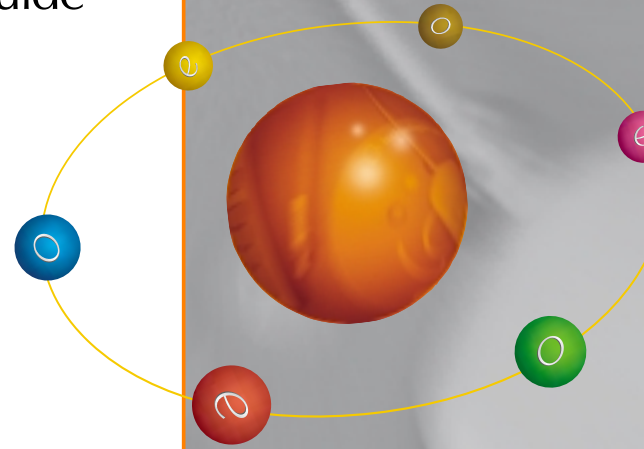


Objecteering/UML

Objecteering/UML Test Designer User Guide

Version 5.2.2



Objecteering

Software

www.objecteering.com

Taking object development one step further

Information in this document is subject to change without notice and does not represent a commitment on the part of Objecteering Software. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the express written consent of Objecteering Software.

© 2003 Objecteering Software

Objecteering/UML version 5.2.2 - CODOBJ 001/003

Objecteering/UML is a registered trademark of Objecteering Software.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>).

UML and OMG are registered trademarks of the Object Management Group. Rational ClearCase is a registered trademark of Rational Software. CM Synergy is a registered trademark of Telelogic. PVCS Version Manager is a registered trademark of Merant. Visual SourceSafe is a registered trademark of Microsoft. All other company or product names are trademarks or registered trademarks of their respective owners.

Contents

Chapter 1: Introduction to Objectteering/UML Test Designer	
Introduction	1-3
Structure of the Objectteering/UML Test Designer user guide	1-5
Overview of the two test modules.....	1-6
Developing applications.....	1-17
Glossary	1-18
Chapter 2: Working with the Objectteering/UML Test Designer module	
Using the Objectteering/UML Test Designer module.....	2-3
Chapter 3: Objectteering/UML Test Designer First Steps	
Overview of the First Steps	3-3
Initializing the First Steps project.....	3-5
Creating a UML Test Designer test suite	3-10
Defining tests in the test suite.....	3-17
Maintaining your test model.....	3-26
Creating a Tests For Java/EJB generation work product	3-29
Generating and compiling test code.....	3-32
Running tests with JUnit	3-36
Generation work product commands	3-39
Examples	3-42
Chapter 4: Frequently asked questions (FAQ)	
How do I test abstract class implementations?	4-3
How do I test interface implementations?	4-5
How do I generate tests for functions with primitive parameters?	4-8
How do I add a new context to a test suite?.....	4-12
How do I add a new test sequence to a context?	4-15
How do I write a test method in Java code?	4-19
How do I define a Setup() called before each test method?	4-21
How do I test a message with an assert note?	4-23
How do I get a returned value with an assert note?	4-26
How do I insert an assertion into the test?	4-28
How do I insert target code into the test?	4-29
How do I generate and compile a new test suite?	4-30
How do I generate and compile a new context?	4-31
How do I test an EJB?	4-32

Chapter 5: Principles of Objectteering/UML Test Designer	
Selecting and parameterizing the Objectteering/UML Test Designer module	5-3
Work products	5-4
Tagged values, notes and stereotypes	5-5
Consistency checks	5-6
Chapter 6: Functions of the Objectteering/UML Test Designer module	
Overview of Objectteering/UML Test Designer functions	6-3
Tagged value types	6-4
Note types	6-5
Stereotypes	6-6
Chapter 7: Customizing the Objectteering/UML Test Designer module	
Defining module parameters	7-3
Sets of parameters	7-6
Chapter 8: UML Test Designer documentation generation	
Overview of UML Test Designer documentation generation	8-3
Functions of UML Test Designer documentation generation	8-7
Index	

Chapter 1: Introduction to
Objecteering/UML Test
Designer

Introduction

Overview

Welcome to the *Objectteering/UML Test Designer* module!

Objectteering/UML Test Designer consists of two modules:

- ◆ The *Objectteering/UML Test Designer* module
- ◆ The *Objectteering/UML Tests for Java/EJB* module

The *Objectteering/UML Test Designer* module is used to generate a test model from the model of your application and then to define test cases and test suites. This module is not subject to an Objectteering/UML license.

