierRole Instance	Abstraction of these more concrete Instances.
ClassifierRole	ClassifierRole	Same concept
Collaboration	Collaboration	Same concept
CollaborationMessage	Stimuli	Same concept
Communication	Association	In Objecteering/UML, Communication links have a dedicated metaclass.
ComponenetInstance	ComponentInstance	Same conceppt.
Component	Component	Same concept
Condition	Guard	
Constraint	Constraint	Same concept
DataFlow	Does not exist	Extension to UML. Flow diagrams are provided by this feature
DataType	DataType	Same concept
Diagram	Does not exist	Diagram model is not specified in the UML standard
Element	Element	In Objecteering/UML, elements are identified, and in the space of a project
Enumeration	Enumeration	Same concept
EnumerationLitteral	EnumerationLitteral	Same concept
Event	Event	Same concept

The ... Objecteering/UML metaclass	**Represents in UML ...**	**comments ...**
Feature	Feature	Same concept, but covers *AssociationEnd* in Objecteering/UML
GeneralClass	Does not exist	This does not exist in UML. This abstract intermediary class is convenient for handling elaborated classifiers
Generalization	Generalization	Same concept. In Objecteering/UML, only *NameSpaces* can be generalized
Instance	Instance	Same concept
Interaction	Interaction	Same concept
InternalProduct	Does not exist	Specific to the tool facilities, not to the supported model
InternalTransition	Transition	Distinguishes the Transitions that are internal to a State
Link	Link	Same concept
LinkEnd	LinkEnd	Same concept
Message	Message	Same concept
ModelElement	ModelElement	Same concept
MpGenProduct	Does not exist	Specific to Objecteering/UML, and its generation capabilities (work products)
NameSpace	NameSpace	Same concept
Node	Node	Same concept
NodeInstance	NodeInstance	Same concept
Note	Does not exist. Kind of TaggedValue	Notes are supported in Objecteering/UML by a specific metaclass. Their definition is structured by Profiles
NoteType	Does not exist	Defines the permitted Notes, in a Profile

The ... Objecteering/UML metaclass	Represents in UML ...	comments ...
ObjectFlowState	ObjectFlowState	Same concept
Operation	Operation	In Objecteering/UML, Operations include the Method notion.
Package	Package	Same concept
Parameter	Parameter	Same concept
Partition	Partition	Same concept
Project	Model	Projects include the configuration definition (Modules, Parameter values, etc.)
PseudoState	PseudoState	Same Concept
Realization	Abstraction and *realize* stereotype	Same concept, more concrete in Objecteering/UML.
Sequence Message	Message	Message dedicated to sequence diagrams
Signal	Signal	Same concept
State	State	Same concept
StateMachine	StateMachine	Same concept
StateVertex	StateVertex	Same concept
Stereotype	Stereotype	Same concept
SubActivityState	SubActivityState	Same concept
TaggedValue	TaggedValues	Same concept, more developed in Objecteering/UML
TagParameter	Value of a UML Tag	Tagged Values are more sophisticated in Objecteering/UML
TagType	Does not exist	Tagged values are more developed in Objecteering/UML. In particular, they are defined by Profiles
TemplateParameter	TemplateParameter	Same concept
Transition	Transition	Same concept

The ... Objecteering/UML metaclass	Represents in UML ...	comments ...
Use	Permission dependency	Specific metaclass in Objecteering/UML for this kind of dependency.
UseCase	UseCase	Same concept
UseCaseDependency	Include and Extend	Same concept more represented as a dependency in Objecteering/UML
UsesInheritance	Association	Specific metaclass defined in Objecteering/UML for this purpose (dependencies among UseCases)
ViewBox	Does not exist	Diagrams are notes specified in UML
ViewElement	PresentationElement	No further specification in UML
ViewLink	Does not exist	Diagrams are not specified in UML

Index