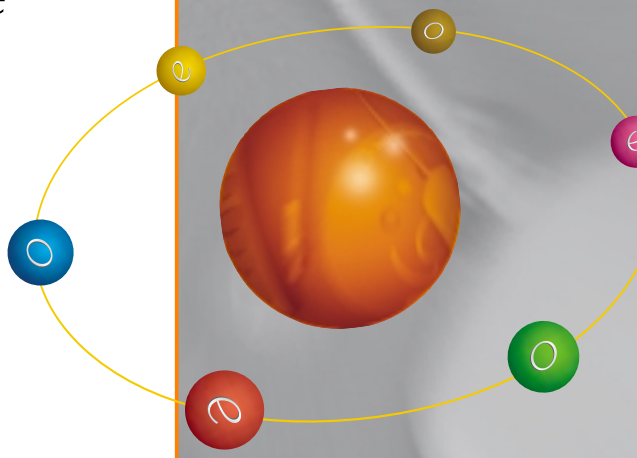
Objecteering/UML

Objecteering/Java Developer User Guide

Version 5.2.2



Objecteering

Software

www.objecteering.com

Taking object development one step further

Objecteering/UML version 5.2.2 - CODOBJ 001/002

Contents

Chapter 1: Introduction

Overview of the Objecteering/Java module

Introduction

Welcome to the *Objecteering/Java* user guide!

The *Objecteering/Java* module allows the generation of a Java application from an Objecteering/UML model, as well as its compilation and the generation of its documentation.

Its integrated design patterns are used to automate the designed model.

Furthermore, during the modeling phase, a reverse feature is provided to enable you to use classes in existing libraries, especially the JDK (Java Development Kit).

Module functions

The *Objecteering/Java* module groups together five main features:

- Java code generation
- generated code compilation
- Java documentation generation
- reversing of the existing libraries
- dynamic Java design patterns

Furthermore, powerful means of parameterization are provided using:

- the *Objecteering/Java* module parameters
- the generation document template accessible from Objecteering/UML Profile Builder
- the parameterization of the basic types and accessors

Adapting to your development environment

The *Objecteering/Java* module provides two functioning modes, to ensure consistency at all times between the model built in Objecteering/UML and the code produced:

- The *model driven* mode generates the entire Java application from the model and retrieves the code inserted using markers.
- The *round trip* mode combines code generation and reverse operations, retrieves the freely typed code and creates modeling elements when needed.

Objecteering/UML can be coupled to other integration development environments (for example, IBM's Visual Age), which can facilitate application development.

Tools used

The 2.1 version of the *Objecteering/Java* module is delivered with versions 1.1, 1.2, 1.3 and 1.4 of the JDK.

For Windows, the cygnus cyg-win 32 tools are also delivered. This is because it is possible, on this platform, to generate makefiles, either in a format which is understood by the Microsoft make tool, or in a format which is understood by the GNU make tool included in the cyg-win 32.

To obtain an Objecteering/UML database containing the entire reversed JDK 1.2, 1.3 or 1.4, please contact the Objecteering/UML technical support team (support@objecteering.com).

The properties editor for Java

The properties editor is essentially a window designed to aid the user in his modeling, by providing rapid access to various information and services he may need to use.

The properties editor contains a number of tabs, including a "*Java*" tab when the *Objecteering/Java* module has been selected for the current UML modeling project. This tab is used to:

- enter or modify certain information relevant to Java generation on the element selected in the explorer, such as notes, tagged values or stereotypes
- run generation, visualization and compilation operations

For further general information on the properties editor, please refer to the "*The Properties editor*" section in chapter 3 of the *Objecteering/UML Modeler* user guide.

For information on the "*Java*" tab of the properties editor, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Code generation

General remarks

This operation consists of producing Java files for a model's classes. Before actually generating, it may be necessary to annotate the model with tagged values and to fill in notes.

Generating types and accessors

The Java generator suggests a default translation of the types modeled in Java types. In the same way, it has a default strategy for generating parameter and attribute declarations, and for generating accessors. It is possible to choose other translations and other strategies without having to redefine the module, using the *Objecteering/UML Profile Builder* tool.

Main topics in this user guide

This user guide will explain:

- the correspondence between Java notions and modeling notions
- the note and tagged value types the module provides
- parameterization of the types of the parameters and attributes generated
- parameterization of the generated parameters and attributes declaration
- parameterization of the generated attributes accessors

The related *Objecteering/Design Patterns for Java* module automates the most important design patterns, in particular those described by E. Gamma.

Other Objecteering/Java features

Overview

In addition to code generation, the *Objecteering/Java* module provides related functions, which are the compiling of generated files, the production of documentation with the *javadoc* tool and a reverse which allows the existing libraries to be used during the modeling phase.

Compiling

This operation consists of producing the *class* files from generated *java* files. For this, the module suggests generating the makefile files, as well as running them from the tool.

Analyzing compilation

After a makefile has been run, the compilation analysis feature allows rapid access to the model elements which caused a compilation error.

Generating documentation

This operation uses the JDK *javadoc* tool to produce HTML files from the generated *java* files. This operation is therefore related to the generated *java* files. Special notes, inserted during the modeling phase and taken into account by the code generation, are then used.

Reverse engineering

This operation consists of incorporating classes of existing libraries, especially the JDK, into a model. This allows you, during the modeling phase, to:

- specialize these classes
- implement them
- create associations towards them
- create attributes which have one of these classes as type
- create method parameters which have one of these classes as type

This operation also allows you to reverse your developed code into Objecteering/UML modeling.

Presentation of Java Patterns

Presentation

Java Patterns provide solutions to problems which all Java developers have come across. In addition, the automation they provide also allows the risks of errors or omissions to be considerably reduced.

The use of patterns allows you to focus on the part of your application that is specific to your domain, and avoids having to deal with problems for which a proven generic solution exists, at a lower level.

Remote Method Invocation

Calling methods on distant objects is a powerful service provided by Java. Having to code not only the concrete class but also the interface requires having to do the same task twice, and entails further risk of errors.

These risks no longer exist with the RMI pattern. In addition, the RMI pattern is incremental, i.e. it can be applied each time the concrete class is modified.

Sending events

It often happens in a Java application that the user wants to trigger an event when the state of an object changes or when it is found in a specific state.

The "*Sending events*" pattern creates the event class and the interface for the objects that will listen. It also manages notifications.

Listening to events

It is almost impossible to write a Java application without listening to events, such as clicking on a button, selecting a text field or moving a gauge.

The "*Listening to events*" pattern allows you to find out when events will occur ("subscribing" to events), and to create internal classes which will manage the listening, as well as methods used to answer notifications.

Non-derivable method

The optimization problem is common to all developers and therefore common to all Java developers.

It is sometimes possible to help the virtual Java machine. For example, when a method is not redefined, it will run faster if it is specified as "*final*".

It is recommended that you carry out this optimization at the end of the development phase. However, it would be tiresome to have to go through all the methods of all the classes to check each time whether a method is redefined or not in a subclass.

The "*Non derivable method*" pattern allows the automation of this task without risks of errors.

Creating methods to be implemented

Launching this pattern on a class has the effect of recursively scanning the implemented interface classes collecting their methods. These methods are then created in the class. This process is carried out for each class of the package.

The processing only takes into account the modeled interfaces and not the {*JavaImplements*} tagged values.

Creating methods to be redefined

Launching this pattern on a class has the effect of going recursively through the parent classes collecting their abstract methods. These methods are then created in the class. This process is carried out for each class of the package.

Transforming a class into a non-primitive class

This pattern is especially useful for reversed classes. The reverse only produces primitive classes. If a user wishes to make a reversed class non-primitive, especially to create associations towards this class, he can use this pattern which will have the effect of transforming the attributes with this type into associations.

Transforming a class into a primitive class

This pattern is the same as the previous operation in the other direction. Its effect is to transform the associations towards this class into attributes.

Glossary

Java generation work product: Object that can be created on a package or a class and which possesses *Objecteering/Java* code generation and compilation features.

Type and accessor generation project: Project that contains information which allows the mapping of the Objecteering/UML types into Java types, and the generation of declarations and accessors for attributes, navigable associations and parameters. The advantage of this technique lies in the fact that the type package is separate from the generator itself.

External edition: Operation which allows the entry of notes with an editor other than the Objecteering/UML text editor. The texts entered between the pre-positioned markers are re-incorporated into the Objecteering/UML repository when the editing ends.

JDK (Java Development Kit): Library containing Java classes and tools, distributed by SUN for Java developers.

Properties editor: Tool which facilities the entry and/or modification of information, as well as code generation, visualization and compilation.

Chapter 2: Using the Objecteering/Java module

Using the Objecteering/Java module

Installation prerequisites

The *Objecteering/Java* module requires a valid license. Furthermore, the JDK 1.2 or later version must have been installed and the tools must be accessible.

Installing the Java module

The *Objecteering/Java* module is delivered to and installed on your Objecteering/UML site automatically during the Objecteering/UML installation procedure. For further information, please refer to chapter 2 of the *Objecteering/Introduction* user guide.

Note: The complete module delivery and installation procedure is fully explained in the "*Detailed view of the Configuration menu*" in chapter 3 of the *Objecteering/Administrating Objecteering Sites* user guide.

Selecting the Java module

The only operation the user has to carry out in order to be able to use the *Objecteering/Java* module is the actual selection of the module itself for the current UML modeling project. This operation is described in the "*Selecting modules in the current project*" section in chapter 3 of the *Objecteering/Introduction* user guide.

Please note that because installation of the *Objecteering/Java* module is carried out automatically during installation of the Objecteering/UML tool itself, module parameters are standard. To customize these module parameters (for example, the JDK path), simply edit the "*Modify module parameter configuration*" window. For further information on modifying module parameters, please refer to the "*Configuration window*" section in chapter 17 of this user guide.

The properties editor for Java

It should be noted that where a version of the *Objecteering/Java* module which adds a tab to the properties editor is selected in place of an earlier version of the module which did not provide this service, you should quit and restart Objecteering/UML, in order for the properties editor to be correctly displayed.

Selecting the Java model type

When creating a new UML modeling project, it is possible to select the "*DefaultJava*" UML model type in the "*UML model type*" field. By doing this, the *Objecteering/Java* module is automatically selected for use in your new UML modeling project.

The properties editor and the Java module

The "Java" tab of the properties editor on a package

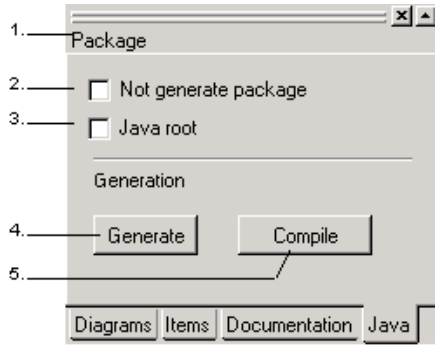


Figure 2-1. The "*Java*" tab of the properties editor on a package

Key:

1 - This gives the name of the package selected in the explorer.

2 - This field is used to add the *{JavaNoPackage}* tagged value.

3 - This field is used to add the *{JavaRoot}* tagged value.

4 - The "*Generate*" button is used to launch code generation on the selected package.

5 - The "*Compile*" button is used to launch compilation.

If the package is the root package, or is tagged {*JavaRoot*}, an additional button, "*Generate documentation*", appears. This button is used to launch Javadoc generation on the package.

Note: For generation and compilation operations, a Java generation work product must exist for the element on which generation and/or compilation operations are to be run. If this is not the case when you click on the "*Generate*" button in the properties editor, the "*Java generation work product*" dialog box appears, allowing you to create a work product there and then.

The "Java" tab of the properties editor on a class

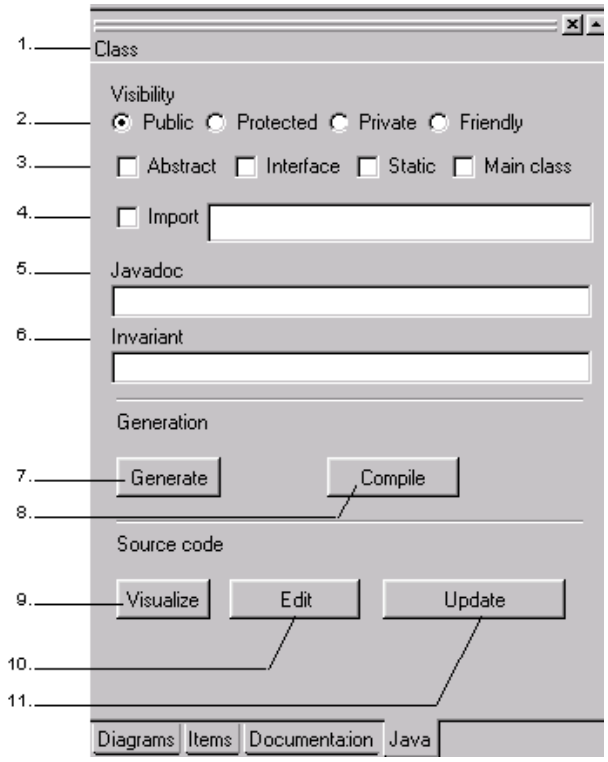


Figure 2-2. The "*Java*" tab of the properties editor on a class

Key:

1 - This gives the name of the class selected in the explorer.

2 - The "*Visibility*" radio buttons are used to select the visibility of the class.

3 - The tickboxes which appear here are used to indicate the nature of the class:

- "*Abstract*": This means that the class is abstract.
- "*Interface*": This adds the <<*interface*>> stereotype to the class.
- "*Static*": This annotates the class using the *{JavaStatic}* tagged value.
- "*Main class*": This indicates that the class should contain a main method. When you check this tickbox, a "public static void main (String[] argu)" is added to the class. The user must define the contents of this method.

4 - When the "*Import*" tickbox is checked, you can specify what you wish to import in the field to the right of the tickbox. If you wish to specify several elements, separate them using a comma.

5 - The "*Javadoc*" field is used to enter "*Javadoc*" type notes for the class in question.

6 - The "*Invariant*" field is used to enter invariants for the class in question.

7 - The "*Generate*" button is used to launch code generation on the selected class.

8 - The "*Compile*" button is used to launch compilation.

9 - The "*Visualize*" button is used to visualize the newly generated code.

10 - The "*Edit*" button is used to open a code edition window, in which you can edit your code.

11 - The "*Update*" button is used to update your generated code.

Note 1: When the selected class is an inner class in the modeling, a tickbox "*InnerBox*", figures in the properties editor. If this tickbox is not checked, the class is generated as a non-public class for its container class.

Note 2: When the "*Main class*" tickbox is checked, an additional button, "*Run application*" appears. This button is used to run the application. When you click on this button, a dialog box appears, inviting you to enter your application parameters. Enter and confirm to run the application.

The "Java" tab of the properties editor on an operation

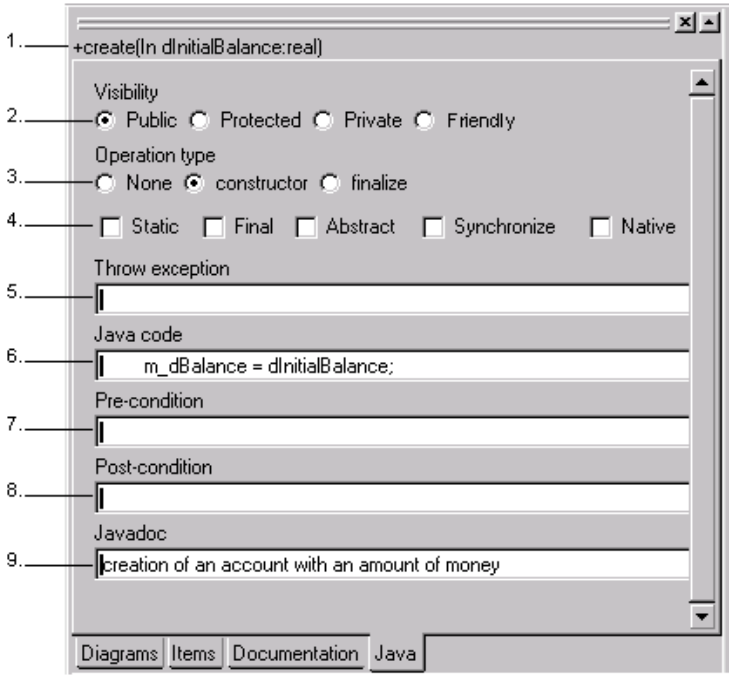


Figure 2-3. The "*Java*" tab of the properties editor on an operation

Key:

1 - This gives the name of the operation selected in the explorer.

2 - The "*Visibility*" buttons are used to select the visibility of the operation.

3 - The "*Operation type*" buttons which appear here are used to indicate the operation type:

- "*None*": This means that the operation has no special type.
- "*constructor*": This adds the <<*create*>> stereotype to the operation, and means that it is a constructor of the class. The generator ignores the modeling name of the operation.
- "*finalize*": This adds the <<*destroy*>> stereotype to the operation, and indicates that the name of the operation is "finalize".

4 - The tickboxes which appear here are used to indicate the nature of the operation:

- "*Static*": This indicates that the operation is "*class*", in other words, shared by all its instances.
- "*Final*": This adds the *{JavaFinal}* tagged value to the operation.
- "*Abstract*": This indicates that the operation is abstract.
- "*Synchronize*": This adds the *{JavaSynchronize}* tagged value to the operation.
- "*Native*": This adds the *{JavaNative}* tagged value to the operation.

5 - The "*Throw exception*" field is used to add the *{JavaThrownException}* tagged value to an operation.

6 - The "*Java code*" field is used to enter or modify the actual Java code of the operation.

7 - The "*Pre-condition*" field is used to enter a constraint stereotyped <<JavaPreCondition>>.

8 - The "*Post-condition*" field is used to enter a constraint stereotyped <<JavaPostCondition>>.

9 - The "*Javadoc*" field is used to enter "*Javadoc*" type notes for the operation in question.

The "Java" tab of the properties editor on an attribute

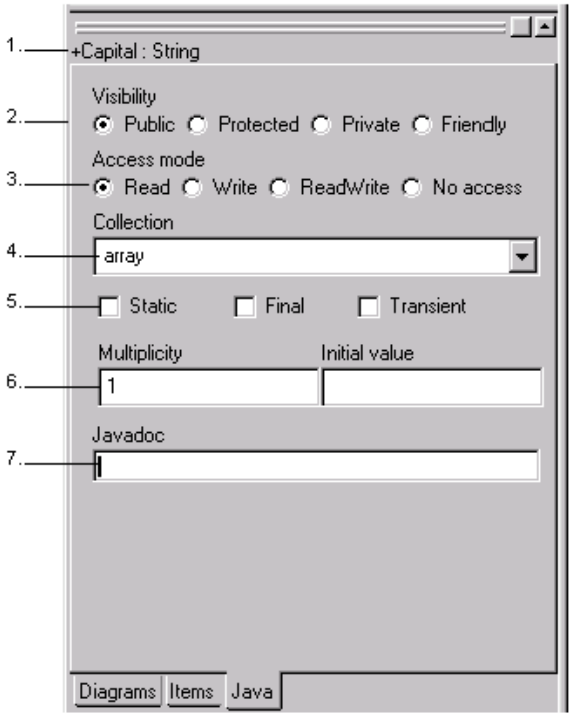


Figure 2-4. The "*Java*" tab of the properties editor on an attribute

Key:

1 - This gives the name of the attribute selected in the explorer.

2 - These buttons are used to select the visibility of the attribute.

3 - The buttons which appear here are used to indicate the attribute's access mode.

4 - The "*Collection*" field is used to add {type(x)}, where x can be one of a group of strings in a list.

5 - The tickboxes which appear here are used to indicate the nature of the attribute:

- "*Static*": This indicates that the attribute is static.
- "*Final*": This adds the *{JavaFinal}* tagged value to the attribute.
- "*Transient*": This adds the *{JavaTransient}* tagged value to the attribute.

6 - The "*Multiplicity*" field is used to enter the attibute's multiplicity, whilst the "*Initial value*" field is used to give its initial value.

7 - The "*Javadoc*" field is used to enter "*Javadoc*" type notes for the attribute in question.

Note: When the attribute is of base type, such as real, int, etc, additional, explicit tickboxes appear.

The "Java" tab of the properties editor on an association

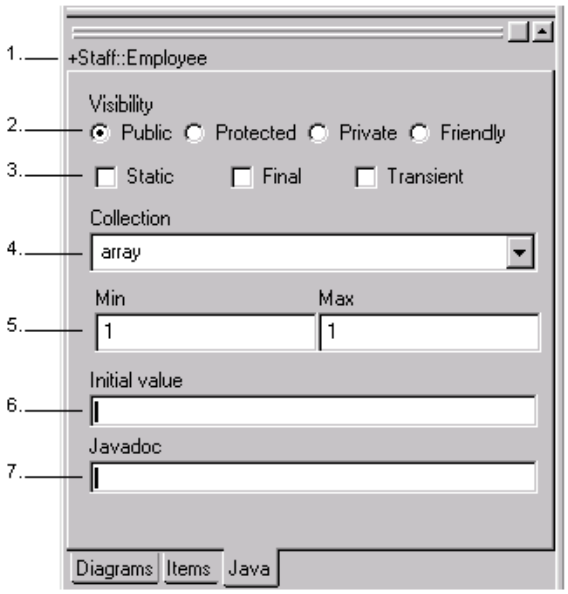


Figure 2-5. The "*Java*" tab of the properties editor on an association

Key:

1 - This gives the name of the association selected in the explorer.

2 - These buttons are used to select the visibility of the association.

3 - The tickboxes which appear here are used to indicate the nature of the association:

- "*Static*": This adds the *{IsClass}* tagged value to the association.
- "*Final*": This adds the *{JavaFinal}* tagged value to the association.
- "*Transient*": This adds the *{JavaTransient}* tagged value to the association.

4 - The "*Collection*" field is used to add {type(x)}, where x can be one of a group of strings in a list.

5 - The "*Min*" and "*Max*" fields are used to enter the association's multiplicity.

6 - The "*Initial value*" field is used to enter a "*JavaInitValue*" note for the selected association.

7 - The "*Javadoc*" field is used to enter "*Javadoc*" type notes for the association in question.

The "Java" tab of the properties editor on a parameter

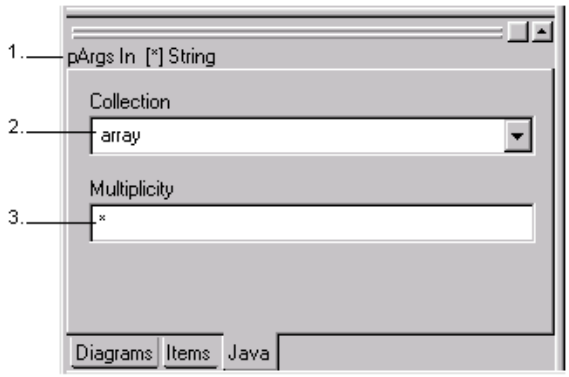


Figure 2-6. The "*Java*" tab of the properties editor on a parameter

Key:

1 - This gives the name of the parameter selected in the explorer.

2 - The "*Collection*" field is used to add the {type(x)}.

3 - The "*Multiplicity*" field is used to define the multiplicity of the parameter.

Chapter 3: First Steps - Java

Java First Steps - Introduction

General remarks

By following an example of a UML modeling project, you are now going to discover step by step the different features of the *Objecteering/Java* module.

Sources

This example is a simple bank account application, extracted from *Learn Java Now,* by *Stephen R.Davis, MicrosoftPress*.

Getting the example model

The first operation we shall look at is how to retrieve the example model (as shown in Figure 3-1).

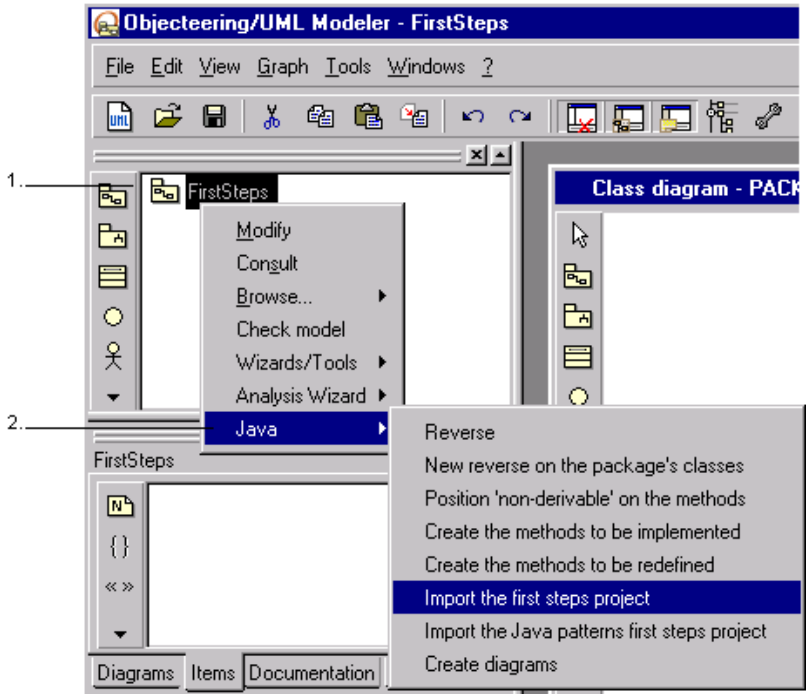


Figure 3-1. Importing the "*First Steps*" project

Steps:

1 - Run *Objecteering/UML Modeler*, and click on the root package using the right mouse-button.

2 - Select the "*Java/Import the first steps project*" commands from the context menu which then appears.

Creating a Java generation work product

Overview

In Objecteering/UML, the Java generation work product provides the commands for generating code, compiling and generating documentation. It can also be used to administrate the files which have been produced. Thus, if you destroy the generation work product, you will also destroy the files produced.

Note: In the "*Java*" tab of the properties editor, code generation and compilation can be directly launched, by clicking on the relevant buttons. When these commands are run via the properties editor and a generation work product does not already exist on the selected element, the "*Java generation work product*" dialog box is automatically opened, to facilitate generation work product creation.

Creating a Java generation work product

Using this example (shown in Figure 3-2), we are going to create a Java generation work product for the "*FirstSteps*" package.

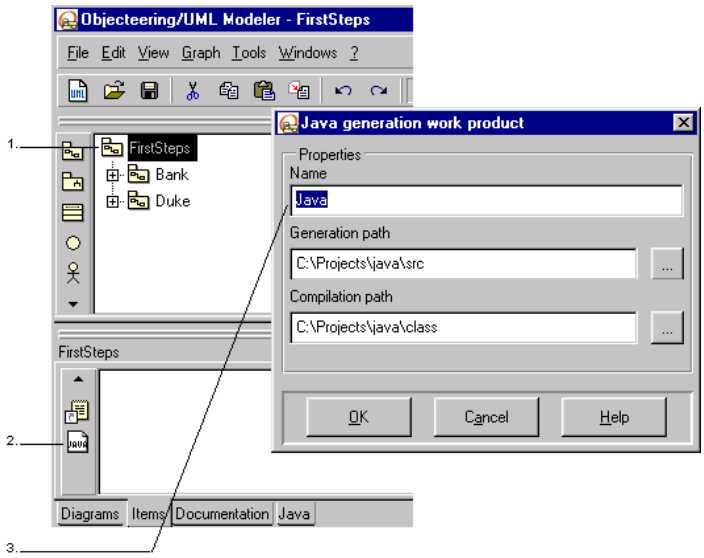


Figure 3-2. Creating a Java generation work product

Steps:

1 - Select the package in the explorer.

2 - Click on the  "*Java generation work product*" button in the "*Items*" tab of the properties editor.

3 - Enter the necessary information and confirm by clicking on the "*OK*" button. The newly created work product will then appear in the "*Items*" tab of the properties editor.

Note: Java generation work products can also created for elements other than the root package. A work product can be created for each class or package.

The propagation of a generation work product

When a Java generation work product on a package or a class propagates, a Java code generation work product is created on each package and class which features at a lower level in the tree structure.

In our example (Figure 3-3), we can make the work product created for the "*FirstSteps*" package propagate the "*Bank*" package and all the classes of this package.

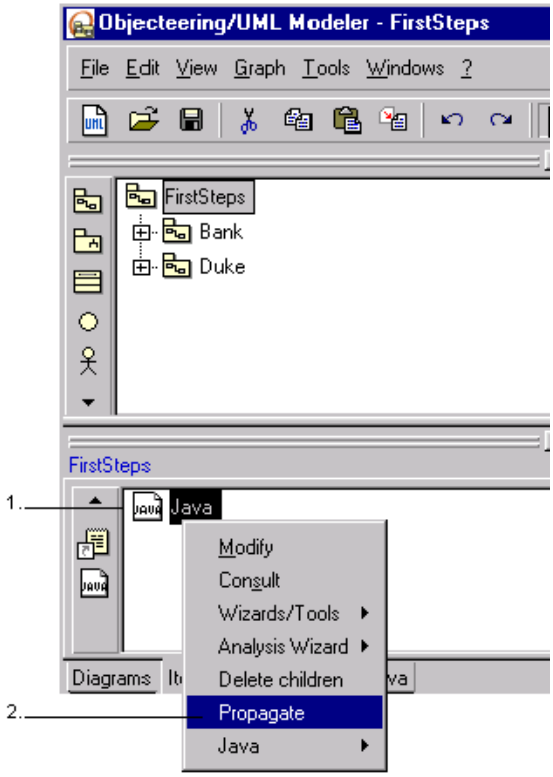


Figure 3-3. Propagating a generation work product

Steps:

1 - Select the generation work product with the right mouse-button.

2 - Choose *"Propagate*" in the context menu. A message in the console informs you that propagation has been successfully carried out.

Java generation work product menus

Overview

The Java generation work product provides all the commands used to generate Java code, to compile and to generate documentation (see Figure 3-4).

The "*Java*" tab of the properties editor also provides the user with easy access to certain commands. In this first steps modeling project, code generation, visualization and compilation operations are going to be run directly from the "*Java*" tab of the properties editor. Subsequent operations will then be run from the context menu on a work product.

Java generation work product commands

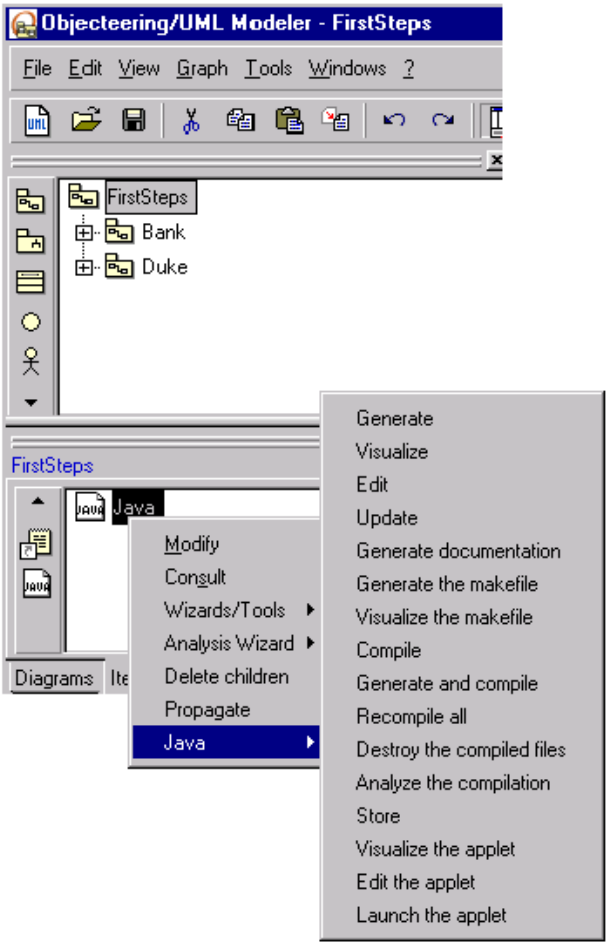


Figure 3-4. Generation work product services

All the commands for editing and generating Java are grouped together in the "*Java*" menu. These commands are detailed in the following table.

The ... menu	**is used to ...**
Generate	generate the Java code for a class (when the work product is linked to a class) or for a set of classes (when the work product is linked to a package).
Visualize	visualize the generated Java code. This action is only carried out for work products linked to a class.
Edit	edit the generated code. The editor used is the one defined by the parameter : "Command invoking the external editor" in the Java generation module. This action is only carried out for work products linked to a class.
Update	incorporate modifications made from outside the tool to the generated files in the Objecteering/UML repository.
Generate documentation	generate an HTML documentation using the "javadoc" tool.
Generate the makefile	produce the makefile file which allows the production of *.class* files from the generated *.java* files.
Visualize the makefile	visualize the generated makefile file
Compile	produce the *.class* files from the *.java* files generated since the last compilation.
Generate and compile	generate the Java code and produce the *.class* files from the *.java* files generated since the last compilation.
Recompile all	produce the *.class* files from the generated *.java* files even if they have not been re-generated since the last compilation.
Destroy the compiled files	destroy the *.class* files
Analyze the compilation	open the window for analyzing the results linked to the last compilation.
Store	build a *.jar* containing the *.class* of a package or the *.class* of a class as well as the files indicated by the "*JavaBeanResource"* tagged values
Visualize the applet	visualize the HTML code of the file for launching the applet defined by the class to which the work product is linked.
Edit the applet	edit the HTML code of the file for launching the applet defined by the class to which the work product is linked.
Launch the applet	launch the applet defined by the class to which the work product is linked.

Note: The "*Visualize*", "*Edit*" and "*Launch the applet*" commands have no impact on a work product that is not linked to a class which generalizes the "*Applet*" class.

Java code generation

Launching Java code generation

A *.java* type file is generated for each class of the model for which the "*Generate*" command is run.

However, it is possible to generate code from a package. A *.java* file is then generated for each of the package's classes.

No code is generated for elements annotated using the *{JavaExtern}* tagged value (as is the case, for example, for an element produced by the reverse of an *Objecteering/Java* module) or the *{nocode}* tagged value. This also applies to classes annotated using the *{JavaNonPublic}* tagged value or to inner classes. The code of these classes is to be incorporated in the .java of the class which contains them.

To generate code, carry out the steps shown in Figure 3-5.

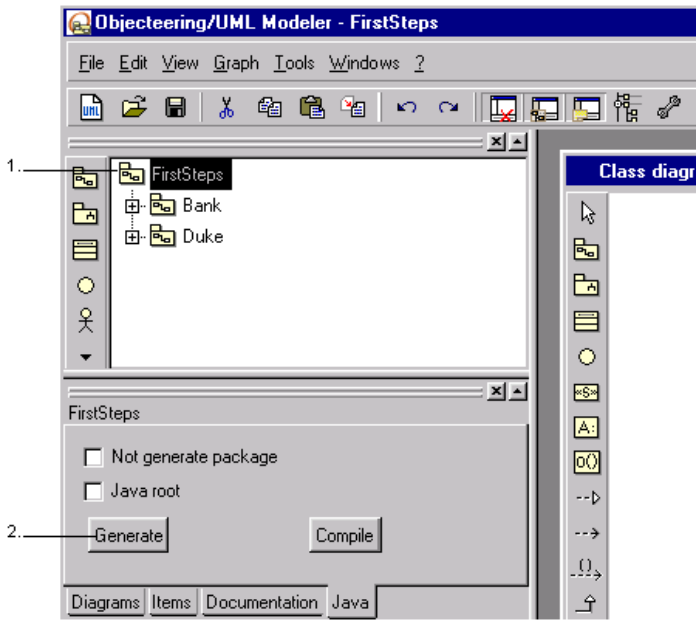


Figure 3-5. Running Java code generation

Steps:

1 - Select the "*FirstSteps*" package in the explorer.

2 - In the "*Java*" tab of the properties editor, click on the "*Generate*" button. In this case, a Java generation work product already exists for the "*FirstSteps*" package. However, if this had not been the case, the "*Java generation work product*" dialog box would have been opened automatically, thus allowing you to easily create the work product.

Note: Exactly the same operation can be carried out by right-clicking on the "*FirstSteps_Java*" work product in the "*Items*" tab of the properties editor, and then running the "*Java/Generate*" commands from the context menu which appears.

You can monitor the progress of the code generation in the console (as shown in Figure 3-6).

Figure 3-6. Monitoring the progress of code generation in the console

Note: You can go up and down in the console by using the scroll bars.

Visualizing the generated Java code

After code generation, you can visualize the generated Java code. This command can only be launched on a work product to which a *.java* file is associated, i.e. on a work product of a class for which the code has been generated.

To visualize generated Java code, carry out the steps shown in Figure 3-7.

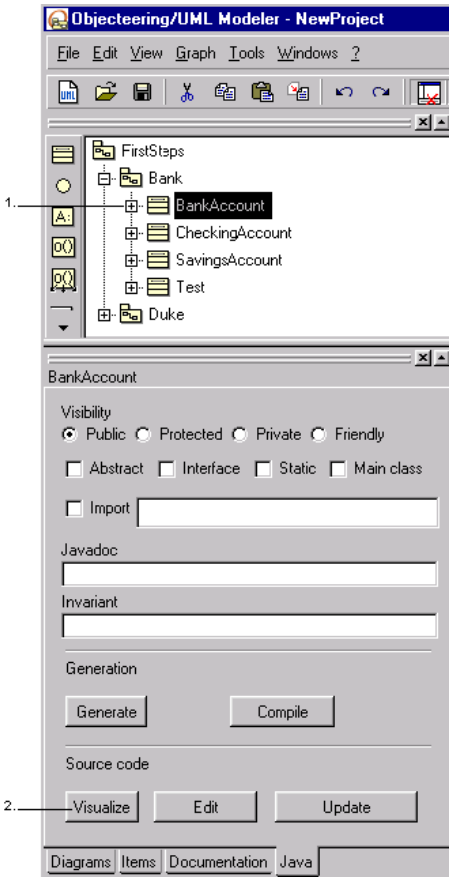


Figure 3-7. Running the "*Visualize*" command.

Steps:

1 - Select the "*BankAccount*" class in the explorer.

2 - In the "*Java*" tab of the properties editor, click on the "*Visualize*" button.

Note: Exactly the same operation can be carried out by right-clicking on the "*BankAccount_Java*" work product in the "*Items*" tab of the properties editor, and running the "*Java/Visualize*" commands from the context menu which then appears.

This command opens a window containing the generated Java code (shown in Figure 3-8). It is not possible to modify the code directly in this window.

It is possible, however, to modify certain elements directly from the model. The elements you can modify are highlighted in blue. Double-click on the blue text to open the dialog box of the model element which allows this text to be typed.

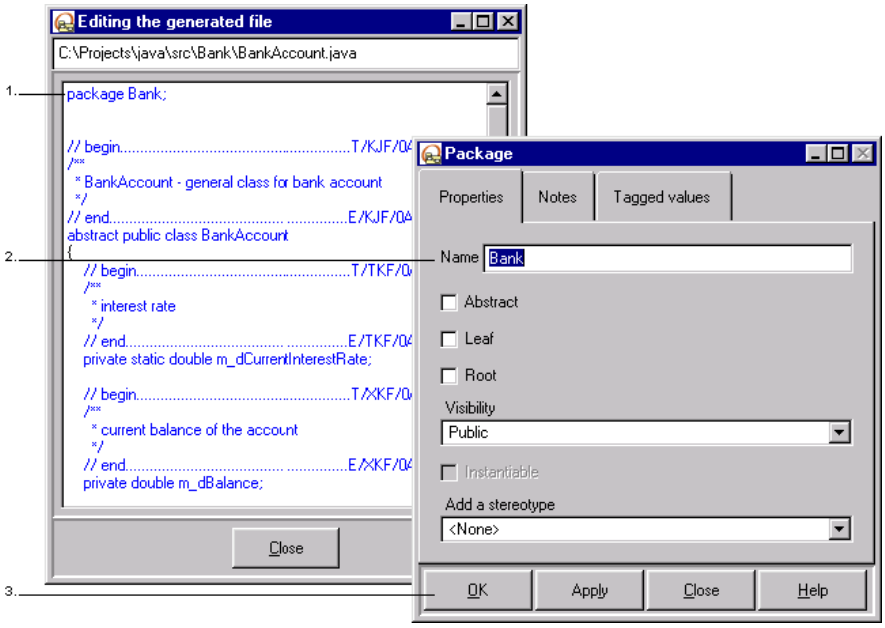


Figure 3-8. Window displaying the Java code of the "*BankAccount*" class

Steps:

1 - Double-click on the blue "*Bank*" text. The dialog box of the "*Bank*" package will open.

2 - Make the necessary modifications.

3 - Confirm by clicking on the "*OK*" button.

Editing the generated code

It is also possible to edit the generated Java code for a class using the editor chosen ("*Command for invoking the external editor*" parameter of the "*External edition*" group).

It is possible to modify zones represented between markers. Modifications are directly incorporated into the model when the editor closes.

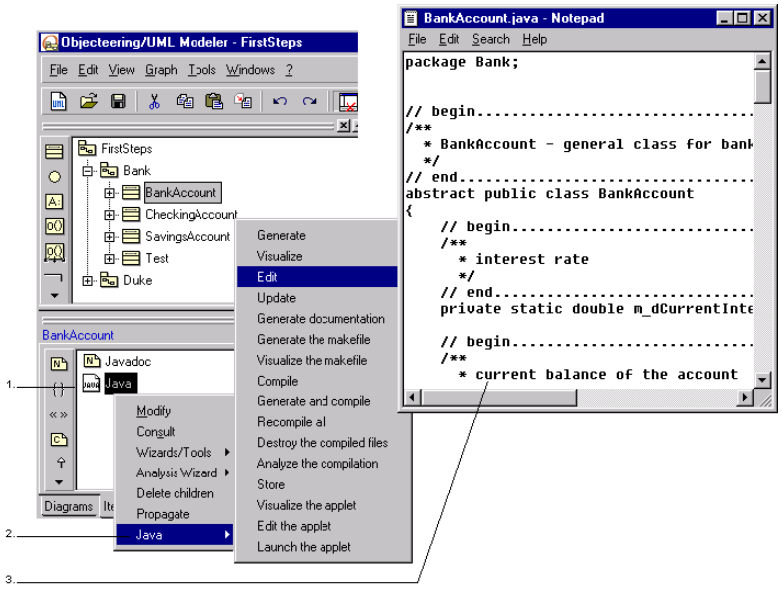


Figure 3-9. Editing the generated code using an external editor

Steps:

1 - Select the generation work product in the "*Items*" tab of the properties editor, using the right mouse button.

2 - Run the "*Java/Edit*" commands from the context menu which then appears.

3 - The code is then edited in the external editor specified at module parameter configuration level.

Updating the model

The Java code associated to a model's class can also be modified completely outside Objecteering/UML. It will be possible to update the model, in order to incorporate those modifications made externally.

1 - Modify the "*CheckingAccount.java*" class outside Objecteering/UML.

2 - Run the "*Java/Update*" command from the work product of the "*CheckingAccount*" class.

3 - Make sure the zone of text has been modified in Objecteering/UML.

In round trip mode, the "*Update*" mode performs the same functions as in model driven mode, i.e. the update of the model from the code contained between the markers. Any modifications carried out outside the marked zones will not be taken into account by this command. In this case, the "*New Reverse*" command should be used after the compilation of the model.

Compilation

Generating the makefile

Generating a makefile from a package's generation work product produces a makefile which allows the recursive compilation of all the classes of this package.

From the Java generation work product, we can generate the makefile necessary to compiling a class or a set of classes (Figure 3-10). The console will display the generation steps.

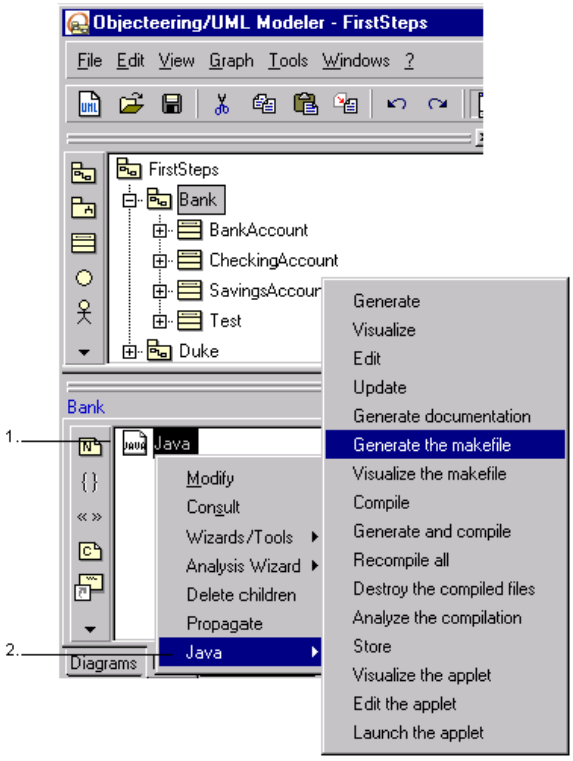


Figure 3-10. Generating the makefile

Steps:

1 - Select the Java generation work product for the "*Bank*" package in the "*Items*" tab of the properties editor, by right-clicking.

2 - Run the "*Java/Generate the makefile*" command from the context menu which then appears.

Visualizing the makefile

It is possible to visualize a makefile that has previously been generated (as shown in Figure 3-11). This command opens a window containing the generated makefile code.

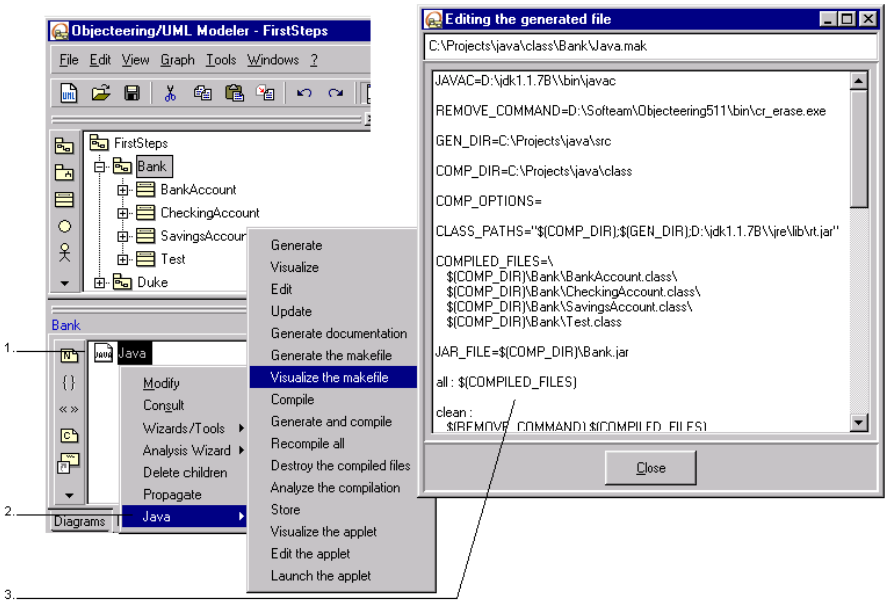


Figure 3-11. Visualizing the makefile of the "*Bank*" package

Steps:

1 - Select the generation work product of the "*Bank*" package in the *Items*" tab of the properties editor by right-clicking.

2 - Run the "*Java/Visualize the makefile*" command from the context menu which then appears.

3 - The makefile is then displayed.

Compilation

The *"Compile" command* in the context menu of a generation work product is used to compile all the files of a class or a package using the makefile produced (Figure 3-12). A file containing the compilation results is then produced. This file is used to analyze the compilation.

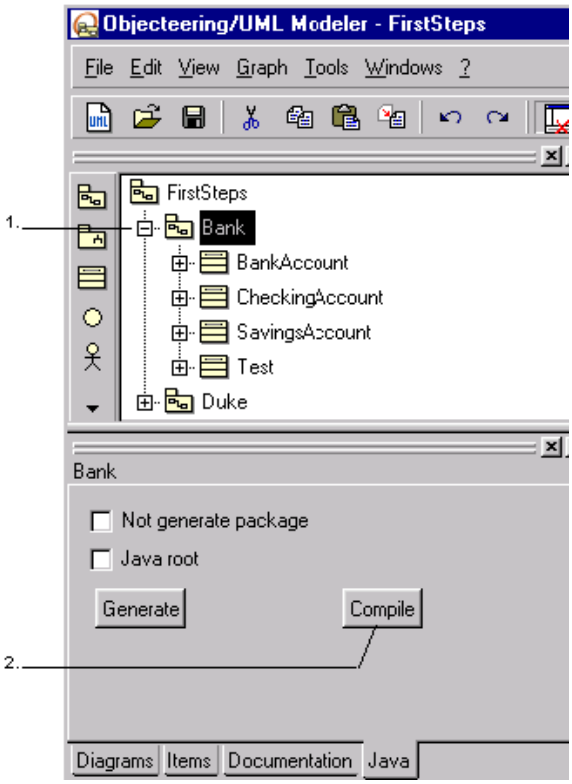


Figure 3-12. Running the "*Compile*" command for the "*Bank*" package

Steps:

1 - Select the "*Bank*" package in the explorer.

2 - In the "*Java*" tab of the properties editor, click on the "*Compile*" button.

Note: Exactly the same operation can be carried out by right-clicking on the "*Java*" work product in the "*Items*" tab of the properties editor, and then running the "*Java/Compile*" commands from the context menu which appears.