The *Objectteering/Tests for Java/EJB* module allows the generation, compilation and execution of tests. This module requires a license.

The *Objectteering/UML Test Designer* module is designed to facilitate the definition of tests directly from your model. A certain level of knowledge in Java code generation and UML is presumed, and you must be familiar with the Objectteering/UML environment, as detailed in the following user guides:

- ◆ *Objectteering/Administrating Objectteering Sites*
- ◆ *Objectteering/UML Modeler*
- ◆ *Objectteering/Java*

Chapter 3 of this user guide, "*First Steps*", provides the user with a step-by-step demonstration of the workings of the *Objectteering/UML Test Designer* module. We also recommend that all users try out the general "*First Steps*" project, detailed in the *Objectteering/Introduction* user guide, in order to get to know the various general functions of the Objectteering/UML tool.

Functions of the Objecteering/UML Test Designer module

Using the *Objecteering/UML Test Designer* module, it is possible to:

- ◆ create a test suite from your test model:
 - ◆ select in your model those classes which are to be tested
 - ◆ select in your model those classes which are to be stubbed
- ◆ define a test model made up of several test cases and test suites
- ◆ define tests inside this model, using UML diagrams or Java code
- ◆ generate the Java code which will be executed, in order to run tests and checks results
- ◆ generate documentation of the test model and a test report after the execution of the test

Complete model-driven generation

Unlike other CASE tools, Objecteering/UML generates an entire *UML Test Designer* application, from the model and its textual specifications and tagged values, right up to the executables. This means that the user need develop nothing outside Objecteering/UML itself.

With Objecteering/UML, everything is entered directly into the CASE tool. Generated source zones can be modified, and model sections corresponding to the generated code presented. In this way, Objecteering/UML ensures consistency at all times between the model and the code.

Structure of the Objecteering/UML Test Designer user guide

The *Objecteering/UML Test Designer* user guide is structured as follows:

- ◆ Chapter 1 - Introduction to the *Objecteering/UML Test Designer* module
 - ◆ Chapter 2 - Working with the *Objecteering/UML Test Designer* module
 - ◆ Chapter 3 - *Objecteering/UML Test Designer* First Steps
 - ◆ Chapter 4 - Frequently asked questions (FAQ)
 - ◆ Chapter 5 - Principles of *Objecteering/UML Test Designer*
 - ◆ Chapter 6 - Functions of the *Objecteering/UML Test Designer* module
 - ◆ Chapter 7 - Customizing the *Objecteering/UML Test Designer* module
 - ◆ Chapter 8 - *UML Test Designer* documentation generation
-

Overview of the two test modules

The Objecteering/UML range now provides a complete test environment, from modeling right through to execution, thanks to the integration of the JUnit framework into the *Objecteering/UML Test Designer* module.

Two powerful tools working together

The JUnit testing framework allows you to write and run test cases and test suites for Java applications. JUnit tests can be organized into a hierarchy of test suites, which can be run and re-run automatically after changing the code. This is useful for automatic regression testing, and ensures complete synergy between coding and testing.

The *Objecteering/UML Modeler* tool, based on the MDA (Mode Driven Architecture) concept, generates code from the UML model of your application. Its *Objecteering/Java* module generates Java code from your model, and permanently maintains consistency between this model and the code generated.

To successfully integrate JUnit into Objecteering/UML, the concept of the test model has been introduced. JUnit provides complete synergy between application code and testing code, whilst Objecteering/UML provides complete consistency between the application's UML model and the code generated. The "test model" allows you to define test architecture in accordance with the model of your application. From this test model, test code is automatically generated for JUnit and can be executed. The user is helped through test definition, and does not have to write even a single line of code!

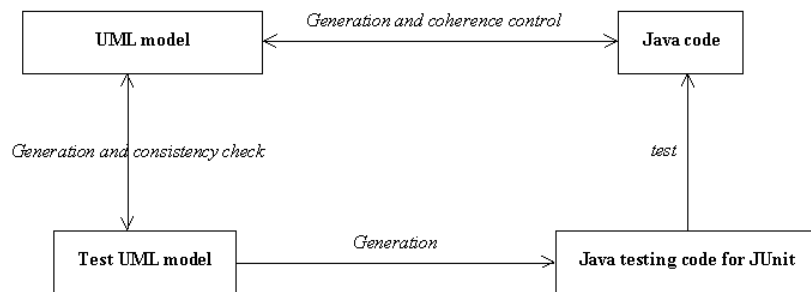


Figure 1-1. The test procedure

Overview of the easy definition of tests

With *Objecteering/UML Test Designer*, it is incredibly simple to define and run tests in a user-friendly way, simply by carrying out the steps below:

Creating a test model

The first step is the creation of a test model from the model of your application, which is carried out by running the "Create a test suite" command (as shown in Figure 1-2).

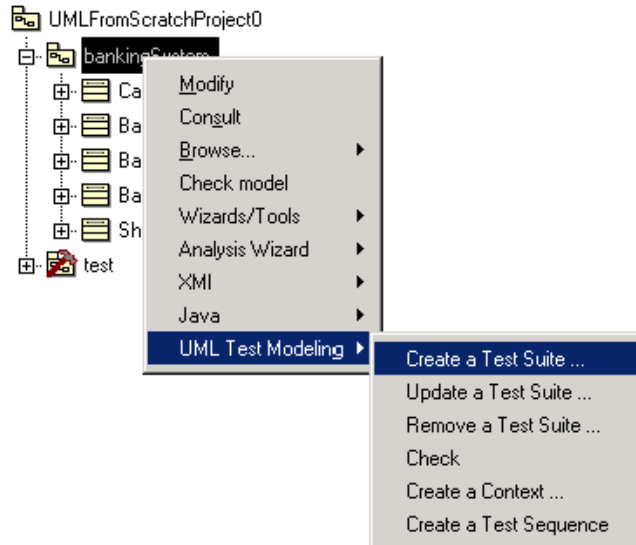


Figure 1-2. Creating a test suite

Selecting classes to be tested

The next step is selecting the classes you wish to test (as shown in Figure 1-3).

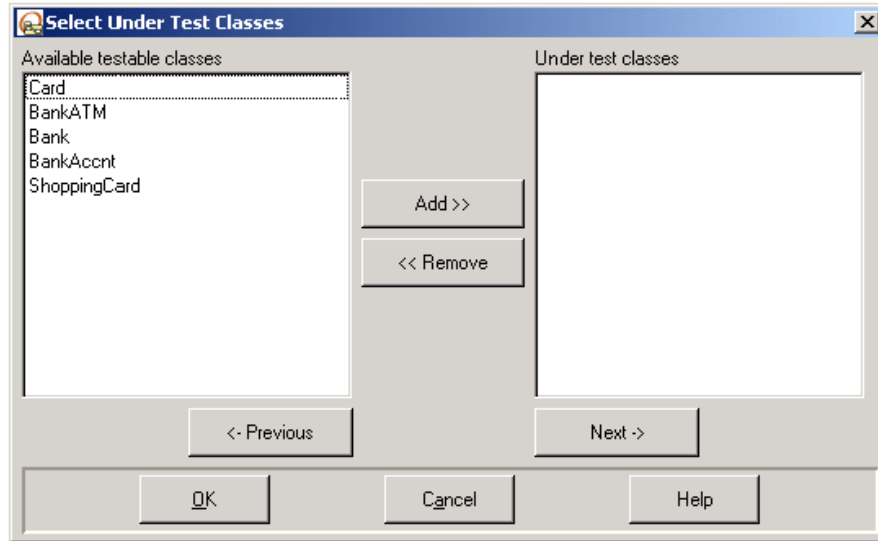


Figure 1-3. Selecting classes to be tested

Selecting classes to be stubbed

The next step is selecting the classes you wish to stub (as shown in Figure 1-4).