Analyzing compilation

Compilation errors can be analyzed directly from the tool (as shown in Figure 3-13). The "*Analyze the compilation*" command opens a window containing the compilation results.

Run the "*Java/Analyze the compilation*" command from the "*Bank*" package work product. This command opens a window containing the results of this compilation.

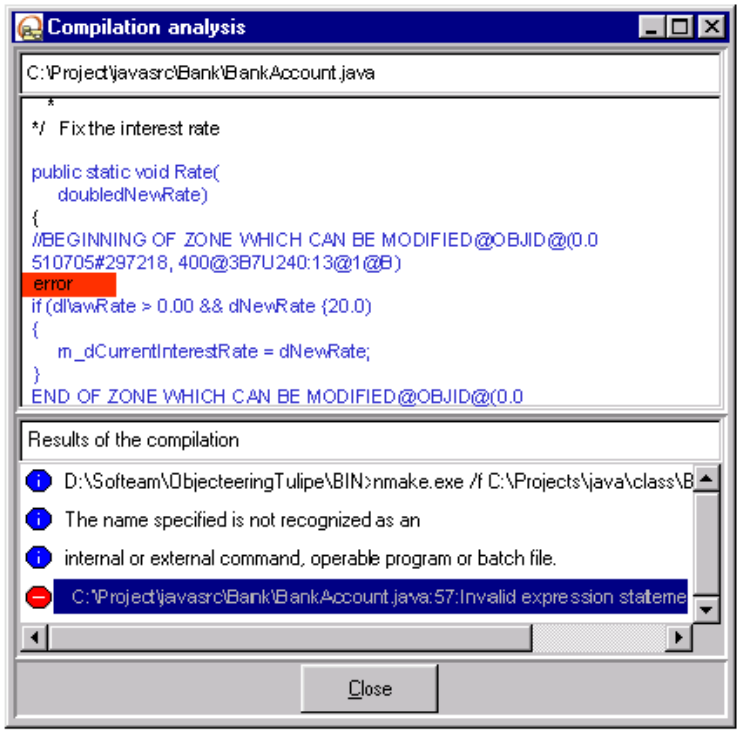


Figure 3-13. Window displaying the compilation analysis of the "*Java*" generation work product

This window is divided into two sections:

1 - The section displaying generated sources which present errors. If you double-click on the blue text, a dialog box for the generation element used to generate this zone will open.

2 - The lower section called "*Compilation results*", displays the errors prefixed by the  icon. The other lines are prefixed with the  icon. If you select a line of errors, the file containing the faulty code will be displayed in the upper section of the window. If you double-click on this code, the model element that generated it will be edited.

Documentation generation

Generation

The javadoc tool can be launched on a generation work product of a class for which code has been generated.

This feature uses the *Javadoc* type notes suggested for the model's *classes*, *attributes*, *associations* and *operations*.

You can choose whether a module parameter should process *description* type notes in the same way as *Javadoc* type notes or not.

The produced HTML file is then opened by the editor designated by the "*Command for editing HTML files*" parameter.

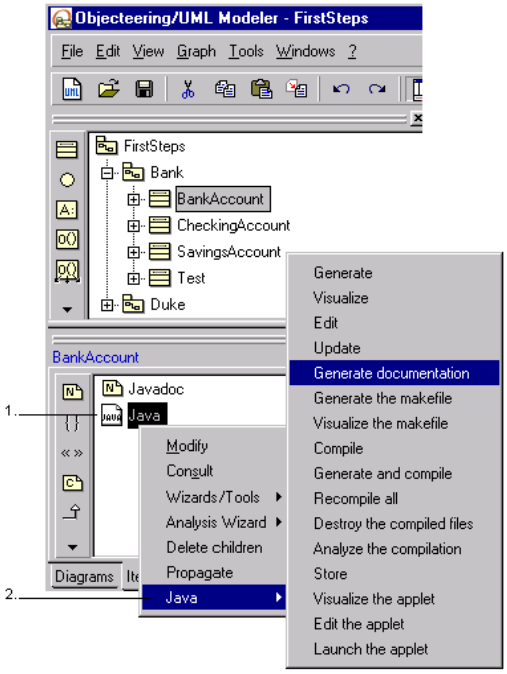


Figure 3-14. Generating documentation

Steps:

1 - Select the Java generation work product for the "*BankAccount*" class in the "*Items*" tab of the properties editor, by right-clicking.

2 - Run the "*Java/Generate documentation*" commands from the context menu which then appears.

The result is shown in Figure 3-15.

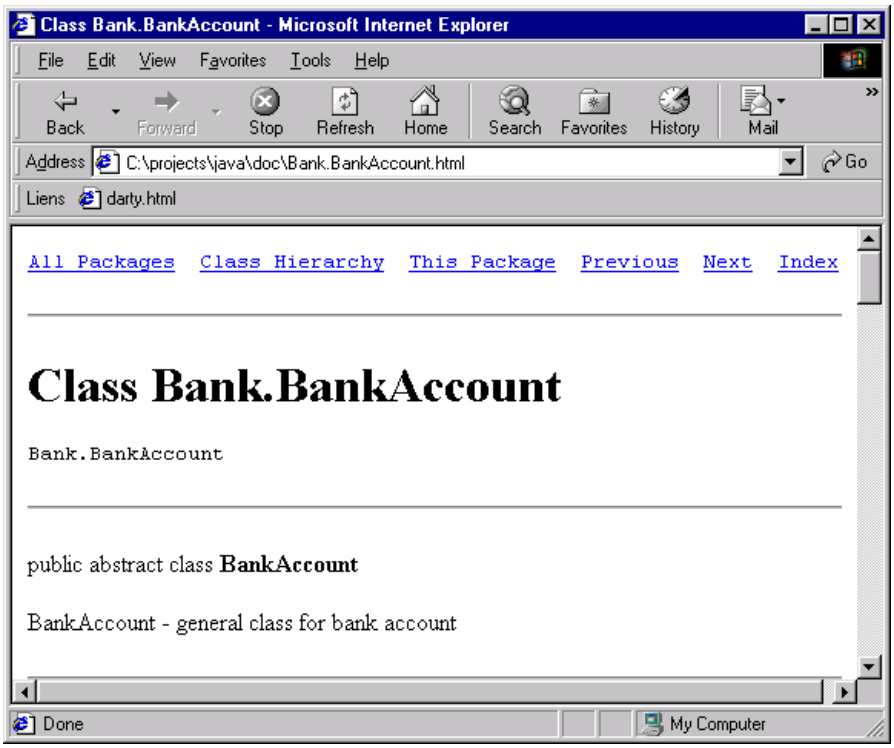


Figure 3-15. File edited by the "*Generate documentation*" command for the "*BankAccount*" class

Storage

Introduction

This feature is used to make "*jar*" format files. The "*jar*" format is compressed and can group together a set of "*.class*" files that will be coherent with associated resources (sound, pictures, ...).

Using this feature, it is possible, for example, to deliver beans developed using Objecteering/UML.

Storage

The "Store" command in a Java work product context menu can be used to create the "*.class*" in the element's lower hierarchy (package or class), to which the work product is linked. The resources belonging to the elements (classes and packages) of this lower hierarchy are added to these files (see the {*JavaBeanResource*} tagged value on a class and package).

Run this command, by selecting the "*Java/Store*" commands from the context menu on the "*Bank*" package work product.

Note: This feature uses the makefile and the compiled files. It is, therefore, necessary to generate the makefile and run the compiling before running the store command.

Running an applet

Introduction

We are now going to look at how to run an applet from Objecteering/UML. First, we have to position ourselves in the "*Duke*" package's "*DukeApplet*" class in the "*FirstSteps*" project. We are going to:

1 - create a Java generation work product

2 - generate the code

3 - generate the makefile

4 - compile

Note: If you are the JDK 1.2 or more recent, you may have to copy the *.java.policy* file from the *$OBJING_PATH/modules/JavaModule/2.1.f/FirstSteps* into your home directory. Alternatively, you may have to concatenate its contents with your *.java.policy* file, if it already exists.

Visualizing the HTML file

The "*Visualize the applet*" command in the Java work product context menu displays the current content of the HTML file used to run the applet (Figure 3-16). This command does not function for a work product that is not linked to a class which generalizes the "*Applet*" class.

Run the "*Java/Visualize the applet*" command on the "*DukeApplet*" class generation work product.

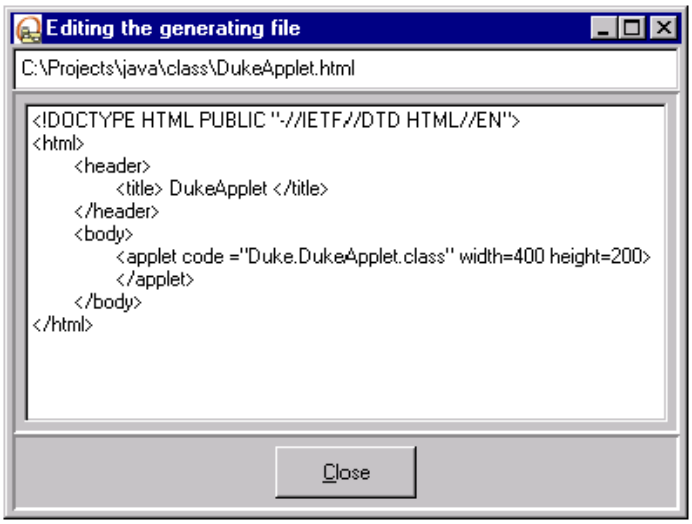


Figure 3-16. Window for visualizing the applet of the "*DukeApplet*" class generation work product

Editing the HTML file

The "*Edit the applet*" command in the Java work product context menu runs an editor on the HTML file, allowing the applet to be modified (Figure 3-17). This command does not function for a work product which is not linked to a class which generalizes the "*Applet*" class.

Run the "*Java/Edit the applet*" command on the "*DukeApplet*" class generation work product.

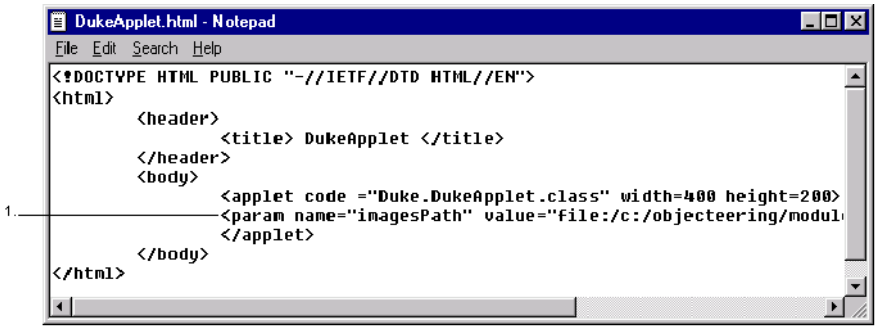


Figure 3-17. Window for editing the applet of the "*DukeApplet*" class generation work product.

Steps:

1 - Include the following line:

For Windows:

```
<param name="imagesPath"
value="file:/c:/objecteering/modules/JavaModule/2.2.a/FirstSteps/gif
/">
```

if Objecteering/UML is located in "*c:\objecteering*"

For Unix:

```
<param name="imagesPath"
value="file:/usr/objecteering/modules/JavaModule/2.2.a/FirstSteps/gi
f/">
```

if Objecteering/UML is located in "*/usr/objecteering*"

Running an applet

The "*Launch applet*" command in the Java work product context menu runs the applet (Figure 3-18). This command does not function for a work product which is not linked to a class which generalizes the "*Applet*" class.

To run the Java applet, you must have the correct rights in the $JDK_PATH/jre/lib/security/java.policy file. To this end, you may have to add the following line to the afore-mentioned file, before being able to run the applet correctly:

```
grant { permission java.security.AllPermission;};
```

Now run the "*Java/Launch applet*" command on the "*DukeApplet*" class generation work product.

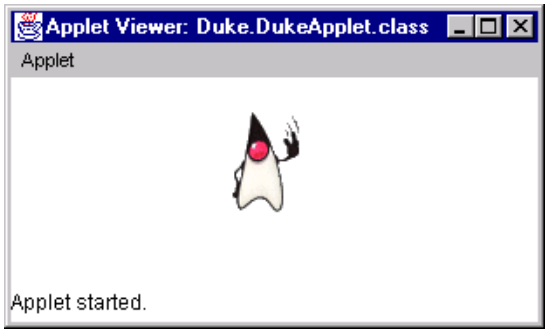


Figure 3-18. Applet run on "*DukeApplet*" class work product

Reverse

Introduction

The reverse allows existing library classes to be used during the modeling phase. The purpose of this is to allow the generalization of these classes, their implementation or their use in modeling.

To reverse a class, you must have its .class file, either as such or compressed in a .zip or .jar file. If you also have the .java source code, a complete reverse of the code and javadoc comments can be carried out. If this is not the case, only an interface reverse will be concerned.

We will now reverse the "*Applet*" class and the "*Button*" class.

If you do not have the source file for these classes, make sure that the "*ReverseJavaCode*" and "*ReverseJavaDoc*" parameters have not been checked.

Warning

A class created through a reverse will receive an identifier in the same way as an object created in an explorer or in a graphic editor.

If two people reverse the same class in two different UML modeling projects, Objecteering/UML will consider them as two different objects.

To avoid losing links to the reversed classes during the reverse, run the reverse operation in the reference UML modeling project from which all the classes used are imported.

Running the command

The command for running the reverse can be found in any package's context menu (as shown in figure 3-19).

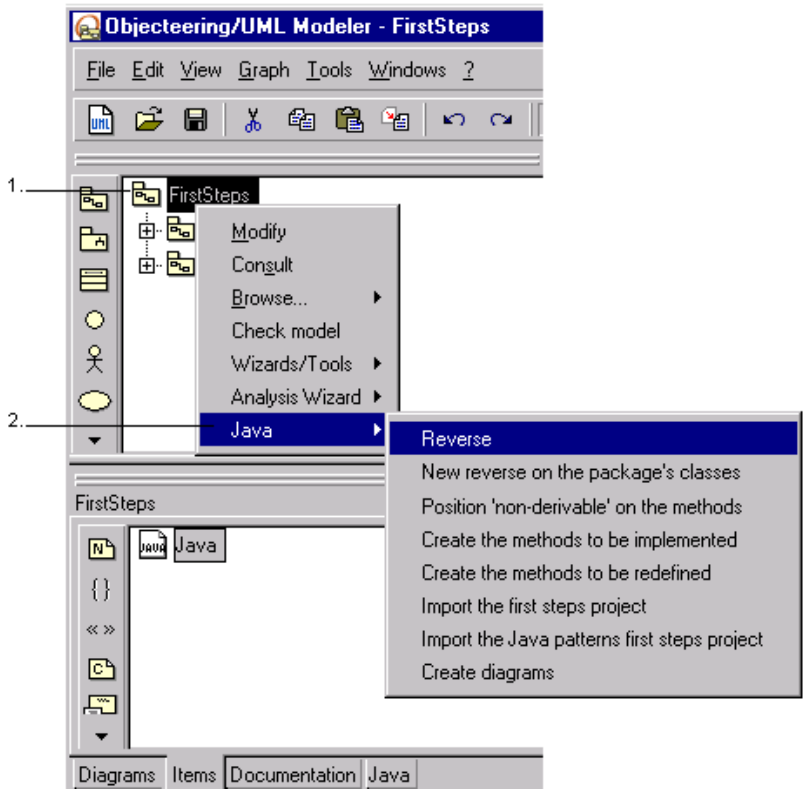


Figure 3-19. Running the reverse command

Steps:

1 - Select the "*FirstSteps*" package using the right mouse-button.

2 - Choose the "*Java/Reverse*" options from the context menus which appear.

When the reverse is launched, a message informs the user that consistency checks are active and that construction of the model, which may potentially not conform to the UML modeling rules checked by Objecteering/UML, may be refused (as shown in Figure 3-20).

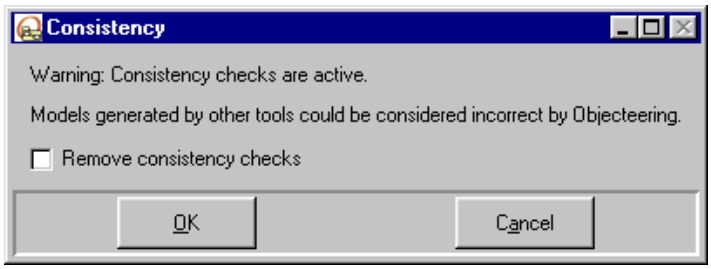


Figure 3-20. Message informing the user that consistency checks are active

If the user wishes to deactivate consistency checks, he should simply switch off the "*Remove consistency checks*" icon.

Selecting the classes to be reversed

The reverse command opens the dialog box used to select the sources, as represented below in figure 3-21. Remember that the path must be specified in the "*Reverse*" set of module parameters, in order to allow Objecteering/UML to find the classes or the classes in the packages. For further information on configuring the *Objecteering/Java* module, please refer to the "*Configuration window*" section in chapter 17 of this user guide.

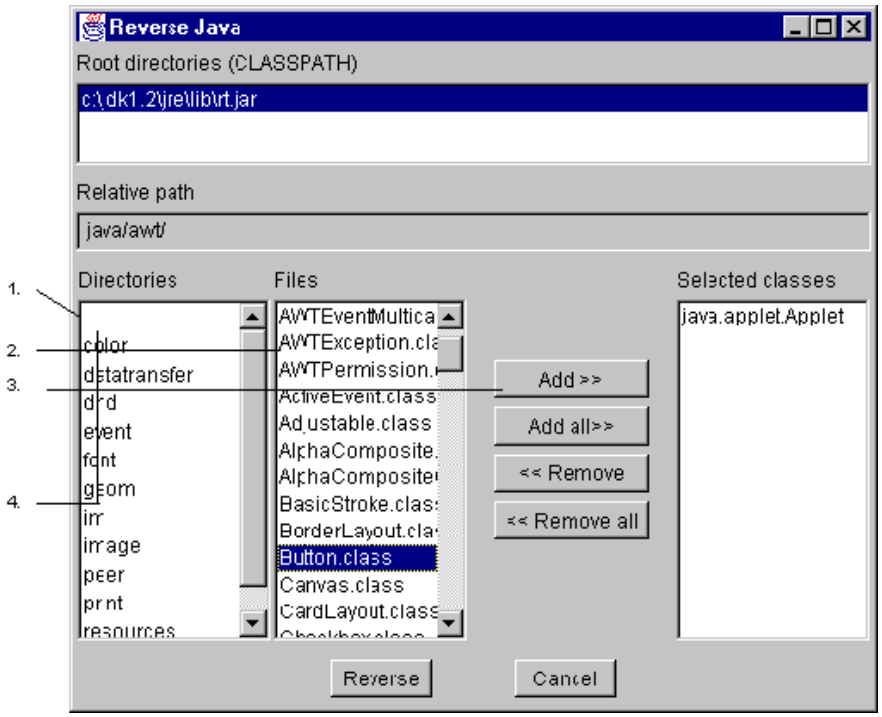


Figure 3-21. Selecting the classes to be reversed.

Steps:

1 - Select the "*java*" directory
2 - Select the "*applet*" directory
3 - Select the "*Applet.class"* file.
4 - Click on the "*Add*" button.
5 - Click on ".." to return to the *"java*" directory.
6 - Select the "*awt*" directory.
7 - Click on "*Button.class*".
8 - Click on "*Add*".
9 - Click on the *"Reverse* " button.

Result

A package named "*java*" appears in the "*FirstSteps*" package (see figure 3-22). It contains the "*applet*" package which contains the "*Applet*" class and the "*awt*" package containing the "*Button*" class.

It is now possible to create a class which generalizes "*Applet*" and which contains "*Button*" type attributes.

Other classes have also appeared. "*Applet*" and "*Button*" need these classes either as attributes, or as method parameters.

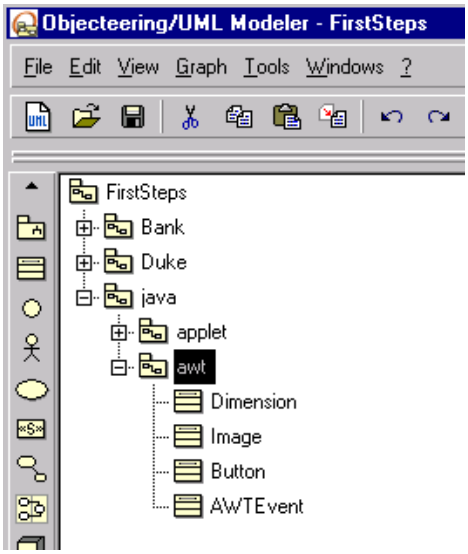


Figure 3-22. Result of the reverse as seen in the explorer

Round trip mode

We shall now look at the round trip functioning mode with regard to reverse operations.

Select the round trip functioning mode and generate the code for the "*Bank*" package. Then, using the editor of your choice, edit the "*BankAccount.java*" file and add an attribute declaration, for example, "*private int toto;*". Now compile the class.

If you proceed by trying to generate code in Objecteering/UML, a warning message informs you that the file has been modified outside Objecteering/UML. If you continue, you will lose the work you have previously carried out. You should, therefore, choose not to continue, and instead run the "*Reverse*" command for the class you have modified. The result of this is that the model changes (for example, the new attribute is added to the model of the "*BankAccount*" class).

Chapter 4: First Steps - Java Patterns

Introduction

General remarks

This First Steps UML modeling project allows you to learn how to use the different patterns of the *Objecteering/Java* module.

Initializing the UML modeling project

A First Steps UML modeling project for Java patterns is delivered with the *Objecteering/Java* module.

To use it, you have to:

1 - Create a UML modeling project (for example "*FirstSteps*").

2 - Once this UML modeling project has been opened, select the *Java* module, and carry out the steps shown in Figure 4-1.

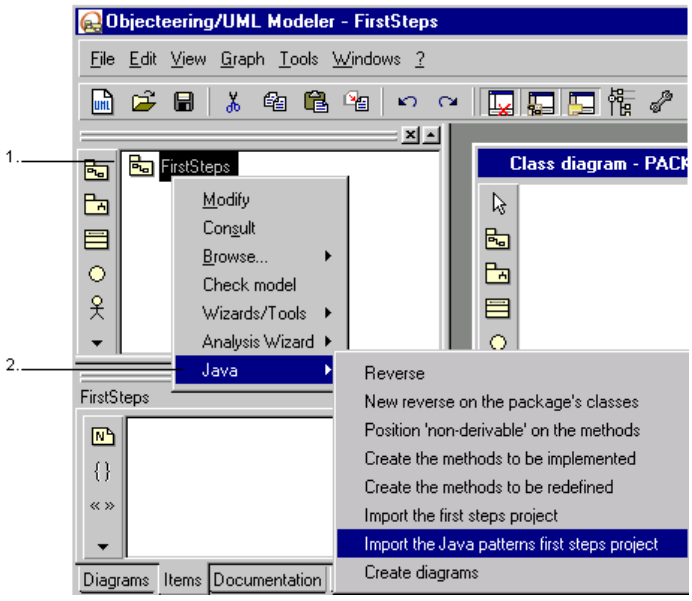


Figure 4-1. Importing the *First Steps* UML modeling project

Steps:

1 - Select the "*FirstSteps*" package by right-clicking.

2 - Run the "*Java/Import the Java patterns first steps project*" commands from the context menu which then appears.

Remote method invocation

Introduction

We are going to work with the "*PatternRMI*" package. It will allow us to create a server and customer application that will communicate through "*RMI*".

The package includes:

- a "*Server*" class containing the server's "*main*" method
- a "*Client*" class containing the client's "*main*" method
- a "*TableEntry*" class representing an information atom
- a "*Table*" class containing access methods to the information

The "*TableEntry*" class represents the couple capital-country. The "*Table*" class offers two services, one gives the capital of a given country, the other gives the country of a given capital.

We will apply the *pattern* to the "*Table*" class in order to provide remote access to its services.

Note:If you are using the JDK 1.2 or more recent, you may have to copy the *.java.policy* file from the *$OBJING_PATH/modules/JavaModule/2.1.f/FirstSteps* into your home directory. Alternatively, you may have to concatenate its contents with your *.java.policy* file, if it already exists.

Model before transformation

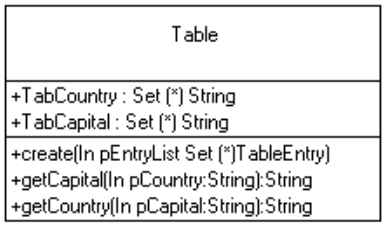


Figure 4-2. Initial model

Running the RMI pattern

After having selected the "*Table*" class in the explorer, activate the pattern (Figure 4-3).

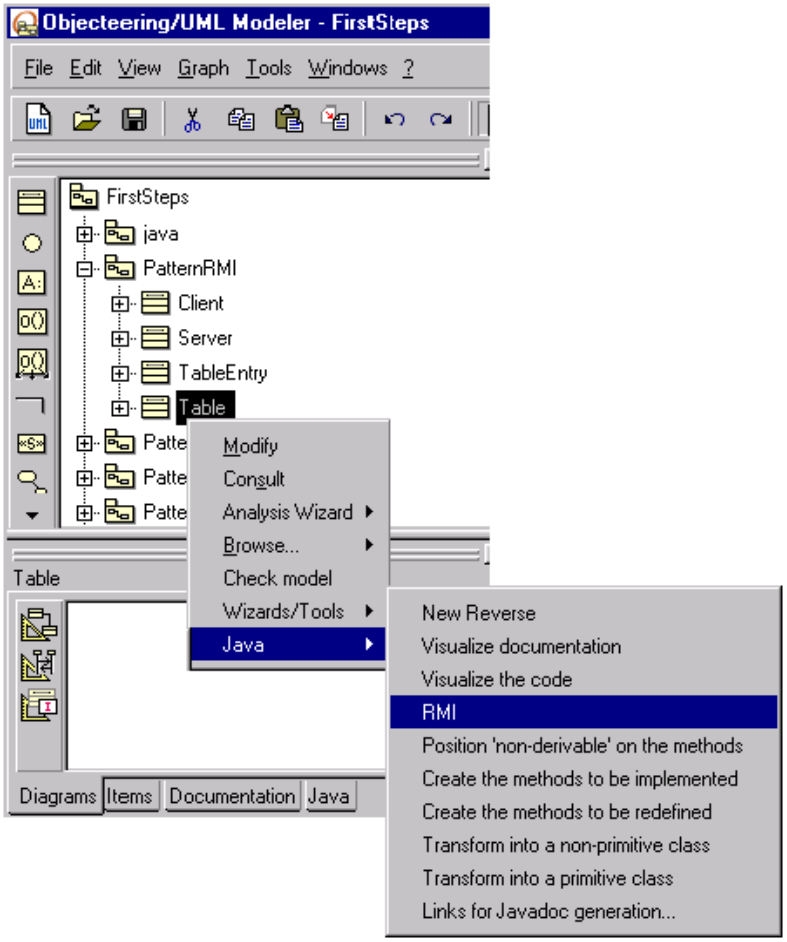


Figure 4-3. Running the pattern

A dialog box opens, suggesting default names for the class and interface that will be deduced from the "*Table*" class. These names can be modified.

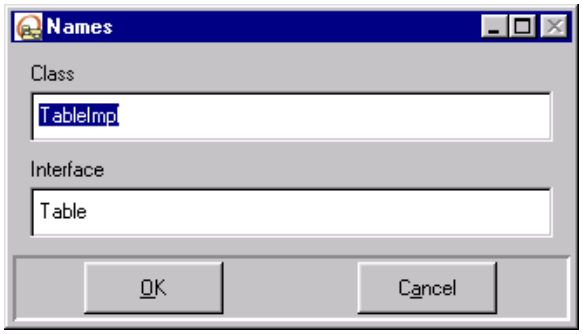


Figure 4-4. Entering class and interface names

Confirm by clicking on OK.

Model after transformation

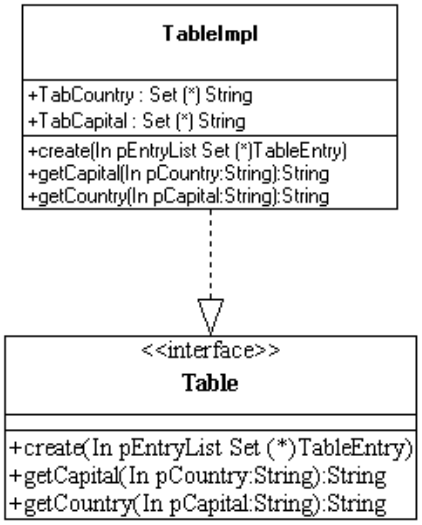


Figure 4-5. Transformed model

Summary of the transformations

The following modifications were made to the concrete class:

- the "*Table*" class was renamed "*TableImpl*"
- a "*TableImpl*" generalization towards "*UnicastRemoteObject*" was added
- an import towards "*java.rmi.**" on "*TableImpl*" was added
- the {*JavaThrownException*} tagged value was added to each method, to indicate that it may run the "*RemoteException*" exception
- an implementation link towards the "*Table*" interface was added

Furthermore, a "*Table*" interface was created with:

- all the "*TableImpl*" public methods
- a generalization towards "*Remote*"
- an import towards "*java.rmi**"

Generating code/Compiling

It is now necessary to generate and compile. Follow the steps below:

1 - Create a Java generation work product on the "*PatternRMI*" package.

2 - Propagate the work product to the classes.

3 - Generate the code of the package's classes.

4 - Generate a makefile on the "*PatternRMI*".

5 - Compile.

For further information on these operations, please refer to chapter 3 of this user guide.

Execution

On the server machine:

- add your file compilation directory to your CLASSPATH variable (the last Java work product field)
- launch the *rmiregistry* executable if it has not already been launched
- launch the *PatternRMI.Server* server application

On the client machine:

- launch the *PatternRMI*.Client application with the server machine name as the parameter

The following result is displayed in the client console:

```
The capital of China is Beijing.
The country whose capital is Rome is Italy.
```

Sending events

Introduction

We are going to work with the "*PatternEventSource*" package. We will thus be able to create events on an attribute and on an association.

The "*PatternEventSource*" package contains:

- a "*Source*" class containing a "*name*" attribute and a "*data*" association to which we will apply the pattern
- a "*Data*" class, the target of the data association
- a "*SpyApplet*" which generalizes Applet and which will allow us to visualize the pattern's action
- an "*InnerSpy*" class, internal class of "*SpyApplet*", which will act as subscriber to the events that we will have created

In Java, events are grouped into event classes. Therefore in Java you have the following basic event classes: "*Window*", "*Mouse*" and *"Key*".

The event sending pattern follows the same principle.

Initial model of the Source class

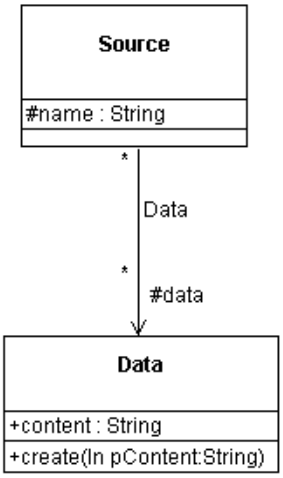


Figure 4-6. Calling the pattern on the "*name*" attribute

We are going to run the event creation pattern on the "*name*" attribute of the Source class.

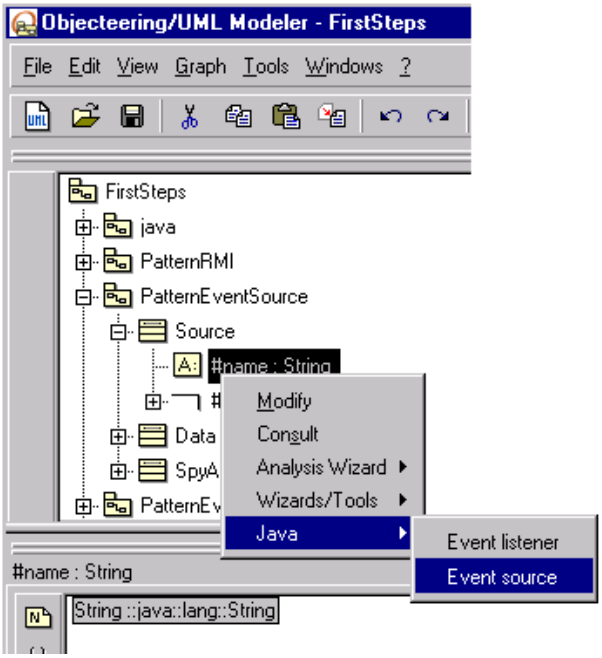


Figure 4-7. Creating events on the "*name*" attribute

A dialog box, used to create new events, opens.

We are going to create a "*NameChanged*" event to be launched each time the value of an attribute is modified. We are also going to create an "*EmptyName*" event aimed at being launched when the attribute takes an empty string as its value.

To do this, carry out the following actions:

1 - Enter "*Name*" in the "*Event class*" field.

2 - Enter "*NameChanged*" in the "*New event*" field.

3 - Click on the "*Add*" button.

4 - Enter "*EmptyName*" in the "*New event*" field.

5 - Click on the "*Add*" button.

6 - Click on the "*OK*" button.

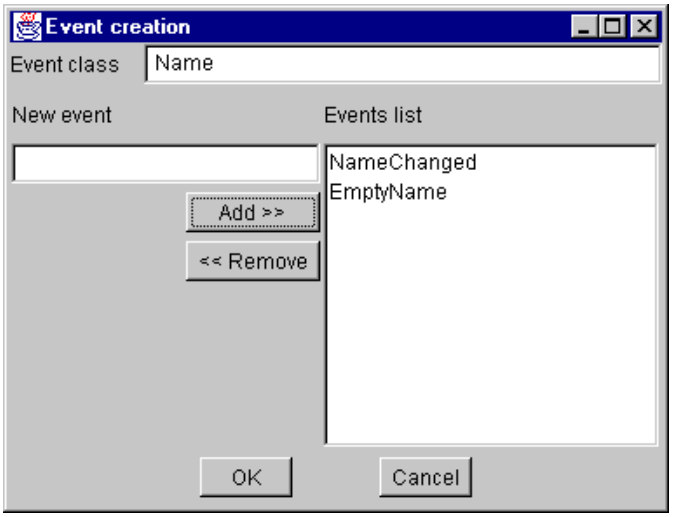


Figure 4-8. Dialog box for creating events

Model of the source class after the first transformation

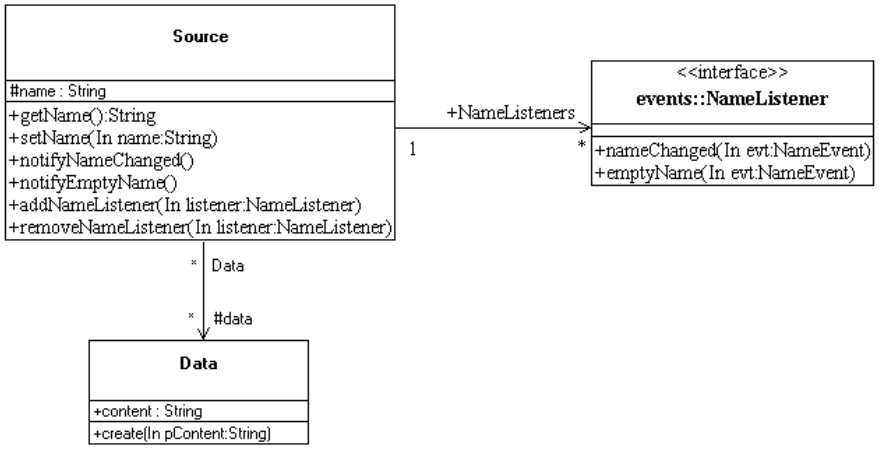


Figure 4-9. Transformed model

Summary of the elements created during the first transformation

New elements appear in our UML modeling project:

- an "*event*" package in the "*PatternEventSource*" package
- a "*NameEvent*" class, which generalizes the JDK's "*EventObject*" class, in the "*event*" package. This class represents the "*Name*" type event class we have just created. Objects with this type will be created each time a "*Name*" type event occurs, in other words, the changing of a name or an empty name.

The class contains ...	**function ...**
a "*name*" attribute	designed to contain the new value of the attribute after the event has occurred.
a "*NAME_CHANGED*" constant	represents the name change event.
an "*EMPTY_NAME*" constant	represents the empty name event.
two constants, "*NAME_FIRST*" and "*NAME_LAST*"	allows you to enumerate the events of the "Name" class.
an "*id*" attribute	designed to take the value of one of the constants representing the events, i.e. "*NAME_CHANGED*" or "*EMPTY_NAME*".
a constructor	takes the event's source object, the type of event and the new value of the "name" attribute as parameters.

- a "*NameListener*" interface in the "*event*" package. This interface defines the methods to be redefined to react to the "*Name*" type events. It contains a "*nameChanged*" method linked to the name change event and an "*emtpyName*" method linked to the empty name event.
- a "*NameAdapter*" class in the "*event*" package. This class implements the "*NameListener*" interface and defines the "*nameChanged*" and "*emptyName*" methods that have no effect. A class such as this is defined each time an event class contains at least two events. Indeed, the subscription is carried out through an event class and not through one event after the other. However, sometimes we are not interested in all the events of an event class. In this way, instead of implementing the interface, a listener can generalize this class and only define the methods corresponding to the events to which it wants to react.

Furthermore, the "*Source*" class has been enriched with the following elements:

- an instance association towards the "*NameListener*" interface described above. It will allow an instance of the "*Source*" class to find out which objects have subscribed to the "*Name*" type events it produces
- a "*getName*" accessor which allows the recovery of the current value of the "*name*" attribute
- a "*setName*" mutator which allows the modification of the value of the "*name*" attribute and the notification of the subscribers to the events of the "*Name*" class on the current instance
- a {*JavaNoAccessor*} tagged value on the "*name*" attribute indicating to the generator that it should not generate accessors. Indeed, the pattern created them
- a "*notifyNameChanged*" method that tells the subscribers to the "*Name*" type events of the current instance that the name has changed
- a "*notifyEmptyName*" method that tells the subscribers to the "*Name*" type events of the current instance that the name is emtpy
- a "*addNameListener*" method which allows an object to subscribe to "*Name*" type events on the current instance
- a "*removeNameListener*" method which allows an object to be removed from the list of subscribers to the "*Name*" type events of the current instance

Modifying the mutator

We are going to look at the "*setName*" method of the "*Source*" class. The method created by the pattern must be modified.

Indeed, the pattern does not know the semantics of the events. Thus, we can see that the "*setName*" method systematically notifies the name change event, which is correct but also gives the empty name event, which is not correct.

Therefore a test needs to be added.

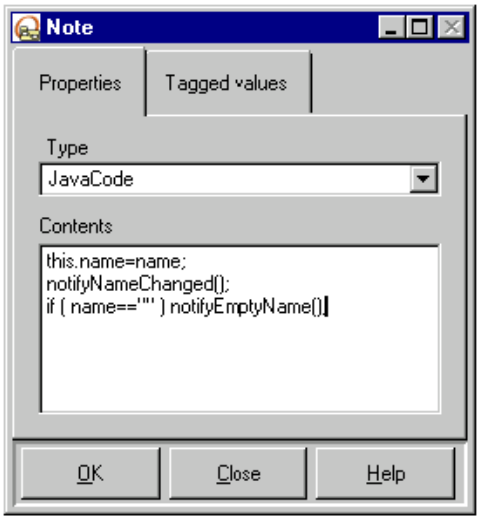


Figure 4-10. Modifying the "*setName"* method

Calling the pattern on the "*data*" association

We are going to run the event creation pattern on the "*data*" association of the "*Source*" class.

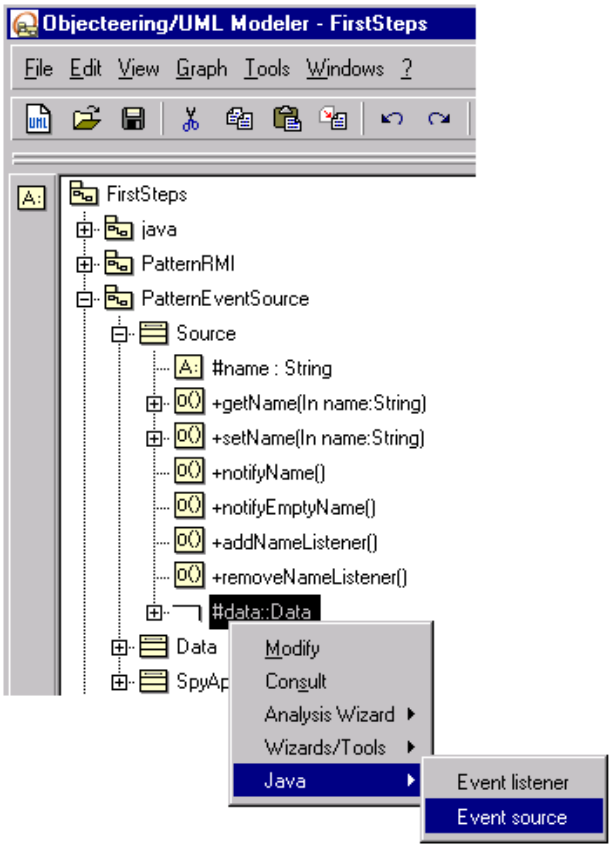


Figure 4-11. Creating events on the "*data*" association

Creating events on the "data" association

A dialog box opens which allows the creation of new events.

We are going to create a "*DataChanged*" event that will be launched each time data are added or removed.

To do this, carry out the following actions:

1 - Enter *"Data*" in the "*Class of events*" field.

2 - Enter " *DataChanged*" in the "*New event*" field.

3 - Click on the "*Add*" button.

4 - Click on the "*OK*" button.

Model of the source class following the second transformation

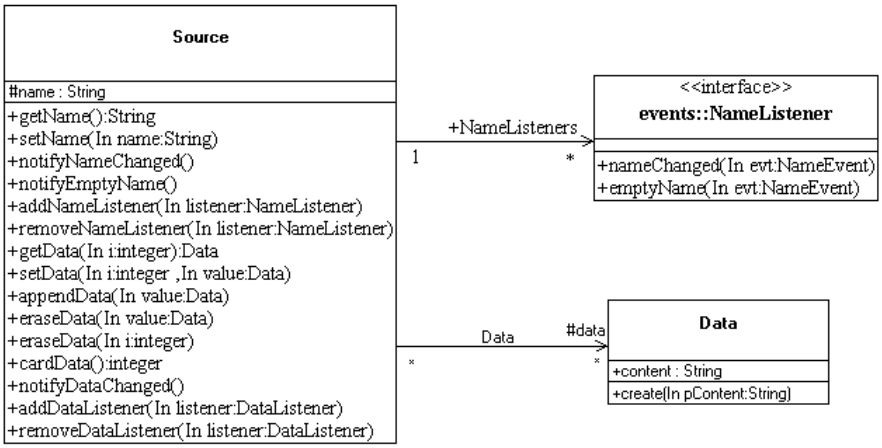


Figure 4-12. Transformed model

Summary of the elements created during the second transformation

New elements appear in our UML modeling project:

- a "*DataEvent*" class, which specializes the JDK's "*EventObject*" class, in the "*event*" package. This class represents the "*Data*" type event class we have just created. Objects with this type will be created each time a "*Data*" type event occurs.

The class contains ...	which ...
a "*data*" association with the same cardinality as the one of the Source class	contains the image of the object's association after the event has occurred
a "*DATA_CHANGED*" constant	represents the name change event
two constants "*DATA_FIRST*" and "*DATA_LAST*"	enumerates the events of the "*Name*" class

- an "*id*" attribute designed to take the value of one of the constants representing the events, i.e. "*DATA_CHANGED*" only
- a constructor that takes the event's source object, the type of event and the new value of the "data" association as its parameters
- a "*DataListener*" interface in the "*event*" package. This interface defines the methods to be redefined to react to "*Data*" type events. It contains a "*dataChanged*" method linked to the association's modification event.

Furthermore, the Source class has been enriched with the following elements:

- an instance association towards the "*DataListener*" interface described above. It will allow an instance of the "*Source*" class to find out which objects have subscribed to the "*Data*" type events it produces
- a "*getData*" accessor allowing the recovery of the Data type object which has the "data" association data rank
- a "*cardData*" accessor allowing the recovery of the current multiplicity of the "*data*" association
- a "*setData*" mutator allowing you to modificy the value of the Data type object with the "data" association data rank and to notify the subscribers to the "*Data*" class events on the current instance
- an "*appendData*" mutator allowing the addition of a Data type object to the "data" association and the notification of the subscribers to the "*Data*" class events on the current instance
- an "*eraseData*" mutator allowing the deletion of a Data type object from the "data" association and the notification of the subscribers to the "*Data*" class events on the current instance
- an "*eraseData*" mutator allowing the deletion of a Data type object from the "data" association data rank and the notification of the subscribers to the "*Data*" class events on the current instance
- a {*JavaNoAccessor*} tagged value on the "*data*" association indicating to the generator that it should not generate accessors. Indeed, the pattern created them
- a "*notifyDataChanged*" method that tells the subscribers to the "*Data*" type events of the current instance that the name has changed
- a "*addDataListener*" method allowing an object to subscribe to "*Data*" type events on the current instance
- a "*removeDataListener*" method allowing an object to be removed from the list of subscribers to the "*Data*" type events of the current instance

Subscription to the "*InnerSpy*" class

It is first necessary to create an implementation link from the *"InnerSpy*" class towards the "*NameListener*" interface and another towards the "*DataListener*" interface. The pattern for creating methods to be implemented on the "*InnerSpy*" class is then applied.

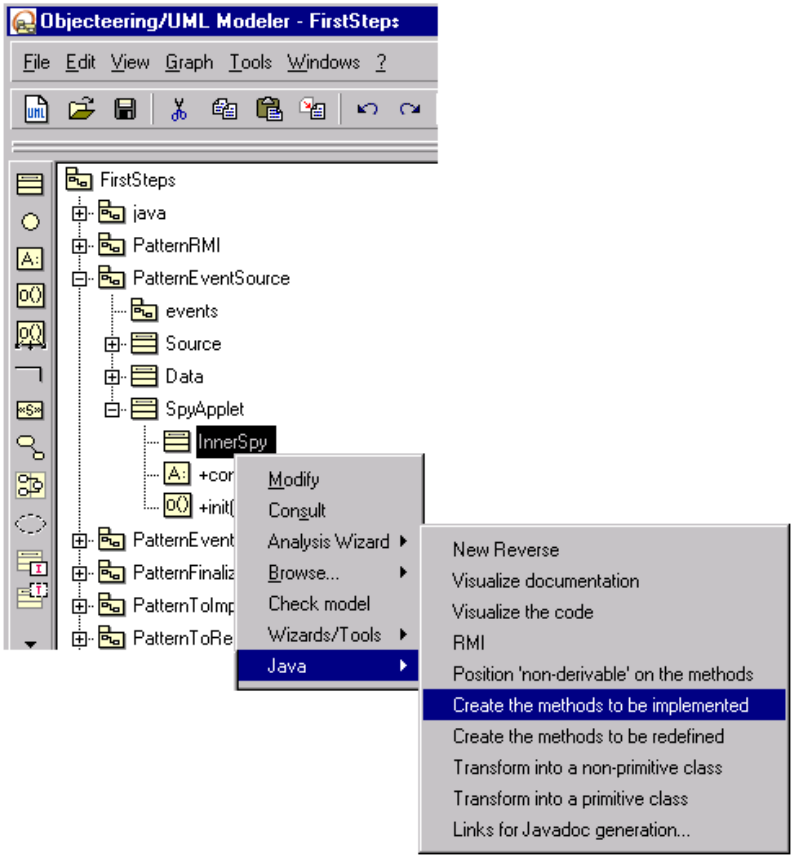


Figure 4-13. Creating methods to react to events

The "*nameChanged*", "*emptyName*" and "*dataChanged*" methods have been created in "*InnerSpy*". You now have to enter the code of these methods. In the example, we are simply going to display traces in the applet's text zone.

"*nameChanged*" method:

```
getConsole().append ( "The new name is " + evt.getName() +
"\n" );
```

"*emptyName*" method:

```
getConsole().append ( "Warning ! Non identified element !\n"
);
```

"*dataChanged*" method:

```
getConsole().append ( "The current data is :\n" );
for ( int i = 0 ; i < evt.cardData() ; i++ )
{
   getConsole().append ( "   - " +
evt.getData(i).getContent() + "\n" );
}
```

Generating code/Compiling

You now have to generate and compile. Follow the steps below:

1 - Create a Java generation work product on the "*PatternEventSource*" package

2 - Propagate the work product to the classes of the "*PatternEventSource*" package

3 - Generate the code of the "*PatternEventSource*" package's classes

4 - Generate a makefile on the "*PatternEventSource*" package

5 - Compile the "*PatternEventSource*" package

For further information on these operations, see the "*First Steps*" chapter in this user guide.

Execution

We are now going to be able to launch the applet. You simply have to activate the menu for launching the applet on the Java work product of the "*SpyApplet*" class.