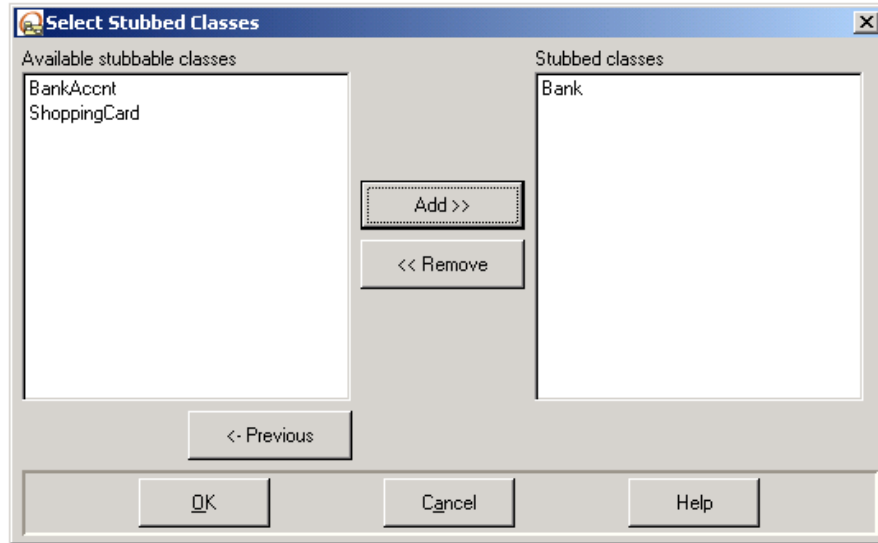


Figure 1-4. Selecting classes to be stubbed

Result

A test model (shown in Figure 1-5) has been created.

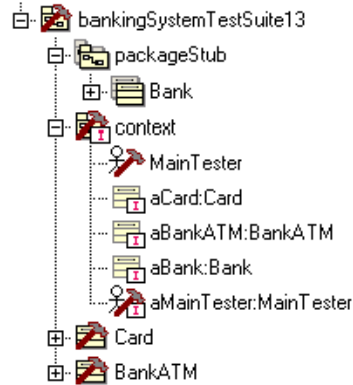


Figure 1-5. The newly created test model

This test model contains:

- ◆ a stub package in which you can define your stubs
- ◆ a context package which corresponds to a test case. Test methods can be defined on this package using sequence diagrams. It also contains an instance of each class to test or stub. These instances will be used in the test sequence diagrams.
- ◆ a copy of each class to test

Defining tests

Tests can now be defined in the test model. This model is organized as a testing hierarchy familiar to JUnit users:

test: the top level test suite

test Suite 1

context1: test case 1

sequence diagram 1: test method 1

sequence diagram 2: test method 2

...

context1: test case 2

sequence diagram 1: test method 1

sequence diagram 2: test method 2

...

test Suite 2

...

The main particularity is that tests methods are defined in a sequence diagram. You can add as many test methods as you wish to a context, as many contexts as you wish to a test suite and as many test suites as you wish to the top level test suite.

A test method is created with a sequence diagram. "Assert" notes, which allow the specification of a test, can be added to return messages. The code corresponding to that test method is automatically generated and inserted into a JUnit test case class.

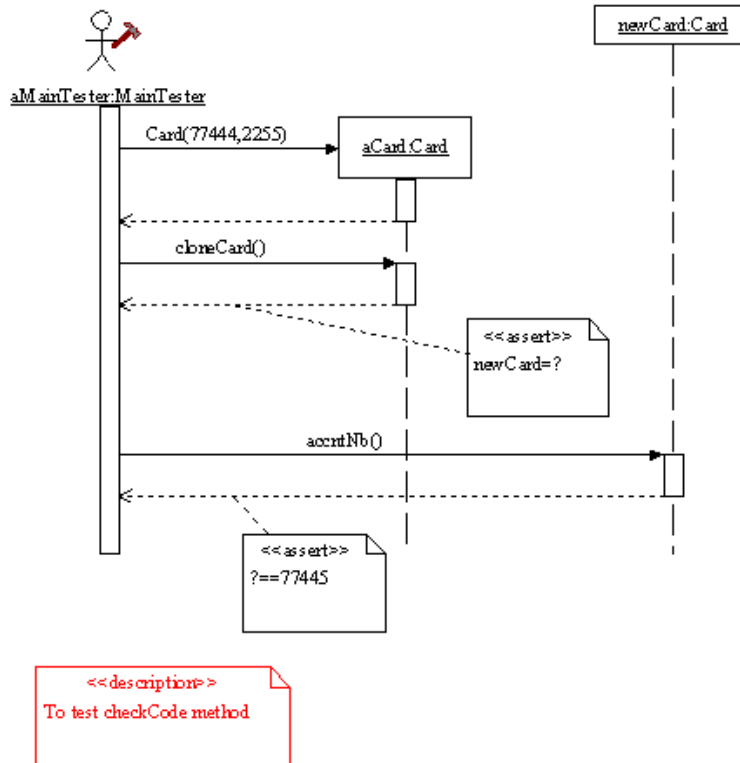


Figure 1-6. An example of a test method

Execution with JUnit

To execute tests, you need not write a single line of code! All you have to do is simply generate and compile the tests in the test model, before launching JUnit from the test model. The test hierarchy presented in the JUnit window corresponds to the test hierarchy of your test model.

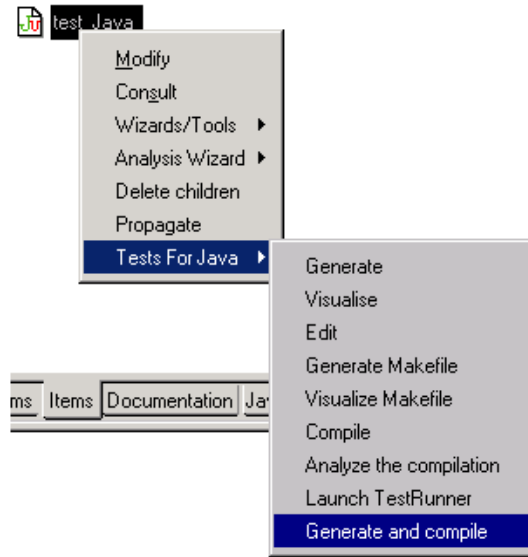


Figure 1-7. Generate compilable code for tests

From your test model, you can now launch the JUnit test runner and check your tests.

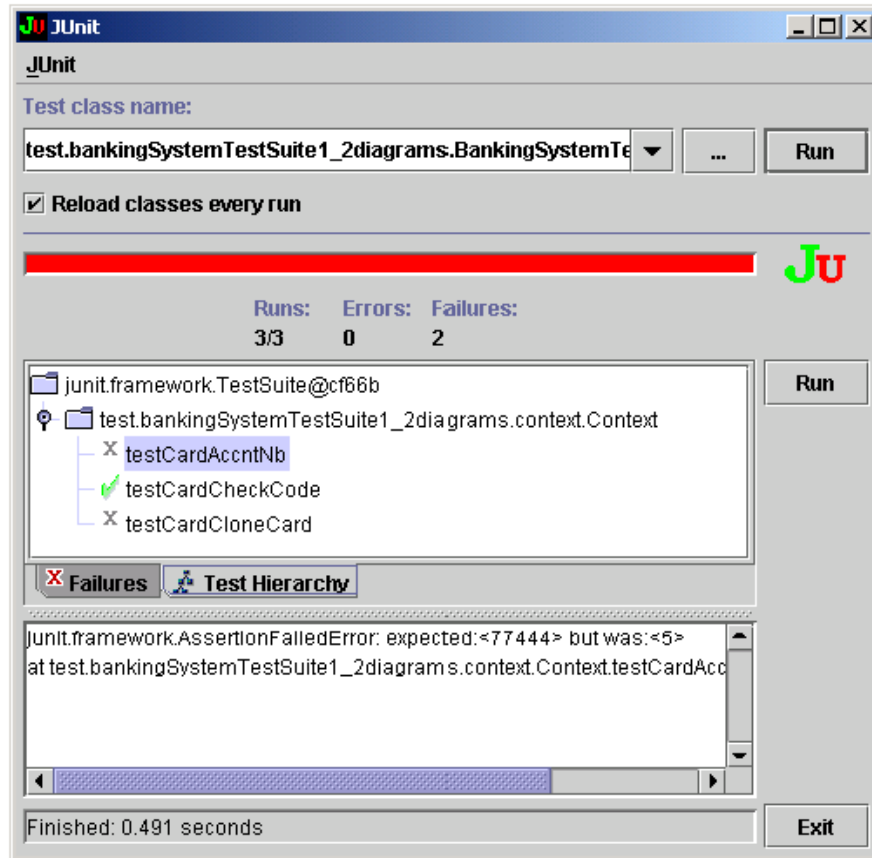


Figure 1-8. Running the test with the JUnit test runner

Other features

It is also possible to automatically generate documentation of your tests in HTML format, describing your test suites and your test cover.

An additional command allows you to check the consistency between your test model and your application model.

Many user facilities are available:

- ◆ the creation of a test sequence diagram
 - ◆ the creation of a context
 - ◆ the updating of a test suite
-

Developing applications

The development of an application and its test model consists of:

- ◆ creating a UML test model from your UML model
- ◆ defining tests in the generated test model either using sequence diagrams or Java code in "testCase" notes
- ◆ generating and compiling tests
- ◆ launching the JUnit test runner

Notes are used to enter text, which will be inserted into the generated code. For example, the *UML Test Designer* "testCase" note is used to enter Java code for an method, which will be executed while running tests.

Tagged values inform the generator of the implementation rules applied. For example, the `{TestedPackage}` tagged value on a test suite package indicates the package tested by the test suite.