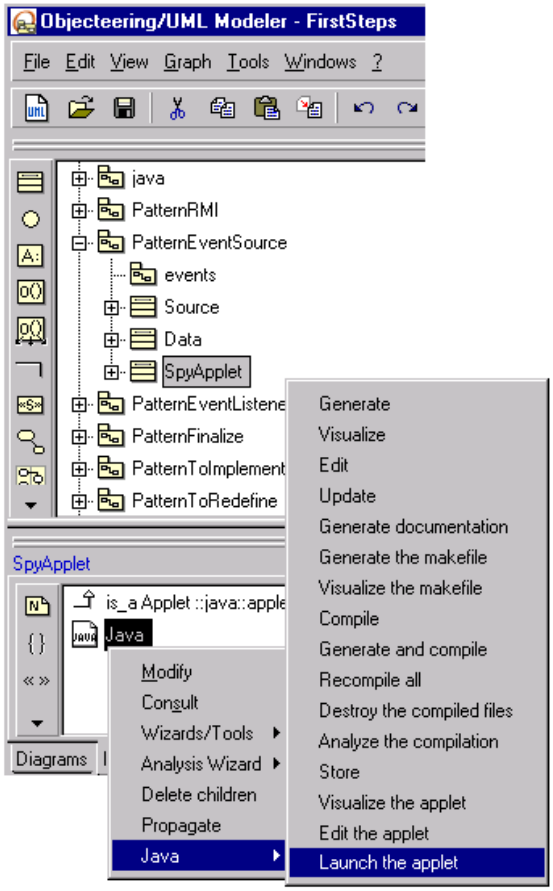


Figure 4-14. Executing the applet

The applet runs a scenario that creates a "*Source*" class object, subscribes to its "*Name*" and "*Data*" type events and modifies the "*name*" attribute and the "*data*" association of this object. The result is as follows:

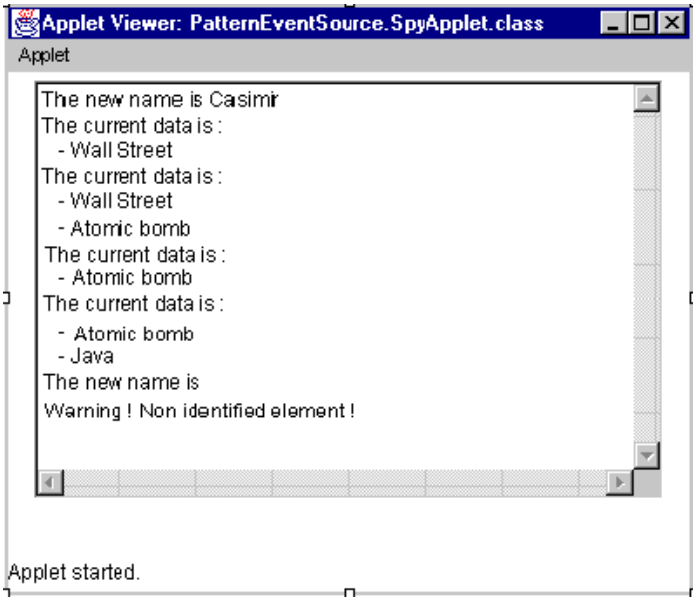


Figure 4-15. Result of the applet execution

Listening to events

Introduction

We are going to work with the "*PatternEventListener*" package, which will allow us to listen to events generated by one of its attributes.

The "*PatternEventListener*" package contains a "*TestApplet*" class which generalizes the JDK's Applet class and which contains a "*Button*" type attribute (which is a JDK class), to which we will apply the pattern, as well as a text field that will be used to visualize the events we will listen to.

Initial model

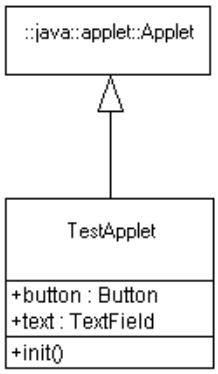


Figure 4-16. Initial model

Calling the pattern

We are going to run the pattern on the "*button*" attribute of the "*TestApplet*" class (Figure 4-17).

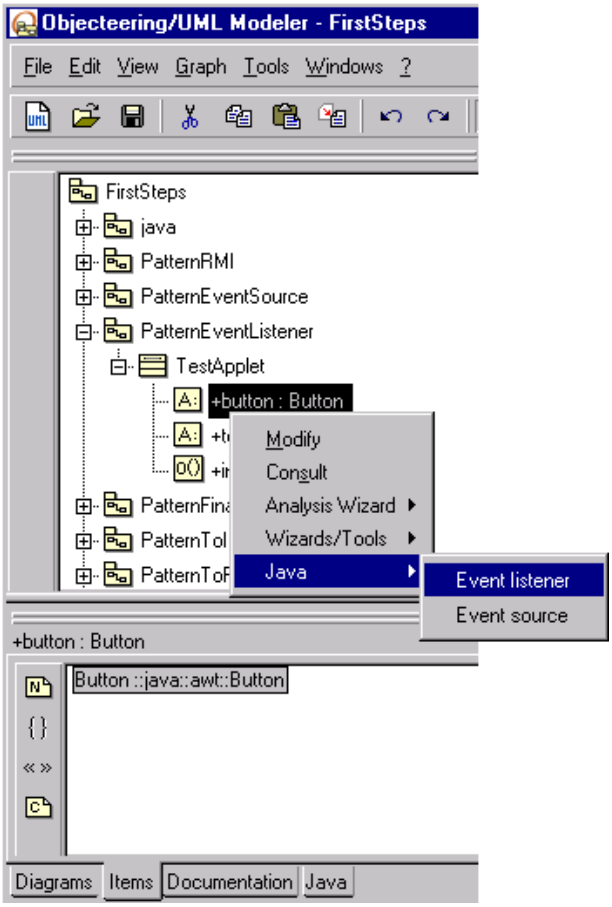


Figure 4-17. Running the pattern

Selecting events

When the dialog box opens, carry out the following steps:

1 - Select the "*focusGained*" event in the list of available events.

2 - Click on the "*Add*" button.

3 - Select the "*focusLost*" event in the list of available events.

4 - Click on the "*Add*" button.

5 - Select the "*keyPressed*" event in the list of available events.

6 - Click on the "*Add*" button.

7 - Select the "*keyReleased*" event in the list of available events.

8 - Click on the "*Add*" button.

9 - Click on the "*OK*" button.

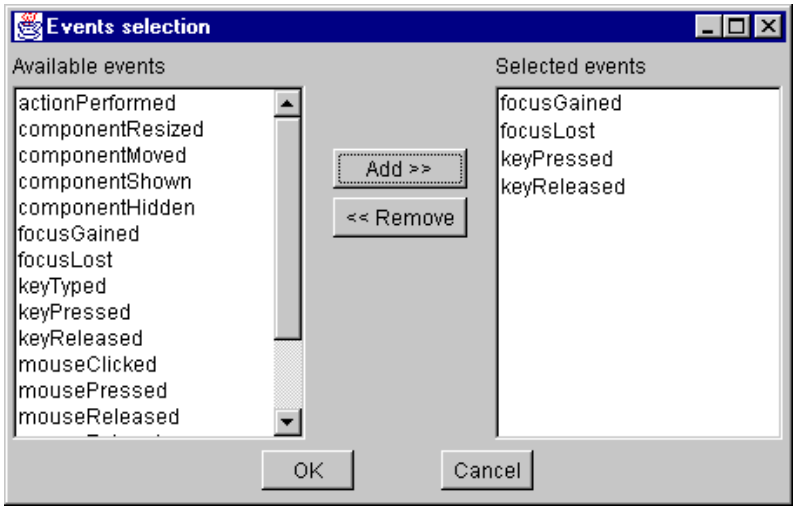


Figure 4-18. Dialog box for selecting events to listen to

Model after transformation

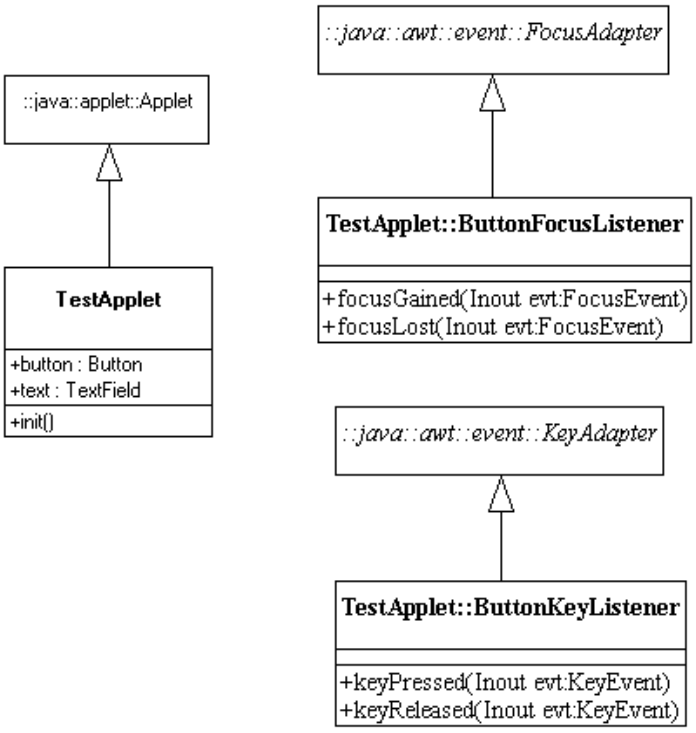


Figure 4-19. Transformed model

Summary of the elements created during the transformation

Application of the pattern created:

- an inner class, "*ButtonKeyListener*", which generalizes the JDK "*KeyAdapter*" class, to react to "*keyPressed*" and "*keyReleased*" events
- an inner class, "*ButtonFocusListener*", which generalizes the JDK "*FocusAdapter*" class, to react to the "*focusGained*" and "*focusLost*"" events

Entering methods of reaction to events

The bodies of the methods reacting to events remain to be entered. We will simply display here the traces in the applet's text field.

"*focusGained*" method of the "*ButtonFocusListener*" class:

```
getText().setText ( "The button has the focus" );
```

"*focusLost*" method of the "*ButtonFocusListener*" class:

```
getText().setText ( "The button has lost the focus" );
```

"*keyPressed*" method of the "*ButtonKeyListener*" class:

```
getText().setText ( "A key has been pressed" );
```

"*keyReleased*" method of the "*ButtonKeyListener*" class:

```
getText().setText ( "The key has been released" );
```

Generating code/Compiling

You now have to generate and compile. Follow the steps below :

- create a Java generation work product on the "*PatternEventListener*" package
- propagate the work product to the classes of the "*PatternEventListener*" package
- generate the code of the "*PatternEventListener*" package's classes
- generate a makefile on the "*PatternEventListener*" package
- compile the "*PatternEventListener*" package

For further details on these operations, see the "*First Steps*" chapter in this user guide.

Execution

We are now able to launch the applet. To do this, simply activate the menu for launching the applet on the Java work product of the "*TestApplet*" class.

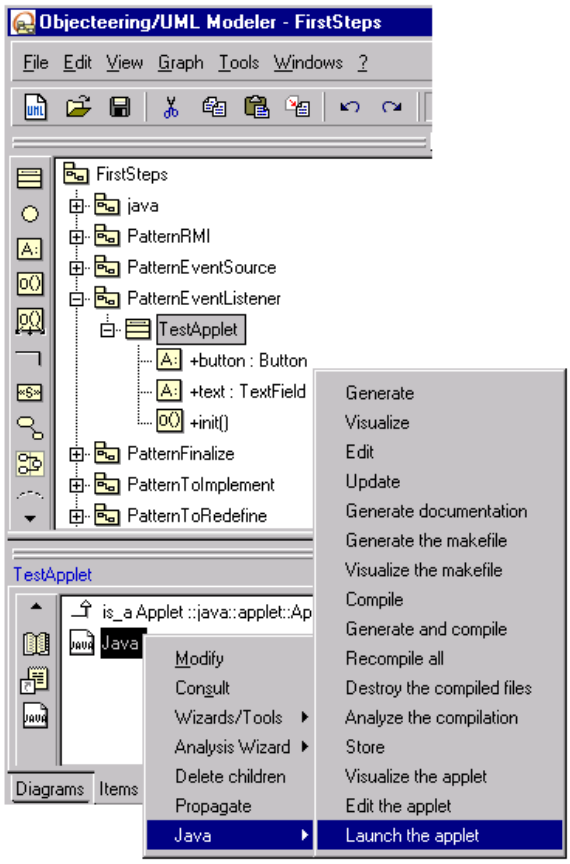


Figure 4-20. Executing the applet

The applet is now activated.

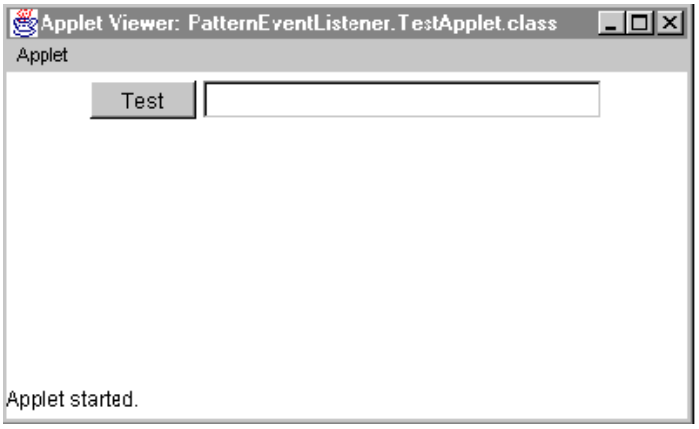


Figure 4-21. Result of the applet execution

1 - Click on the "*Test*" button. The text field now displays "*The button has the focus*".

2 - Click on the text field. The text field now displays "*The button has lost the focus*".

3 - Click on the "*Test*" button. The text field now displays "*The button has the focus*".

4 - Press a key and hold it down. The text field now displays "*A key has been pressed*".

5 - Release the key. The text field now displays "*The key has been released*".

Non derivable operation

Introduction

We are going to work with the "*PatternFinalize*" package. It will allow the positioning of the "*Cannot be specialized*" field of certain operations.

The "*PatternFinalize*" package includes:

- a "*ParentClass*" class containing three operations
- a "*ChildClass1*" class, which is a child of "*ParentClass*", containing two operations, one of which redefines one of the "*ParentClass*" operations
- a "*ChildClass2*" class, which is a child of "*ParentClass*", containing a operation that redefines one of the "*ParentClass*" operations

Model before transformation

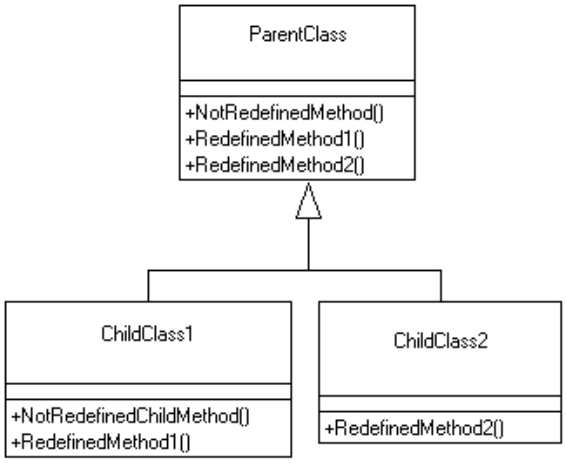


Figure 4-22. Initial model

Calling the pattern

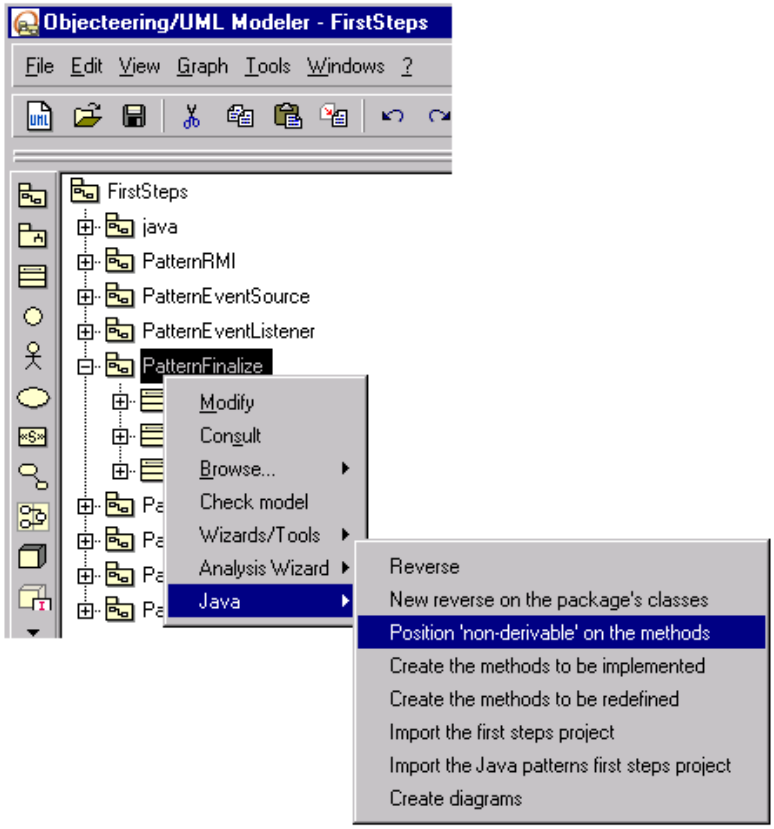


Figure 4-23. Calling the pattern

Model after its transformation

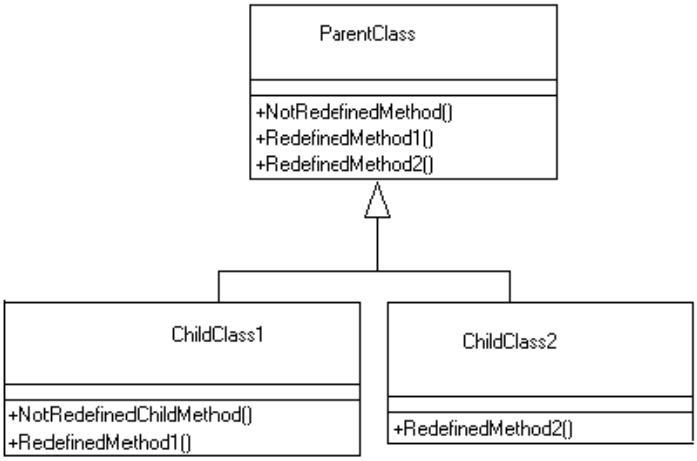


Figure 4-24. Transformed model

Summary of the modifications made to the model

The field of the "*ParentClass*" "*NotRedefinedMethod*" operation that is not redefined in any child class is selected.

The field of the "*ChildClass1*" "*NotRedefinedMethod*" operation that is not redefined in any child class is selected.

The field of the "*RedefinedMethod1*" operation of the "*ChildClass1*" class that redefines the operation with the same name of the "*ParentClass*" class is selected since the operation is not redefined itself.

The field of the "*RedefinedMethod2*" operation of the "*ChildClass2*" class that redefines the operation with the same name of the "*ParentClass*" class is selected since the operation is not redefined itself.

Creating operations to implement

Here we will be working with the "*PatternsToImplement*" package, which will allow us to automatically create the operations that a class must redefine according to its implementation links.

The "*PatternsToImplement*" package contains:

- an "*Appliance*" class interface containing the "*switchOn*" and "*switchOff*" operations
- a "*Screen*" class interface specializing "*Appliance*" and containing the "*displayTest*", "*getCurser*" and "*setCurser*" operations
- a "*GraphicTool*" class interface containing the "*drawRectangle*" and "*drawCircle*" operations
- a "*MyGraphicDisplay*" class which implements the "*Screen*" and "*GraphicTool*" interfaces

We will apply the pattern to the "*MyGraphicDisplay*" class.

Initial model

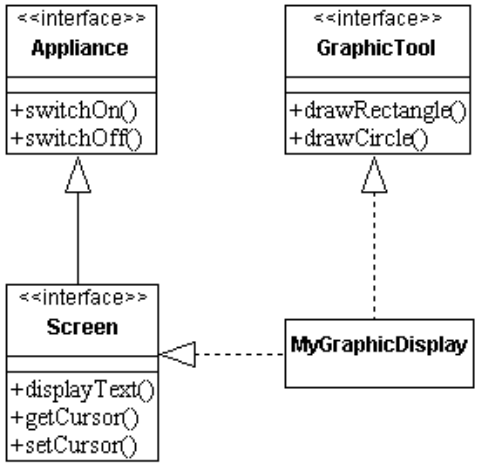


Figure 4-25. The initial model

Calling the pattern

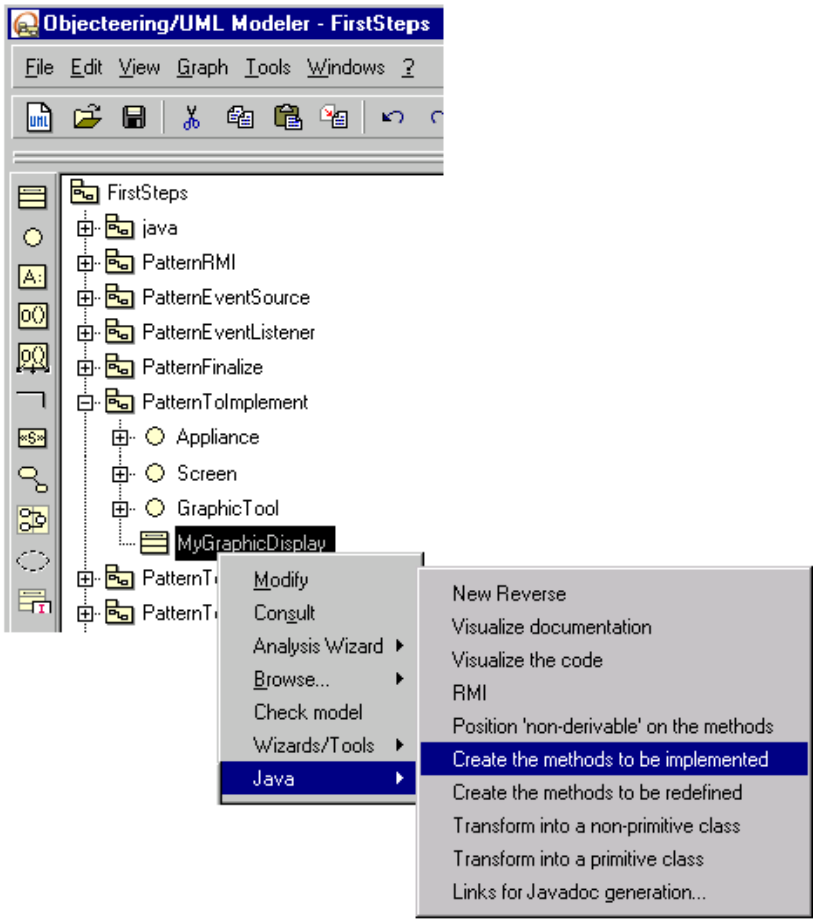


Figure 4-26. Calling the pattern

Final model

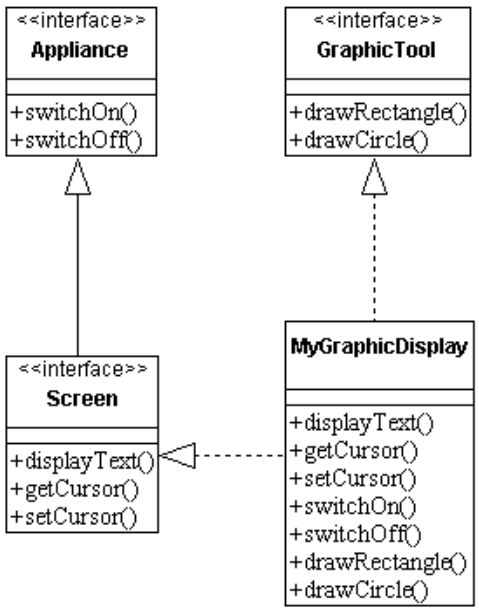


Figure 4-27. The model after transformation

Summary of the modifications made to the model

The "*MyGraphicTool*" class has been enriched with the following operations:

- "*displayTest*" which redefines the operation of the same name of the "*Screen*" class
- "*getCursor*" which redefines the operation of the same name of the "*Screen*" class
- "*setCursor*" which redefines the operation of the same name of the "*Screen*" class
- "*switchOn*" which redefines the operation of the same name of the "*Appliance*" class
- "*switchOff*" which redefines the operation of the same name of the "*Appliance*" class
- "*drawRectangle*" which redefines the operation of the same name of the *GraphicTool*" class
- "*drawCircle*" which redefines the operation of the same name of the "*GraphicTool*" class

Creating operations to be redefined

Here we will be working with the "*PatternToRedefine*" package, which will allow us to automatically create operations which a class must redefine according to its generalization links.

The "*PatternsToRedefine*" package contains:

- an "*Animal*" abstract class containing the "*eat*" abstract operation
- a "*Bird*" abstract class which specializes "*Animal*" and which contains the "*sing*" abstract operation and the "*fly*" non-abstract operation
- a "*crow*" class which specializes "*Bird*"

We will apply the pattern to the "*Bird*" class.

Initial model

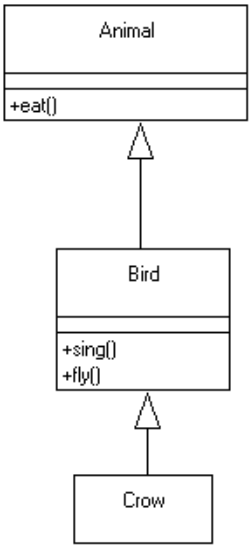


Figure 4-28. The initial model

Calling the pattern

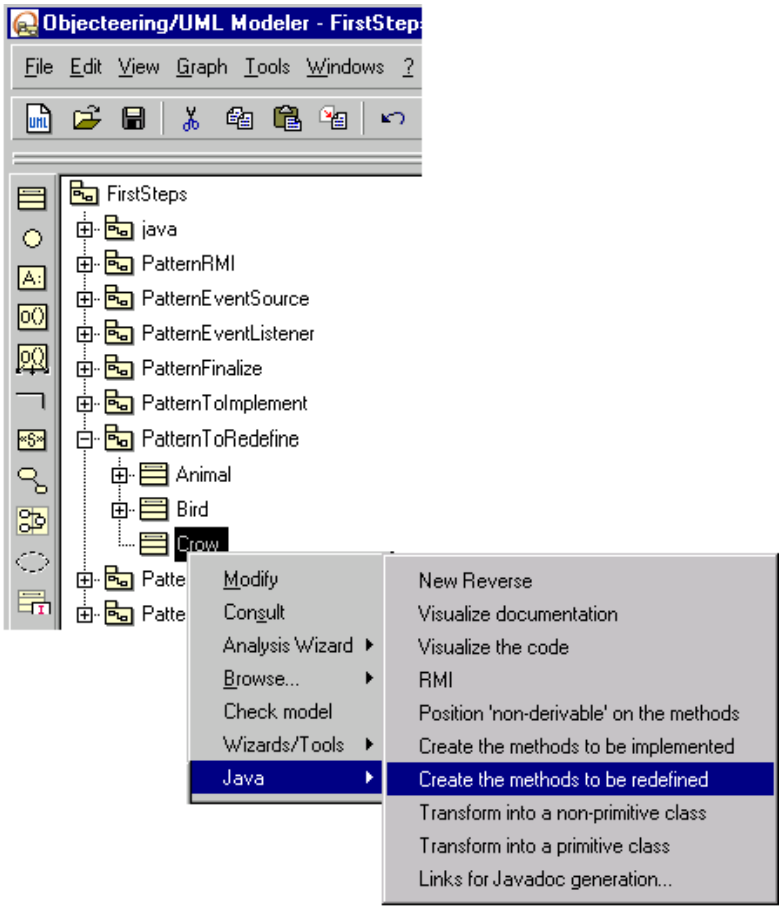


Figure 4-29. Calling the pattern

Final model

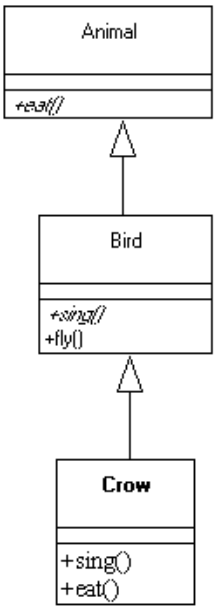


Figure 4-30. The model after transformation

Summary of the modifications made to the model

The "*Crow*" class has been enriched with the following operations:

- "*eat*" which redefines the operation of the same name of the "*Animal*" class
- "*sing*" which redefines the operation of the same name of the "*Bird*" class

However, the non-abstract "*fly*" operation has not been redefined.

Transforming a non-primitive class into a primitive class

Here we will be working with the "*PatternToPrimitive*" package, which will allow us to transform a non-primitive class into a primitive class, and parallel to this, to transform associations directed towards this class into attributes.

The "*PatternToPrimitive*" package contains:

- a "*Button*" non-primitive class
- a "*Window*" class with an association directed towards the "*Button*" class

We will apply the pattern to the "*Button*" class.

Initial model

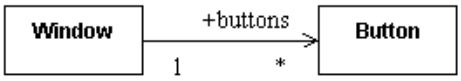


Figure 4-31. The initial model

Calling the pattern

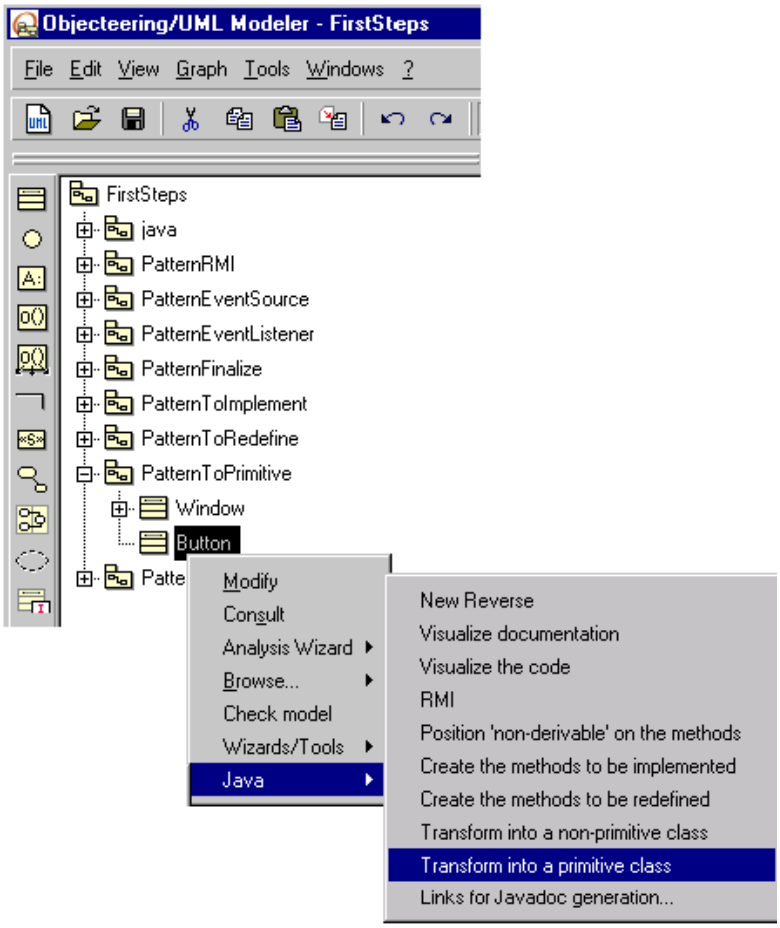


Figure 4-32. Calling the pattern

Final model

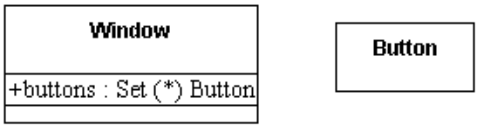


Figure 4-33. The model after transformation

Summary of the modifications made to the model

The "*Button*" class is now primitive. Furthermore, the "*Window*" class association has disappeared and been replaced by an attribute. The attribute takes the name of the erstwhile association role as its name, and the visibility of the association. The size of the attribute corresponds to the maximum multiplicity which the association had.

Transforming a primitive class into a non-primitive class

Here we will be working with the "*PatternToNonPrimitive*" package, which will allow us to transform a primitive class into a non-primitive class, and parallel to this to transform attributes which have this class as type into associations directed towards this class.

The "*PatternToNonPrimitive*" package contains:

- an "*Employer*" primitive class
- a "*Company*" class containing an "*Employee*" type attribute

We are going to apply the pattern to the "*Employee*" class.

Initial model

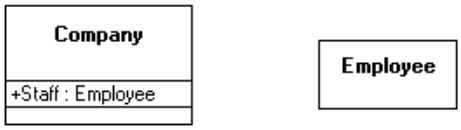


Figure 4-34. The initial model

Calling the pattern

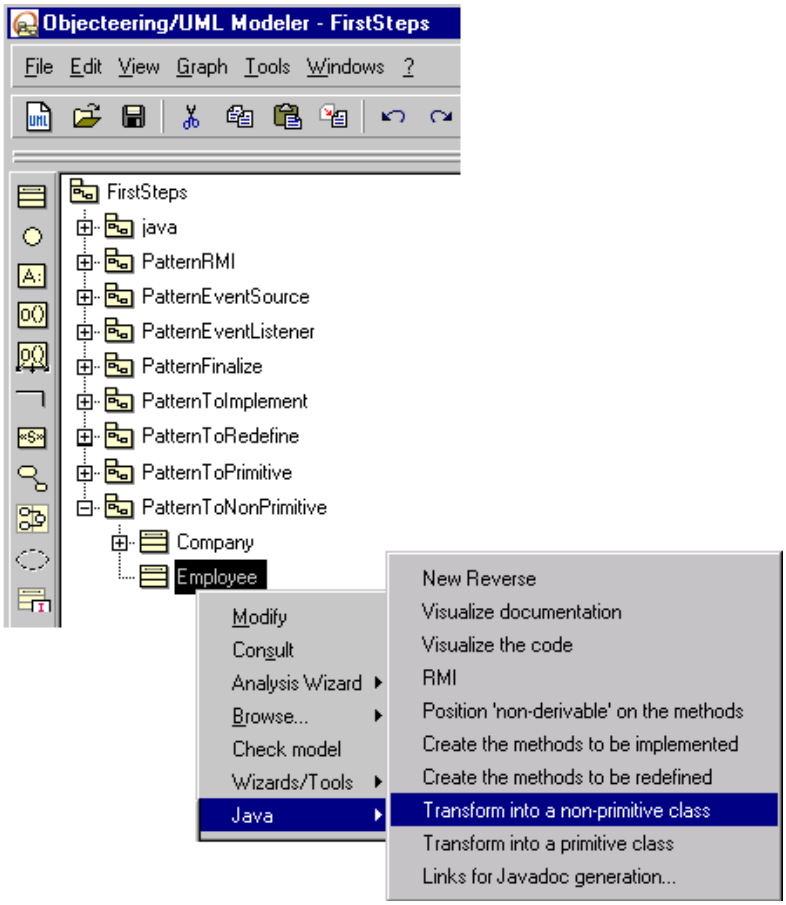


Figure 4-35. Calling the pattern

Final model

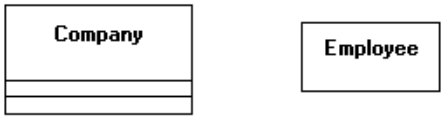


Figure 4-36. The model after transformation

Summary of the modifications made to the model

The "*Employee*" class is no longer primitive. Furthermore, the "*Employee*" type attribute of the "*Company*" class has disappeared and been replaced by a navigable association directed towards "*Employee*". The name of the role is the name which the attribute had, and the visibility of the navigable association link is the same as that of the erstwhile attribute. The minimum multiplicity of the association is 0. Its maximum multiplicity corresponds to the size the attribute had.

Chapter 5: Code generation

Code generation - Overview

Generating types and accessors

The mapping of types and the generation of accessors is possible using a special package called "*JavaTypes*". It is added to the "*predefinedTypes*" project during installation of the *Objecteering/Java* module.

A user may therefore modify the mapping of types and the generation of accessors, without having to re-define the standard module:

- either by modifying the "*JavaTypes*" package
- or by creating a similar package and by modifying the "*Types translation package*" parameter, presented in the "module configuration" dialog box. This parameter designates the package to be used (we recommend this solution)

For further details on the structure of the package and the way it is used by the generator, please refer to the "*Principles of type and accessor generation* " section in chapter 17, "*Customizing Java generation*", of this user guide.

The generation of Java code is based on a UML model, extended by notes and tagged values specific to Java, in order to generate all the code for Java classes.

This generation can be parameterized to a high level, using the following mechanisms:

- Java generation parameters provide general generation options
- the "*JavaTypes*" package allows you to set parameters used to generate types of base, accessors, etc
- the Java "*Generation template*" allows the integral redefinition of the generation through the *Objecteering/UML Profile Builder* tool

Code generation and model consistency checks

Code may be generated from the UML model regardless of whether consistency checks are active or inactive. However, when generation is launched, a message informs the user that he is in the process of generating code on a model which may potentially not conform to the UML modeling rules checked by Objecteering/UML (as shown in Figure 5-1).

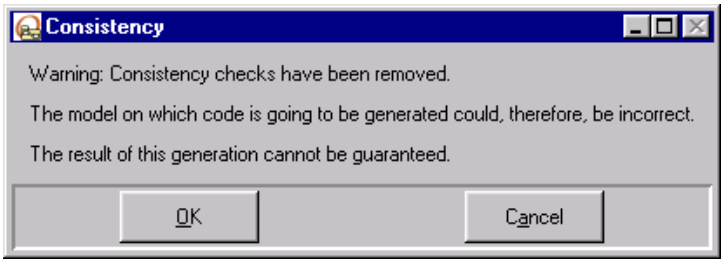


Figure 5-1. Message informing the user that consistency checks have been removed

Note: It should be noted that code generation in command line mode (please see objingcl) is assured, whatever the state of the consistency checks at the time of code generation.

The generation work product

Before generating Java code, it is necessary to create a Java generation work product (see Figure 5-2). This object can be created for:

- packages
- classes

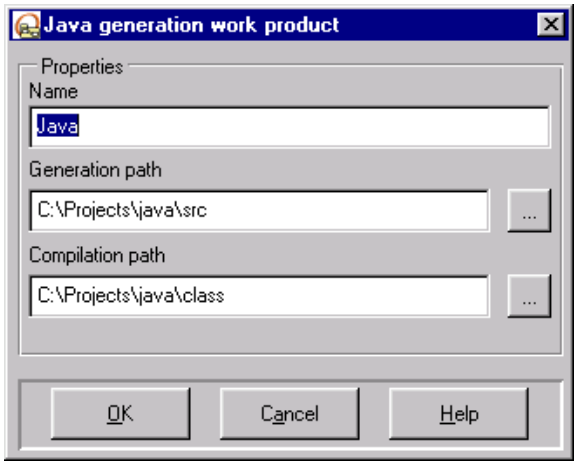


Figure 5-2. Dialog box for a Java generation work product

Key:

- "*Name*": This field is used to enter the name of the work product.
- "*Generation path*": This field is used to enter the root directory for the generation of *.java* files.
- "*Compilation path*": This field is used to enter the root directory for the generation of *.class* files

Note: Code generation can be launched for a package or a class directly through the "*Java*" tab of the properties editor, simply by clicking on the "*Generate*" button. If a Java generation work product already exists, code generation is carried out as normal. However, if no generation work product exists on the selected element, the "*Java generation work product*" dialog box is automatically opened, thus allowing you to easily create the work product. For further details on the properties editor and the *Objecteering/Java* module, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Code generation - Functions

Java generation work product commands

Objecteering/Java generation offers several features related to generation. These features are available via the context menu of a Java generation work product (see Figure 5-3).

Note: Code generation and compilation on a package can be directly launched via the "*Java*" tab of the properties, whilst for a class, code generation, visualization and compilation operations can be launched in the way.

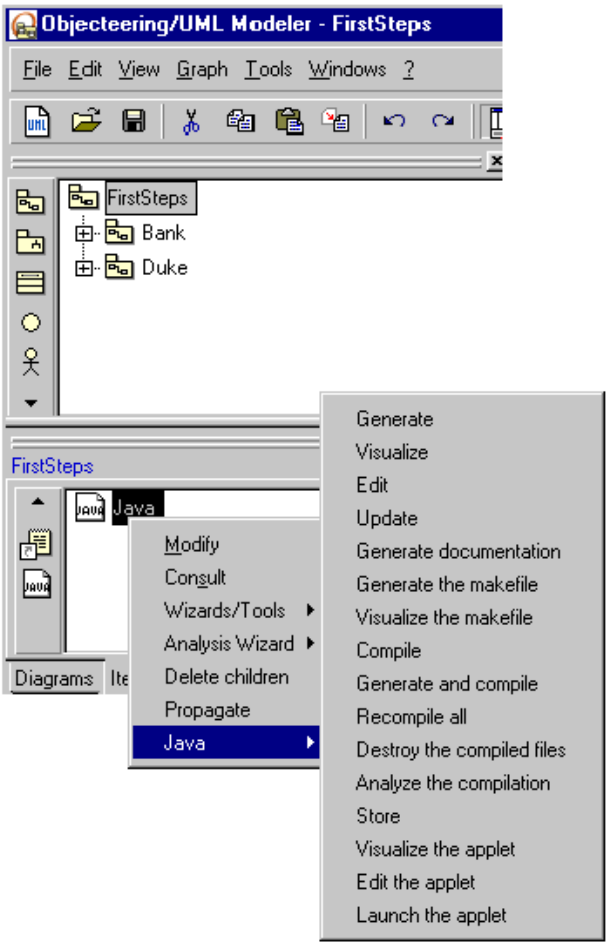


Figure 5-3. Commands available from a Java generation work product

These commands are detailed in chapter 3 of this user guide.

Code generation - Tagged value types

Overview

The tagged values provided by Objecteering/UML allow you to adapt Java semantics to a UML model, in order to generate all Java notions accurately. For example, the notion of native method in Java does not exist in UML. A *{JavaNative}* tagged value on a UML operation allows you to specify this on a model.

Note: Certain tagged values can be added to certain model elements directly through the "*Java*" tab of the properties editor. For further information, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Tagged values on a class

Name ...	Parameters ...	Role ...
JavaStatic	N/A	Used to declare the embedded class as being static.
JavaExtends	name of the mother class	Inherits non-modeled classes
JavaImplements	names of implemented interfaces	Implementation of non modeled interfaces
JavaImport	names of the imported classes and packages	Imports non-modeled packages and classes
JavaName	the class's Java name	During the generation phase, the parameter of this tagged value takes precedence over the modeling name.
JavaNonPublic	N/A	Indicates a non public class. This tagged value is only taken into account for a class contained within a class.
JavaExtern	N/A	Indicates a class for which code must not be generated.
JavaBean	N/A	Indicates a class as being a bean. This information is used for storage.
JavaBeanResource	resource file	indicates a resource to be stored at the same time as the class.
JavaNoAccessor	N/A	Non generation of the accessors on the class attributes and associations.

Tagged values on a package

Name ...	Parameters ...	Role ...
JavaRoot	N/A	Causes the generation of the package structure to begin from the annotated package.
JavaNoPackage	N/A	Causes the generation of the package structure to ignore the annotated package.
JavaImport	names of the imported classes and packages	Imports non-modeled packages and classes.
JavaName	the package's Java name	during the generation phase, the parameter of this tagged value takes precedence over the modeling name. This is particularly useful when developing packages, since it is used to develop a sub-branch of a larger modeling project.
JavaExtern	N/A	Indicates a package for which code must not be generated from its classes.
JavaBeanResource	resource file	Indicates a resource to be stored at the same time as the package's classes.

Tagged values on an operation

Name ...	Parameters ...	Role ...
JavaStrict	N/A	Causes the representation of the Java keyword "strictfp" for the generated operation.
JavaName	the operation's Java name	During the generation phase, the parameter of this tagged value takes precedence over the modeling name.
JavaSynchronized	N/A	Representation of the Java keyword : "*synchronized*" meant for the multi-threading.
JavaNative	N/A	Representation of the "*native*" Java keyword for the methods written in languages other than Java.
JavaThrownException	name of the exceptions that may be called	Lists the exceptions the operation can generate.
JavaNoInvariant	N/A	Means that the invariant method is not called for this operation.
JavaParentConditions	N/A	Allows the generation in the operation of the conditions (pre and/or post) of the redefined operation on the parent class.

Tagged values on an attribute

Name ...	Parameters ...	Role ...
JavaPublic	N/A	Generates the visibility of access methods as being public.
JavaWrapper	N/A	Uses the Wrapper class as base type (for example, Integer for int).
JavaName	the attribute's Java name	During the generation phase, the parameter of this tagged value takes precedence over the modeling name.
JavaLong	N/A	Carries out modifications in the case of an integer type attribute. This can also be applied
JavaShort	N/A	Carries out modifications in the case of an integer type attribute.
JavaByte	N/A	carries out modifications in the case of an integer type attribute.
type	name of the class of the types mapping and accessor generation package to be used for the attribute.*	Indicates a class other than the default class for the generations related to the attribute.
JavaTypeExpr	text for specifying the attribute's Java type	Generation of the types that cannot be represented in Objecteering/UML (ex : int [ ] [ ]).
JavaNoAccessor	N/A	Non-generation of the accessors.
JavaFilterAccessor	name of an accessor that must not be generated.	No generation of accessors given as parameters.
JavaGenerateAccessor	name of an accessor not generated by default and that should be generated	Generation only of accessors given as parameters.

Name ...	Parameters ...	Role ...
JavaFinal	N/A	Representation of the Java keyword "*final*".
JavaVolatile	N/A	Representation of the Java keyword "*volatile*".
JavaTransient	N/A	Representation of the Java keyword "*transient*".
JavaNoInvariant	N/A	Means that no invariants will be generated for the attribute in question.

The *Attribute* section of the table in the "*Type and Accessor Generation - Project Overview*" section of chapter 14 of this user guide resumes the values authorized as parameters for a {*type*} tagged value on an attribute.

	Simple attribute ...	Finite multiple attribute ...	infinite multiple attribute ...
vector	N/A	X	X (default)
array	N/A	X	X
hashtable	N/A	X	X
stack	N/A	X	X
hashSet	N/A	X	X
set	N/A	X	X
collection	N/A	X	X
linkedList	N/A	X	X
list	N/A	X	X
map	N/A	X	X
hashMap	N/A	X	X

These values correspond to the possibilities for the "*JavaTypes*" package. In the case of a simple attribute, the class used is the class named *<parameter value>SimpleAttribute*.

For a finite multiple attribute, the class used is the class named *<parameter value>FiniteAttribute*.

Where an infinite multiple attribute is concerned, the class used is the class named *<parameter value>MultipleAttribute*.

A user wishing to enrich the "*JavaTypes*" package or to write his package must, therefore, respect these naming rules.

Tagged values on a navigable association

Name ...	Parameters ...	Role ...
type	name of the class of the types mapping and accessor generation package which is to be used for the association *	Indicates a class other than the default class for the generations related to the association.
JavaNoAccessor	N/A	Non-generation of the accessors
JavaFilterAccessor	name of an accessor that must not be generated.	No generation of accessors given as parameters.
JavaGenerateAccessor	name of an accessor not generated by default and that should be generated	Generation only of accessors given as parameters.
JavaFinal	N/A	Representation of the Java keyword : "*final*".
JavaVolatile	N/A	Representation of the Java keyword : "*volatile*"
JavaName	the association's Java name	during the generation phase, the parameter of this tagged value takes precedence over the modeling name.
JavaPublic	N/A	Generates the visibility of access methods as being public.
JavaTransient	N/A	Representation of the Java keyword : "*transient*".
JavaTypeExpr	text for specifying the attribute's Java type	Generation of the types that cannot be represented in Objecteering/UML (ex : int [] []).
JavaNoInvariant	N/A	Means that no invariants will be generated for the association in question.

The *Association* table in the "*Type and Accessor Generation - Project Overview*" section of chapter 14 of this user guide resumes the values authorized as parameters for a {*type*} tagged value on an association.

	Simple attribute ...	**Finite multiple attribute ...**	**infinite multiple attribute ...**
vector	N/A	X	X (default)
array	N/A	X	X
hashtable	N/A	X	X
stack	N/A	X	X
hashSet	N/A	X	X
set	N/A	X	X
collection	N/A	X	X
linkedList	N/A	X	X
list	N/A	X	X
map	N/A	X	X
hashMap	N/A	X	X

These values correspond to the possibilities for the "*JavaTypes*" package. In the case of a simple association, the class used is the class named *<parameter value>SimpleAssociation*.

For a finite multiple association, the class used is the class named *<parameter value>FiniteAssociation*.

Where an infinite multiple association is concerned, the class used is the class named *<parameter value>MultipleAssociation*.

A user wishing to enrich the "*JavaTypes*" package or to write his package must, therefore, respect these naming rules.

Tagged values on a parameter

Name ...	Parameters ...	Role ...
JavaWrapper	N/A	Uses Wrapper as its base type (for example, Integer for int).
JavaByte	N/A	Carries out modifications for an *integer* type parameter.
JavaShort	N/A	Carries out modifications of an *integer* type parameter.
JavaLong	N/A	Carries out modifications for an *integer* type parameter.
JavaTypeExpr	text for specifying the parameter's Java type	Generation of the types that cannot be represented in Objecteering/UML (ex : int [] []).
type	name of the class for generating the parameter declaration *	Indicates a class other than the default class to generate a parameter declaration.

The *Parameter* table in the "*Type and Accessor Generation - Project Overview*" section of chapter 14 of this user guide resumes the values authorized as parameters for a {*type*} tagged value on a parameter.

	Simple attribute ...	Finite multiple attribute ...	infinite multiple attribute ...
vector	N/A	X	X
array	N/A	X (default)	X (default)
hashtable	N/A	X	X
stack	N/A	X	X
hashSet	N/A	X	X
set	N/A	X	X
collection	N/A	X	X
linkedList	N/A	X	X
list	N/A	X	X
map	N/A	X	X
hashMap	N/A	X	X

These values correspond to the possibilities for the "*JavaTypes*" package. In the case of an in parameter (or an out parameter), the class used is the class named *<parameter value>IOParameter*.

For a return parameter, the class used is the class named *<parameter value>ReturnParameter*.

A user wishing to enrich the "*JavaTypes*" package or to write his package must, therefore, respect these naming rules.

Tagged values on an assocation end

Name ...	Parameters ...	Role ...
JavaPublic	N/A	Generates the visibility of access methods as being public.
JavaName	the association's Java name	During the generation phase, the parameter of this tagged value takes precedence over the modeling name.

Tagged values on a data type

Name ...	Parameters ...	Role ...
JavaName	the data type's Java name	During the generation phase, the parameter of this tagged value takes precedence over the modeling name.

Code generation - Note types

Overview

Objecteering/UML notes are used to complete the UML model with texts expressed in Java syntax. Usually, the internal processing of a UML method is entered in Java in a "JavaCode" type note associated to a method's implementation.

Note: Certain notes can be added to certain model elements directly through the "*Java*" tab of the properties editor. For further information, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Type of notes on a class

The ... type of note	is used to ...
JavaHeader	insert a text before declaring the class
JavaMembers	insert a text in the class body
JavaBottom	insert a text after declaring the class
JavaDoc	add a comment to be used by javadoc
JavaCode	insert Java code used for the class invariant
JavaImport	generate and bring back imports in JavaImport notes, if the "*Use Java import notes to generate imports*" tickbox is checked, or if the {JavaNoImport} tagged value is present on the class in question. This can only be used in model-driven mode, and not in round-trip mode.

Type of notes on an operation

The ... type of note	is used to ...
JavaDoc	add a comment which will be used by javadoc.
JavaCode	provide Java code for implementing the operation.
JavaSuper	insert Java code before the pre-condition (especially for calling the builder of the parent class).
JavaReturned	insert Java code after the post-condition (especially for instructing the method return).

Type of notes on an attribute

The ... type of note	is used to ...
JavaDoc	add a comment to be used by javadoc

Type of notes on a navigable association

The ... type of note	is used to ...
JavaDoc	add a comment to be used by javadoc
JavaInitValue	set initialization for the association

Code generation - Stereotypes

Overview

Objecteering/UML defines stereotypes which allow you to designate certain objects as being concerned by the generation of Java code. In this way, a constraint concerning an operation will be used as a pre-condition if it carries the <<JavaPreConditions>> stereotype.

Note: Certain stereotypes can be added to certain model elements directly through the "*Java*" tab of the properties editor. For further information, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Stereotypes on a constraint

Name	is used to ...
JavaDocInvariant	designate a constraint as being an invariant expressed in JavaDoc. Such a constraint is only taken into account if it concerns a class.
Invariant	When the "description note processed as Javadoc" parameter is selected, the constraint is taken into accoutn in JavaDoc as for <<JavaDocInvariant>>.
JavaInvariant	designate a constraint as being an invariant expressed in Java. Such a constraint is only taken into account if it concerns a class.
JavaPreCondition	designate a constraint as being a pre-condition expressed in Java. Such a constraint is only taken into account if it concerns an operation.
JavaPostCondition	designate a constraint as being a post-condition expressed in Java. Such a constraint is only taken into account if it concerns an operation.

Java code and modeling correspondence

Introduction

Listed below are the usual concepts for Java development, together with the way in which they can be expressed in Objecteering/UML.

Notions on a package

Java notion ...	UML model mapping ...
generating a sub-package from a larger modeling project, without having the entire package hierarchy	the {JavaName} tagged value with the parameter containing the complete name on the package in question

Notions on a class

Java notion ...	UML model mapping ...
interface	«interface» stereotype on a class
abstract class	"*Abstract*" field of a class
final class	"*Leaf*" field of a class.
generalization	class generalization, or {*JavaExtends*} tagged value
interface implementation	implementation link, or {*JavaImplements*} tagged value
imports classes	use and reference links of its package, or {*JavaImport*} tagged value on the class and its package
class invariant	constraint on class, stereotyped «JavaInvariant»
inner class	*JavaMembers* type of note, or contained class not carrying the {JavaNonPublic} tagged value
non public class	*JavaBottom* note type, or contained class carrying the {JavaNonPublic} tagged value

Notions on an operation

Java notion ...	mapping in the model ...
public visibility	*Public* value of the *Visibility* field on an operation.
protected visibility	*Protected* value of the *Visibility* field on an operation.
private visibility	*Private* value of the *Visibility* field on an operation.
friendly visibility	*None* value of the Visibility field on an operation.
constructor	«create» stereotype on an operation
finalize() method	«destroy» stereotype on an operation
abstract	"*Abstract*" operation field.
static	"*Class*" operation field.
final	"*Cannot be specialized*" operation field.
synchronized	{*JavaSynchronized*} tagged value
native	{*JavaNative*} tagged value
exceptions generated by an operation	{*JavaThrownException*} tagged value. You can model the throwing of exceptions through a dependency link towards a class which derives from java.lang.Exception or towards a signal stereotyped <<Exception>>.

Notions on an attribute

Java notion ...	mapping in the model ...
visibility	same mapping process as for the methods.
static	*Class* field of an attribute.
final	{*JavaFinal*} tagged value

Code generation for an attribute or association

For an attribute or association in a class, a variable declaration and a number of access methods to this variable are generated. There are two groups of access methods, one for reading ("*getX()*","*cardX()*") and the other for modification ("*setX()*", "*appendX()*","*eraseX()*" and so on). In order to simplify presentation in the following table, "*get()*" refers to the read group of access methods, whilst "*set()*" indicates the modification group of access methods.

Code generation for an attribute takes into account the UML modeling of "*Visibility*", "*Access Mode*" and the *{public}* tagged value on the attribute, and is summarized in the following table.

For an association, there is no "Access Mode" in Objecteering/UML. Code generation is the same as the R/W column in the table.

	No access	**Read**	**Write**	**R/W**
+	+declare	+get() #declare	+set() #declare	+get() +set() #declare
{JavaPublic} and +	+declare	+get() +declare	+set() +declare	+get() +set() +declare
#	#declare	#get() -declare	#set() -declare	#get() #set() -declare
{JavaPublic} and #	#declare	+get() -declare	+set() -declare	+get() +set() -declare
-	-declare	-get() -declare	-set() -declare	-get() -set() -declare
{JavaPublic} and -	-declare	+get() -declare	+set() -declare	+get() +set() -declare
Package	-declare	get() -declare	set() -declare	get() set() -declare
{JavaPublic} and Package	declare	+get() declare	+set() declare	+get() +set() declare

Code generation for an enumeration

Since there is no "*enum*" in Java, a special class is generated for a UML enumeration.

For example, if a Color "*Enumeration*" with a number of "*EnumerationLiterals*" (red, green, blue) is defined, the code generated is as follows:

```
public class Color

{
    private int value;
    public static final Color red = new Color(0);
    public static final Color green = new Color(1);
    public static final Color blue = new Color(2);
    private Color(int code) {
        value = code;
    }
}
```

An Enumeration of Package is generated as a Class in the package, and an Enumeration of Class is generated as a non public class in the file of its owned class.

Note: If you choose to select the "*Compatible 432*" option, an Enumeration of Package generates no code, and an Enumeration of Class is generated as an Interface which is implemented by its owned class.

For example, for an Enumeration Color in the class C, the generated code is as follows:

```
public class C implements Color
{
}

interface Color
{
    static int red = 0;
    static int green = 1;
    static int blue = 2;
}
```

```
public java.lang.String toString ()
```

This returns the value of the enumeration.

```
Public int toInt ()
```

This returns the integer associated with the enumeration.

```
Public static Color fromString (String str)
```

This returns an enumeration from its string.

Chapter 6: Compilation

Compilation - Overview

Introduction

The compilation of a Java generation work product consists of producing the *.class* files from the *.java* files previously generated on this work product. The method used is to produce a makefile file.

Compilation can be run:

- directly from the "*Java*" tab of the properties editor on packages and classes
- from the context menu available on Java generation work products on classes and packages

For a package, the makefile produced recursively compiles all the package's classes and sub-packages.

For a class, the makefile produced compiles the class.

Compilation - Functions

Compilation services

Objecteering/Java generation provides several functions related to compilation, as detailed in the following table:

The ... command	is used to ...
Generate a makefile	generate a makefile file to compile *.java* files into *.class* files.
Visualize the makefile	visualize a generated makefile.
Compile	generate the *class* files from the *.java* files generated since the last compilation.
Generate and compile	generate a makefile file and compile.
Recompile all	force the generation of all the *.class*, even if the *.java* hasn't been regenerated.
Destroy the compiled files	destroy the *.class* files
Analyze the compilation	open a window with the compilation errors and offering direct access to the faulty modeling elements.

Access

These functions are available from the context menu of a Java generation work product detailed in chapter 3 in this manual.

Storage

Objecteering/Java provides a storage command, "*Store*", available in the context menu on Java generation work products and used to make "*.jar*" format archive files.

Where the application to be stored contains one main class (a class for which the "*Main class*" tickbox has been checked in the "*Java*" tab of the properties editor, or for which the "*Main*" tickbox has been checked in the class' dialog box), this class is generated in the manifest file of the .jar file in the following form:

```
Main-Class : name of main class
```

The same is true for applications containing more than one main class, except that the *Objecteering/Java* tool will manage this, by asking you to select the class you wish to be the main class (as shown in Figure 6-1 below).

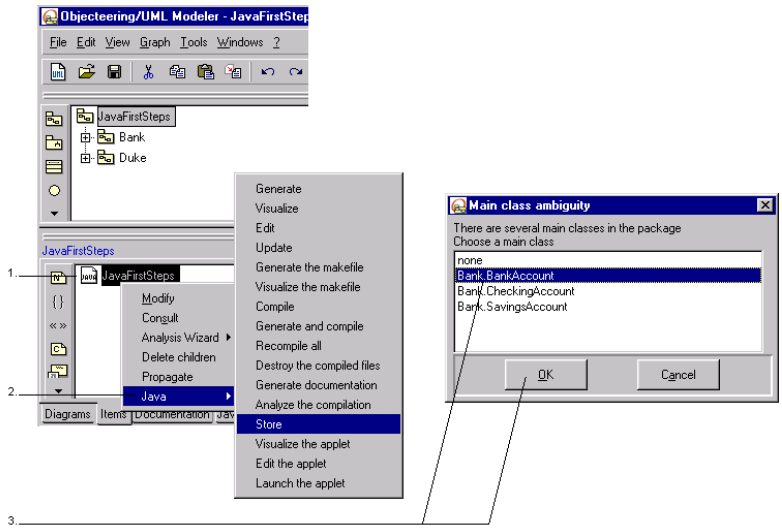


Figure 6-1. Managing main classes when running the "*Store*" command

Steps:

1 - Right-click on the Java generation work product for which generation and compilation has just been run.

2 - Run the "*Store*" command from the context menu which then appears.

3 - In the list of main classes, select the one you are interested in, and confirm by clicking on the "*OK*" button.

Chapter 7: Java documentation generation

Java documentation generation - Overview

Introduction

Java documentation generation consists of producing HTML files from the generated *.java* files. The JDK *javadoc* tool is used for this purpose. The produced documentation can be visualized using an HTML explorer.

Java documentation generation work product

The Java documentation generation work product is the same work product as the one used to launch code generation. It also has features related to documentation generation.

A work product can be created for a package or a class.

For ...	the documentation generation command ...
a package	generates documentation for all the package's classes
a class	generates documentation for the class only

Java documentation generation - Functions

Launching documentation generation

The *Objecteering/Java* module offers a command which allows you to launch documentation generation.

If the documentation exists, the HTML explorer is launched directly. Otherwise, documentation should be generated before opening the explorer.

Access

This feature is available in the Java generation work product context menu.

Java documentation generation - Note types

Overview

Javadoc notes are used in Objecteering/UML to enter information which will be included in Java documentation, at class, attribute, operation, operation parameter, operation return parameter and navigable association level. Javadoc notes can also be created on links stereotyped <<throw>>. A parameter allows the use of "*description*" type notes, as well as "*Javadoc*" type notes

In Objecteering/UML, a Javadoc note on...	**produces Java documentation ...**
a class	for the deduced class.
an attribute	for the attribute and its accessors.
an operation	for the deduced operation.
an operation parameter or return parameter	for the deduced parameter or return parameter, concatenated with the operation's documentation. Documentation generation options at module configuration leve are used to specify whether or not you wish to generate this information in your application's source code. If you check the tickboxes at module parameter level, the corresponding @param and @return markers will be generated in the Javadoc zone of the operation.
a navigable association	for the attribute and its accessors.
a link stereotyped <<throw>>	for the link. Documentation generation options are used to specify whether or not you wish to generate this information in your application's source code. If you check the tickbox, the corresponding @throws marker will be generated in the Javadoc zone of the operation.

Note 1: For classes, operations, attributes, associations, parameters and return parameters, javadoc type notes can be entered directly in the "*Java*" tab of the properties editor, simply by entering them in the "*Javadoc*" field. For further information on the properties editor, please refer to the "*The properties editor and the Java module*" section in chapter 2 of this user guide.

Note 2: For further information on @param, @return, @throws and @see markers, please refer to the related Javadoc documentation provided by Sun.

Java documentation generation - Javadoc @param markers

Customizing Javadoc documentation

Javadoc @param markers are used in the generation of Java documentation for parameters which have Javadoc notes. If you want the information contained in Javadoc notes on parameters to appear in the generated Java documentation, you must specify this at module parameter level (as shown in Figure 7-1 below).

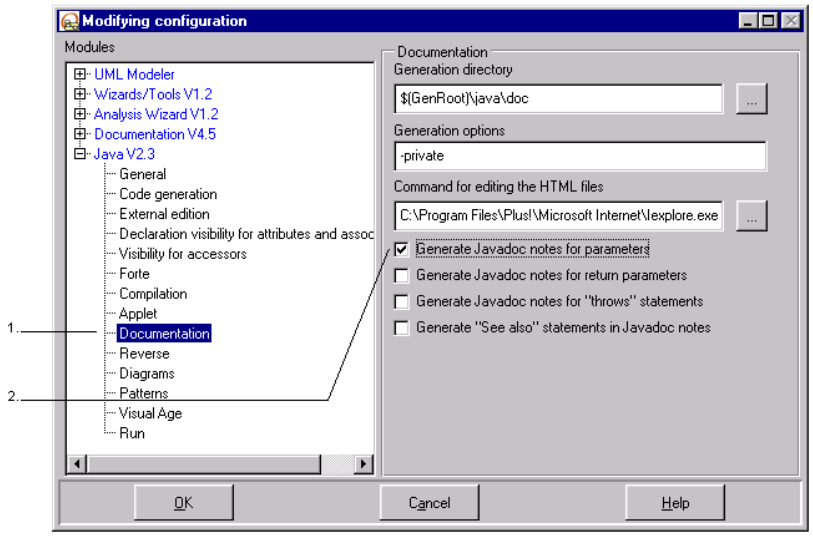


Figure 7-1. Parameterizing generation of Javadoc notes on parameters in Java documentation

Steps:

1 - Select the "*Documentation*" set of Java module parameters.

2 - Check the "*Generate Javadoc notes for parameters*" tickbox.

Javadoc notes entered for parameters will now be generated in Java documentation.

Entering Javadoc notes on parameters

To enter Javadoc notes on parameters, the "*Javadoc*" field in the "*Java*" tab of the properties editor is used (as shown in Figure 7-2).

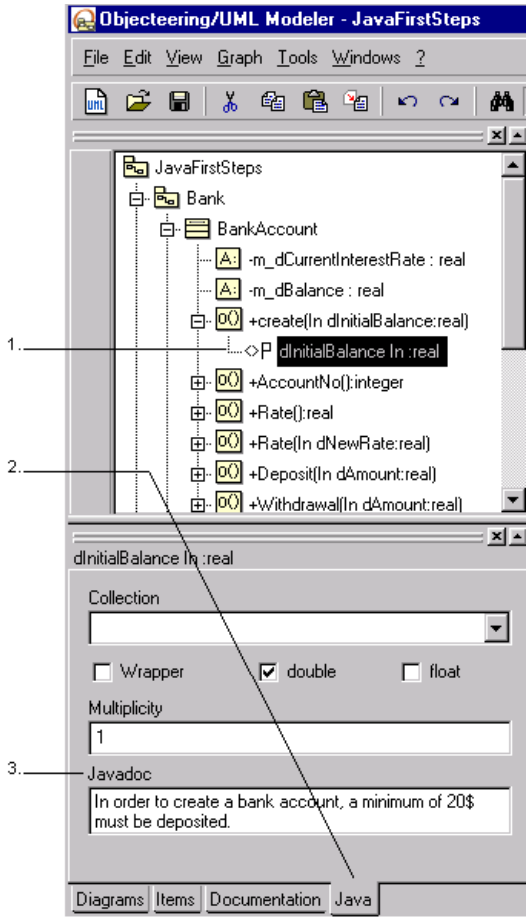


Figure 7-2. Entering a Javadoc note for a parameter

Steps:

1 - In the explorer, select the parameter for which you wish to enter a Javadoc note.

2 - Click on the "*Java*" tab of the properties editor.

3 - In the "*Javadoc*" field, enter the contents of the Javadoc note.

The result of the operations shown in Figures 7-1 and 7-2 is shown in Figure 7-3 below.

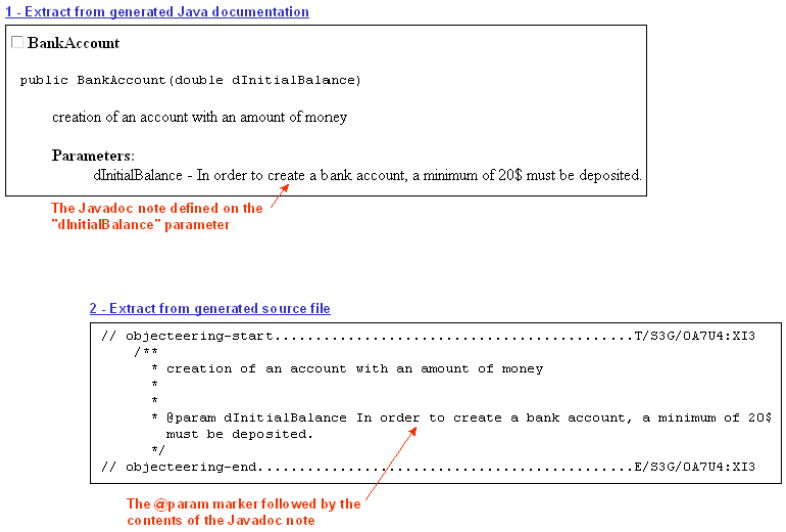


Figure 7-3. Extracts from generated Javadoc documentation and the generated source file

The information you entered in the Javadoc note zone of the properties editor is presented in the generated Java documentation. It is introduced by the "*Parameters*" title, followed by the name of the parameter and then its description.

The same information is present in the generated source file, preceded by first the @param marker and then the name of the parameter.

Java documentation generation - Javadoc @return markers

Customizing Javadoc documentation

Javadoc @return markers are used in the generation of Java documentation for return parameters which have Javadoc notes. If you want the information contained in Javadoc notes on return parameters to appear in the generated Java documentation, you must specify this at module parameter level (as shown in Figure 7-4 below).

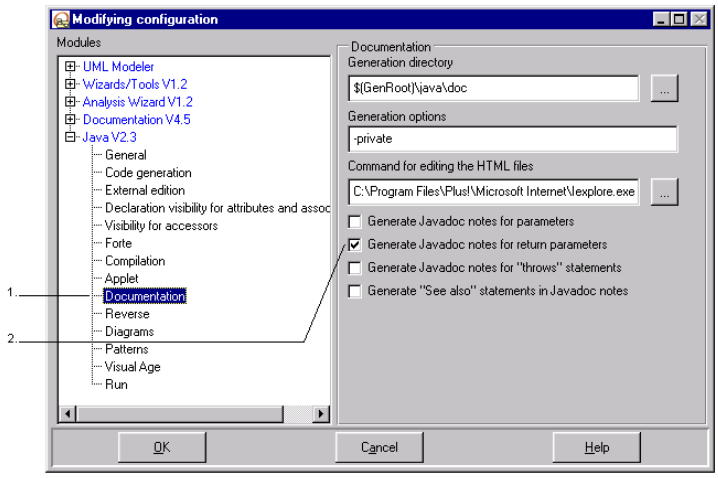


Figure 7-4. Parameterizing generation of Javadoc notes on return parameters in Java documentation

Steps:

1 - Select the "*Documentation*" set of Java module parameters.

2 - Check the "*Generate Javadoc notes for return parameters*" tickbox.

Javadoc notes entered for return parameters will now be generated in Java documentation.

Entering Javadoc notes on return parameters

To enter Javadoc notes on return parameters, the "*Javadoc*" field in the "*Java*" tab of the properties editor is used (as shown in Figure 7-5).

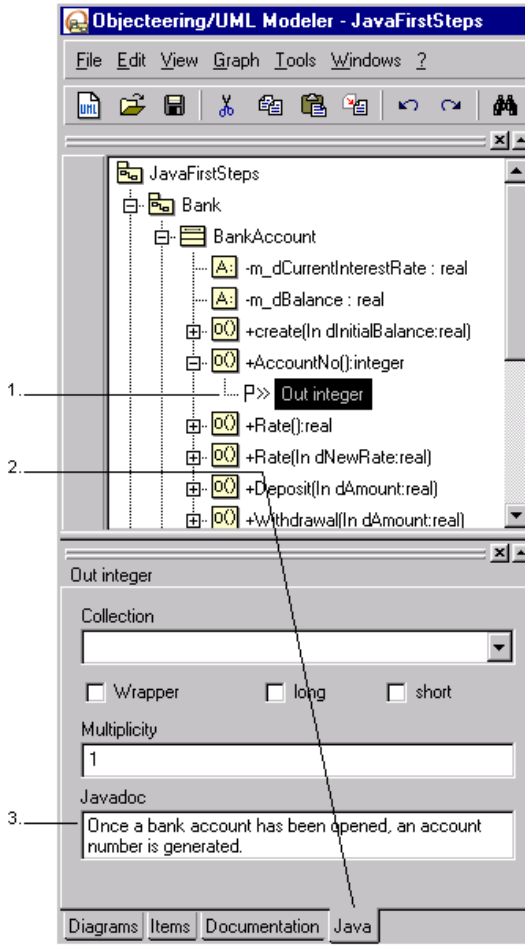


Figure 7-5. Entering a Javadoc note for a return parameter

Steps:

1 - In the explorer, select the return parameter for which you wish to enter a Javadoc note.

2 - Click on the "*Java*" tab of the properties editor.

3 - In the "*Javadoc*" field, enter the contents of the Javadoc note.

The result of the operations shown in Figures 7-4 and 7-5 is shown in Figure 7-6 below.

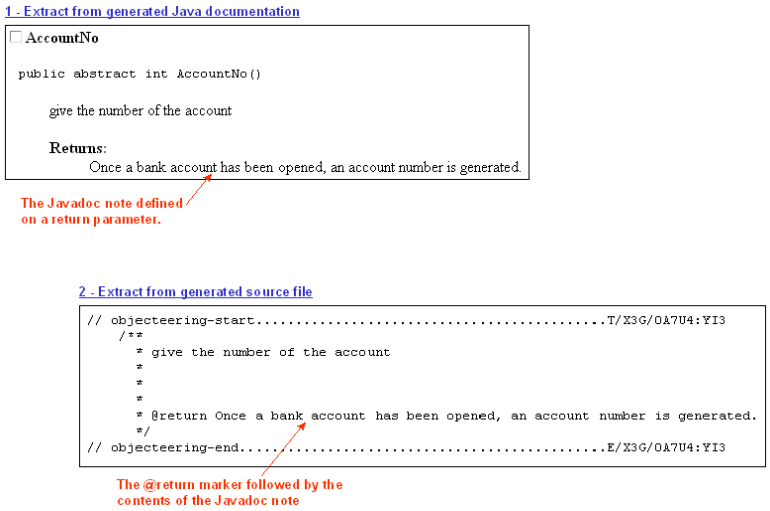


Figure 7-6. Extracts from generated Javadoc documentation and the generated source file

The information you entered in the Javadoc note zone of the properties editor is presented in the generated documentation. It is introduced by the "*Returns*" title, followed by the description of the return parameter.

The same information is present in the generated source file, preceded by the @return marker.

Java documentation generation - Javadoc @throws markers

Customizing Javadoc documentation

Javadoc @throws markers are used in the generation of Java documentation for exceptions which have Javadoc notes. If you want the information contained in Javadoc notes on exceptions to appear in the generated Java documentation, you must specify this at module parameter level (as shown in Figure 7-7 below).

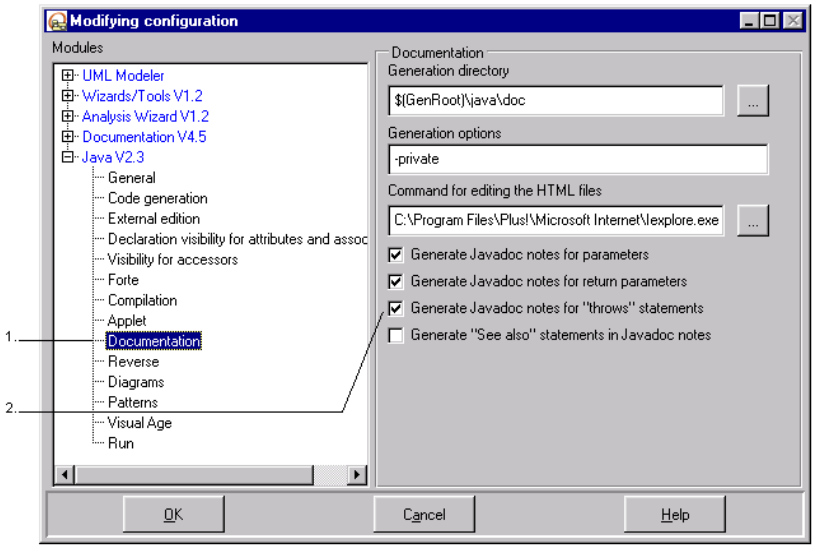


Figure 7-7. Parameterizing generation of Javadoc notes on exceptions in Java documentation

Steps:

1 - Select the "*Documentation*" set of Java module parameters.

2 - Check the "*Generate Javadoc notes for "throws" statements*" tickbox.

Javadoc notes entered for thrown exceptions will now be generated in Java documentation.

Entering Javadoc notes on exceptions

To enter Javadoc notes on exceptions, the "*Notes*" tab of the thrown exception's dialog box is used (as shown in Figure 7-8).

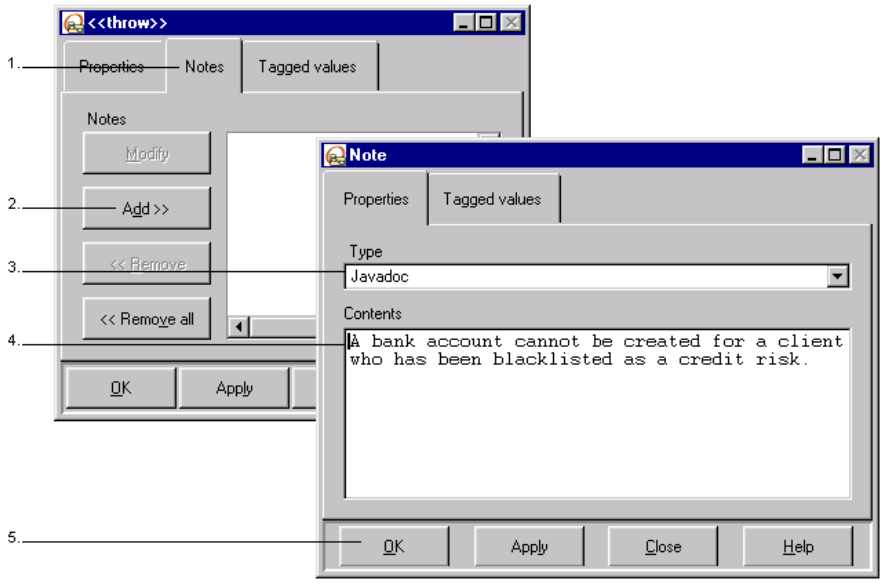


Figure 7-8. Entering a Javadoc note for a thrown exception

Steps:

1 - After opening the thrown exception's dialog by double-clicking, click on the "*Notes*" tab.

2 - Click on the "*Add*" button to create a new note.

3 - Select the "*Javadoc*" type of note.

4 - Enter the contents of the Javadoc note.

5 - Confirm by clicking on "*OK*".

The result of the operations shown in Figures 7-7 and 7-8 is shown in Figure 7-9 below.

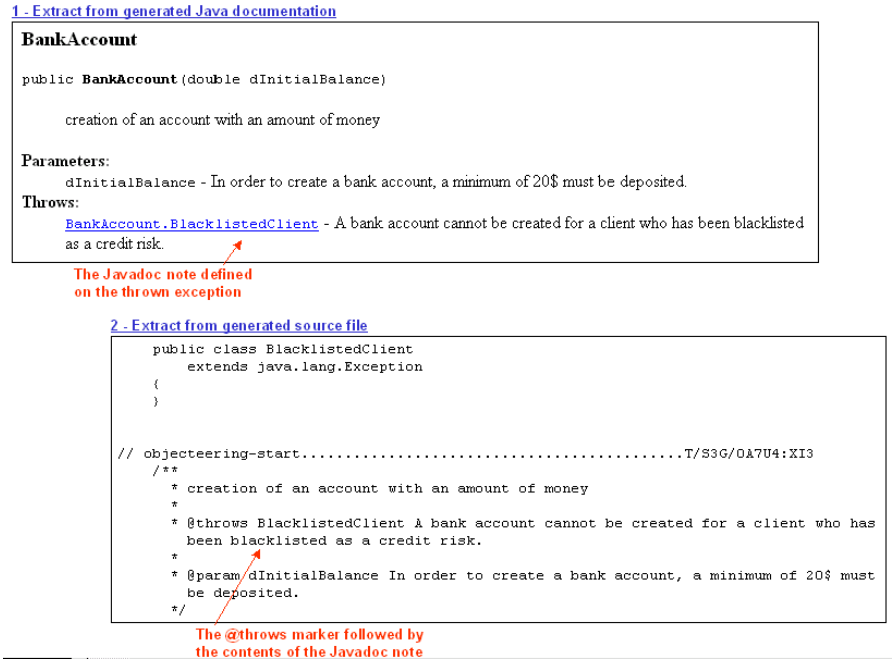


Figure 7-9. Extracts from generated Javadoc documentation and the generated source file

The information you entered in the Javadoc note is presented in the generated Java documentation. It is introduced by the "*Throws*" title, followed first by the elements involved in the thrown exception and then by its description.

The same information is present in the generated source file, preceded by the @throws marker.

Java documentation generation - Javadoc @see markers

Customizing Javadoc generation

Javadoc @see markers are used in the generation of "*See also*" links in Java documentation. If you want "*See also*" statements to appear in Javadoc notes and therefore in the generated Java documentation, you must specify this at module parameter level (as shown in Figure 7-10 below).

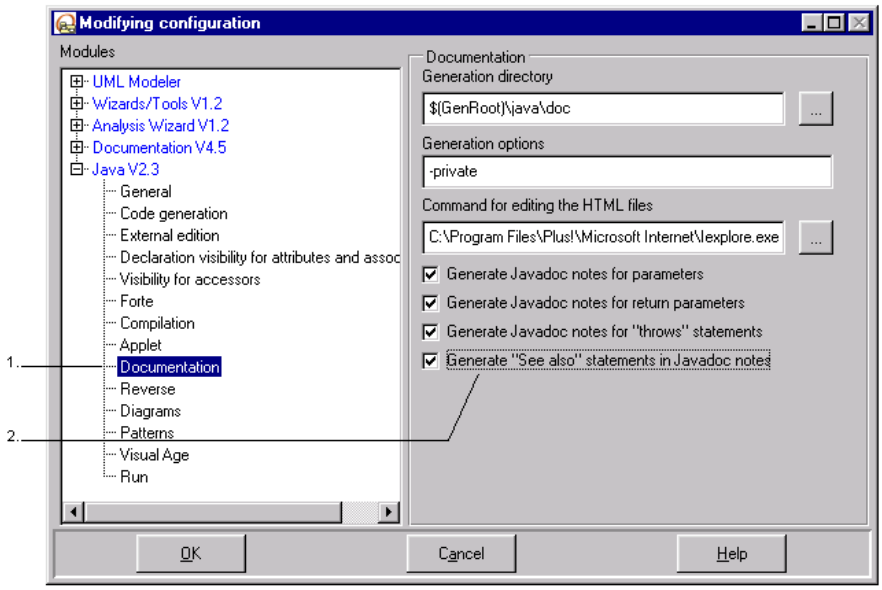


Figure 7-10. Parameterizing generation of "*See also*" statements in Javadoc notes

Steps:

1 - Select the "*Documentation*" set of Java module parameters.

2 - Check the "*Generate "See also" statements in Javadoc notes*" tickbox.

Modeling see also links

Javadoc @see markers are used to create "*See also*" links in the documentation created by Javadoc. Using *Objecteering/Java*, these links are modeled through a dialog box which can be accessed via the context menu available on classes and operations.

By using the drag and drop feature, you can link a class or an operation to a package, class, operation, attribute or association. For every linked element, it is possible to add a description text.

Where operations are concerned, you can choose to display the type of the link. Possible types are:

- "*Generate the object name*": this is used to generate only the name of the operation.
- "*Generate the object name with argument types*": this is used to generate the name of the operation and the list of types of the parameters.
- "*Generate the object name with argument types and names*": this generates the name of the operation and the list of types and names of the parameters.

The "*Add*", "*Apply*" and "*Remove*" buttons are used to add a new "*See also*" link, modify an existing "*See also*" link or delete an existing "*See also*" link.

Figure 7-11 below shows an example of the creation of a new see also link.

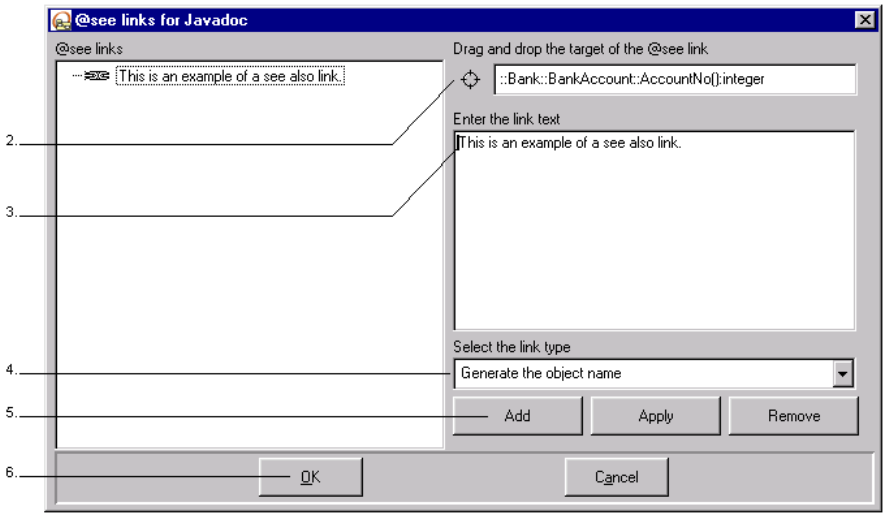


Figure 7-11. Creating a new "*See also*" link

Steps:

1 - Select the class or operation for which you want to create a see also link by right-clicking, and run the "*Links for Javadoc generation*" command.

2 - From the explorer, drag and drop the element you wish to link the selected class or operation to.

3 - Enter a description of the new see also link.

4 - Select the link type. By default, this is set to "*Generate the object name*".

5 - Click on "*Add*" to confirm your creation and add the new link to the left hand "*@see links*" column.

6 - Confirm by clicking on "*OK*".

The result of this operation is shown in Figure 7-12 below.

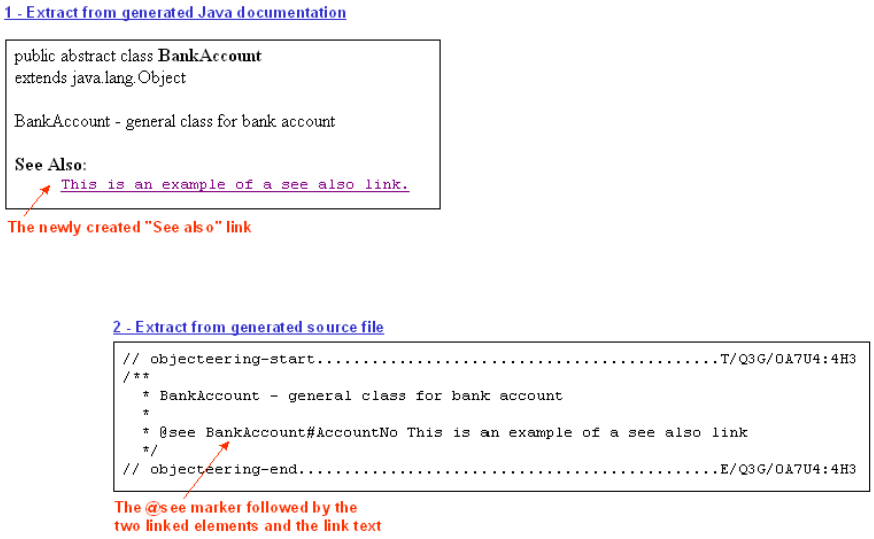


Figure 7-12. Extracts from generated Javadoc documentation and the generated source file

The information you entered in the "*Enter the link text*" zone of the "*@see links for Javadoc*" window is presented in the generated Java documentation. It is introduced by the "*See Also*" title, followed by the text entered.

The same information is present in the generated source file, preceded by first the @see and then by the linked elements.

Chapter 8: Reverse

Reverse - Overview

Introduction

Java class libraries are essential elements in Java programming. The reverse tool allows the integration of these basic libraries into Objecteering/UML, by recreating Objecteering/UML classes from Java files.

During this operation, classes and packages modeled are rebuilt from the sources available in the class libraries. In this way, the user can use library elements within his own application.

Reversable sources

The Objecteering/UML Java reverse uses *.class* files to build class structure. *.class* files can either exist independently or can be compressed in *.zip* files or *.jar* files. If you are using a JDK 1.1, you can reverse only *.class* files which are independent or compressed in *.zip* files. However, where a JDK 1.2 or 1.3 is being used, you can also reverse *.class* files which are compressed in *.jar* files.

Configuring reverse operations

Java reverse operations are based on certain module parameters, defined and configured through the  ("*Modify module parameter configuration*") icon. For further information on these parameters, please refer to the "*Configuration window*" section in chapter 17 of this user guide, which contains a paragraph detailing the "*Reverse*" group of parameters.

Using reversed classes

Integrating these software components into the repository allows:

- the visualization of the reversed classes (structure of the packages, attributes, associations, operations)
- the specialization of the reversed classes and the redefinition of their methods
- the visualization of documentation and code associated with the reversed elements, if they are accessible
- the continuation of the development of your UML modeling project in Objecteering/UML

Status of reversed classes

Whether or not the reversed class is taken into account depends on the {*JavaExtern*} tagged value associated with the class. Reverse levels are decided by the "*ReverseJavaCode*" and "*ReverseJavaDoc*" parameters and the available java source file.

- If the java source file is available, the java code is reversed when the "*ReverseJavaCode*" parameter is selected. If this is not the case, the reversed class is set with the {*JavaExtern*} tagged value. In any case, parameter names, as well as attribute and association initializations, are created correctly.
- If the java source file is not available, the reversed class is set with the {*JavaExtern*} tagged value. The reverse can only take place when the "*ReverseJavaCode*" parameter has not been selected. In this case, information on the name of the operation parameters is not available in the repository. The operation parameters are named *p* followed by the number of their location in the parameter declaration. The first operation parameter will, therefore, be called p0.
- Java documentation information is reversed only when the "*ReverseJavaDoc*" parameter is selected and the java source file is available.

Code reverse and model consistency checks

A model can be reversed from files external to Objecteering/UML regardless of whether consistency checks are active or inactive. We recommend, however, that removable consistency checks be deactivated, since if they are left active, the reverse operation may fail before a model is produced.

When the reverse is launched, a message informs the user that consistency checks are active and that construction of the model, which may potentially not conform to the UML modeling rules checked by Objecteering/UML, may be refused (as shown in Figure 8-1).

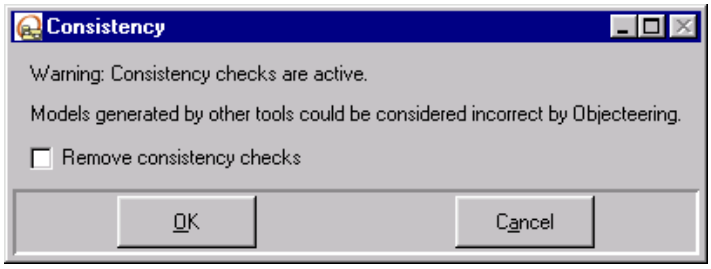


Figure 8-1. Message informing the user that consistency checks are active

If the user wishes to deactivate consistency checks, he should simply check the "*Remove consistency checks*" box.

After the reverse, we advise that you reactivate all consistency checks, by clicking on the  ("*Switch on/Switch off consistency checks*") icon, and correct any problems in consistency.

Note: It should be noted that code reversal in command line mode (please see objingcl) is assured, whatever the state of the consistency checks when the reverse command is run.

Warning

A class created through reverse receives an identifier just like an object created in an explorer or in a graphic editor.

If two people reverse the same class in two different UML modeling projects, Objecteering/UML will consider there to be two different objects.

To avoid losing the links towards the reversed classes during the import, it is necessary to run the reverse in a reference modeling project from which everyone imports the used classes.

Reverse - Functions

Reverse features

The *Objecteering/Java* module offers several features related to the reverse.

- the reverse itself on a package
- re-launching the reverse on a class already reversed (with different options)
- re-launching the reverse on a package which has already reversed classes (with different options)
- visualizing the documentation on a reversed class
- visualizing the code on a reversed class

Launching reverse operations

The reverse command can be run from any package, as shown in figure 8-2 below.

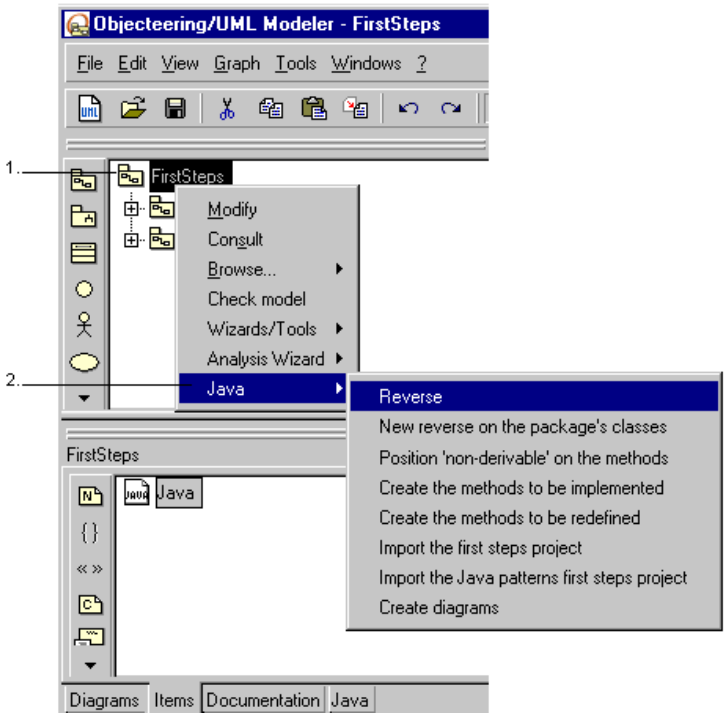


Figure 8-2. Running the reverse command on a package

Steps:

1 - Select the package on which code generation is to be run in the explorer, by right-clicking.

2 - Run the "*Java/Reverse*" commands from the context menu which then appears.

Selecting classes to be reversed

A selection window then appears, in which the user selects those classes he wishes to reverse (as shown in Figure 8-3).

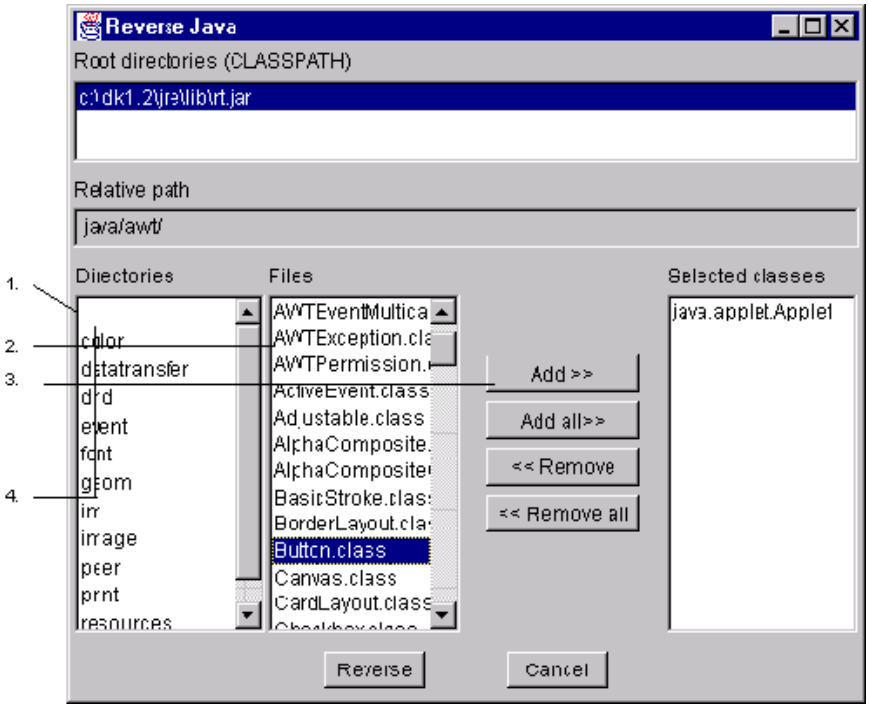


Figure 8-3. Window used to select those classes which are to be reversed

Steps:

1 - Select the "*java*" directory

2 - Select the "*applet*" directory

3 - Select the "*Applet.class"* file.

4 - Click on the "*Add*" button.

Note 1: The user can browse packages and select those classes to be reversed, using the "*Add*" and "*Add all*" buttons. Classes belonging to different packages may be selected.

Note 2: Clicking on the "*Reverse*" button triggers the reverse of the selected classes.

Re-running the reverse operation

If a package has already been reversed and you wish to update the model (after further developing the Java files, for example), you should simply run the "*New reverse on the package's classes*" command on the package in question. This command (illustrated in Figure 8-4) directly triggers the reverse operation on the package's classes, without displaying the selection window shown in Figure 8-3.

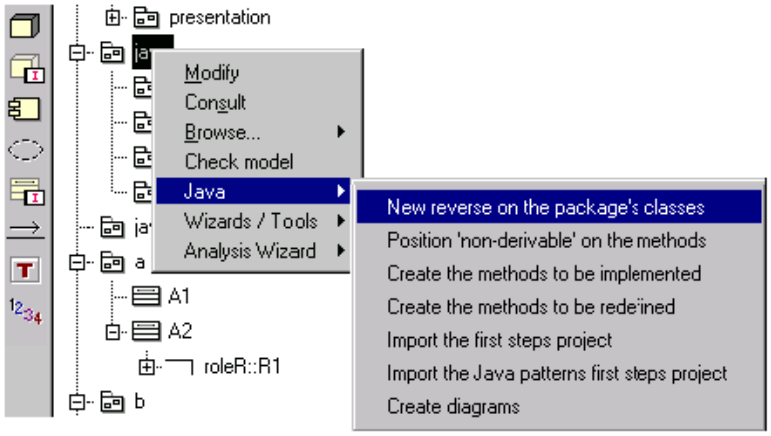


Figure 8-4. The "*New Reverse on the package's classes*" command

Visualizing information related to a class

There are two commands available in a class context menu (as shown in Figure 8-5).

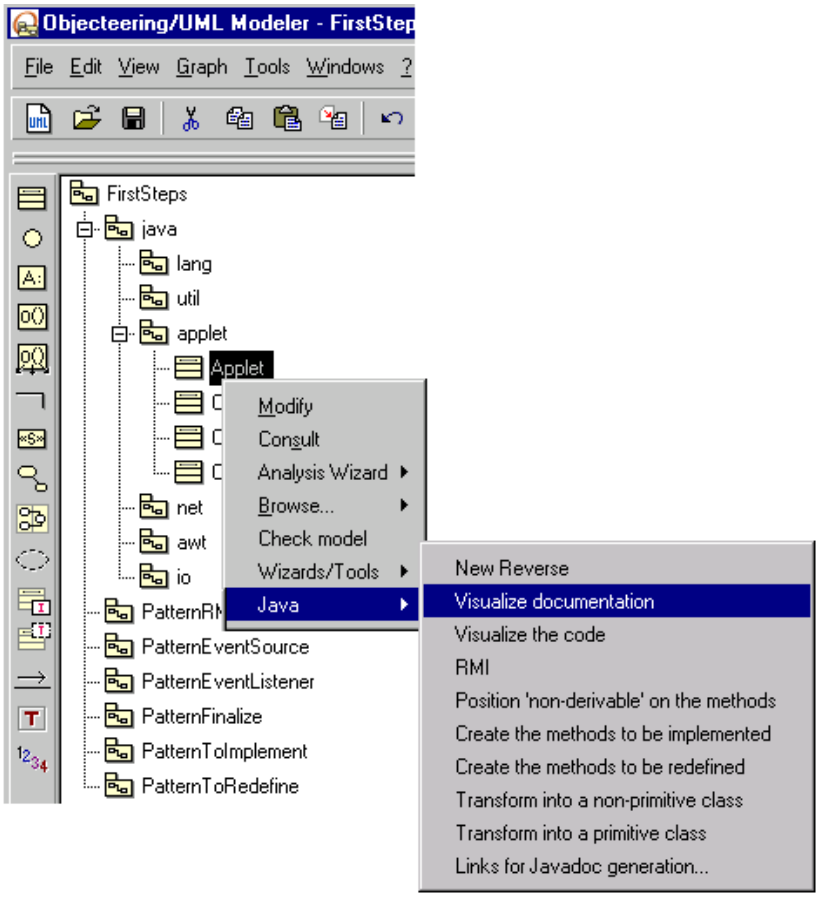


Figure 8-5. Visualizing the documentation of the "*Applet*" class

The ... command	opens ...
Visualize documentation	an HTML editor containing the original HTML documentation related to the current class.
Visualize the code	a dialog box containing the Java code corresponding to the current class.

Steps involved in the reverse of a model

Operating mode

From any package, select the "*Java/Reverse*" commands from the context menu. After choosing to window then appears.

- The list of *source directories* indicates the directories from which the user can select the classes to be reversed. This list is filled by the configuration parameter, whose name is "*Class files paths*".
- The user can browse these directories. However, only directories containing classes that can be selected are displayed in the explorer. Only the files with the *.class* extension appear.
- Clicking on a class selects it in the file explorer. Clicking on "*Add*" adds it to the list of selected classes. Multiple selection is possible.
- Clicking on the paths of the source directories automatically positions the current *path* on the selected path.
- Selecting the "*Reverse*" button launches the import of the selected classes.

Classes used

A reversed class can have attributes, associations or operation parameters, whose type corresponds to the non-reversed class. If so, these classes are created empty in Objecteering/UML, and can be reversed later.

Java/Objecteering correspondence

Certain relationships between Java and Objecteering/UML are direct, others require the creation of tagged values, shown in the list below.

Metaclass ...	Java modifier ...	tagged value ...
Attribute, AssociationEnd	final	JavaFinal
N/A	transient	JavaTransient
N/A	volatile	JavaVolatile
Method	synchronized	JavaSynchronized
N/A	native	JavaNative

The constructors of a Java class, which have the name of this class in Java, are created with the <<*create*>> stereotype in Objecteering/UML.

As the passing mode of the parameters in Java is through reference, all the parameters of an operation, except the return parameter, have the In/Out type. A java class may contain fields. Their type decides the reversed field which will be created as an attribute or an association. When the type is primitive java type or string, the field corresponds to an attribute. Otherwise, it is an association.

Correspondence with the Java primitive types

When you import an attribute or a parameter with a primitive Java type (there are eight of them), correspondence is established with Objecteering/UML's predefined types. Certain Java type characteristics require the addition of a tagged value.

Primitive Java type ...	type in the repository ...	tagged value ...
boolean	boolean	N/A
char	char	N/A
int	integer	N/A
short	integer	JavaShort
long	integer	JavaLong
byte	integer	JavaByte
float	real	N/A
double	real	JavaLong

Restrictions

When a type declaration cannot be expressed in Objecteering/UML, it is encapsulated in a {*JavaTypeExpr*} tagged value and the attribute, association or parameter type gets the *undefined* value. This {*JavaTypeExpr*} tagged value is not included in the signature of a method. Therefore, if an "*m1*" method is defined containing the only parameter which has a type that cannot be expressed (a *JavaTypeExpr*), the creation of a second "*m1*" method with only one *JavaTypeExpr* parameter will not be accepted by Objecteering/UML. This situation, however, is very rare.

Automatic diagram creation

If you check the "*Automatically create diagrams on reverse*" tickbox, the reverse operation will automatically create, at package level, diagrams of generalizations, associations, actors, use cases and package relations. Each time you run the "*New reverse on the package's classes*" command, new diagrams will be created after every round trip cycle, with the old diagrams being renamed using the "*old*" suffix.

Chapter 9: Choosing the functional mode

Java functioning modes

Introduction

Two functioning modes are available with the *Objecteering/Java* module:

- Model driven, which is based on the total generation of declarative code from the model itself. Application code (in other words, the programming of operations) is modeling by notes, or typing in marked zones. This mode does not concern reverse engineering.
- Round trip, which combines reverse engineering and code generation. Code is generated in exactly the same way as with the model driven mode. The user can freely modify the code nearby in the generated code or work in his favorite IDE. The model is updated by running the "*Reverse*" command.

Configuring the functioning mode

The Java functioning mode is configured through the standard Objecteering/UML configuration interface, which is launched either by selecting *the* "*File/Edit Configuration/Modify Configuration*" menu or by clicking on the  "*Modifies the configuration of module parameters*" icon. The generation mode is selected through the definition of the "*Generation mode*" field (as shown in Figure 9-1).

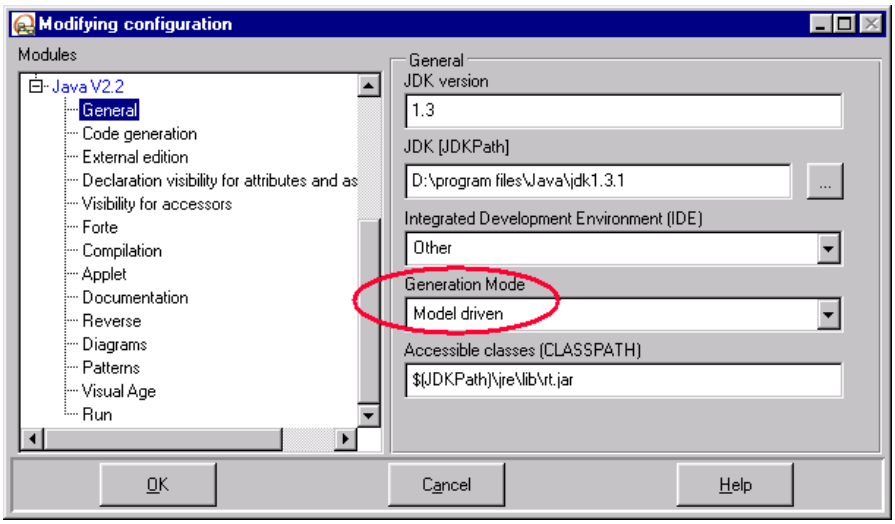


Figure 9-1. Configuring the generation mode

Model driven mode

Principles of model driven engineering

Model driven engineering is the default Objecteering/UML functioning mode and is based on complete generation of declarative code from the UML model.

Marked zones are carefully inserted into the generated Java code, with each of these zones corresponding to a note which can be added to the model (see Figure 9-2). Application code (virtually reduced to the programming of operations) is written in these marked zones (using with the aid of an external editor or an IDE).

Once the zones have been completed, their contents can be transferred back into the model in the form of corresponding notes. This operation can be carried out as many times as possible.

The model driven engineering mode does not, therefore, involve reverse engineering. All "declarative" modifications, such as, for example, the addition of a new attribute, are in fact model modifications and as such must be carried out within Objecteering/UML, before subsequently regenerating the code.

Certain mechanisms exist to guarantee that the regenerated code does not write over any modifications made outside Objecteering/UML.

Thus, the entire application can be fine-tuned outside Objecteering/UML, as long as the model itself is not modified.

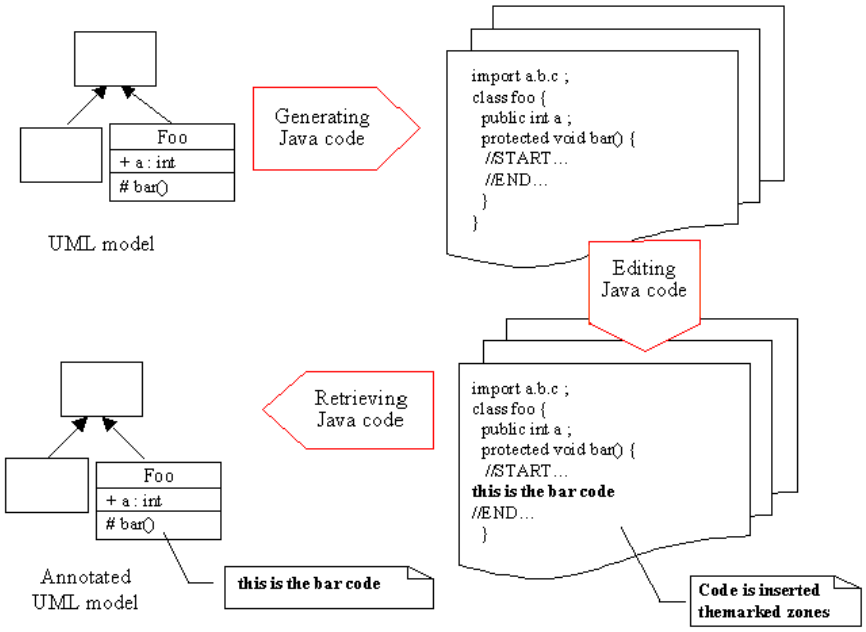


Figure 9-2. Principles of model driven engineering

Advantages of the model driven engineering mode

The main advantage of this mode is that the code always corresponds to the model. In fact, it is only possible to modify declarations by modifying the model itself.

Other advantages are:

- A relatively small modification to the model can lead to significant modifications at declarative code level. These code modifications are automatically taken into account by the code generator.
- Total model/code consistency leads to up-to-date documentation, as well as other generation work products (metrics, SQL, …), which are consistent with the code at no additional cost.

Drawbacks of the model driven engineering mode

The obvious drawback of the model driven engineering mode is that the user is obliged to come back to Objecteering/UML after every modification made to the model, in order to regenerate the code.

Round trip mode

Principles of round trip engineering

Round trip engineering is a functioning mode which combines code generation and reverse engineering. Code is generated in exactly the same way as with model driven engineering. Markers, which are used during the reverse engineering phase to differentiate between code written by the user and code generated by Objecteering/UML, are also generated.

Application code (virtually reduced to programming operations if the model is complete) is written in these marked zones (with the aid of an external editor or an IDE).

Unlike the model driven mode, the round trip mode allows the model to be directly modified at code level. The code is then fully reversed, in order to update the model.

In the example shown in Figure 9-3, the String type attribute appears in the model.

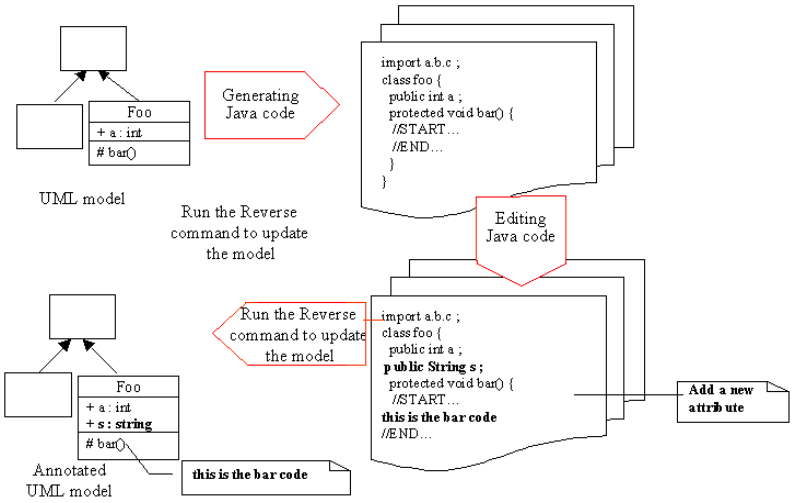


Figure 9-3. Principles of round trip engineering

Advantages of the round trip engineering mode

The principal advantage of the round trip mode is that modifications can be carried out within the code itself, with the model only being updated periodically.

Drawbacks of the round trip engineering mode

The fundamental disadvantage is:

- either that the code will very rapidly differ from the model
- or that the model returned will be physical and will lose its "overview"
- or a mix of the two previously described disadvantages

Frequently encountered problems: N-ary associations

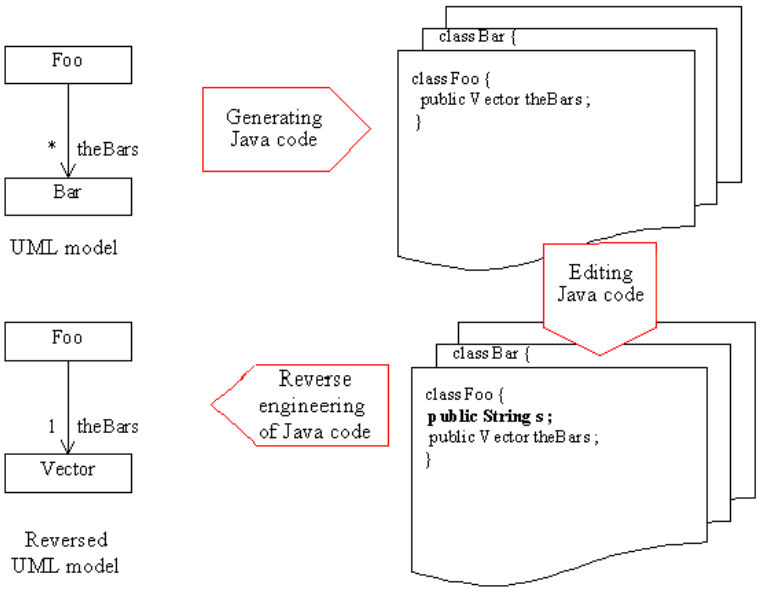


Figure 9-4. The problem of n-ary associations

Starting with a 0..* association called "*theBars*" from the "*Foo*" class to the "*Bar*" class, after the round trip we have a 1..1 association, still called "*theBars*" but going from the "*Foo*" class to the "*Vector*" class. The semantics of this association have, therefore, been lost. The model has regressed from its logical level to its physical level.

Certain mechanisms exist to get round these problems. Thus, if the "*theBars*" association has a "*public Bar getTheBars (int idx);*" form accessor, which is normally the case where you have requested that Objecteering/UML automatically generate accessors, the reverse engineering tool will find the correct association again.

Frequently encountered problems: Renaming a class

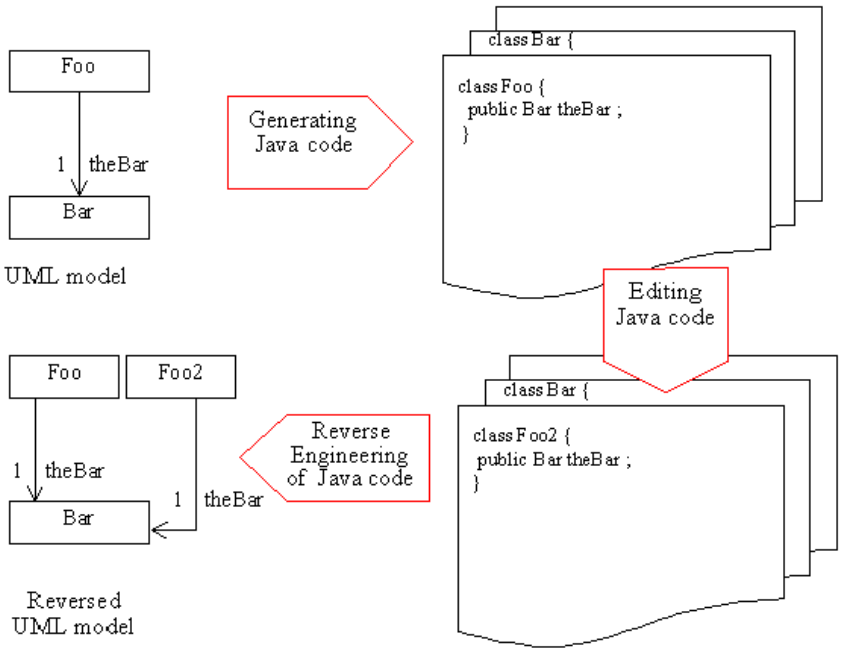


Figure 9-5. The problem of renaming a class

We start with a 1..1 association called "*theBar*" going from the "*Foo*" class to the "*Bar*" class. After carrying out a round trip operation and renaming the "*Foo*" class "*Foo2*", we have a 1..1 association going from the "*Foo2*" class to the "*Vector*" class, but at the same time the "*Foo*" class with its association has been retained. In this case, the model is no longer valid. It differs from the code.

Frequently encountered problems: Divergence in generation

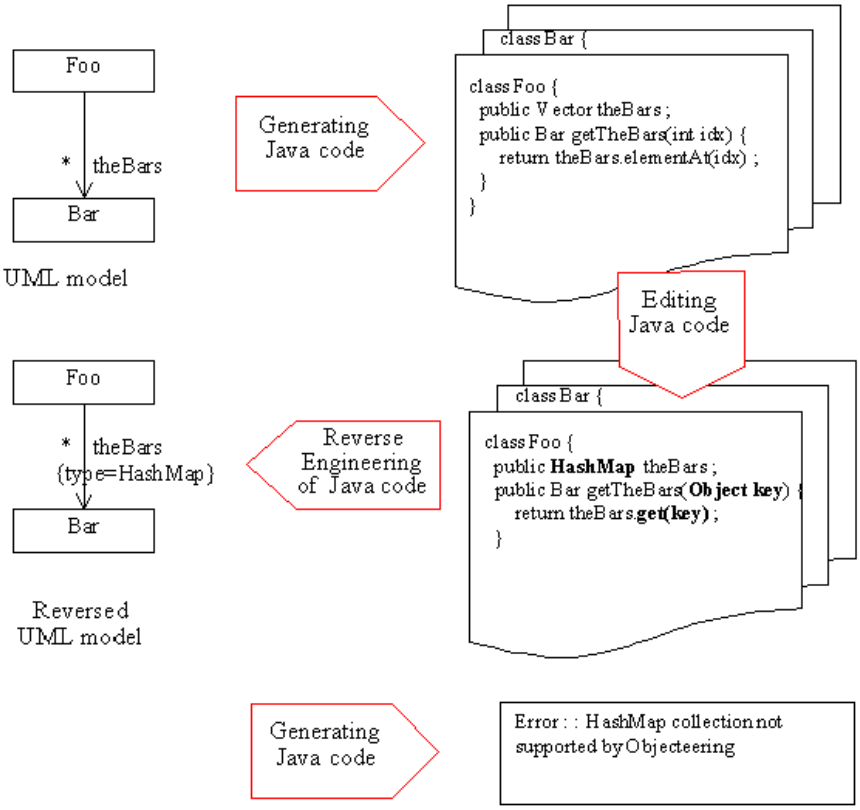


Figure 9-6. The problem of generation divergence

Starting with a 1..1 association called "*theBar*" from the "*Foo*" class to the "*Bar*" class, default generation creates a Vector. If this Vector is changed into another collection type in the Java code, two case scenarios are possible:

- Reverse engineering recognizes the new type and transfers it in the form of the {type=…} tagged value. During the next generation, if the generator recognizes the type, all is well.
- Reverse engineering does not recognize the new type (as illustrated in Figure 9-6). Generation is stopped and the round trip cycle is interrupted.

To avoid this problem, try to choose a type known to our generator (for details on the list of types known to the *Objecteering* Java generator, please refer to the "*Code generation - Tagged value types*" section in chapter 5 of this user guide).

Handy tips

Reverse

Before all reverse operations, make sure that the classes you wish to reverse are properly compiled and can be accessed through the ClassPath specified in the reverse configuration. The reverse command uses *.java* and *.class* files to carry out its operations.

Multiple multiplicity associations and attributes

For multiple multiplicity associations and attributes created in Java, Objecteering/UML can, in round trip mode, recreate the correct association or attribute through an accessor of C getXXX() form.

In this case, if C is a non-primitive class, an association towards C, with XXX as its role, is created.

If C is a type or a primitive class, a C type attribute called XXX is created.

Visibility is that of the accessor.

Generation markers

Even in round trip mode, the code generator generates "*user*" code between the "//START OF MODIFIABLE ZONE …" and "//END OF MODIFIABLE ZONE …" markers, in order to be able to differentiate between the code generated by Objecteering/UML (invariants, pre and post-conditions, amongst other things) and the code written by the user.

During the reverse operation, these markers are deleted. For methods, as well as markers, the code before the start marker and after the end marker is deleted, in order not to generate invariants, pre-conditions and post-conditions several times.

Managing imports

During the reverse, imports are transformed (where possible, in other words, where the imported element is modeled) into a use link in the model.

Thus, an "import P1.P2.C1;" line in the reversed Java source for the "*C*" class is translated in the model by a use link from the "*C1*" class to the "*C*" class, if the "*C1*" class exists in the model and is contained in the "*P2*" package, which is itself contained in the "*P1*" package.

Collections

If you are adding a collection type attribute to your Java code, remember to create a "standardized" accessor, so that reverse engineering will be properly carried out.

For example, do not write:

```
class Company {
public Vector Addresses ;
}
```

but instead:

```
class Company {
private Vector Addresses ;
public Address getAddresses (int idx) { return
Addresses.elementAt(idx) ; }
}
```

In this way, reverse engineering will detect a 0..* association from the "*Address*" class to the "*Vector*" class.

Only types which can be handled by the Java generator should be used (for details on the list of types known to the *Objecteering/Java* generator, please refer to the "*Code generation - Tagged value types*" section in chapter 5 of this user guide).

Filtering accessors

Generate your accessors through Objecteering/UML and do not modify them. Check the "*Filter Accessors on Reverse*" tickbox, so that they will not be transferred into your model when reverse engineering is run.

If you wish to write your accessors yourself, do not check the "*Filter Accessors on Reverse*" tickbox. All the accessors will then be transferred as methods into the classes. In certain cases, this can provoke errors, for example, "Several methods have the same signature".

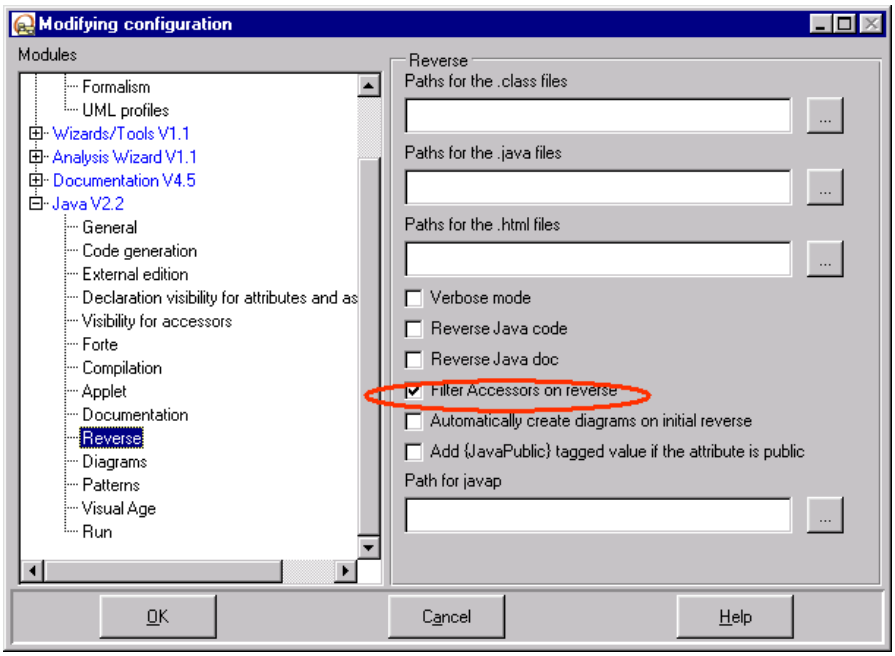


Figure 9-7. Filtering accessors

Chapter 10: Working with other IDEs

Introduction to working with other IDEs

The *Objecteering/Java* module provides a means of working with other IDEs, which allows you to import developments carried out in other IDEs into the Objecteering/UML model, or to export the code generated into your favorite IDE. Imports and exports are implicitly carried out using the "*Generate*", "*Reverse*" or "*Reverse again*" commands.

Visual Age/Objecteering integration

Introduction

The module available for the integration of Visual Age and Objecteering/UML allows the user to model his Java applications in Objecteering/UML and then, using the standard Objecteering/UML Java generation commands, to generate the corresponding Java code and to import it into Visual Age.

It can also be used to reverse code coming from Visual Age (still using the standard Objecteering/UML commands), by automatically exporting Visual Age sources and classes, before running the "*normal*" reverse.

These operations can be carried out in both model driven and round trip modes.

Installation

To be operational, the two tools must be activated in the correct environment selected by the user on the same machine.

Objecteering/UML servlet class files provided in the *<objecteeringDir>/modules/JavaModule/bin/VAServlets*.jar jar file must be extracted in the *<vadir>/ide/tools/com-ibm-ivj-toolserver/servlets/* directory, using winzip or an MSDOS command.

In order for this modification to be taken into account, Visual Age must be launched after the installation of these files.

Before all code generation operations, the user must first create a Visual Age project within Visual Age itself and then activate the "*Remote Access to Tool API*" window (through the *Windows*/*Options* menus). Access can be automatically initialized when Visual Age is started, by checking the "*Start remote access to Tool API on Visual Age startup*" and "*Use system generated port*" tickboxes in the Visual Age *Window/Options/remote access to Tool API* menu.

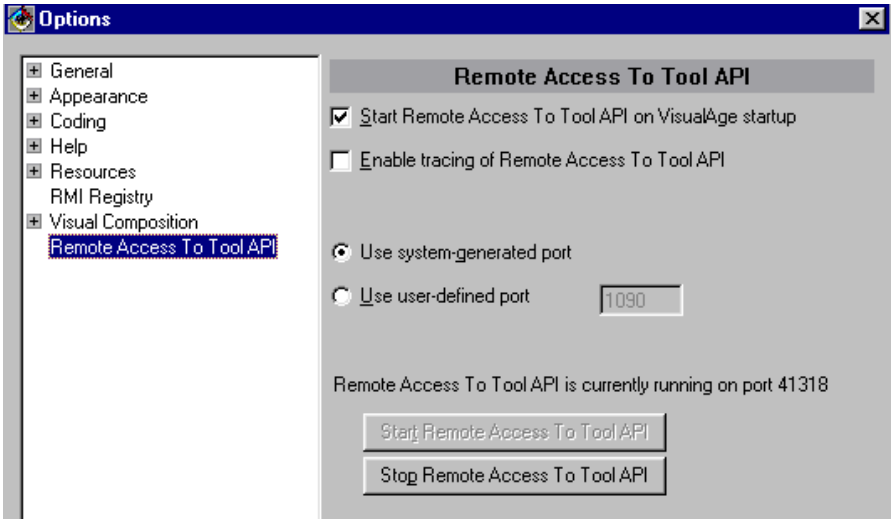


Figure 10-1. Configuring remote access to the tool API

Configuration

To configure Objecteering/Visual Age integration, simply activate the "*Edit configuration*" window by clicking on the  "*Modifies the configuration of module parameters*" icon. Objecteering/Visual Age integration is configured in the "*General*" sub-set of Java module parameters.

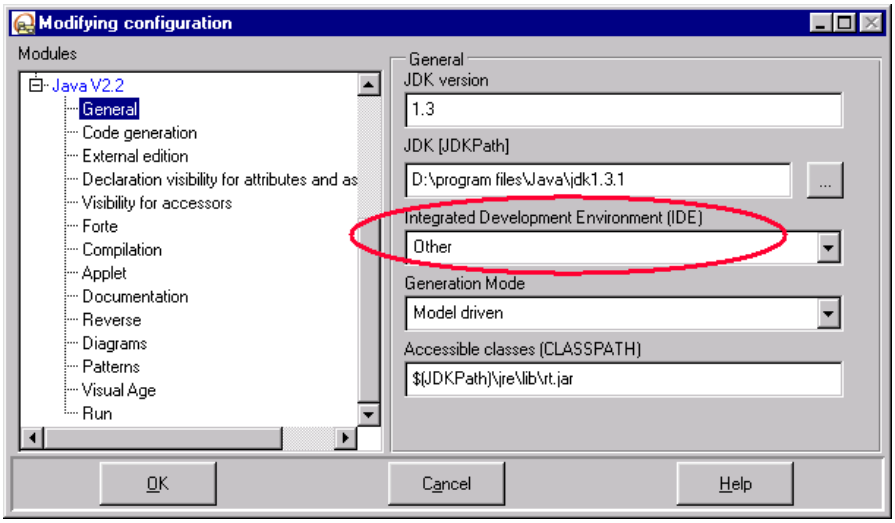


Figure 10-2. Configuring the choice of integrated development environment in the "*General*" subset of Java module parameters

- *Integrated Development Environment (IDE)*: This field is used to indicate the choice of development environment which is to be used. In order to use Visual Age, select the *Visual Age* value from the scrolling list.

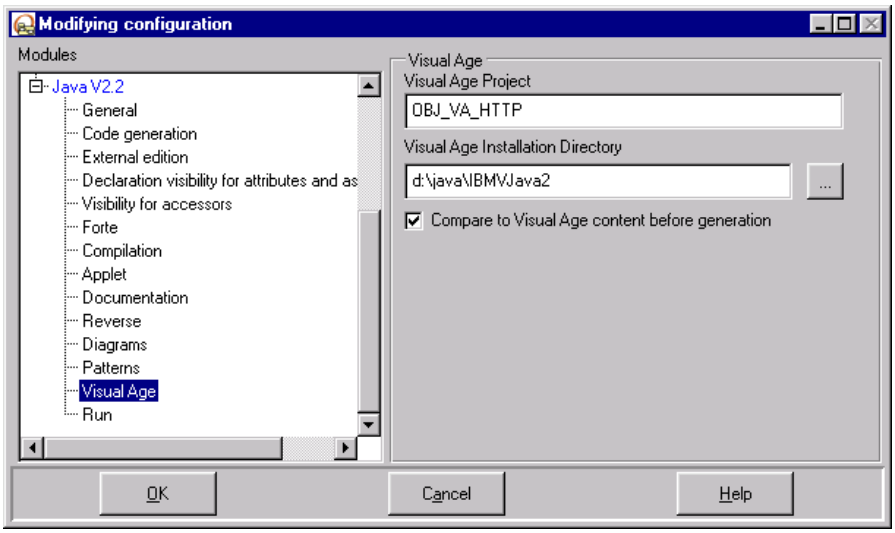


Figure 10-3. Configuring the Visual Age project in the "*Visual Age*" subset of Java module parameters

- *Visual Age Project*: During initial code generation or Reverse Engineering operations, the name of the Visual Age project is requested. This name is then systematically reused. This name is memorized as a Java module parameter (Visual Age section). In order to modify the Visual Age project, the user must simply modify this name or leave it blank, in order to make an interactive selection during the next code generation or reverse engineering operations.
- *Visual Age Installation Directory*: This indicates the Visual Age installation directory. This directory is memorized as a Java module parameter (Visual Age section, as shown in Figure 10-3).
- *Compare to Visual Age content before generation*: Before regenerating code in Objecteering/UML, this field is used to check whether or not the corresponding code has been modified in Visual Age. This mode is memorized as a Java module parameter (Visual Age section).

Visual Age/Objecteering integration - Operating mode

Introduction

There are no particular commands necessary regarding the operating mode selected. The processing described in the following paragraphs is integrated at Java code generation and Java reverse levels.

Generating Java code

For processing operations common to both the model driven and round trip modes, a request for the import of Java files is automatically triggered after the files are generated from Objecteering/UML.

In order to optimize performances, import processing in Visual Age is carried out only once on all generated files.

Certain preliminary processing differs according to the operating mode. Before code generation, if the "*Compare to Visual Age content before generation*" tickbox is checked, a request for the export of files from Visual Age is run in an intermediary directory. Those files exported are compared with those present in the Objecteering/UML generation directories.

- Model driven mode: if the files are different, they are copied in order to allow the automatic reintegration of modifications within the Objecteering/UML markers.
- Round trip mode: if the files are different, a message warns the user that information could be lost (modifications in the two environments). He can then choose whether or not to continue with code generation. No code retrieval is carried out. The user can then run the "*Reverse again*" command in order to update the model.

Note: In certain cases (non-compilable sources, for example), Visual Age refuses to import or export its files. If this is the case, an error is displayed in the Objecteering/UML console. Objecteering/UML continues with the operation in progress where this error is recoverable.

Reverse

Before all reverse operations, a request for the export of Java source files and compiled files is sent to Visual Age, in order to make all the information necessary to reverse operations available.

The "Update" command

Before all "*update*" operations, a request for the export of Java source files is sent to Visual Age, in order to make all the information necessary to the update available.

Note: In round trip mode, the "*Update*" command retains the same functions as in model driven mode, in other words, model update from the code contained between the markers. Any modifications carried out outside these marked zones will not be taken into account by this command. If this is the case, then the "*New Reverse*" command should be used.

Compiling and archiving

The *Objecteering/Java* module compilation and archiving commands must not be used in Visual Age mode. Visual Age internal commands should be used instead.

Forte/Objecteering integration

Forte markers

It is possible to activate the generation of Forte markers. In Forte, these markers prevent the modification of code zones generated by the Java module.

This function is activated at module parameter level (as shown in Figure 10-4).

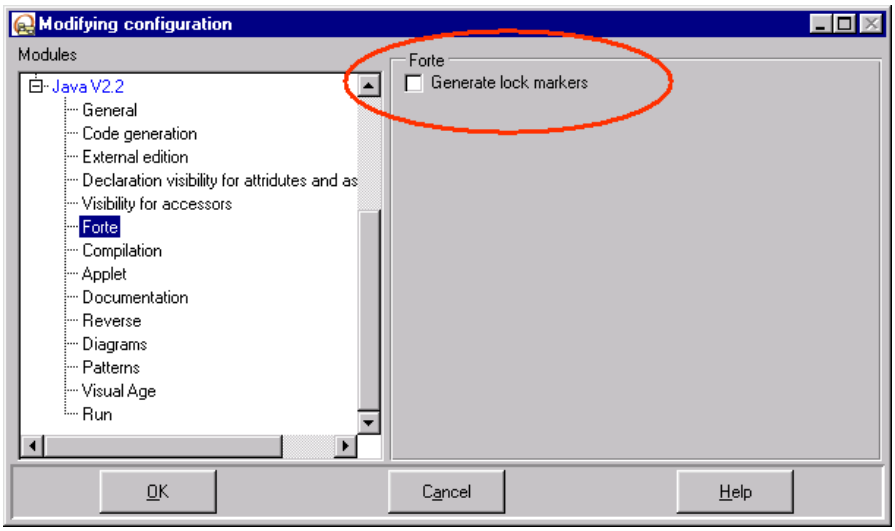


Figure 10-4. Activating the generation of Forte markers

Simply select the "*Forte*" parameter sub-set, and check the "*Generate lock markers*" tickbox.

Note: You must also select "*Forte*" as the "*Integrated Development Environment (IDE)*" parameter in the "*General*" sub-set of Java parameters (for further information, please see the "*Configuration window*" section in chapter 16 of this user guide).

Chapter 11: Objecteering/Eclipse

Introduction to Objecteering/Eclipse

Overview

Objecteering/Eclipse is a plug-in for Eclipse, used to incorporate Objecteering/UML into your Eclipse workspace. It gives you the possibility of synchronizing your UML model and the code in your Eclipse project and simplifies the writing of Java code.

Objecteering/Eclipse is a fully functional version of Objecteering/UML, with an extended version of the *Objecteering/Java* module which supports better integration in Eclipse.

From Objecteering/UML to Eclipse

The *Objecteering/Java* module used with *Objecteering/Eclipse* has a certain number of features to help its integration into Eclipse.

- Every modification of the model regenerates impacted Java files, and all views of these files in Eclipse are updated.
- When a file is added to the Eclipse project directory by a generation operation, Eclipse is notified and adds the file concerned to its project.

From Eclipse to Objecteering/UML

Eclipse can also notify Objecteering/UML of modifications made to the code, which can impact the model.

- When a Java file is saved, Objecteering/UML is notified to retrieve modifications and insert them into its UML model.
- When a class is created by Eclipse, Objecteering/UML is notified to add it to the model.

To retrieve information about the model, Objecteering/UML extracts data from the compiled class file and from source files. If your Java file is incorrect when you save it, an error appears in the Objecteering/UML console and your modifications cannot be added to your model. To update your model, you should correct your code.

Functions

Using *Objecteering/Eclipse*, it is possible to:

- generate your Java code from your model
- edit your code in Eclipse
- synchronize your code and your UML model

Parameterization

Furthermore, Objecteering/UML provides powerful means of parameterization, through the following:

- *Objecteering/Eclipse* module parameters
- generation and document templates, accessible from *Objecteering/UML Profile Builder*

These parameterization capabilities allow you to adapt and enrich modeling elements or even to redefine generation rules, in order to adapt generated code to your style of programming.

Objecteering/Eclipse glossary

- *Eclipse*: Open source Java IDE which can be extended by a plug-in system. Objecteering/Eclipse is a plug-in for Eclipse.
- *WSAD:* Websphere Studio Application Developer. A special version of Eclipse by IBM.
- Perspective: A set of editors in Eclipse for a task. For example, a Java perspective or a J2EE perspective.
- View: A window opened in a perspective.

Using Objecteering/Eclipse

Prerequisites

In order to use *Objecteering/Eclipse*, the following prerequisites must be respected:

- Objecteering/UML must already be installed
- the *Objecteering/Java* module must already be installed
- the Eclipse framework must already be installed

Creating a UML modeling project

For information on how to create a new UML modeling project, please refer to the "*Creating a new UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide.

Installing Objecteering/Eclipse

To install *Objecteering/Eclipse*, simply double-click on the associated .prof file. *Objecteering/Eclipse* will then be installed on your site and will be available for selection in your UML modeling project.

Selecting Objecteering/Eclipse in a UML modeling project

This operation is detailed in the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide.

Creating an Eclipse project

The model creation wizard is integrated into project creation wizards. Use the "*File/New/Project*" menu to open the project type selection box, choose "*New Objecteering project*", and click on the "*Next*" button to access project options.

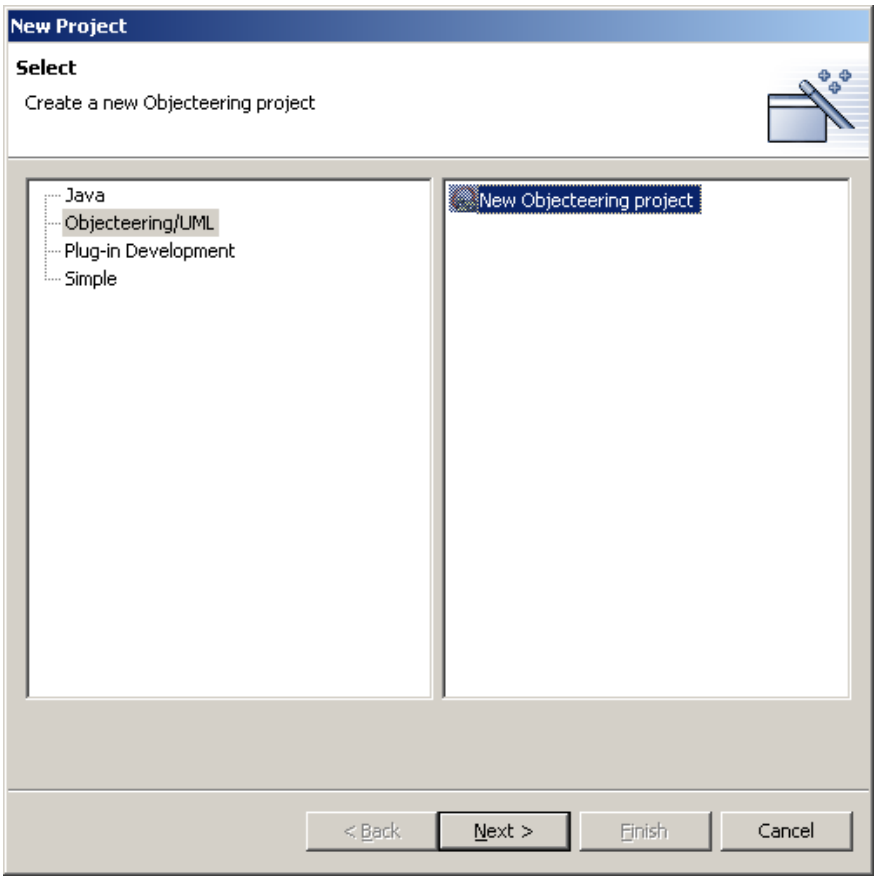


Figure 11-1. Creating an Eclipse project

Eclipse Java project creation

The screen shown in Figure 11-2 is used to define where your Java files will be generated.

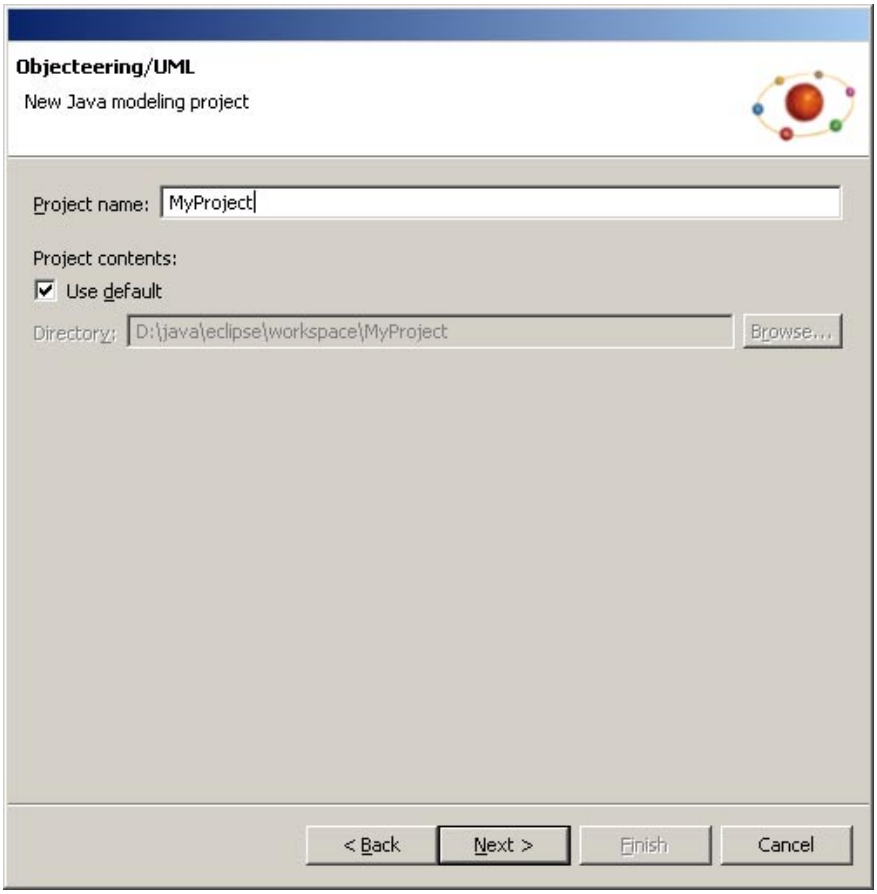


Figure 11-2. Eclipse Java project creation - defining where your Java files will be generated

Figure 11-3 presents advanced options.

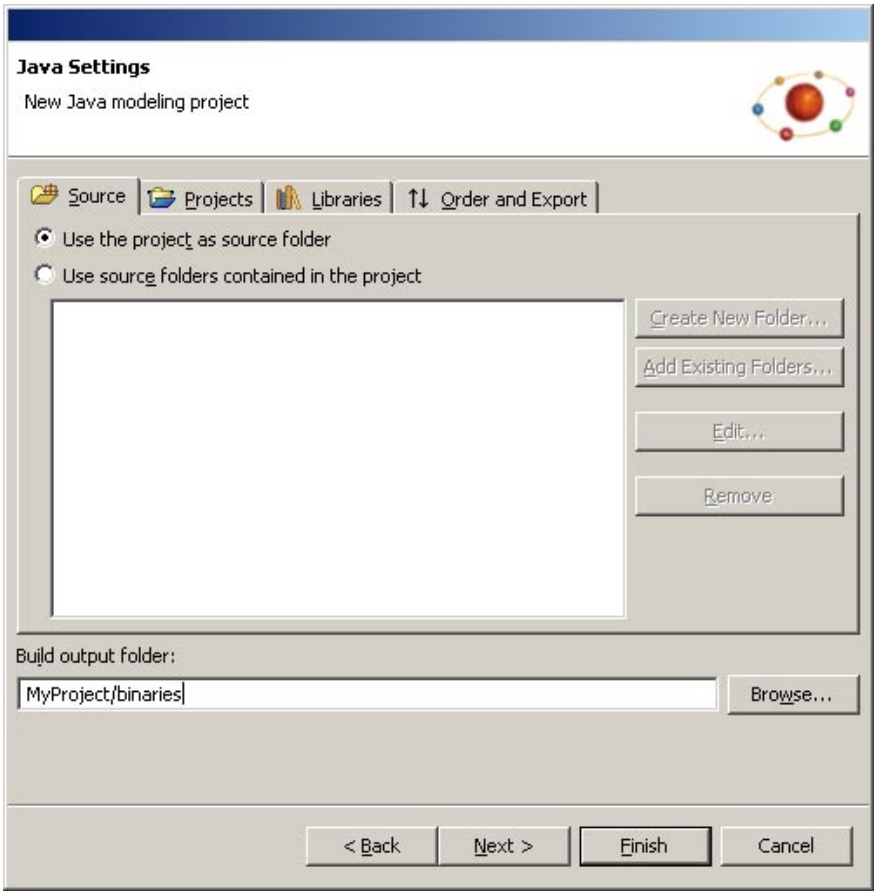


Figure 11-3. Eclipse Java project creation - advanced options

For further information, please refer to the Eclipse documentation.

Objecteering/UML basic options

The screen shown in Figure 11-4 is used to define basic Objecteering/UML options:

- the location where .ofp files should be created
- the name of the project file
- the name of the root package

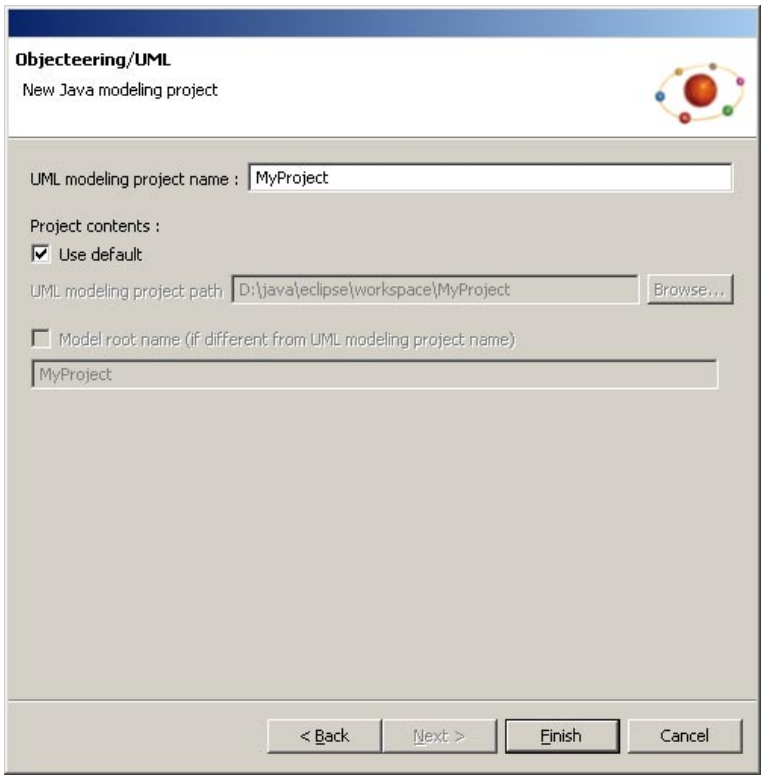


Figure 11-4. Objecteering/UML basic options

Objecteering/Eclipse first steps

Introduction

Objecteering/Eclipse first steps present a demonstration project, designed to help you discover the different features of *Objecteering/Eclipse*, step by step.

We recommend that before starting every user carry out the general Objecteering/UML first steps in the *Objecteering/Introduction* user guide.

The *Objecteering/Eclipse* first steps will demonstrate:

- how to initialize the *First Steps* project
- how to generate Java code
- how to edit generated Java code
- how to compile
- how to synchronize your code and your model

Sources

This example is a simple bank account application, extracted from *Learn Java Now,* by *Stephen R.Davis*, *MicrosoftPress*.

Preparing the Eclipse working environment

Before starting work with *Objecteering/Eclipse*, you must first prepare the working environment.

1 - Launch the Eclipse environment.

2 - Open the Objecteering/UML perspective, by selecting *Objecteering/UML* in the "*Window/Open perspective/Other*" menu.

3 - Create an Objecteering/UML project in Eclipse, by choosing "*New Objecteering project*" in the "*File/New/Project*" menu.

4 - Name your Eclipse project "*FirstSteps*", and leave the other default values.

Importing a model into the Eclipse FirstSteps project

We are now going to import the "*Bank*" demonstration project, which will be our model, into our project. This is done by running the "*Import the first steps project*" command (as shown in Figure 11-5).

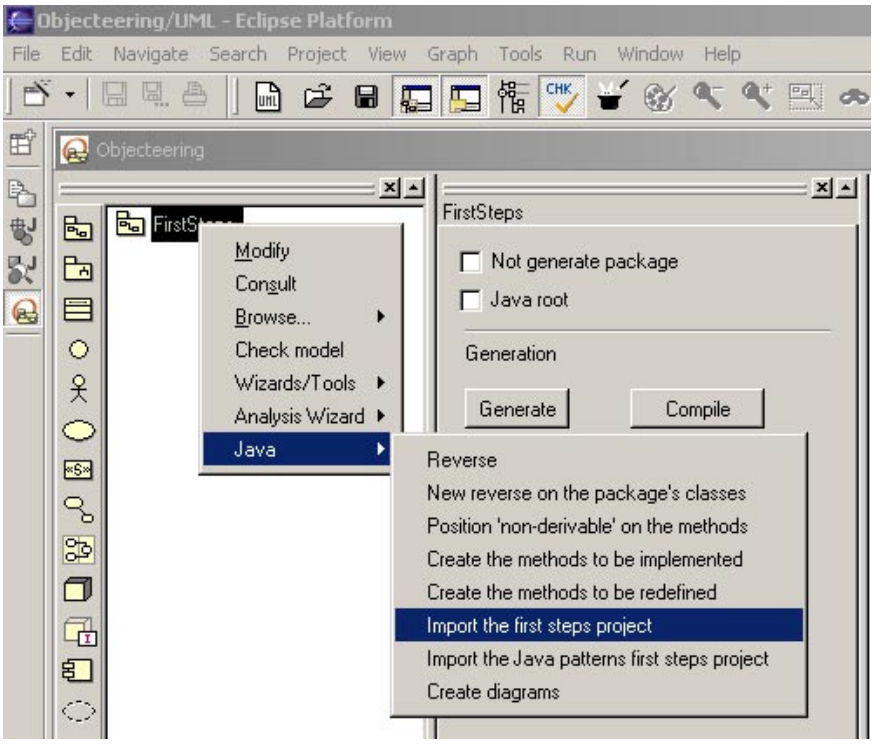


Figure 11-5. Importing the first steps project

Generating Java code

When the "*Generate*" command is run on the bank package, a set of Java files is created.

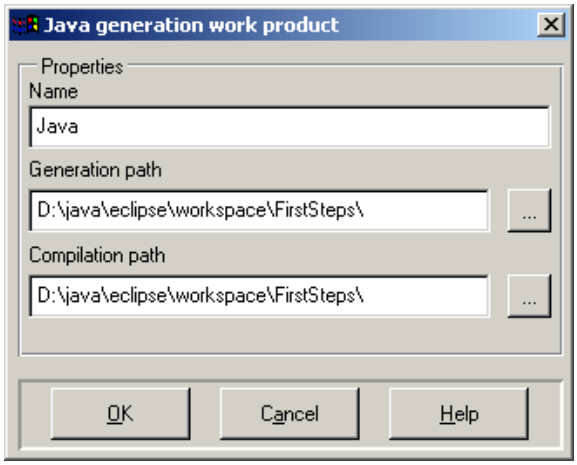


Figure 11-6. Creating a Java generation work product

After the generation, all the files generated appear in the Eclipse navigator.

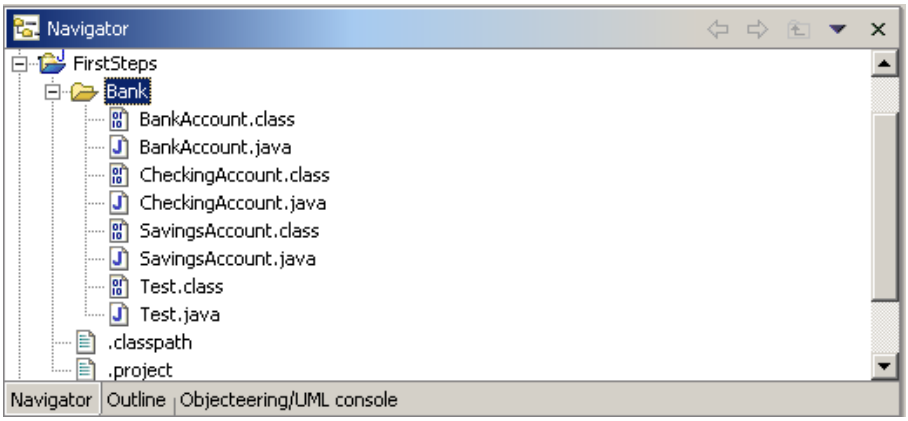


Figure 11-7. The Eclipse navigator

Note: Double-clicking on the generation work product executes the "*Generate*" command for the selected work product.

Editing generated code

The Java code generated for the element in question can be edited using the "*Edit*" button in the properties editor. This command will open a new window in Eclipse with the code of the class.

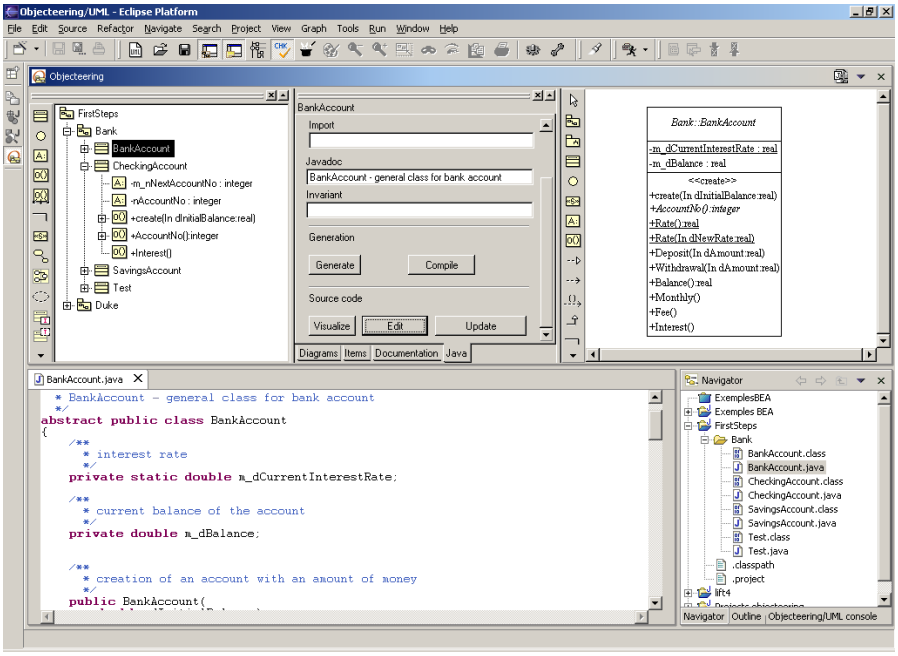


Figure 11-8. Editing generated Java code

Now, you can edit the file you have just opened, for example, by adding a method:

```
/**
 * This is a sample method
 */
public void sampleMethod () {
System.out.println ("This is a sample");
}
```

When you save your file, if the Java file is correct, your model is automatically updated with the addition of this new method.

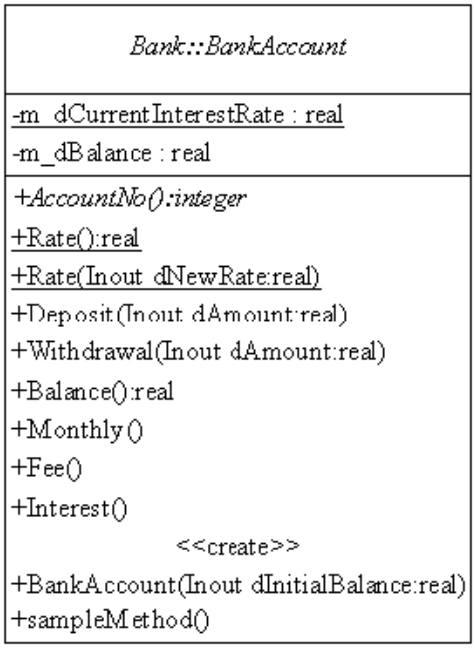


Figure 11-9. The updated model

You can also modify the model in Objecteering/UML. After every model modification, impacted files are automatically re-generated. If you add an integer parameter to your sampleMethod, it appears in the source file opened in Eclipse.

The Objecteering/UML perspective in Eclipse

Launch the perspective

To open the perspective, use the "*Window/Open perspective/Other*" menu, and select the Objecteering/UML perspective. In the perspective shown in Figure 11-10, there are two views:

- Objecteering/UML
- the Objecteering/UML console.

To open these views, use the "*Window/Show view/Other*" menu.

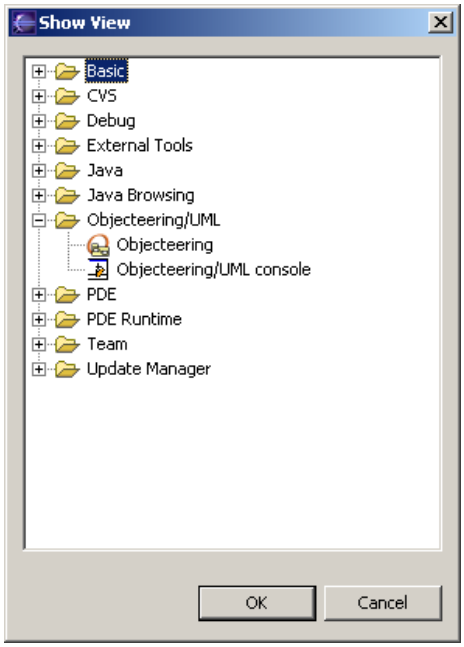


Figure 11-10. Showing views

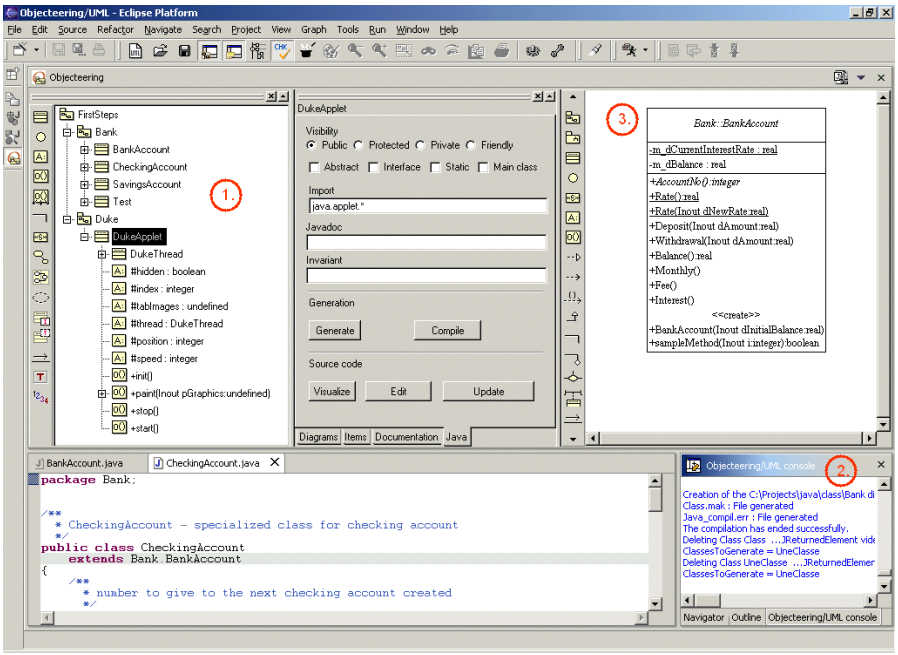


Figure 11-11. The Objecteering/UML perspective

This is the Objecteering/UML perspective, divided into three views:

1 - the Objecteering/UML view

2 - the Objecteering/UML console

3 - the editor view

The Objecteering/UML view

The Objecteering/UML view displays the UML browser, the properties editor and an opened diagram. This is the traditional Objecteering/UML interface.

On the right of this view's title bar, there are a number of buttons. The window management button is used to close opened windows and navigate between Objecteering/UML windows.

Other buttons can be added by modules, such as the "*Macros*" module, as shown in Figure 11-12 above.

Note: For further information on the Objecteering/UML interface, please refer to the *Objecteering/UML Modeler* user guide.

Note: For further information on the Java options of the properties editor, please refer to the *Objecteering/Java* user guide.

The  button on the right of the title bar contains a pull-down menu to navigate between Objecteering/UML views (diagrams and browsers), and to close them.

The Objecteering/UML console

On the bottom right, the Objecteering/UML console displays information provided by Objecteering/UML. It is an Eclipse view, so you can dock it anywhere in Eclipse.

The editor view

On the bottom left is the Java editor from Eclipse, opened by Objecteering/UML. You can use every editing function of Eclipse in this window, such as coloring or coding wizards.

Parameterizing Objecteering/Eclipse

Introduction

The configuration of Objecteering/Eclipse is split between the *Objecteering/Java* module in Objecteering/UML, and the Objecteering/UML perspective in Eclipse.

Parameterization in Objecteering/Java

Parameters in the *Objecteering/Java* module have an impact in Objecteering/Eclipse.

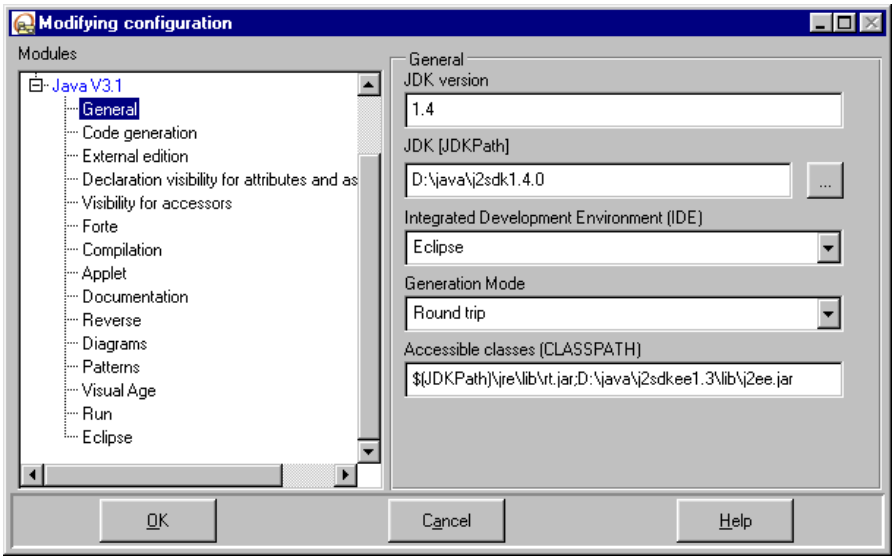


Figure 11-12. The "*General*" module parameter sub-set

In the "*General*" Java module parameter sub-set, the generation mode must be selected. For more information, please refer to chapter 9, "*Choosing the functional mode*", of this user guide. Objecteering/Eclipse can work in two modes.

Your IDE must also be selected. By default, this is Eclipse.

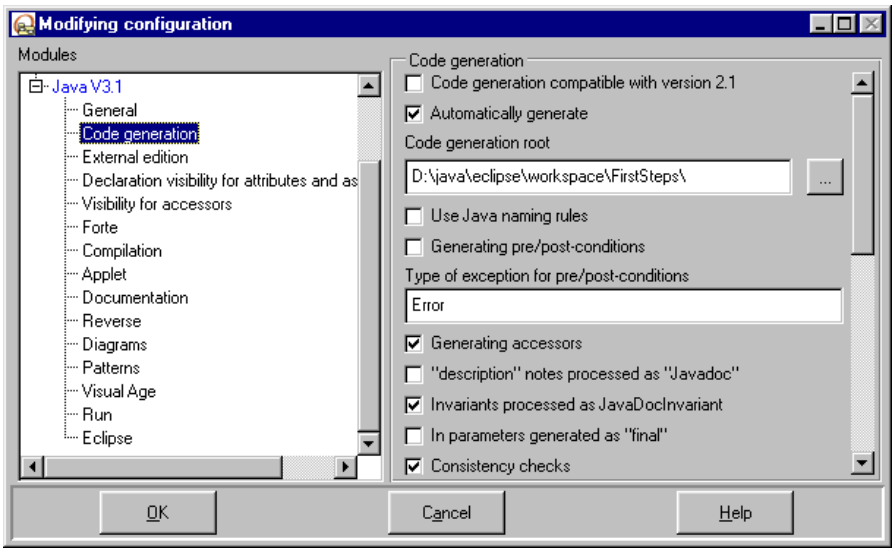


Figure 11-13. The "*Code generation*" module parameter sub-set

In the "*Code generation*" Java module parameter sub-set, the "*Code generation root*" option must be the root of your Eclipse project, and we recommend the use of the "*Automatically generate*" tickbox.

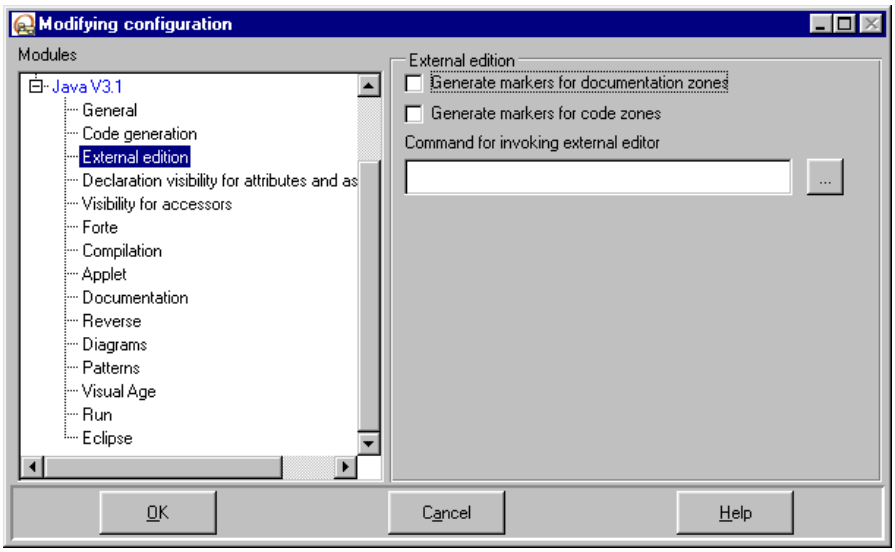


Figure 11-14. The "*External edition*" module parameter sub-set

In the "*External edition*" Java module parameter sub-set, there are options regarding external edition. In Objecteering/Eclipse, we recommend that you do not specify an external editor (the Eclipse editor is automatically used). We also recommend that you activate markers in model-driven mode, and deactivate them in round-trip mode.

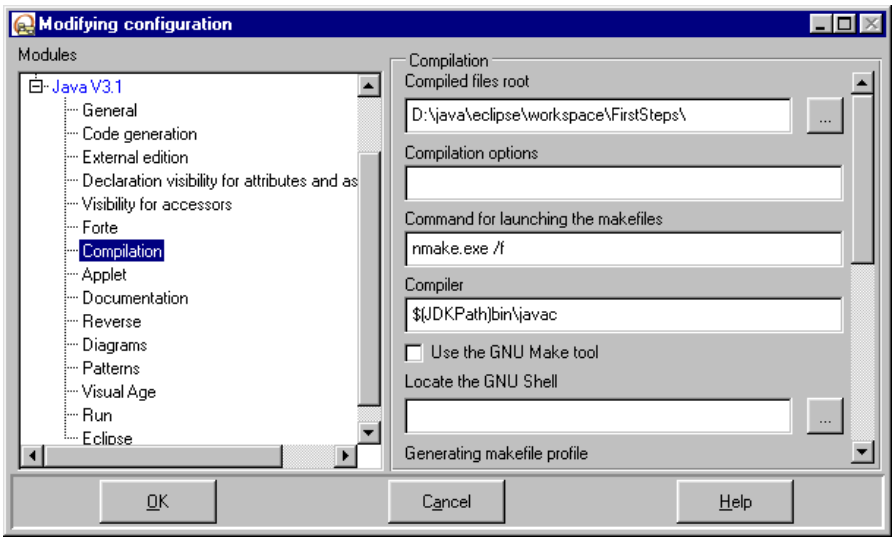


Figure 11-15. The "*Compilation*" module parameter sub-set

In the "*Compilation*" Java module parameter sub-set, the "*Compiled files root*" must be set to the "build output folder" directory of your Eclipse project.

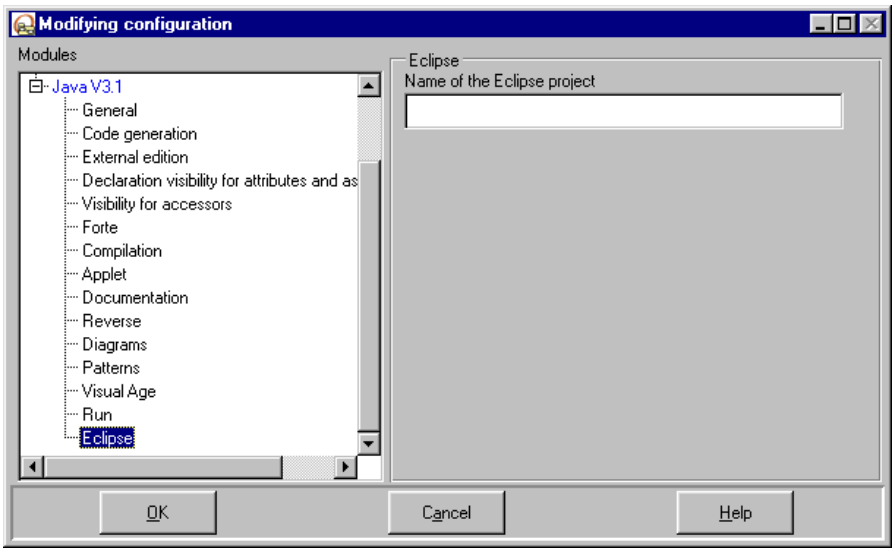


Figure 11-16. The "*Eclipse*" module parameter sub-set

The "*Name of the Eclipse project*" parameter is the name of your Java project in Eclipse.

For further information on *Objecteering/Java* parameters, please refer to the "*Configuration window*" section in chapter 17 of this user guide.

Parameterization in Eclipse

Options of the Objecteering/UML plug-in are accessible through the "Window/Preference" menu in Eclipse.

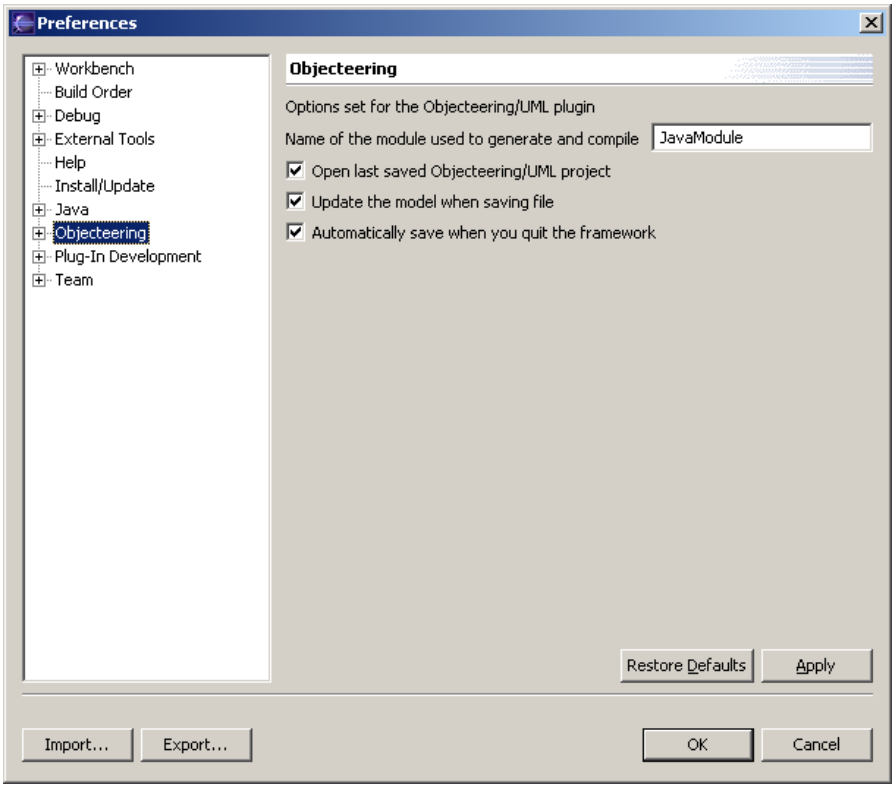


Figure 11-17. The "*Preferences*" window in Eclipse

In the "*Preferences*" window, you must declare the name of the module used to generate in Objecteering/UML. By default, this is set to the *Objecteering/Java* module.

Chapter 12: Design Patterns

Design Patterns - Overview

General remarks

The *Objecteering/Java* module is delivered with design patterns and programming patterns that the Java designer will find very useful. These patterns result in the modification of a model by creating objects or by modifying existing objects.

The patterns delivered with the *Objecteering/Java* module are:

- the creation of methods to be implemented
- the creation of methods to be redefined
- the transformation of a primitive class into a non-primitive class
- the transformation of a non-primitive class into a primitive class
- RMI
- event source
- event listener
- non-derivable method

Access

These patterns are available in the context menu of a class, a package, an attribute or an association end.

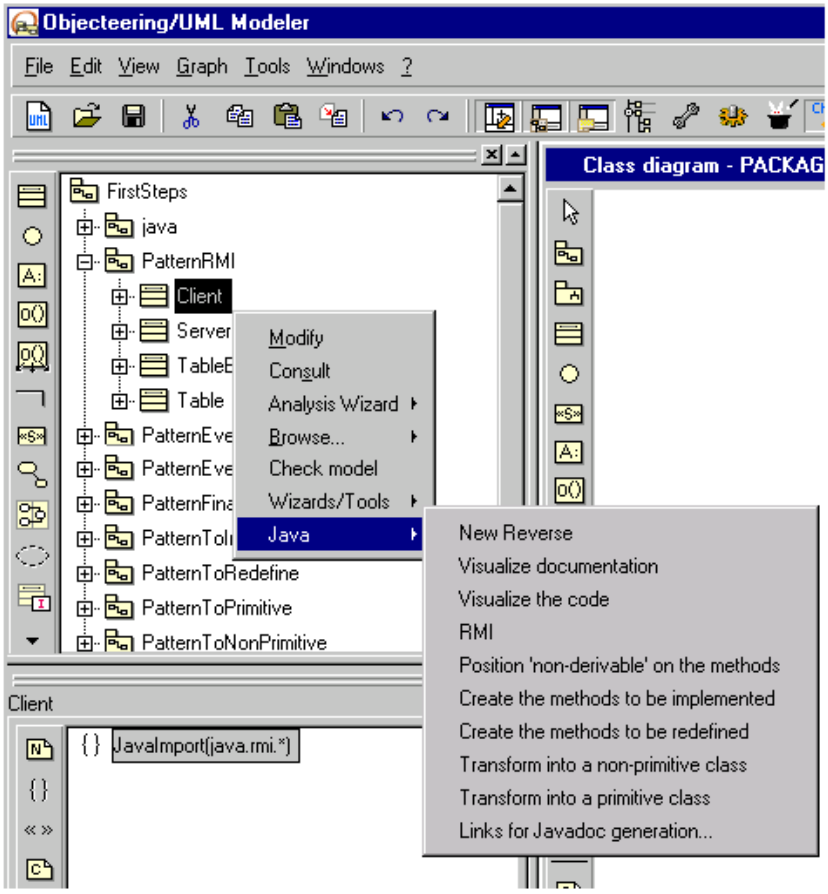


Figure 12-1. Menu for launching patterns

Design Patterns - Detailed view

Remote Method Invocation

Calling methods on remote objects is a powerful service provided by Java. Having to code not only the concrete class but also the interface requires having to do the same task twice, and entails further risks of errors.

These risks no longer exist with the *RMI pattern*. Furthermore, it is incremental, in other words, it can be applied each time the concrete class is modified.

Sending events

It often happens in a Java application that the user wants to trigger an event when the state of an object changes or when it is found in a specific state.

The "*Sending events*" pattern creates the event class and the interface for the objects that will listen. It also manages notifications.

Listening to events

It is almost impossible to write a Java application without having to listen to events, such as clicking on a button, selecting a text field or moving a gauge.

The "*Listening to events* " pattern allows you to find out when the events will occur ("subscribing" to the events), by creating internal classes that will manage the listening as well as the methods for answering the notifications.

Non-derivable method

The optimization problem is common to all developers and therefore common to all Java developers.

It is sometimes possible to help the virtual Java machine. For example, when a method is not redefined, it will run faster if it is specified as "*final*".

We recommend that you carry out this optimization at the end of the development phase. However, it would be tiresome to have to go through all the methods of all the classes to check each time whether a method is redefined or not in a subclass.

The "*Non derivable method*" pattern allows the automation of this task without risks of errors.

Creating the methods to be implemented

Launching this pattern on a class has the effect of recursively scanning the implemented interface classes collecting their methods. These methods are then created in the class. This process is carried out for each of package's classes.

Processing only takes into account the modeled interfaces and not the {*JavaImplements*} tagged values.

Creating the methods to be redefined

Launching this pattern on a class has the effect of going recursively through the parent classes and collecting their abstract methods. These methods are then created in the class. This process is carried out for each of the package's classes.

Transforming a class into a non-primitive class

This pattern is especially useful for reversed classes. The reverse only produces primitive classes. If a user wishes to make a reversed class non-primitive, especially to create associations towards this class, he can use this pattern which will have the effect of transforming attributes with this type into associations.

Transforming a class into a primitive class

This pattern is the opposite of the previous operation. Its effect is to transform associations towards this class into attributes.

Chapter 13: Remote method invocation

Remote method invocation - Overview

General remarks

Calling methods on distant objects is a powerful service offered by Java. Having to code not only the concrete class but also the interface requires having to do the same task twice, and thus entails further risk of errors.

These risks no longer exist with the RMI pattern. In addition, it is incremental, in other words, it can be applied each time the concrete class is modified.

Aim of the pattern

The use of the RMI pattern aims at producing an interface from a class, designed to provide the services on this class which may be invoked from a distance. These services are composed of the class' public methods.

Prerequisite

The pattern applies to a class. Applying this pattern will, amongst other things, add to this class a generalization towards the JDK's "*UnicastRemoteObject*" class. The class must not, therefore, own any generalizations. If it does, the generation cannot be run.

Applying the pattern

Model before use of the pattern

The initial class is a class containing a certain number of services.

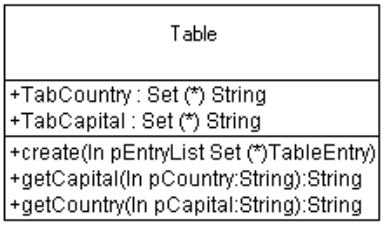


Figure 13-1. Initial model

Operating mode

The pattern is run from the class context menu.

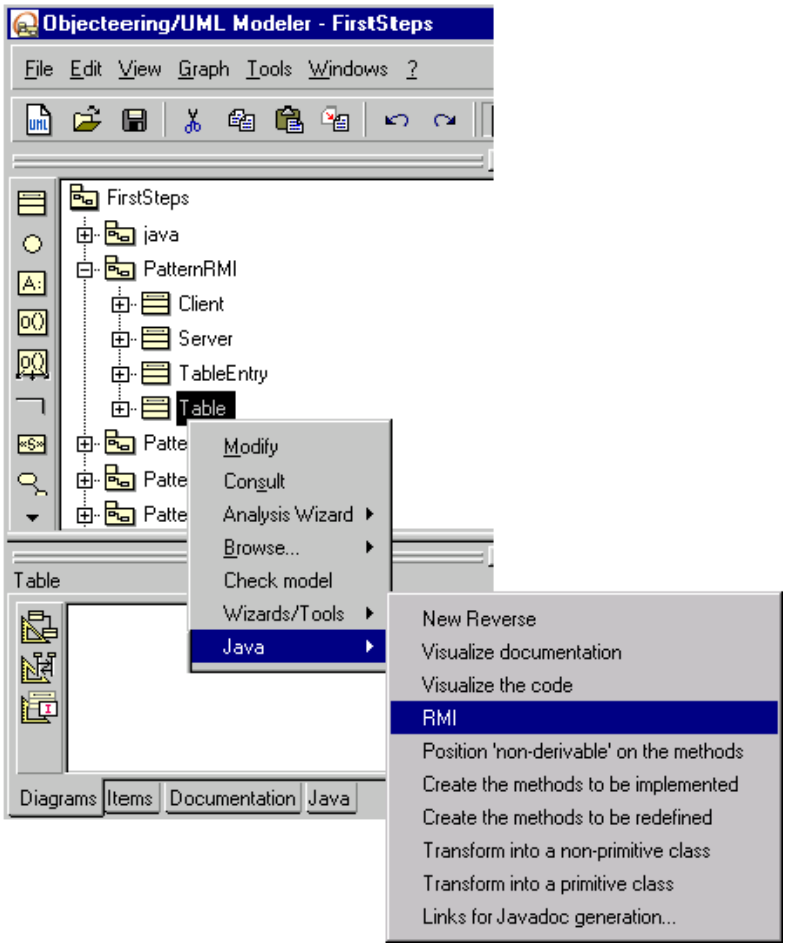


Figure 13-2. Calling the pattern

A dialog box opens suggesting default names for the class and interface that will be deduced from the initial class. It is possible to modify these names.

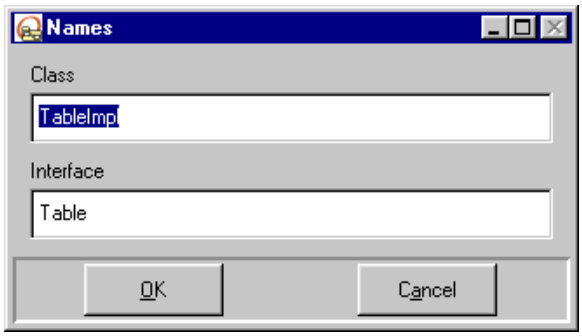


Figure 13-3. Dialog box for entering class and interface names

Once you have made your modifications, confirm them by clicking on "*OK*".

Consequences of applying the pattern

Model after use of the pattern

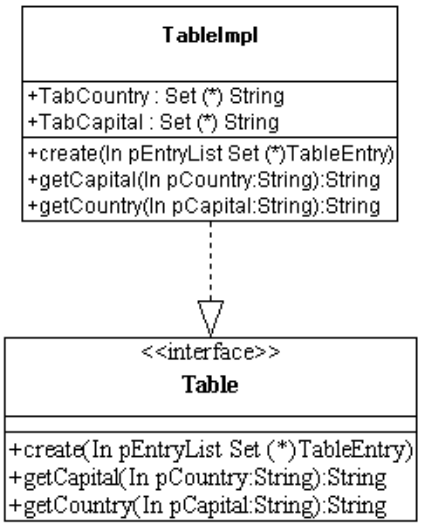


Figure 13-4. Transformed model

Summary of the transformations

The following modifications were made to the concrete class:

- the class was renamed with the name that was entered in the dialog box
- a generalization was added towards "*UnicastRemoteObject*"
- adding of an import towards *java.rmi**
- the {*JavaThrownException*} tagged value was added on each public method to indicate that it may run the "*RemoteException*" exception
- an implementation link was added towards the interface

Furthermore, an interface was created with:

- the name given in the dialog box
- all the public methods of the initial class
- a generalization towards "*Remote*"
- an import towards "*java.rmi**"

Furthermore, it is possible to re-apply the pattern later, either on the class, or on the interface. It is necessary to give the correct names of the class and interface in the dialog box if you have not kept the default names. The interface is then updated taking into account the modifications made to the class.

Chapter 14: Sending events

Sending events - Overview

General remarks

It often happens in a Java application that the user wants to trigger an event when the state of an object changes or when it is found in a specific state.

The "*Sending events*" pattern creates the events class, which is the interface for the objects that will want to listen and manage notifications.

In Java, events are grouped into event classes. Therefore in Java you have the following basic event classes: "*Window*", "*Mouse*" and "*Key*".

The sending events pattern follows the same principle.

Aim of the pattern

The use of the "*Sending events*" pattern allows you to generate events when an attribute or a simple association changes value, or when an attribute or a multiple association has elements added to it or removed from it.

Prerequisite

The "*Sending events*" pattern can be applied to any attribute or association.

Applying the pattern

Model before use of the pattern

As the pattern is applied to attributes and associations, an example of a class with both is presented here.

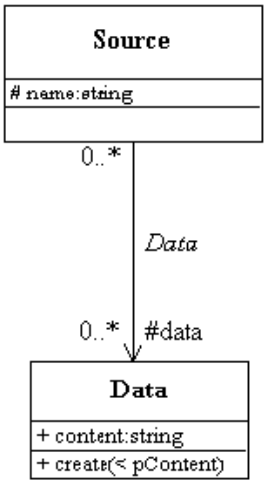


Figure 14-1. Initial model

Operating mode

The "*Sending events*" pattern can be run on an attribute or an association.

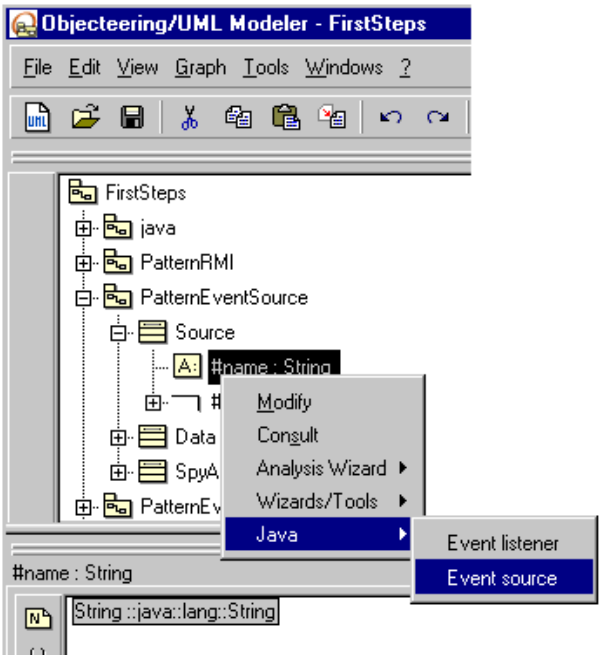


Figure 14-2. Calling the pattern

A dialog box used to create new events then appears.

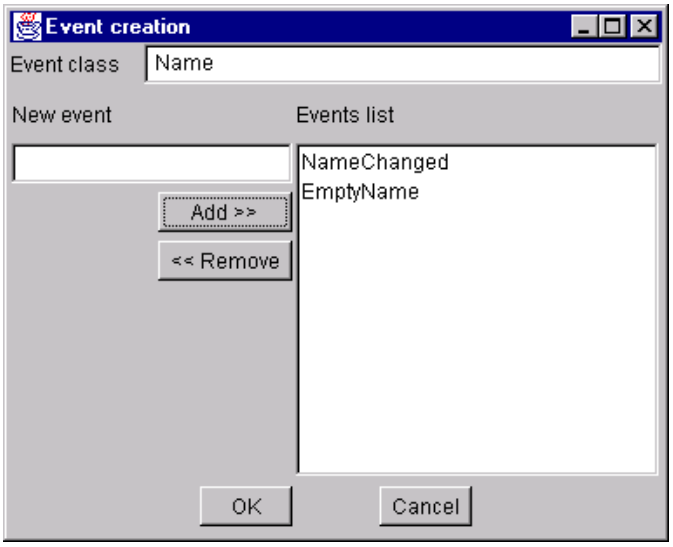


Figure 14-3. Dialog box for selecting events to create.

The ... field or button	is used to ...
New event	enter a new event for the given class
Events list	display the events added to the given class
Add	enter a new event for the given class
Remove	remove the selected event from the list of events
OK	run the pattern
Cancel	stop the processing

Consequences of applying the pattern

Model after use of the pattern

The result below is obtained through the creation of the "*Name*" class of events on the "*name*" attribute, with the "*NameChanged*" and "*EmptyName*" events on the "*data*" association of the "*Data*" event class, with the "*DataChanged*" event.

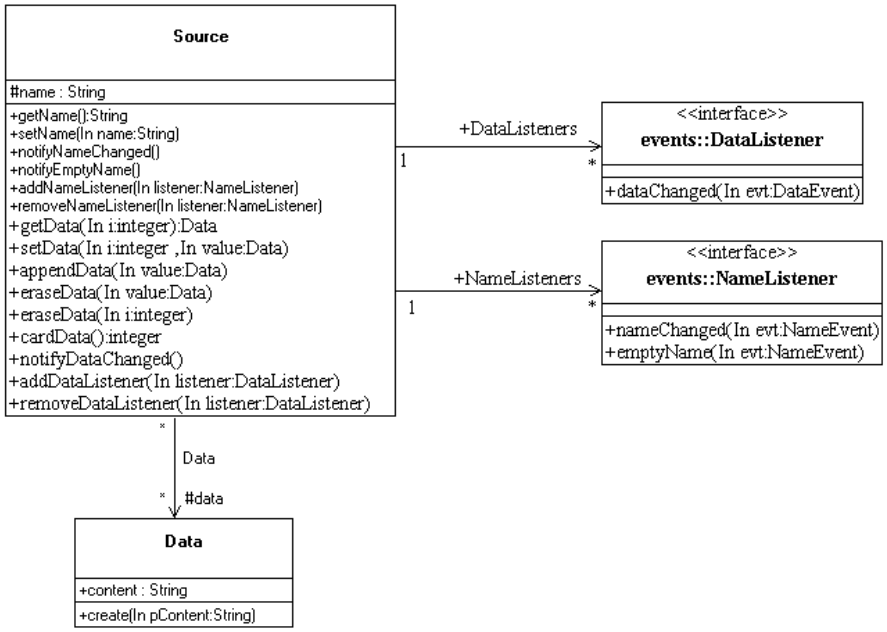


Figure 14-4. Transformed model

Summary of transformations

The execution of the pattern results in the creation or the update of an "*events*" package in the package of the class which owns the attribute or the association, from which the pattern was launched.

In this package, the following elements are created or updated:

- an event class
- an interface which is complemented by classes used to listen to the type of events above
- an adapter class to be redefined, which would be an alternative to the interface above, to listen to the type of events above

Furthermore, the class which contains the attribute or association on which the pattern was launched is enriched with methods and associations.

Details of the contents of the events package

The "events" package

An "*events*" package is created (if it does not already exist) in the package of the class which owns the attribute or the association, on which the pattern was launched. It is designed to contain a class representing each created class of events, as well as the interfaces and the adapters to be implemented or to be redefined, in order to listen to the events of these event classes.

The "Event" class

A class of events; which specializes the JDK's "*EventObject*" class, is created in the "*events*" package. This class represents the event class previously entered. Its name is obtained by concatenating the entered event class, (the first letter is converted to upper case if needed), with the "Event" string. Objects of this class will be created each time an event, which has the type it represents, occurs. This class contains:

- an attribute or an association, with the same name as the element on which the pattern has been run, aimed at containing the new value of the attribute or association after the event has occurred.
- a constant for each event of the event class. The name of the constant is obtained from the event's name by adding "_" characters in front of the upper case letters, except for the first letter, and when putting the result in upper case.
- two constants giving the interval in which the constants of the event class are defined, and which allow you to enumerate these events. Their names are made up of the name of the event class put in upper case, and concatenated with "*_FIRST*" and "*_LAST*" respectively.
- an "*id*" attribute designed to take the value of one of the constants representing the events, in other words, the type of event which occurred.
- a constructor that takes the event's source object, the type of event and the new value of the attribute or association as parameters.

Interface

An interface which defines the methods to be redefined to react to the events of the event class is created in the package. The name of this interface is made up of the event class name, the first letter of which is put into upper case if necessary, and concatenated with the "*Listener*" string. This interface contains one method per event. The names of these methods are made up of an event name. The first letter is put in lower case if necessary.

Adapter

An adapter class implements the interface above and defines each method with an empty body. A class such as this is defined each time an event class contains at least two events. Indeed, the subscription is carried out through an event class and not through one event after the other. However, sometimes we are not interested in all the events of an event class. In this way, instead of implementing the interface, a listener can specialize this class and can only define the methods corresponding to the events to which it wants to react.

Details of the emitter class transformations

Transformation of the emitter class

The following elements are added to the emitter class:

- an instance association towards the interface described above. This will allow an instance of the class to find out which objects have subscribed to the events produced by the modifications of the attribute or association value.
- the attribute's or association's mutators and accessors. It is important to note that compared to what a standard generation would have produced, mutators are enriched by the notification of an event that can be produced by an association or an attribute. The user may have to modify the body of these mutators to add conditions to the calling of these notifications.
- a {*JavaNoAccessor*} tagged value on the attribute or association notifying the generator that it should not generate accessors, since the pattern has already created them.
- a method of notification by event that notifies all the subscribers to the events with this type and belonging to the current instance that an event occurred. The method's name is obtained by concatenating the "*notify*" string with the name of the event, the first letter of which is put into upper case if required.
- a method which allows an object to subscribe to the events with the corresponding type on the current instance. The method's name is obtained by concatenating the "*add*" string with the name of the event class, the first letter of which is put into upper case if required, and with the "*Listener*" string"
- a method which allows an object to be removed from the list of subscribers to the events with the corresponding type of the current instance. The method's name is obtained by concatenating the "*remove*" string with the event class name, the first letter of which is put in upper case if required, and with the "*Listener*" string".

Chapter 15: Listening to events

Listening to events - Overview

General remarks

It is almost impossible to write a Java application without having to listen to events, such as clicking on a button, selecting a text field or moving a gauge for example.

Aim of the pattern

The "*Listening to events*" pattern allows you to subscribe to events, and to create the internal classes that will manage the listening, as well as the methods for answering the notifications.

Prerequisite

The "*Listening to events*" pattern can be applied to any simple association or attribute, provided the events exist on the attribute's class or the association's target class.

The classes for which events exist are the classes of the JDK's AWT or SWING components, as well as the classes on the attributes or associations to which you have applied the event sending pattern described above.

Applying the pattern

Model before applying the pattern

Here we have the case of an applet with graphic components.

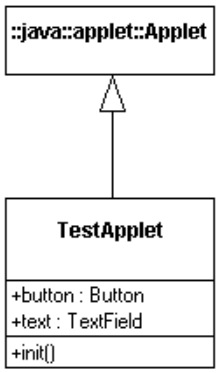


Figure 15-1. Initial model

Operating mode

The pattern can be run from an attribute or an association.

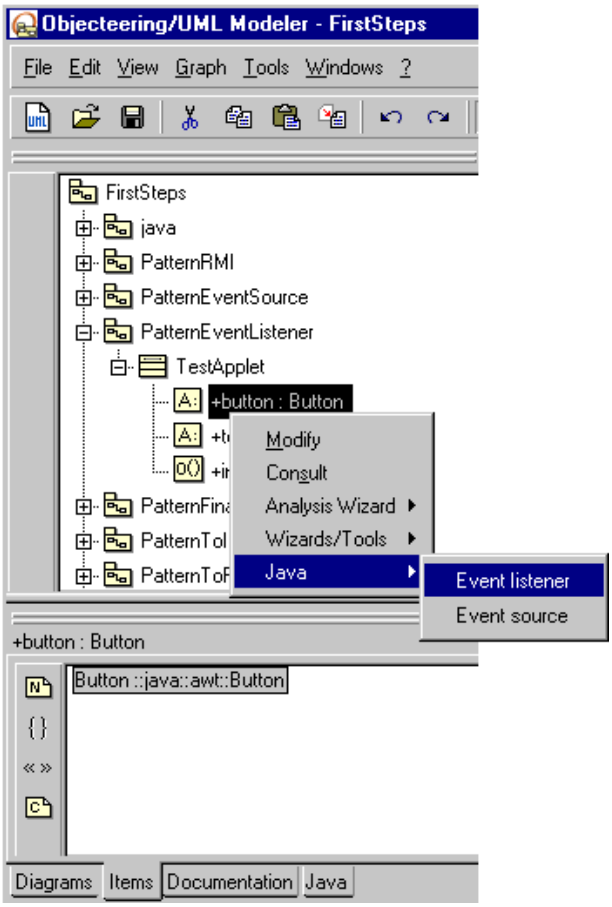


Figure 15-2. Calling the pattern

Selecting events

A dialog box appears listing the events available on the attribute's or association's class. You may select those which interest you.

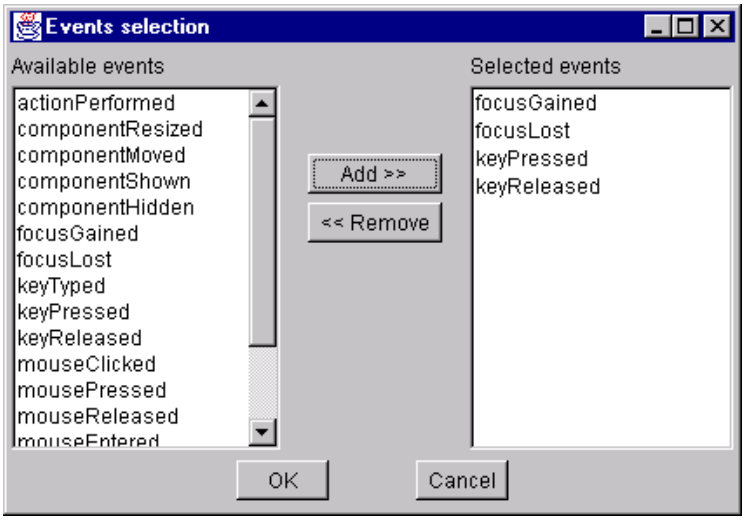


Figure 15-3. Dialog box for selecting events to listen to

The ... field or button	is used to ...
Available events	give the list of events available on the attribute's or association's class.
Selected events	display the list of events currently selected.
Add	add the overscored elements in the list of available events in the list of selected events.
Remove	delete the overscored events from the list of selected events.
OK	run the application of the pattern with the selected events.
Cancel	stop the processing.

Note: The events proposed are those which correspond to the event classes present in the model. They have either been reversed, imported or created, for example, using the sending events pattern.

Consequences of applying the pattern

Model after applying the pattern

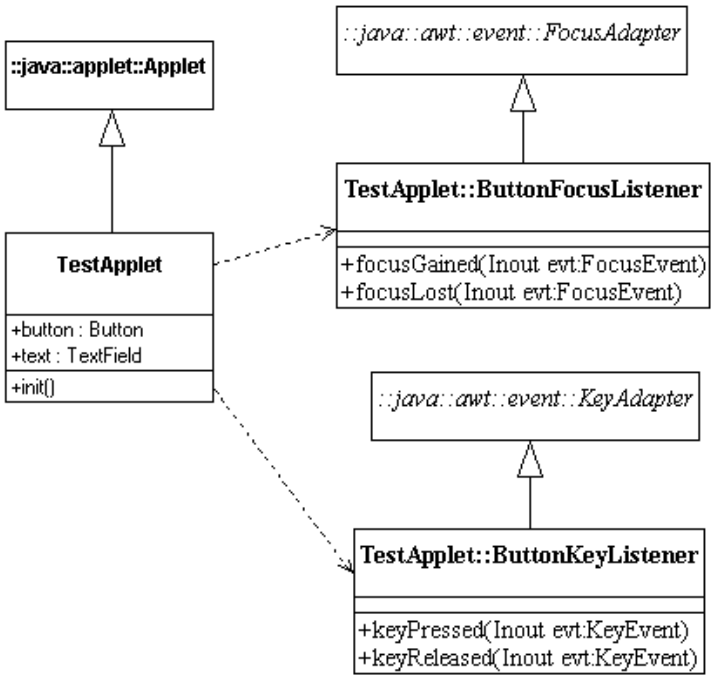


Figure 15-4. Transformed model

Summary of elements created during the transformation

By applying the pattern, the following elements have been created:

- an internal class for each class of events in which an event has been selected. The name of this class is the concatenation of the name of the attribute's or association's class on which the pattern has been run, with the name of the attribute or the name of the association's role, whose first letter is put in upper case if required, with the name of the event class, whose first letter is put in upper case if required, and with the "*Listener*" string.
- on the internal classes described above, an implementation link towards the interface of the listeners of the event class, if this class contains only one event, otherwise a generalization towards the adapter of the listeners of the event class.
- on the internal classes described below, the methods for reacting to the selected events.

The user should now simply enter the body of those methods which react to the events.

Chapter 16: Non derivable method

Non derivable method - Overview

General remarks

The optimization problem is common to all developers, and therefore common to all Java developers.

It is sometimes possible to help the virtual Java machine. For example, when a method is not redefined, it will run faster if it is specified as "*final*".

It is recommended that you carry out this optimization at the end of the development phase. However, it would be a tiresome task to have to go through all the methods of all the classes to check each time whether a method is redefined or not in a subclass.

The "*Non derivable method*" pattern is used to automate this task without the risk of errors.

Prerequisite

The "*Non derivable method*" pattern can be applied to any model class or package.

Applying the pattern

Model before transformation

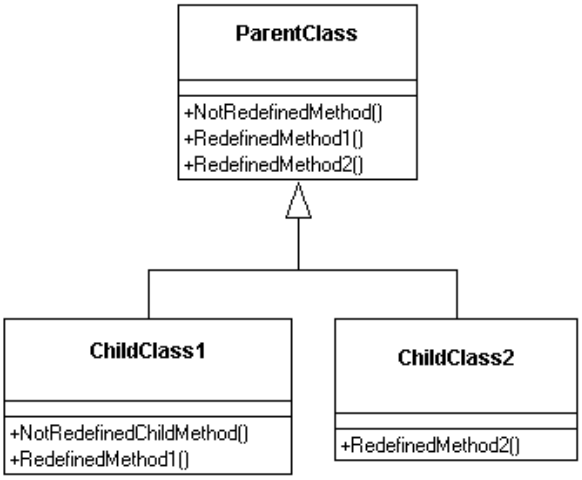


Figure 16-1. Initial model

Calling the pattern

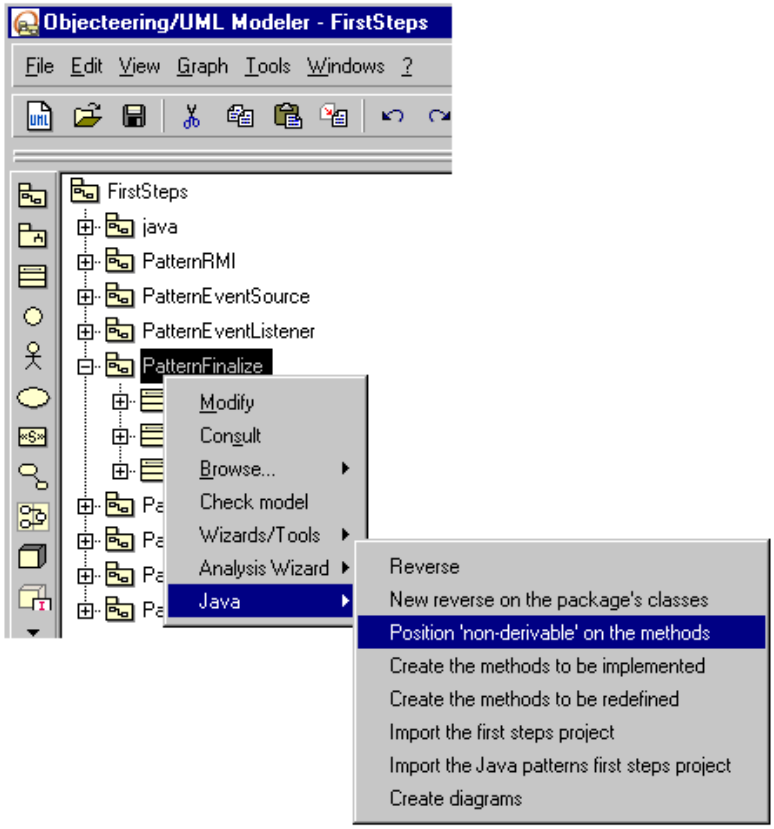


Figure 16-2. Calling the pattern

Methods to implement

Overview

Implementing an interface in Java necessitates the redefinition of all its methods. This detailed task, which can lead to signature copying errors, can be entirely automated.

Prerequisites

The "*Methods to implement*" pattern applies to every model class when the class implements an interface.

Applying the pattern

The model before transformation is as follows:

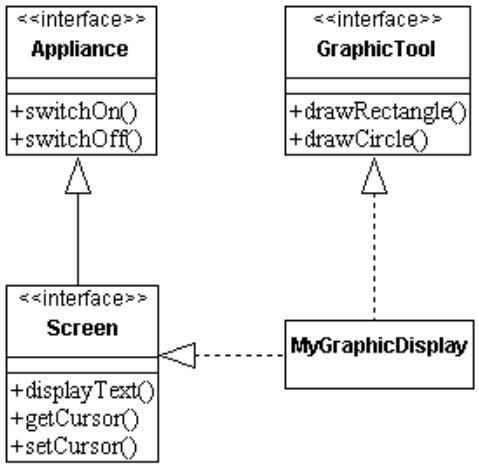


Figure 16-3. The model before transformation

Calling the pattern

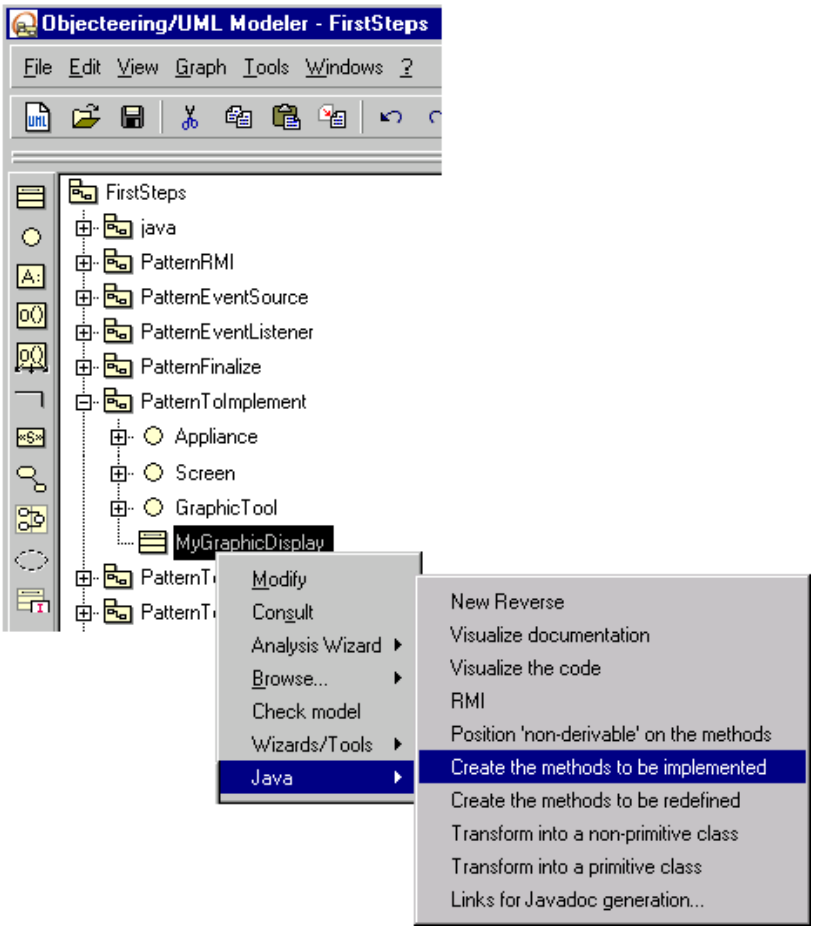


Figure 16-4. Calling the pattern

Consequence of applying the pattern

The model after transformation is as follows:

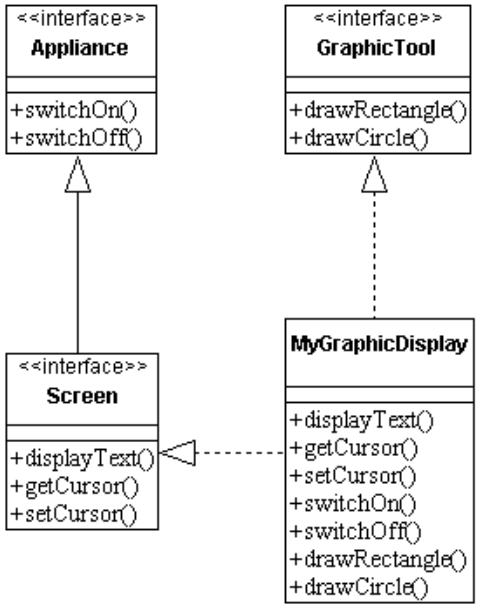


Figure 16-5. The model after transformation

Summary of modifications made to the model

The class to which the pattern is applied is enriched with further operations, which redefine those of the interfaces which it implements.

Methods to redefine

Overview

In Java, for a non abstract class, specializing an abstract class necessitates the redefinition of all its abstract methods. This detailed task, which can lead to signature copying errors, can be entirely automated.

Prerequisites

The "*Methods to redefine*" pattern applies to every non abstract model class when the class has a generalization.

Applying the pattern

The model before transformation is as follows:

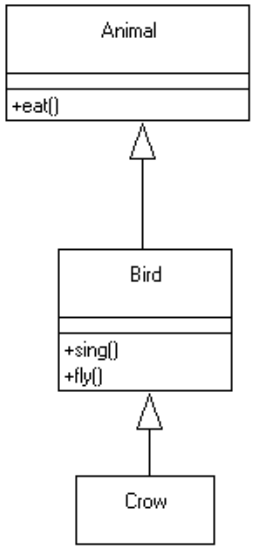


Figure 16-6. The model before transformation

Calling the pattern

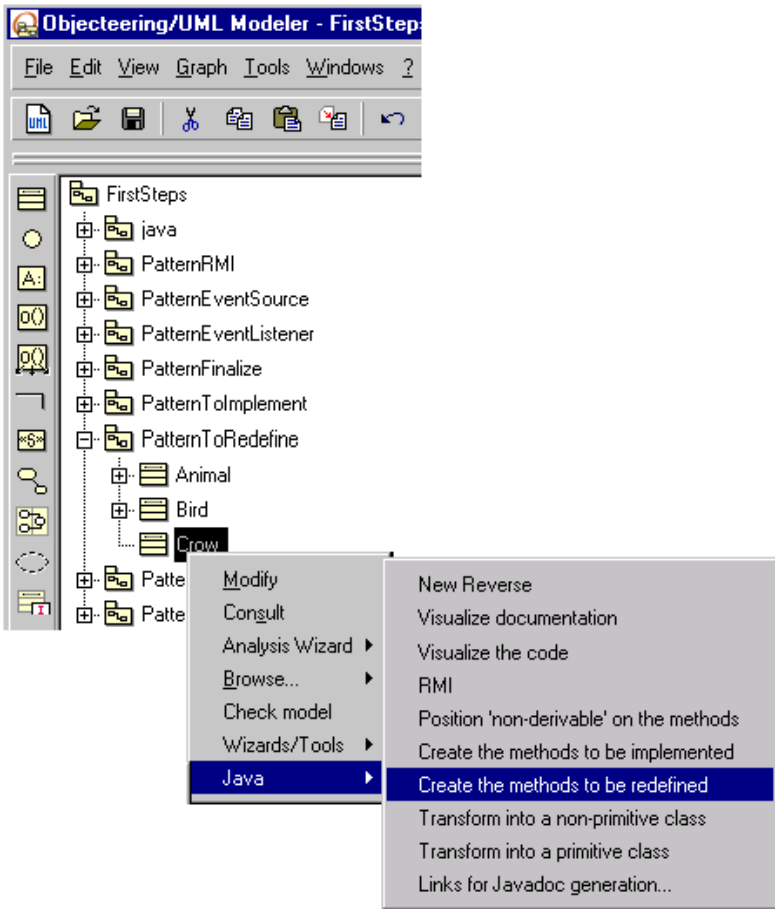


Figure 16-7. Calling the pattern

Consequence of applying the pattern

The model after transformation is as follows:

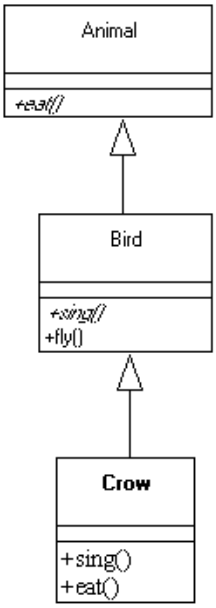


Figure 16-8. Model after transformation

Summary of modifications made to the model

The class to which the pattern is applied is enriched with further operations which redefine the abstract operations of its generalization graph.

Primitive class

Overview

Objecteering/UML is used to provide a semantic nuance between primitive classes, usually intended to be used as a type for attributes, and non-primitive classes, generally used as association destinations.

In the first phase of modeling, this is a distinction which we need not pay attention to.

The present pattern is used in a unique command to transform a non-primitive class into a primitive class, and parallel to this, to transform associations which are directed towards it into attributes.

Prerequisites

This pattern is applied to every non-primitive model class.

Applying the pattern

The model before transformation is as follows:

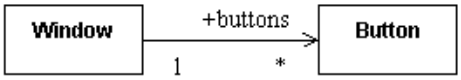


Figure 16-9. The model before transformation

Calling the pattern

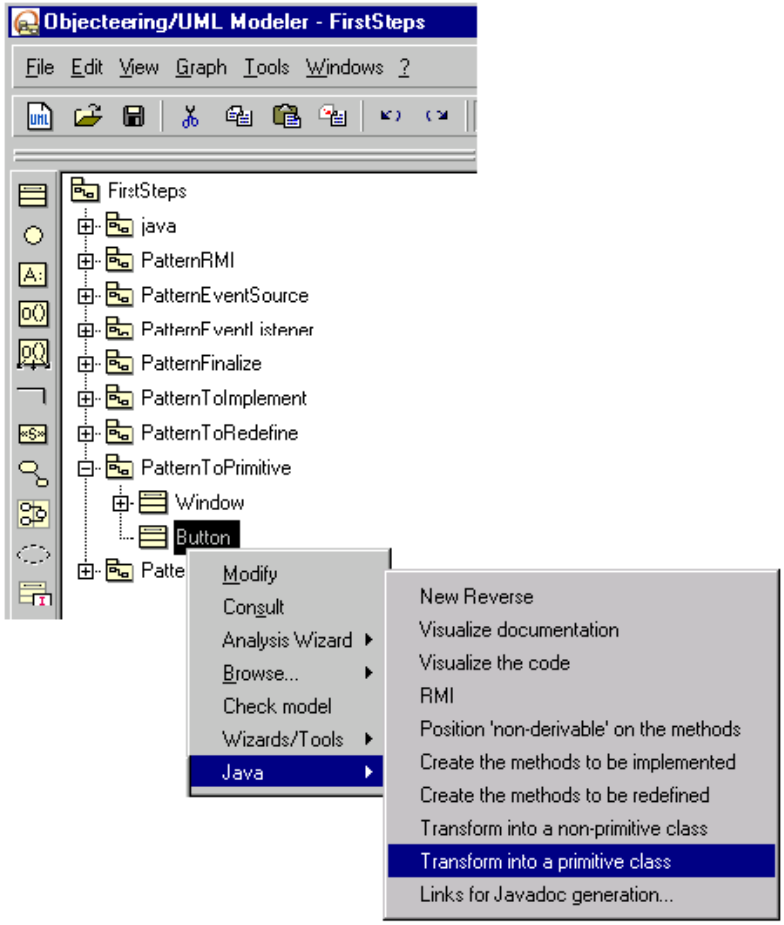


Figure 16-10. Calling the pattern

Consequence of applying the pattern

The model after transformation is as follows:

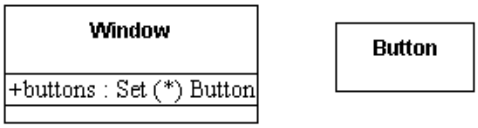


Figure 16-11. The model after transformation

Summary of modifications made to the model

The class to which the pattern is applied is transformed into a primitive class. Parallel to this, navigable associations directed towards this class are transformed into attributes which have this class as its type.

The attribute takes the same name and the same visibility as the association role. Tagged values and notes are carried over, inasmuch as their types exist on two types of object. Stereotypes are carried over inasmuch as they exist on two types of object.

Non-primitive class

Overview

Objecteering/UML is used to provide a semantic nuance between primitive classes, usually intended to be used as a type for attributes, and non-primitive classes, generally used as association destinations.

In the first phase of modeling, this is a distinction which we need not pay attention to.

The present pattern is used in a unique command to transform a primitive class into a non-primitive class, and parallel to this, to transform attributes with this type into associations.

Prerequisites

This pattern is applied to every primitive model class.

Applying the pattern

The model before transformation is as follows:

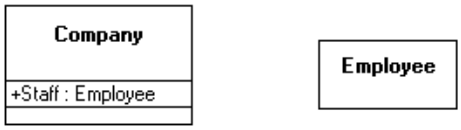


Figure 16-12. The model before transformation

Calling the pattern

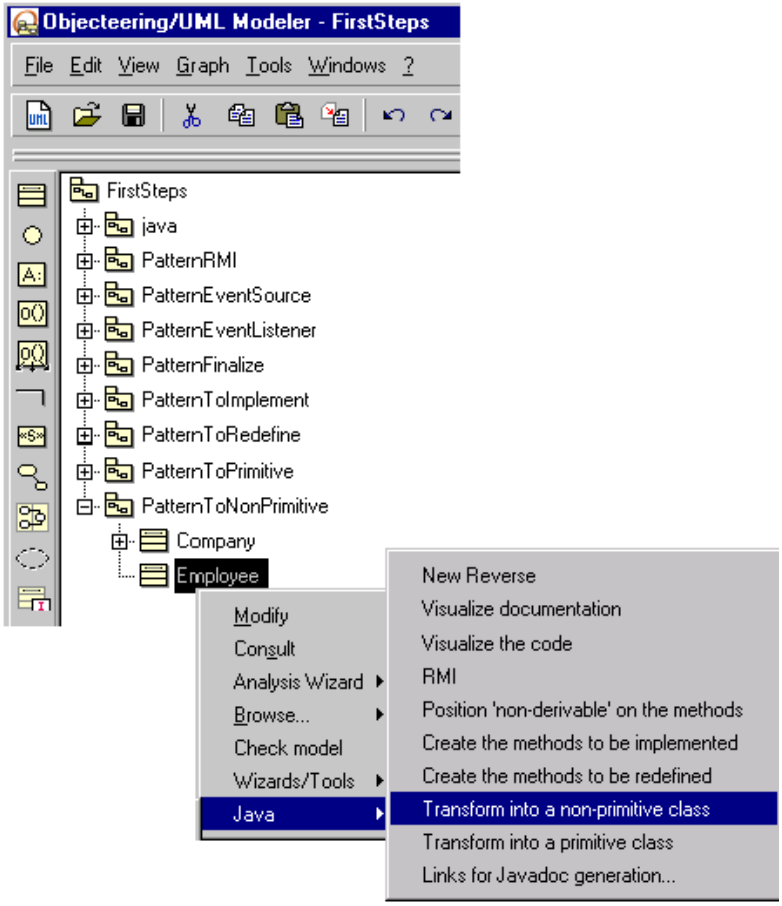


Figure 16-13. Calling the pattern

Consequence of applying the pattern

The model after transformation is as follows:

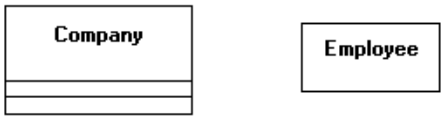


Figure 16-14. The model after transformation

Summary of modifications made to the model

The class to which the pattern is applied is transformed into a non-primitive class. Parallel to this, attributes which have this class as type are transformed into navigable associations directed towards this class.

The navigable link takes as its role name the same name as the attribute, as well as taking the same visibility as the attribute. Its minimum multiplicity is set to 0, with the maximum multiplicity being the size of the attribute. Tagged values and notes are carried over, inasmuch as their types exist on two types of object. Stereotypes are carried over inasmuch as they exist on two types of object.

Chapter 17: Customizing Java generation

Customizing Java generation - Overview

Overview

Objecteering/Java provides several modes of customization, addressed to different categories of users. The very complete Java generation customization allows the user to generate the exact programming form desired, by adapting the *Objecteering/Java* module. For example, it is possible to change the mapping of types and the generation of accessors used by default, to create specific generations or to respect forms of imposed programming, and so on.

The customization modes

Depending on the level of detail of the desired parameterization, the user may:

- modify configuration parameters and simply change the general options of the *Objecteering/Java module*.
- configure the mapping of types and the generation of accessors. The choice of basic types and the way of using them in code generation is then adapted to all generations.
- parameterize the generation document template. This is the most detailed level of parameterization. Code generation can be completely customized, by adapting the structure of a generated Java source, as well as its content.

Configuration window

Overview

The Java module configuration window can be used to modify the behavior of the generator regarding the following elements:

- generation directories
- generation options
- project used to map types and to generate accessors
- generation document templates (code and makefile)
- UML profiles which contain the J rules
- visibility parameters

Note: The names and internal names of each parameter are detailed below. The internal name is only useful for the "J" designer and is used to call the *getCurrentModuleParameterValue* method.

The "General" group

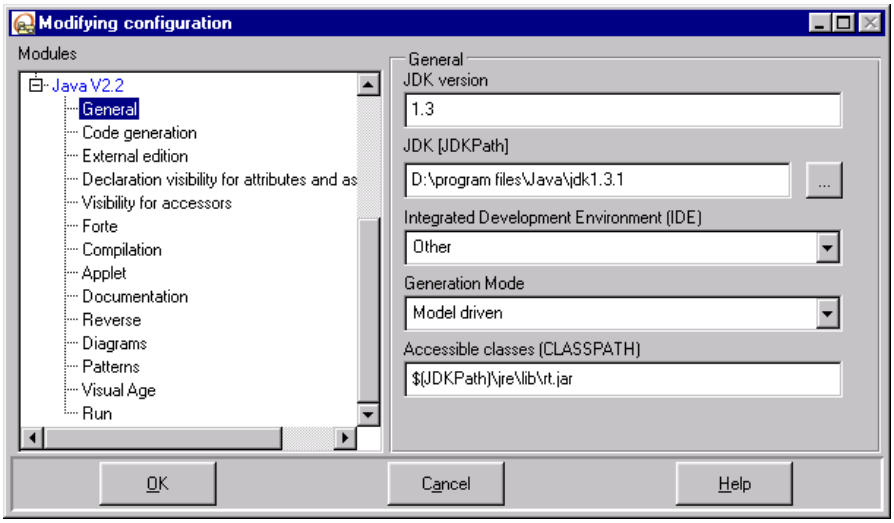


Figure 17-1. The "*General*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
JDK version	JDKVersion	the version of the JDK which is used.
JDK (JDKPath)	JDKPath	the root directory of the JDK.
Integrated Development Environment (IDE)	IDE	the integrated development environment which is to be used during generation.
Generation Mode	GenerationMode	the mode selected for generation (*model driven* or *round trip*).
Accessible classes (CLASSPATH)	AccessibleClasses	the paths for searching for imports. The separator is ";" for Windows and ":" for UNIX

Note1: For further information on the model driven generation mode and the round trip generation mode, please refer to chapter 9 of this user guide.

Note 2: For further information on integrated development environments, please refer to chapter 10 of this user guide.

The "Code generation" group

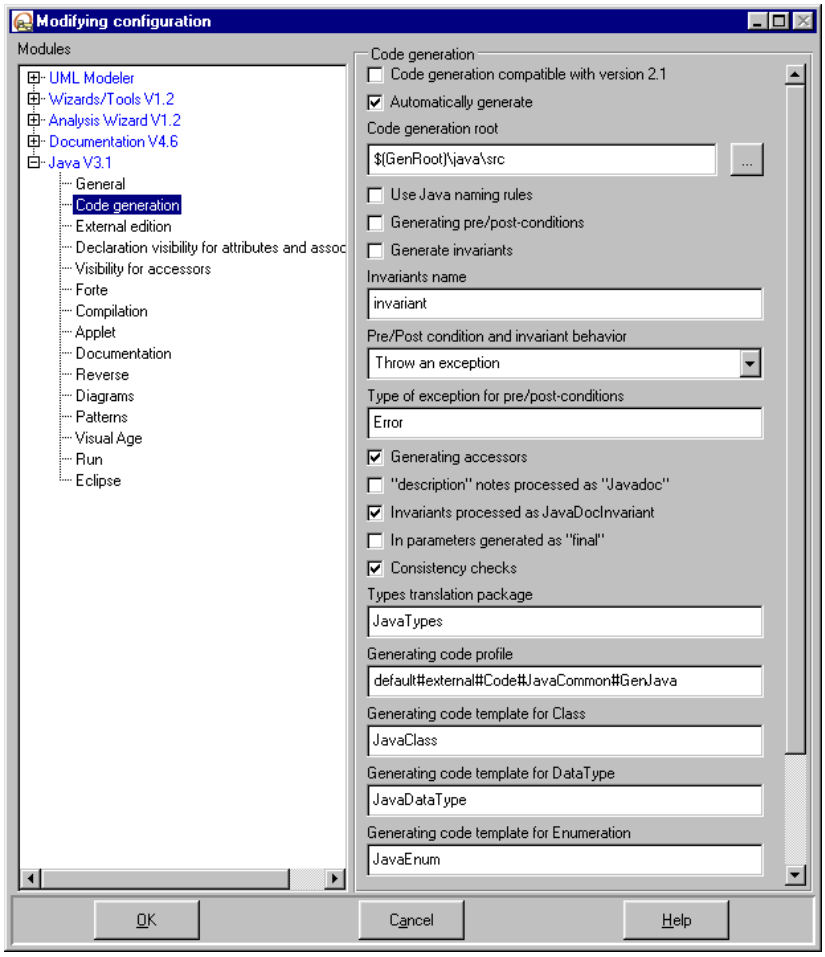


Figure 17-2. The "*Code generation*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Code generation compatible with version 2.1	Compatible	whether or not to generate the same code for Enumerations as earlier versions, the visibility of attributes and associations, and their access methods.
Automatically generate	autoGenerate	whether or not generation should be run automatically.
Code generation root	GenerationPath	the root in which .java files are generated.
Use Java naming rules	UseJavaNaming	that during generation, the names of packages, classes, methods, attributes and parameters will be generated, whilst respecting Java recommendations concerning the case (upper or lower) of the first letter.
Generating pre/post-conditions	PrePostGeneration	whether the pre and post conditions should be generated on the methods.
Generate invariants	generateInvariant	whether not not invariants should be generated.
Invariants name	invariantsName	the names of the generated invariants methods.
Pre/Post condition and invariant behavior	errorType	whether a throw exception or an assertion should be generated.
Type of exception for pre/post-conditions	PrePostExceptionClass	the exception class used in the generated code for the pre and post conditions.
Generating accessors	AccessorsGeneration	the global running of accessor generation.
"*Description*" notes processed as "*Javadoc*"	DescriptionAsJavadoc	the processing of the "*description*" type notes as well as the "*Javadoc*" type notes.
Invariants processed as JavaDocInvariant	generateInvariantAsJavaDocInvariant	that the invariant is generated as if it were in a constraint stereotyped JavaDocInvariant.
In parameters generated as "*final*"	FinalForIn	the generation of the "*final*" keyword for the "in" parameters.
Consistency checks	ConsistencyControl	if the specific Java consistency controls should be executed during a generation.

The ... parameter	internal name ...	indicates ...
Types translation package	TypesTranslationPackage	the name of the project used for translation types and the generation of accessors.
Generating code profile	JavaProfile	UML profile used to generate Java code.
Generating code template for Class	JavaTemplate	Java code generation template used for Classes.
Generating code template for DataType	dataTypeTemplate	Java code generation template used for DataTypes.
Generating code template for Enumeration	EnumTemplate	Java code generation template used for Enumerations.
Use "JavaImport" notes to generate imports	useJavaImportNotes	that "JavaImport" note contents should be used in generation, if the note is present.
Optimize "import" statements	ImportOptimization	whether or not the necessary imports should be generated by simply calculating on classes and not packages. By optimizing Java imports, only the import of necessary classes is generated in the list of java file imports. In this mode, use links between packages are generated in the form of complete package imports. The dependencies of the class indicated by the user or calculated are browsed, and an import is generated for each class which participates in the dependency.
Strong encapsulation for access methods (modifier)	ModifierEncapsulation	whether or not to raise visibility one level to modify access methods of attributes and associations.
Generator behavior if generated file present but not managed	manageMode	generator behavior in the case where the generated file is present but is not managed. Choose from "*Ask*", "*Continue*" or "*Cancel*".

Please note that the last four parameters described do not appear in Figure 17-2.

Please also note that generation options are taken into account during reverses.

Note 1: For more information on consistency checks, please refer to the "*Code generation - Overview*" section in chapter 5 of this user guide. A definition of accessors is provided in the "*Glossary*" section in chapter 1. For details on notes, please refer to the "*Java documentation generation - Note types*" section in chapter 7.

Note 2: To find out more about the types translation package, please refer to the "*Principles of type and accessor generation* " section in chapter 17 of this user guide. In the same chapter, the "*Code generation document templates*" section provides details on code generation templates.

The "External edition" group

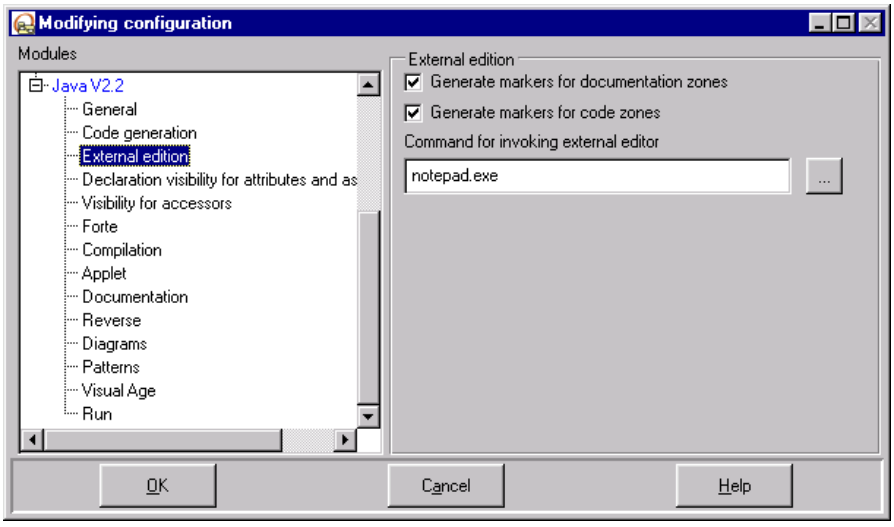


Figure 17-3. The "*External edition*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Generate markers for documentation zones	DocIdGenerated	whether or not markers should be generated for documentation zones.
Generate markers for code zones	IdGenerated	whether or not markers should be generated for code zones.
Command for invoking external editor	ExtEditorCommandLine	the command used to launch an editor to modify the generated code.

The "Declaration visibility for attributes and associations" group

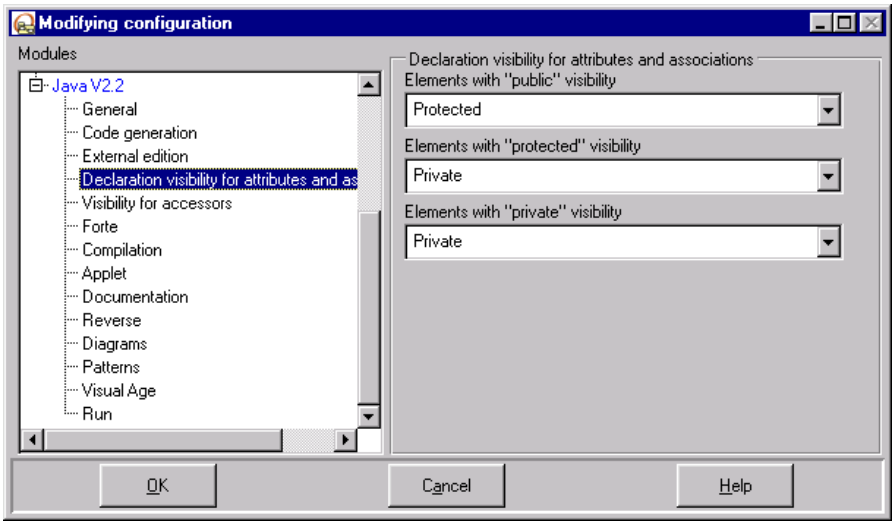


Figure 17-4. The "*Declaration visibility for attributes and associations*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	is used to ...
Elements with "public" visibility	publicDataMemberVisibility	declare the visibility of the Java instance variable generated for public attributes.
Elements with "protected" visibility	protectedDataMemberVisibility	declare the visibility of the Java instance variable generated for protected attributes.
Elements with "private" visibility	privateDataMemberVisibility	declare the visibility of the Java instance variable generated for private attributes.

The "Visibility for accessors" group

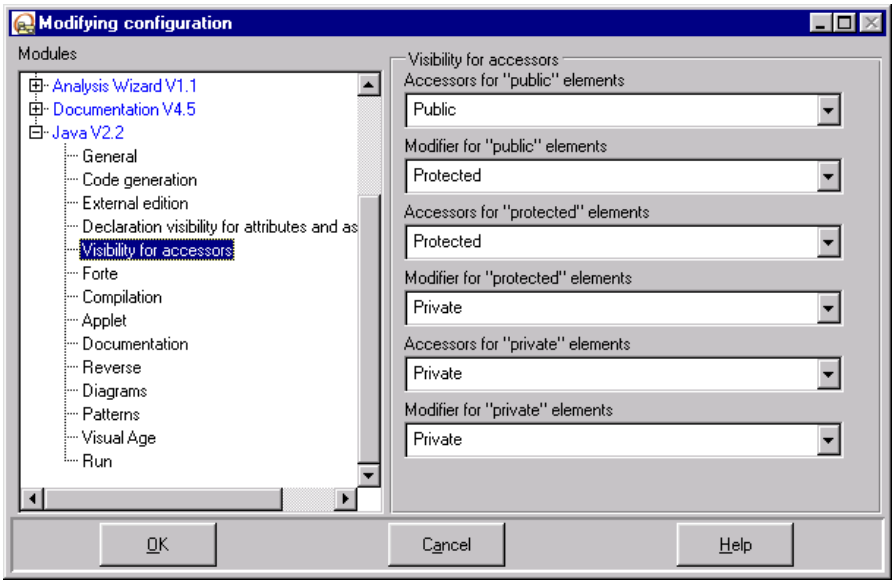


Figure 17-5. The "*Visibility for accessors*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	is used to ...
Accessors for "public" elements	publicAccessorVisibility	declare the visibility of "get" type accessors for all public attributes.
Modifier for "public" elements	publicModifierVisibility	declare the visibility of "set" type accessors for all public attributes.
Accessors for "protected" elements	protectedAccessorVisibility	declare the visibility of "get" type accessors for all protected attributes.
Modifier for "protected" elements	protectedModifierVisibility	declare the visibility of "set" type accessors for all protected attributes.
Accessors for "private" elements	privateAccessorVisibility	declare the visibility of "get" type accessors for all private attributes.
Modifier for "private" elements	privateModifierVisibility	declare the visibility of "set" type accessors for all private attributes.

Note: The {JavaPublic} tagged value still takes priority in visibility management with regard to module parameters.

The "Forte" group

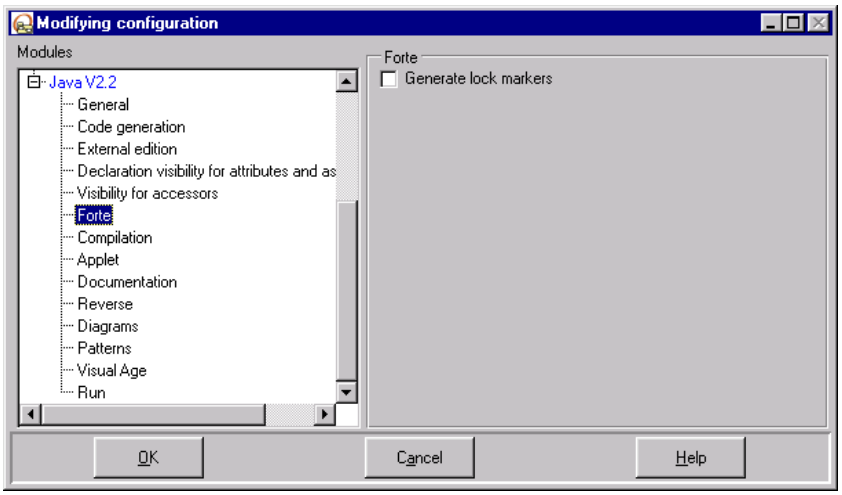


Figure 17-6. The "*Forte*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	is used to ...
Generate lock markers	genForteMarker	activate generation of Forte lock markers, which prevent the modification, in Forte, of code zones generated by the Java module.

The "Compilation" group

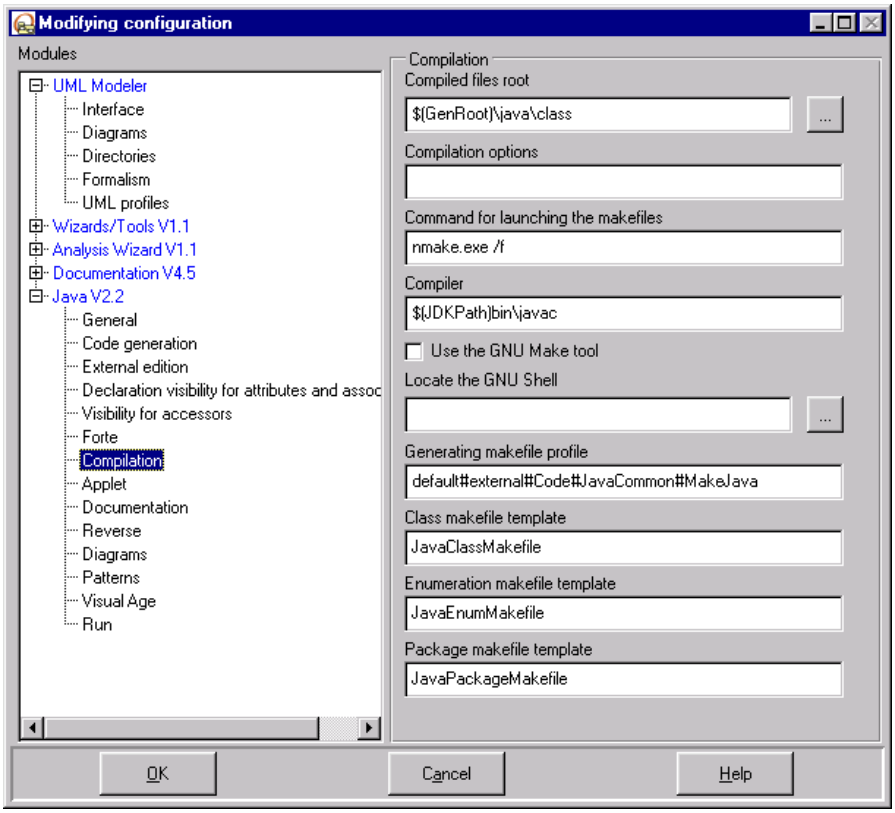


Figure 17-7. The "*Compilation* " sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Compiled files root	CompilationPath	the root in which *.class* files are generated.
Compilation options	CompilationOptions	the compilation options (aimed at Java).
Command for launching the makefiles	MakeCommand	the command used to activate makefile files.
Compiler	JavaCompiler	the Java compiler.
Use the GNU Make tool	UseGNUMake	that the GNU *make* tool should be used for compilation, which has an impact on the makefile generation. This option does not function in UNIX. When this option is selected, it is necessary to give the correct path for the *make* tool in the corresponding parameter of the "*External edition*" group. Where this tickbox is checked, the value of the "*Command for launching the makefiles*" parameter must be in "*make -f*" form.
Locate the GNU shell	GNUShell	path of the sh.exe program. This is located in the *bin* directory of the *cyg-win32* directory, and is only taken into account if the previous one has been selected. For example : if *cygnus* tools have been installed in the *c:\cygnus* directory, the value of the parameter must be in "*c:\cygnus\bin\sh.exe*" form.
Generating makefile profile	MakeProfile	the UML profile used to generate the Java compilation makefile.
Class makefile template	ClassMakefile	the document template which guides makefile generation for a class.
Enumeration makefile template	EnumMakefile	the document template which guides makefile generation for an enumeration.
Package makefile template	PackageMakefile	the document template which guides makefile generation for a package.

Note 1: Where CYGNUS is concerned, Objecteering/UML stores the CYGNUS package in the "$OBJING_PATH\bin" sub-directory. This package can be extracted using "usertools.exe" or "cdk.exe".

Note 2: For further information on makefile generation document templates, please refer to the "*Makefile generation document templates*" section in chapter 17 of this user guide.

The "Applet" group

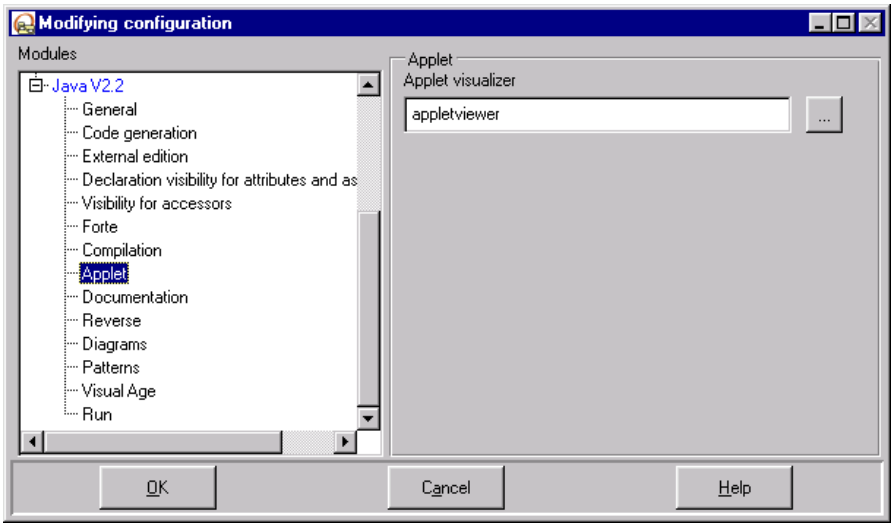


Figure 17-8. The "*Applet*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Applet visualizer	AppletVisualizer	the command used to launch an applet.

The "Documentation" group

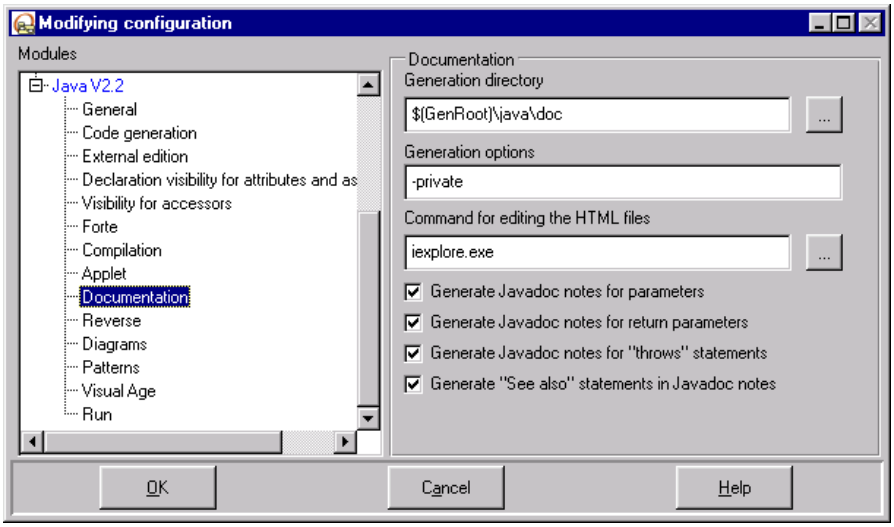


Figure 17-9. The "*Documentation*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Generation directory	GenDocPath	the directory in which HTML files are produced by Java documentation generation.
Generation options	JavaDocOptions	options to transmit to the javadoc JDK tool for the generation of Java documentation.
Command for editing the HTML files	HTMLCommandEditor	the command used to visualize the generated Java documentation
Generate Javadoc notes for parameters	paramJavadoc	whether or not Javadoc notes on parameters should be generated in your application's source code.
Generate Javadoc notes for return parameters	returnJavadoc	whether or not Javadoc notes on return parameters should be generated in your application's source code.
Generate Javadoc notes for "throws" statements	throwsJavadoc	whether or not Javadoc notes on links stereotyped <<throws>> should be generated in your application's source code.
Generate "See also" statements in Javadoc notes	seeJavadoc	whether or not "See also" statements, used to add "See also" links to the documentation generated by Javadoc, should be generated.

Note: For further details on documentation generation, please refer to chapter 7 of this user guide.

The "Reverse" Group

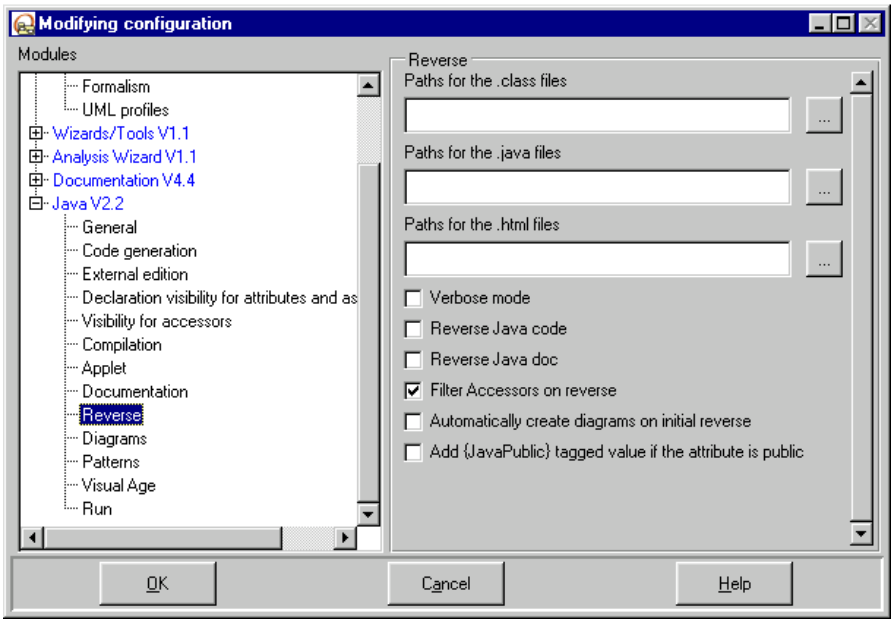


Figure 17-10. The "*Reverse*" sub-set of "*Java*" module parameters

The ... command	internal name ...	has the role of ...
Paths for the *.class* files	CompiledFilesPath	containing the *.class* file root paths to be reversed. This string is made up of a list of paths separated by ":" for UNIX versions and ";" for Windows versions. These paths can be *.zip* files, *.jar* files or directories which have *CLASSPATH* semantics. They cannot, however, contain directories which correspond to Java packages. If you enter the current directory followed by "*", all the directories, .zip files and .jar files contained in the current directory are displayed.
Paths for the *.java* files	SourceFilesPath	containing the *java* files root paths. It is also used to visualize the Java code corresponding to a class. This string is made of a list of paths separated by ":" for UNIX versions, and ";" for Windows versions. They cannot, however, contain directories which correspond to Java packages.
Paths for the *.html* files	HTMLFilesPath	containing the html file paths corresponding to the classes imported into the repository for visualizing their documentation. This string is made up of a list of paths separated by ":" for the UNIX versions, and ";" for the Windows versions. They cannot, however, contain directories which correspond to Java packages.
Verbose mode	VerboseMode	choosing to display an exhaustive trace of object creation operations during a reverse operation.
Reverse Java code	ReverseJavaCode	choosing to reverse operations' code.

The ... command	internal name ...	has the role of ...
Reverse Java doc	ReverseJavaDoc	choosing to reverse documentation data (class documentation is extracted from .html files and put into the model in the form of JavaDoc notes).
Filter Accessors on Reverse	AccessorFilter	filtering accessors when a reverse operation is carried out. "Canonical" access operations (getXXX, setXXX) can be filtered during the reverse. Visibility is, however, retained. In round trip mode, this mode must be used systematically, except in certain particular cases.
Automatically create diagrams on initial reverse	DiagramCreationOnReverse	indicating whether or not diagrams should be automatically created when the initial reverse operation is carried out. Diagrams can also be created on subsequent request using the "*Create Diagrams*" command on packages.
Add {JavaPublic} tagged value when public	AddJavaPublicTagWhenPublic	automatically adding the {JavaPublic} tagged value, where the element is public.

Note 1: The reverse tool searches for the java source file in the directories indicated by the "*Paths for the .java files*" parameter. When more than one source file of the same name exists in these paths, the first file found is taken into account.

Note 2: *.class* file paths can be *.zip* files, *.jar* files or directories which have *CLASSPATH* semantics. For example, if you want to reverse class "*C*" of package "*P*", the *C.class* file is located in "...*class**P**C.class*". However, *CLASSPATH* semantics are simply "...*class*", and this is what should be defined for this parameter, instead of the full "...*class**P*" path.

Note 3: For further information on reverse operations, please refer to the chapter 8 of this user guide.

The "Patterns" group

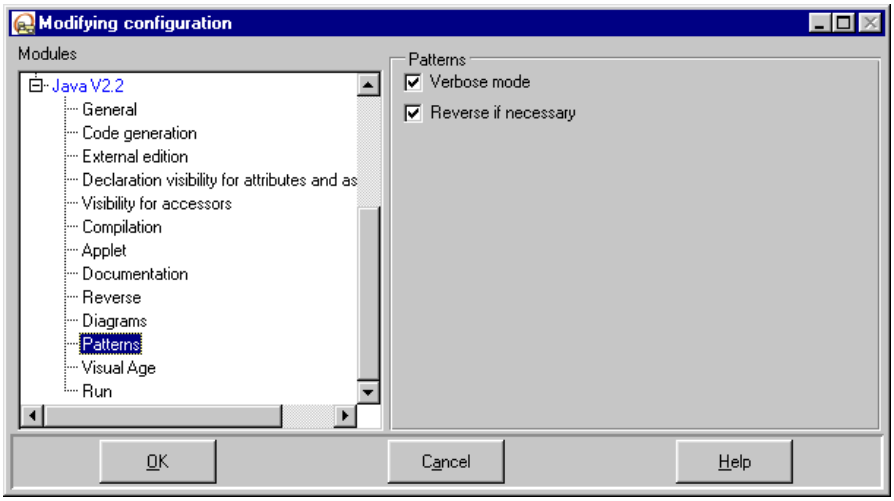


Figure 17-11. The "*Patterns*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Verbose mode	VerboseMode	the activation of traces during the running of the patterns.
Reverse if necessary	DoReverse	whether or not a reverse operation should be run, where necessary.

The "Visual Age" group

For details on the "*Visual Age*" module parameter sub-set, please refer to the "*Visual Age/Objecteering integration*" section in chapter 10 of this user guide.

The "Run" group

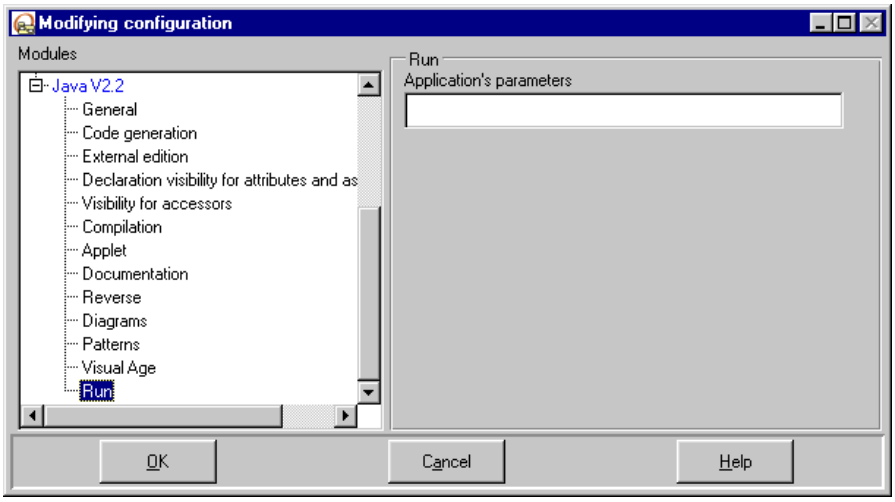


Figure 17-12. The "*Run*" sub-set of "*Java*" module parameters

The ... parameter	internal name ...	indicates ...
Application's parameters	runParameters	the default parameters for the application to be run.

The "Eclipse" group

For details on the "*Eclipse*" module parameter sub-set, please refer to the "*Parameterizing Objecteering/Eclipse* " section in chapter 11 of this user guide.

Principles of type and accessor generation

General remarks

The *Objecteering/Java* module is delivered with the "*JavaTypes*" package. This package is an example of a type mapping and accessor generation package, and can be modified. It is also possible to create another type mapping and accessor generation package.

Role of the JavaTypes package

This kind of package supports:

- the mapping of Java attribute types generated from the attributes of the model classes
- the mapping of the Java attribute types generated from the associations between the model classes
- the mapping of the operation parameter types
- the generation of the Java attribute declarations generated from the attributes of the model classes
- the mapping of the Java attribute declarations generated from the relations between the model classes
- the generation of the operation parameter declarations
- the generation of the Java attribute accessors generated from the attributes of the model classes
- the mapping of the Java attribute accessors generated from the associations between the model classes

Structure of the JavaPredType project

A type mapping and accessor generation package must reference the "*TypesEditor*" module.

This project contains:

- a type's package, named "*BaseTypes*", for mapping basic types
- a package named "*DefaultTranslations*", which defines the generation of default accessors
- a package named "*TranslationClasses*" which contains all accessor generation classes

Mapping types

The "*BaseTypes*" package supports this mapping. The mapping of a given type is represented by an Objecteering/UML type.

This package is used as the mapping for the standard Objecteering/UML types declared in the "*_predefinedTypes*" project, for example *integer* and *string*. As it is not possible to create a type which has exactly the same name as a type of the "*_predefinedTypes*" project, a "_" is added. For example, the mapping of the *integer* type is carried out by the *_integer* type.

This package can also be used as the mapping for any other type, represented by a class, a type or an enumerate in an Objecteering/UML model. In this case, it is necessary to create a type with the same name as the class, the type or the enumerate in the "*BaseTypes*" package.

If the generation does not find either a T1 type or a _T1 in this package, then the name T1 is used in the Java file. Thus, it is generally not necessary to create a type for each class, type or enumerate of a model before launching Java code generation.

If the type which is searched for exists, a *targetType* note is then looked for. If this text does not carry the {*Jeval*} tagged value, the Java type is then contained in the note itself. If it carries this tagged value, the Java type is the J evaluation of the text content. The text is evaluated in the context of the "*GeneralClass*" (metaclass name, representing classes or types) representing the type to map. A communication protocole then exists with the generator. The *JavaElement* variable represents the element (attribute, association or parameter link) which bears the type to be mapped. The Java type must be placed in the *JavaReturn* variable with the "String" J type.

Examples of mapping for the *string and integer* types

The rule for mapping a *string* in Java is given by the *_string* type. Its *targetType* text does not carry the {*Jeval*} tagged value. The Java type is, therefore, the content of the text itself, in other words, *String*.

The rule for mapping an *integer* in Java is given by the *_integer* type. Its *targetType* note carries the {*Jeval*} tagged value. The Java type is therefore provided by the J evaluation of this note.

For example, if the object (attribute or parameter) with this type carries the {*JavaLong*} tagged value, *JavaReturn* gets the "long" value. If the object with this type carries the {*JavaShort*} tagged value, *JavaReturn* gets the "short" value. If the object with this type carries the {*JavaByte*} tagged value, *JavaReturn* gets the "byte" value. If the object does not carry any of these tagged values, *JavaReturn* gets the "int" value. If the multiplicity is multiple, *Long*, *Short*, *Byte* or *Integer* will be used to allow insertion in the lists.

Generating the declarations and accessors

The project's classes support the generation of accessors and declarations. When the generator has to process the case of an attribute, association end, or parameter, it first determines the class that will be the support. If the object (attribute, association end or parameter) carries a {*type*} tagged value, the class that is used is the one whose name corresponds to its parameter. Otherwise, the class used is the one referenced by the package that corresponds to the case of the object.

In the case of a parameter, only the method with the *declare* predefined name is taken into consideration.

In the case of an attribute or association end, methods which are not annotated with the {*notDefault*} tagged value are taken into account. Added to these are the ones whose names correspond to the parameter of a Java tagged value {*JavaGenerateAccessor*} annotating the object. We then delete those classes whose names correspond to the parameter of a {*JavaFilterAccessor*} tagged value carried by the object. If the object carries the tagged value {*JavaNoAccessor*}, only the method named *declare* is taken into account.

For the method named *declare, Objecteering/Java* simply evaluates the J note of its body in J.

For the other selected methods, Objecteering/UML calculates the accessor as follows:

J evaluation of	**provides information about...**
the operation's *JModifiers* note	the accessor's modifiers.
return parameter J note	the accessor's type of return.
the operation's *JName* note	the accessor's name.
the operation's *JExceptions* note	the exceptions sent back by the accessor.
J text of each parameter	an accessor's parameter.
some operation body's J note	the accessor's body.

Each evaluation must inform the variable named *JavaReturn* with the J "String" type.

Two tagged values are available for the definition of accessor types:

The ... type tagged value	designates a ... accessor
{access}	read-only
{modify}	read-write

Note: The accessor which calculates the multiplicity of an association must be called "*card*".

Useful attributes

The following table presents those attributes which can be useful when personalizing a types package.

Metaclass	Type	Name	is ...
Object	int	Val_indent	a J variable used by the Java generator to memorize the current indentation (please see the Indent method below)
AssociationEnd	String	MultiplicityMin	an attribute containing the minimum multiplicity for the association
AssociationEnd	String	MultiplicityMax	an attribute containing the maximum multiplicity for the association
Attribute	String	Multiplicity	an attribute containing the attribute's multiplicity
Attribute	boolean	IsSet	an attribute indicating whether the attribute has simple or multiple multiplicity
Parameter	String	Multiplicity	an attribute containing the parameter's multiplicity
Parameter	boolean	IsSet	an attribute indicating whether the parameter has simple or multiple multiplicity

Useful methods

The following table presents those methods which can be useful when personalizing a types package, either to redefine them or simply to use them.

Metaclass	Name	Role	Default implementation
Object	String firstLetterToLower (in String str)	Puts the first letter of a given string in lower case	Returns a string identical to that used as a parameter, with the possible exception of the first letter, which is in lower case
Object	String Indent (in int nb)	Calculates the indentation	Returns a string containing a number of spaces equal to the parameter (please see the Val_indent attribute above)
ModelElement	boolean isTaggedValue (in String type)	Predicate indicating whether an object carries a tagged value or a given type	Searches the receiving object for a tagged value which has a TagType whose Name attribute has the value of the parameter
Feature	String FieldModifiers()	Calculates the corresponding Java attribute modifiers	Returns the "public static final" if the container is an interface, a string aggregating visibility (please see the getJavaVisibility method) and, optionally, the "static" (please see the StaticModifier method), "final" (if the object carries the {JavaFinal} tagged value), "volatile" (if the object carries the {JavaVolatile} tagged value) and "transient" (if the object carries the {JavaTransient} tagged value) keywords.
Feature	String StaticModifier()	Manages the "static" keyword	Returns the "static" string if the object is "class", and an empty string if this is not the case

Metaclass	Name	Role	Default implementation
Feature	String getJavaVisibility()	Manages the Java visibility of Java attributes generated	If the object has public visibility, returns the "public" string, if the accessors are generated for the generated attribute using the object and if this object does not carry the {JavaPublic} tagged value. If the object has protected or private visibility, call the getVisibility method
Feature	String getVisibility()	Maps UML visibility to Java visibility	Public ➔ public Protected ➔ protected Private ➔ private Undefined ➔ friendly, i.e. empty string
Feature	String getAccessModifiers()	Manages the modifiers of accessors in read-only mode	Concatenation of the returns of the getAccessVisibility and StaticModifier methods
Feature	String getAccessVisibility()	Manages the visibility of accessors in read-only mode	Returns the "public" string.
Feature	String getModifyModifiers()	Manages the modifiers of accessors in read-write mode	Concatenation of the returns of the getModifyVisibility and StaticModifier methods
Feature	String getModifyVisibility()	Manages the visibility of accessors in read-write mode	Calls the getVisibility method

Metaclass	Name	Role	Default implementation
Feature	String MemberName()	Handling string of the corresponding Java attribute	If the object is "class", the owner class is concatenated and the object name is separated by ".". If this is not the case, this is concatenated and the name of the object is separated by ".".
Feature	String SimpleMemberName()	Name to be used to declare the Java attribute and the calculation of accessor names	Name of the object with the first letter in lower case, if the "Use Java naming rules" parameter of the "Code generation" group is checked
AssociationEnd	Class getTranslateClass()	Calculation of the class being used to support the generation of the Java attribute declaration and its accessors	The class is searched for in the package of the "*_predefinedTypes*" project, whose name is given by the "Type translation package" parameter of the "Code generation" group. The class returned depends on the parameter of the {type} tagged value, if the association carries one. If this is not the case, it is the default class (please refer to the "*Overview of type and accessor generation*" section in chapter 17 of this user guide)
AssociationEnd	String AssociationEndType()	Management of the Java type to generate for the corresponding Java attribute	Returns the parameter of the {JavaTypeExpr} tagged value or the destination class

Metaclass	Name	Role	Default implementation
AssociationEnd	String AssociationEndInitValue ()	Management of the initial value of the corresponding Java attribute	If the association carries a {JavaInitValue} tagged value, returns the concatenation of the "=" string and the value of the tagged value. If this is not the case, an empty string is returned
Attribute	boolean useWrapper()	Predicate indicating whether or not a scalar type (ex : int) or the equivalent class (ex : Integer) should be generated By default, this method is used in the BaseTypes package of the types package	Returns true in two cases: The IsSet attribute is true and the isArray method gives false The attribute carries the {JavaWrapper} tagged value
Attribute	boolean isArray()	Predicate indicating whether or not the attribute has a table type	Returns true if the class returned by the getTranslateClass method has a name which begins by "array".

Metaclass	Name	Role	Default implementation
Attribute	Class getTranslateClass()	Calculation of the class used as generation support of the Java attribute declaration and of its accessors	The class is searched for in the package of the "*_predefinedTypes*" project, whose name is given by the "Type translation package" parameter of the "Code generation" group. The class returned depends on the parameter of the {type} tagged value, if the attribute carries one. If this is not the case, it is the default class (please refer to the "*Overview of type and accessor generation*" section in chapter 17 of this user guide)
Attribute	String AttributeType()	Management of the Java type to generate for the corresponding Java attribute	Returns the parameter of the {JavaTypeExpr} tagged value
Attribute	String AttributeInitValue()	Management of the initial value of the corresponding Java attribute	If the attribute has a non-empty initial value, the concatenation of the "=" string and this value is returned.

Metaclass	Name	Role	Default implementation
Parameter	Class getTranslateClass()	Calculation of the class used as generation support of the Java parameter declaration	The class is searched for in the package of the "*_predefinedTypes*" project, whose name is given by the "Type translation package" parameter of the "Code generation" group. The class returned depends on the parameter of the {type} tagged value if the parameter carries one. If this is not the case, it is the default class (please refer to the "*Overview of type and accessor generation*" section in chapter 17 of this user guide)
Parameter	boolean useWrapper()	Predicate indicating whether or not a scalar type (ex : int) or the equivalent class (ex : Integer) should be generated	Returns true if the parameter carries the {JavaWrapper} tagged value
Parameter	boolean isArray()	Predicate indicating whether or not the parameter has a table type	Returns true if the class returned by the getTranslateClass method has a name starting with "array".
Parameter	String ParameterModifiers()	Calculation of the corresponding Java parameter modifiers	Returns "final" if the passing mode of the parameter is "in" and the "In parameters generated as final" parameter of the "Code generation" group is checked.

Metaclass	Name	Role	Default implementation
Parameter	String ParameterType()	Management of the Java type to generate for the corresponding Java parameter	Returns the parameter of the {JavaTypeExpr} tagged value or the parameter type.
Parameter	String ParameterName()	Name to be used for the declaration of the Java parameter	Name of the object with the first letter in lower case if the "Use Java naming rules" parameter of the "Code generation" group is checked

Induced imports

The class can carry one or more {*import*} tagged values, which designate the classes to be imported, after implementation of the declarations and accessors.

Overview of type and accessor generation

Generation of default accessors

The {*type*} tagged value can be used in a model, attribute, association end or parameter to designate the class to use for generating the declaration and/or accessors.

To avoid having to create a tagged value for each attribute, association end or parameter, the generator is informed of default generations by packages with defined names. These reference the class to be used in the absence of the {type} tagged value. These packages are found in the "*JavaTypes*" package's "*DefaultTranslations*" package.

The ... package name	generates by default ...
SimpleAttribute	the declaration and the accessors for an attribute sized 1.
MultipleAttribute	the declaration and the accessors for an attribute sized *.
FiniteAttribute	the declaration and the accessors for an attribute with the size n>1.
OptionalSimpleAssociation	the declaration and the accessors for an 0-1 association.
MandatorySimpleAssociation	the declaration and the accessors for a 1-1 association.
OptionalMultipleAssociation	the declaration and accessors for a 0-* association.
MandatoryMultipleAssociation	the declaration and accessors for a 1-* association.
FiniteAssociation	the declaration and the accessors for a relation n-m, where m>1.
SimpleIOParameter	the declaration for a parameter sized 1.
MultipleIOParameter	the declaration for a parameter sized *.
FinitelOParameter	the declaration for a parameter sized n>1.
SimpleReturnParameter	the declaration for a return parameter sized 1.
MultipleReturnParameter	the declaration for a return parameter sized *.
FiniteReturnParameter	the declaration for a return parameter sized n>1.

Each of these packages references the class which is used by default in the case dealt with by the package. Modifying this referencing is a simple means of modifying the default behavior of the generator.

Classes which customize types and accessors in the JavaTypes package

The ... class	is used to ...
simpleAssociation	generate the declaration and accessors for an association with a simple multiplicity.
VectorMultipleAssociation	generate the declaration and accessors on the Vector class basis for an association with multiple multiplicity.
VectorFiniteAssociation	generate the declaration and accessors on the Vector class basis for an association with finite multiplicity.
ArrayMultipleAssociation	generate the declaration and accessors on the basis of a table for an association with mutliple multiplicity.
ArrayFiniteAssociation	generate the declaration and accessors on the basis of a table for an association with finite multiplicity.
StackMultipleAssociation	generate the declaration and accessors on the Stack class basis for an association with multiple multiplicity.
StackFiniteAssociation	generate the declaration and accessors on the Stack class basis for an association with finite multiplicity.
HashtableMultipleAssociation	generate the declaration and accessors on the Hashtable class basis for an association with multiple multiplicity.
HashtableFiniteAssociation	generate the declaration and accessors on the Hashtable class basis for an association with finite multiplicity.
ListMultipleAssociation	generate the declaration and accessors on the List class basis for an association with multiple multiplicity.
ListFiniteAssociation	generate the declaration and accessors on the List class basis for an association with finite multiplicity.
LinkedListMultipleAssociation	generate the declaration and accessors on the LinkedList class basis for an association with multiple multiplicity.
LinkedListFiniteAssociation	generate the declaration and accessors on the LinkedList class basis for an association with finite multiplicity.
CollectionMultipleAssociation	generate the declaration and accessors on the Collection class basis for an association with multiple multiplicity.
CollectionFiniteAssociation	generate the declaration and accessors on the Collection class basis for an association with finite multiplicity.
SetMultipleAssociation	generate the declaration and accessors on the Set class basis for an association with multiple multiplicity.
SetFiniteAssociation	generate the declaration and accessors on the Set class

The ... class	is used to ...
	basis for an association with finite multiplicity.
HashSetMultipleAssociation	generate the declaration and accessors on the hashSet class basis for an association with multiple multiplicity.
HashSetFiniteAssociation	generate the declaration and accessors on the hashSet class basis for an association with finite multiplicity.
MapMultipleAssociation	generate the declaration and accessors on the Map class basis for an association with multiple multiplicity.
MapFiniteAssociation	generate the declaration and accessors on the Map class basis for an association with finite multiplicity.
HashMapMultipleAssociation	generate the declaration and accessors on the HashMap class basis for an association with multiple multiplicity.
HashMapFiniteAssociation	generate the declaration and accessors on the HashMap class basis for an association with finite multiplicity.
SimpleAttribute	generate the declaration and accessors for an attribute with a simple multiplicity.
VectorMultipleAttribute	generate the declaration and accessors on the Vector class basis for an attribute with multiple multiplicity.
VectorFiniteAttribute	generate the declaration and accessors on the Vector class basis for an attribute with finite multiplicity.
ArrayMultipleAttribute	generate the declaration and accessors on the basis of a table for an attribute with multiple multiplicity.
ArrayFiniteAttribute	generate the declaration and accessors on the basis of a table for an attribute with finite multiplicity.
StackMultipleAttribute	generate the declaration and accessors on the Stack class basis for an attribute with multiple multiplicity.
StackFiniteAttribute	generate the declaration and accessors on the Stack class basis for an attribute with finite multiplicity.
HashtableMultipleAttribute	generate the declaration and accessors on the Hashtable class basis for an attribute with multiple multiplicity.
HashtableFiniteAttribute	generate the declaration and accessors on the Hashtable class basis for an attribute with finite multiplicity.
ListMultipleAttribute	generate the declaration and accessors on the List class basis for an attribute with multiple multiplicity.
ListFiniteAttribute	generate the declaration and accessors on the List class basis for an attribute with finite multiplicity.
LinkedListMultipleAttribute	generate the declaration and accessors on the LinkedList class basis for an attribute with multiple multiplicity.

The ... class	is used to ...
LinkedListFiniteAttribute	generate the declaration and accessors on the LinkedList class basis for an attribute with finite multiplicity.
CollectionMultipleAttribute	generate the declaration and accessors on the Collection class basis for an attribute with multiple multiplicity.
CollectionFiniteAttribute	generate the declaration and accessors on the Collection class basis for an attribute with finite multiplicity.
SetMultipleAttribute	generate the declaration and accessors on the Set class basis for an attribute with multiple multiplicity.
SetFiniteAttribute	generate the declaration and accessors on the Set class basis for an attribute with finite multiplicity.
HashSetMultipleAttribute	generate the declaration and accessors on the hashSet class basis for an attribute with multiple multiplicity.
HashSetFiniteAttribute	generate the declaration and accessors on the hashSet class basis for an attribute with finite multiplicity.
MapMultipleAttribute	generate the declaration and accessors on the Map class basis for an attribute with multiple multiplicity.
MapFiniteAttribute	generate the declaration and accessors on the Map class basis for an attribute with finite multiplicity.
HashMapMultipleAttribute	generate the declaration and accessors on the HashMap class basis for an attribute with multiple multiplicity.
HashMapFiniteAttribute	generate the declaration and accessors on the HashMap class basis for an attribute with finite multiplicity.
SimpleIOParameter	generate the declaration of a parameter with simple multiplicity.
VectorIOParameter	generate the declaration on the Vector class basis for a parameter with finite or multiple multiplicity.
ArrayIOParameter	generate the declaration on the basis of a table for a parameter with finite or multiple multiplicity.
StackIOParameter	generate the declaration on the Stack class basis for a parameter with finite or multiple multiplicity.
HashtableIOParameter	generate the declaration on the Hashtable class basis for a parameter with finite or multiple multiplicity.
ListIOParameter	generate the declaration on the List class basis for a parameter with finite or multiple multiplicity.
LinkedListIOParameter	generate the declaration on the LinkedList class basis for a parameter with finite or multiple multiplicity.
CollectionIOParameter	generate the declaration on the Collection class basis for a

The ... class	is used to ...
	parameter with finite or multiple multiplicity.
SetIOParameter	generate the declaration on the Set class basis for a parameter with finite or multiple multiplicity.
HashSetIOParameter	generate the declaration on the HashSet class basis for a parameter with finite or multiple multiplicity.
MapIOParameter	generate the declaration on the Map class basis for a parameter with finite or multiple multiplicity.
HashMapIOParameter	generate the declaration on the HashMap class basis for a parameter with finite or multiple multiplicity.
SimpleReturnParameter	generate the declaration of a return parameter with simple multiplicity.
VectorReturnParameter	generate the declaration on the Vector class basis for a return parameter with finite or multiple multiplicity.
ArrayReturnParameter	generate the declaration on the Array class basis for a return parameter with finite or multiple multiplicity.
StackReturnParameter	generate the declaration on the Stack class basis for a return parameter with finite or multiple multiplicity.
HashtableReturnParameter	generate the declaration on the Hashtable class basis for a return parameter with finite or multiple multiplicity.
ListReturnParameter	generate the declaration on the List class basis for a return parameter with finite or multiple multiplicity.
LinkedListReturnParameter	generate the declaration on the LinkedList class basis for a return parameter with finite or multiple multiplicity.
CollectionReturnParameter	generate the declaration on the Collection class basis for a return parameter with finite or multiple multiplicity.
SetReturnParameter	generate the declaration on the Set class basis for a return parameter with finite or multiple multiplicity.
HashSetReturnParameter	generate the declaration on the HashSet class basis for a return parameter with finite or multiple multiplicity.
MapReturnParameter	generate the declaration on the Map class basis for a return parameter with finite or multiple multiplicity.
HashMapReturnParameter	generate the declaration on the HashMap class basis for a return parameter with finite or multiple multiplicity.

We will deal later with simple multiplicity for the 0-1 or 1-1 type multiplicities, multiple multiplicity for the n-* type multiplicity and finite multiplicity for the n-m and m>1 type multiplicities.

Customizing association accessors

Introduction

Each class which parameterizes the accessors for an association must contain an operation named "*declare*", used to generate the declaration of the corresponding Java attribute.

The other operations are used to generate attribute or association accessors according to the access mode of their attribute or association.

For associations, one of them must generate an accessor named "*card*" returning an integer containing the association's current multiplicity.

Customizing the methods of the simpleAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to the Java attribute.
set	the modification of the Java attribute.
card	the calculation of the number of elements.

Customizing the methods of the vectorMutipleAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to an element with a given index.
set	the substitution of the element with a given index.
append	the addition of an element.
erase_by_element	the deletion of a given element.
erase_by_index	the deletion of an element with a given index.
card	the calculation of the number of elements.

Customizing the methods of the vectorFiniteAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to an element with the given index.
set	the substitution of the element with a given index.
append	the addition of an element.
erase_by_element	the deletion of a given element.
erase_by_index	the deletion of an element with a given index.
card	the calculation of the number of elements.

Customizing the methods of the arrayFiniteAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to an element with a given index.
get_all	the access to all the elements.
set	the modification of the element with the given index.
set_all	the modification of all the elements.
card	the calculation of the number of elements.

Customizing the methods of the stackMultipleAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to the element on top of the stack.
append	the addition of an element.
erase	the deletion of an element on top of the stack.
card	the calculation of the number of elements.

Customizing the methods of the stackFiniteAssociation class

The ... method	is used to customize ...
declare	the declaration of the Java attribute.
get	the access of the element on top of the stack.
append	the addition of an element.
erase	the deletion of an element on top of the stack.
card	the calculation of the number of elements.

Customizing the methods of the hashtableMultipleAssociation class

The ... method	is used to customize ...
declare	the declaration of the attribute.
get	the access to an element from its key.
append	the adding of an element with its key.
erase_by_key	the deletion of an element from its key.
card	the calculation of the number of elements.

Customizing the methods of the hashtableMultipleAssociation class

The ... method	is used to customize ...
declare	the declaration of the attribute.
get	the access to an element from its key.
append	the addition of an element with its key.
erase_by_key	the deletion of an element from its key.
card	the calculation of the number of elements.

Customizing attribute accessors

Introduction

Each class which parameterizes accessors for a class must contain a method named "*declare*", used to generate the declaration of the corresponding Java attribute. The methods are used to generate accessors in access or modify mode. For finite or multiple attributes, one of them must generate an accessor named "*card*", which returns an integer containing the attribute's current multiplicity.

Customizing the attributes of the defaultSimpleAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to the Java attribute.
set	the modification of the Java attribute.

Customizing the attributes of the vectorMultipleAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the Java attribute.
get	the access to an element with the given index.
set	the substitution of the element with a given index.
append	the addition of an element.
erase_by_element	the deletion of a given element.
erase_by_index	the deletion of the element with the given index.
card	the calculation of the number of elements.

Customizing the attributes of the vectorFiniteAttribute class

The ... attribute	is used to customize ...
get	the access to an element with the given index.
set	the substitution of the element with a given index.
append	the addition of an element.
erase_by_element	the deletion of a given element.
erase_by_index	the deletion of the element with the given index.
card	the calculation of the number of elements.

Customizing the attributes of the arrayFiniteAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the attribute.
get	the access to an element with the given index.
get_all	the access to all the elements.
set	the modification of the element with the given index.
set_all	the modification of all the elements.

Customizing the attributes of the stackMultipleAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the attribute.
get	the access to the element on top of the stack.
append	the addition of an element.
erase	the deletion of an element on top of the stack.
card	the calculation of the number of elements.

Customizing the attributes of the stackFiniteAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the attribute.
get	the access to the element on top of the stack.
append	the addition of an element.
erase	the deletion of an element on top of the stack.
card	the calculation of the number of elements.

Customizing the attributes of the hashtableMultipleAttribute class

The ... attribute	is used to customize ...
declare	the declaration of the attribute.
get	the access to an element from its key.
append	the addition of an element with its key.
erase_by_key	the deletion of an element from its key.
card	the calculation of the number of elements.

Customizing parameter declarations

Introduction

The classes which parameterize the declaration of the parameters have a unique method named "*declare*", used to generate the declaration of the parameter.

Customizing the declarations of the class parameters

The "*declare*" method for the ... class	is used to customize ...
simpleIOParameter	the declaration of the parameter
vectorIOParameter	the declaration of the parameter
stackIOParameter	the declaration of the parameter
hashtableIOParameter	the declaration of the parameter
simpleReturnParameter	the declaration of the parameter
vectorReturnParameter	the declaration of the parameter
arrayReturnParameter	the declaration of the parameter
stackReturnParameter	the declaration of the parameter
hashtableReturnParameter	the declaration of the parameter

Additional elements for customizing types (Summary)

Summary of the types of notes and tagged values

The types of notes and tagged values necessary for defining a UML modeling project for translating the types and accessor generation are defined in the TypesEditor module.

Name ...	Types ...	Metaclass ...	has the role of ...
import	tagged value	Class	indicating the packages necessary for the implementation of the accessors.
notDefault	tagged value	Operation	indicating an accessor that must not be generated by default.
Jeval	tagged value	Note	indicating an expression to evaluate in J.
targetType	note	DataType	the translation of a type.
J	note	Operation	evaluating the J code to calculate a declaration or an accessor.
J	note	Parameter	evaluating the J code to calculate a parameter or a return parameter of an accessor.
JName	note	Operation	calculating the name of an accessor.
JModifiers	note	Operation	calculating the modifiers of an accessor.
JExceptions	note	Operation	the exceptions sent back by the accessor.

Summary of J variables

Global variables must be defined for the passing of parameters to the evaluated J expressions, or the retrieval of the results after evaluation.

Name ...	Type...	has the role of ...
JavaReturn	String	the return of a declaration or accessor calculation method
JavaElement	Object	the name of the object for translating the type

Summary of J methods

For information on all J methods available, please refer to the "*Principles of type and accessor generation*" section in chapter 17 of this user guide.

Code generation document templates

Overview

To use this generation feature, you must have a valid *Objecteering/UML Profile Builder* license. Document templates are used to visualize the structure of generated Java source and to easily modify the Java code produced. The document template editors provided are detailed in the *Objecteering/UML Profile Builder* and *Objecteering/Documentation* user guides.

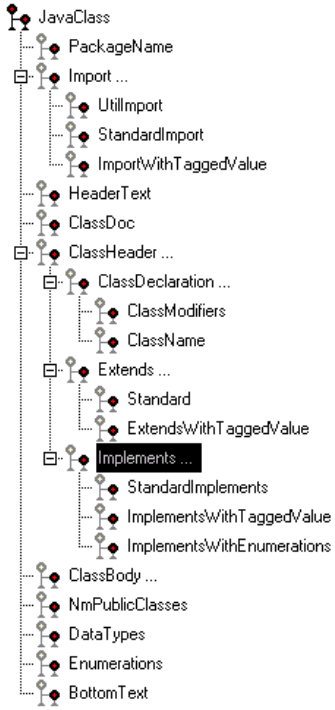


Figure 17-13. Java generation document template

Each document template element is loaded with a file zone to generate. The roles are summed up in the table below.

The ... document template element	is used to generate ...
ClassDoc	the Javadoc type documentation relative to the class
PackageName	the name of the package in which the class is defined.
Import	the processing of the imported classes and packages.
UtilVector	the import of the package when an attribute, an association or a method parameter has the Vector, Stack or Hashtable type.
StandardImport	the processing use links between packages translated by a package import used for all the classes of the user package.
ImportWithTaggedValue	the processing of the *JavaImport* tagged values present on the class or the package to which the class belongs.
HeaderText	the *JavaHeader* type texts.
ClassHeader	the class header.
ClassModifiers	the *abstract*, *final*, visibility.
ClassName	the name of the class.
Extends	the processing of the generalizations.
Standard	the generalization for a generalization link on a model.
ExtendsWithTaggedValue	the translation of the {*JavaExtends}* tagged values.
Implements	the processing of the interface implementations.
StandardImplements	the implementation for an implementation link on the model.
ImplementsWithTaggedValue	the translation of the {*JavaImplements*} tagged values.

The ... document template element	**is used to generate ...**
ImplementsWithEnumerations	the implementations which correspond to the enumerations used in the class.
ClassBody	the class body (see below).
DataTypes	the generation into non-public classes of types defined in the class.
Enumerations	the generation in the form of interfaces of enumerations defined in the class.
NonPublicClasses	the generation of non-public classes.
BottomText	the *JavaBottom* type texts.

Detail of the ClassBody item (part 1)

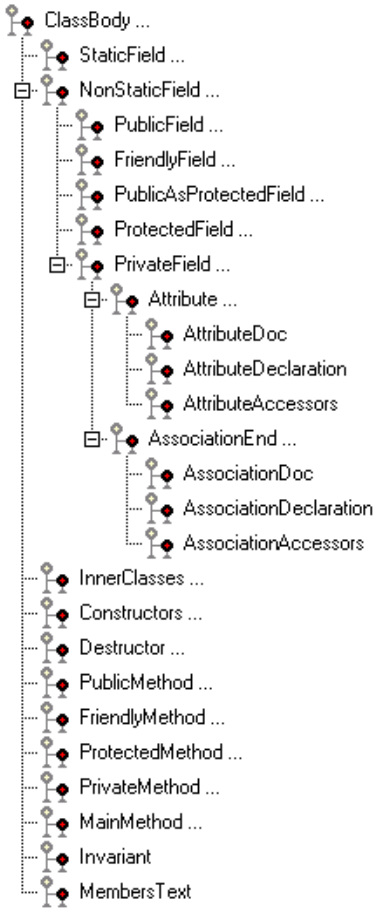


Figure 17-14. Detail of a class body

The ... item	represents ...
ClassBody	the class body
StaticField	the static fields
NonStaticField	the non static fields
PublicField	the public visibility fields
ProtectedField	the protected visibility fields
FriendlyField	the friendly visibility fields
PrivateField	the private visibility fields
Attribute	the attributes
AttributeDoc	the *Javadoc* text for an attribute
AttributeDeclaration	the declaration of an attribute
AttributeAccessors	the accessors and the modifiers on an attribute
AssociationEnd	the associations
AssociationDoc	the *Javadoc* text for an association
AssociationDeclaration	the declaration of the attribute corresponding to an association
AssociationAccessors	the accessors and the modifiers on the attribute corresponding to an association
InnerClasses	the internal classes
Constructors	the constructors
Destructor	the *finalize* method
PublicMethod	the public methods
ProtectedMethod	the protected methods
FriendlyMethod	the friendly methods
PrivateMethod	the private methods
MainMethod	The *main* method
Invariant	the *invariant()* method
MembersText	the *JavaMembers* notes

Detail of the ClassBody item (Part 2)

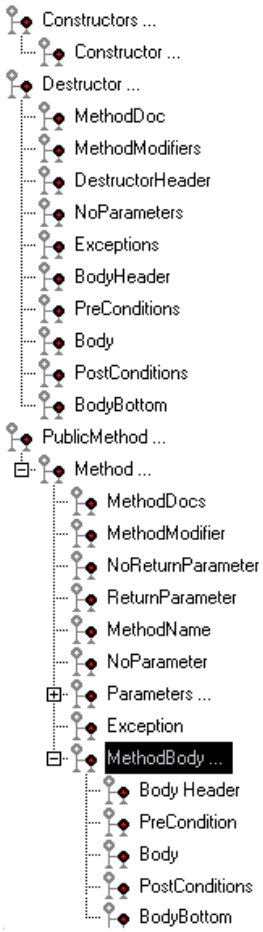


Figure 17-15. Decomposition of the method content

The ... item	represents ...
Constructors	the constructors
Constructor	a constructor
Destructor	the destructor
DestructorHeader	the declaration of the finalizer
PublicMethod	a class's public methods
Method	a method
MethodDoc	the generation of a *javadoc* text for a method
MethodModifiers	the visibility, *static*, *final*, *abstract*, *synchronized*, *native*
NoReturnParameter	the type of return (void)
ReturnParameter	the type of return parameter
MethodName	the method's name
NoParameters	the section related to the parameters of a method when it doesn't have any
Parameters	the section related to the parameters of a method when it has some
ListOfParameters	the parameters
Parameter	a parameter
Exceptions	the declarations of exception launching
MethodBody	the method's body
BodyHeader	the *JavaSuper* type texts
PreConditions	the pre-conditions of the *Javacode* type model
Body	the method's implementation, generation of a *JavaCode* type model implementation
PostConditions	the generation of the post-conditions of the *JavaCode* type model
BodyBottom	the generation of a *JavaReturned* text type

Makefile generation document templates

Overview

The technique used for code generation is also used for the generation of the compilation makefiles.

Three document templates have been developed for this purpose. The first is used to compile the classes in a package and the second to compile only a class.

Structure of the package makefile document template

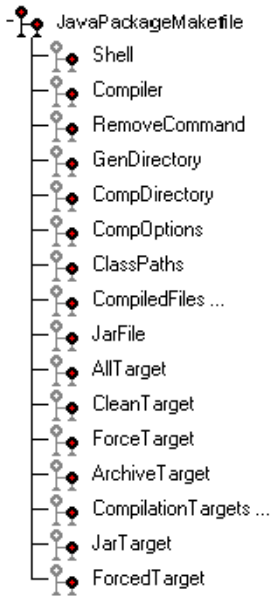


Figure 17-16. Document template for generating the makefile for a package

The ... item	generates ...
Shell	Declaration of the Shell variable (GNU make on Windows only).
Compiler	the definition of the compiler.
RemoveCommand	the definition of the command to delete the files.
GenDirectory	the definition of the .java source files root directory.
CompDirectory	the definition of the root directory for the compilation results.
CompOptions	the definition of the compiler options.
ClassPaths	the definition of the paths that search the Java classes used.
CompiledFiles	the definition of the list of *.class* files to produce.
JarFile	the declaration of the storage file name.
AllTarget	the compilation's global target.
CleanTarget	the target of the *class* files destruction.
ForceTarget	the target of the forced re-generation of all the *.class* files.
ArchiveTarget	the main target for producing file storage.
CompilationTargets	the targets for producing compiled files.
JarTarget	the target that runs the *jar* command for producing the storage file.
ForcedTarget	the target used to force the running of the commands of a target on which the target is dependent.

Contents of the CompiledFiles item

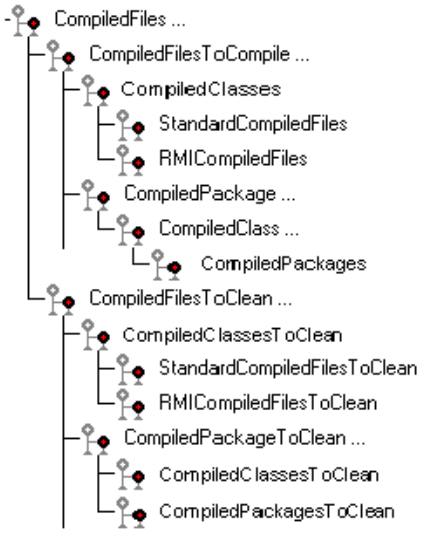


Figure 17-17. Document template for generating the makefile for a package - the "*CompiledFiles*" item

The ... item	generates ...
CompiledFilesToCompile	the definition of the list of *.class* files to produce
CompiledClasses	the files compiled for a class
StandardCompiledFiles	the *.class* file compiled for a class
RMICompiledFiles	the stub and skeleton for a class
CompiledPackages	the files compiled for the classes of a package recursively
CompiledFilesToClean	the definition of the list of *.class* files to be produced (format of the GNU make tool on PC).
CompiledClassesToClean	the files compiled for a class (format of the *destruction* library)
StandardCompiledFilesToClean	the *.class* file compiled for a class (format of the *destruction* library)
RMICompiledFilesToClean	the stub and skeleton for a class (format of the *destruction* library)
CompiledPackagesToClean	the files compiled for the classes of a leaf package (format of the *destruction* library)

Contents of the CompilationTargets item

Figure 17-18. Document template for generating the makefile for a package - "*Compilation Targets*" item

The ... item	generates ...
PackagesTargets	the compilation targets of the sub-packages of a package
ClassesTargets	the compilation targets of a package's classes
CompilationTarget	the compilation target for compiling a *.java* in *.class*
RmicTarget	the target for producing the stub and skeleton

Structure of the class makefile document template

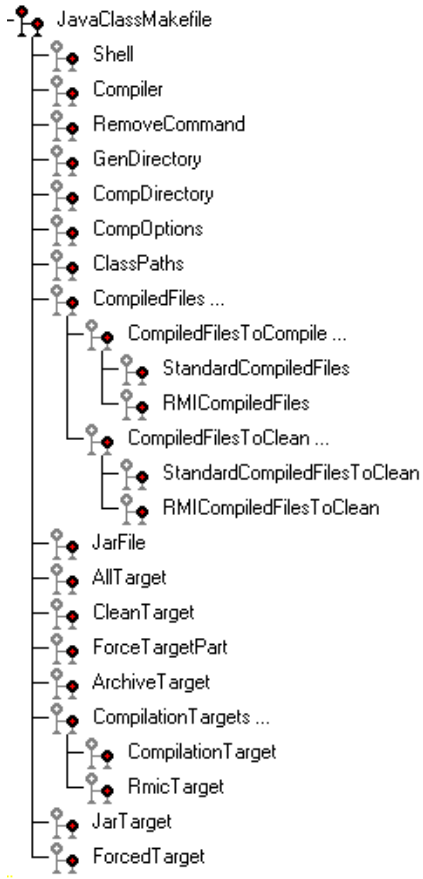


Figure 17-19. Document template for generating the makefile for a class

The ... item	generates ...
Shell	declaration of the Shell variable (GNU *make* on Windows only)
Compiler	the definition of the compiler
RemoveCommand	the definition of the command to delete the file
GenDirectory	the definition of the generation root directory
CompDirectory	the definition of the directory put together for the compilation results
CompOptions	the definition of the compiler options
ClassPaths	the definition of the paths used to search for the used java classes
CompiledFiles	the definition of the list of *.class* files to be produced
CompiledFilesToCompile	the definition of the list of *.class* files to be produced
StandardCompiledFiles	the file compiled for a class
RMICompiledFiles	the stub and skeleton for a class
CompiledFilesToClean	the definition of the list of *.class* files to be produced (format of the destruction binary).
StandardCompiledFilesToClean	the file compiled for a class (format of the destruction binary)
RMICompiledFilesToClean	the stub and skeleton for a class (format of the destruction binary)
JarFile	the declaration of the name of the storage file
AllTarget	the target of the *.class* files destruction
ForceTarget	the target of the forced re-generation of all the *.class* files
ArchiveTarget	the main target for producing file storage
CompilationTargets	the targets of the *.class* generation
CompilationTarget	the target used to produce the stub and skeleton

The ... item	generates ...
RmicTarget	the target used to produce the stub and skeleton
JarTarget	the target that runs the *.jar* command used to produce the storage file.
ForcedTarget	the target used to to force the running of the commands of a target on which the target is dependent.

Chapter 18: Calling module on-line commands

Calling commands - Overview

Module commands which do not require an interface can be launched on a command line using the *objingcl* delivered with Objecteering/UML.

Calling commands

Syntax

An online command is called using an instruction as follows:

```
objingcl   -prj <UML modeling project_name>
           -mdl JavaModule
           -cmd <command_Name>
           <metaclass>:<object_name>
           -db <base_name>
```

Commands that can be invoked

Command ...	Metaclass ...	Action ...
generate	Java	code generation
javaDoc	Java	documentation Generation
generateMakefile	Java	makefile generation
compile	Java	compilation of the *.java* into *.class*
forceCompile	Java	destruction of the *.class* then compilation
cleanCompile	Java	destruction of the *.class*
archive	Java	production of the storage file
operationsToImplement	Class	creation of the methods of the implemented interfaces
operationsToImplement	Package	creation of the methods of the implemented interfaces for all the package's classes
operationsToRedefine	Class	creation of all the abstract methods of the parent classes
operationsToRedefine	Package	creation of all the abstract methods of the parent classes for all the classes of a package
toNonPrimitive	Class	the transformation of a primitive class into a non-primitive class
toPrimitive	Class	the transformation of a non-primitive class into a primitive class
finalize operations	Class	the positioning of the "non derivable" field on non-redefined class operations
toPrimitive	Package	the positioning of the "non derivable" field non-redefined operations of classes of a package and its sub-packages

Index

Java generation work product 5-5 WSAD 11-4