For further information, please refer to chapter 5, "*Principles of Objecteering/UML Test Designer*", of this user guide.

Glossary

- ◆ *Assert note*: this note is used in a test sequence diagram to check the return value of a method and carry out a test.
 - ◆ *Class under test*: a class that will be tested. Tests will be carried out on its methods.
 - ◆ *Context package*: this package is located in a test suite and contains a set of instances (of tested or stubbed classes) used to define the test sequence diagrams.
 - ◆ *Organization test suite*: the package which represents the test suite.
 - ◆ *setUp note*: this may be defined on a context package to set the context (set of instances) before the execution of each test.
 - ◆ *Stubbed class*: if a class is needed to test a class under test, but is not available (for example, not implemented), a stubbed class is created to simulate its behavior.
 - ◆ *Test*: stereotype of a test sequence diagram.
 - ◆ *Test case*: a single test represented by a note or a sequence diagram.
 - ◆ *Test case note*: this note may be defined on a test suite package to write a test method in Java code.
 - ◆ *Test suite*: a set of elements used to test a package (package, sequence diagrams, test case notes).
 - ◆ *Tested package*: a test suite is created on a "tested package", which contains the classes which are to be tested.
-

Chapter 2: Working with the
Objecteering/UML Test
Designer module

Using the Objecteering/UML Test Designer module

Prerequisites

In order to use the *Objecteering/UML Test Designer* module, the Objecteering/UML CASE tool (version 5.1 and later) must already have been installed, and the OBJING_PATH environment variable already positioned.

You also need the JUnit toolkit, available from www.junit.org or in the OBJING_PATH\Objecteering\modules\2.0 directory.

You must have the correct license in order to be able to use *Objecteering/UML Test Designer*.

Note: <\$OBJING_PATH> designates the Objecteering/UML installation directory.

Using the module

Before starting work with *Objecteering/UML Test Designer*, the following steps should be carried out:

- ◆ create a UML modeling project. For further details on this operation, please refer to the "*Creating a UML modeling project*" section in chapter 3 of the *Objecteering/UML Modeler* user guide.
 - ◆ select the module for the current UML modeling project. For further details on this operation, please refer to the "*Selecting modules in the current UML modeling project*" section in chapter 3 of the *Objecteering/Introduction* user guide.
-

Chapter 3: Objecteering/UML Test Designer First Steps

Overview of the First Steps

Introduction

The *Objectteering/UML Test Designer* module First Steps present a *UML Test Designer* demonstration project, designed to help you discover the different features of the *Objectteering/UML Test Designer* module, step by step.

We recommend that before starting, every user carry out the general Objectteering/UML First Steps in the *Objectteering/Introduction* user guide.

The *Objectteering/UML Test Designer* First Steps will demonstrate:

- ◆ how to initialize the First Steps project
- ◆ how to create a UML Test Designer test suite
- ◆ how to define tests in the test suite
- ◆ how to maintain your test model
- ◆ how to create a Tests For Java generation work product
- ◆ the generation work product menus available for Tests For Java
- ◆ how to generate Tests For Java code
- ◆ how to visualize generated Tests For Java code
- ◆ how to compile
- ◆ how to run tests with JUnit

These First Steps, which demonstrate the principal features of the *Objectteering/UML Test Designer* and *Objectteering/Tests For Java* modules, last about 45 minutes for a new user.

Sources

As a basis for your demonstration project, you need a Java application for which code has already been generated.

You can import a demonstration project from the properties editor, by simply selecting your project's root package and choosing a project in the "Tests for Java/EJB" tab (as shown in Figure 3-1 below):

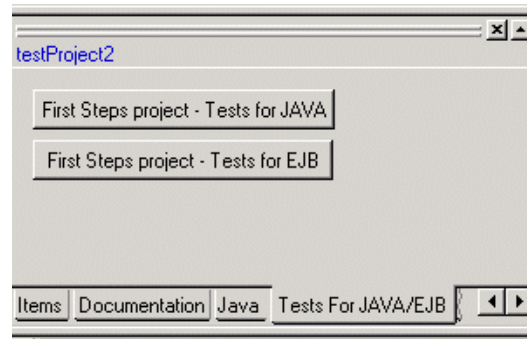


Figure 3-1. Importing a demonstration project from the properties editor

The "Tests for Java" first steps project contains a small Java application which can be generated and compiled with the *Objectteering/Java* module, and its test model which can be generated and compiled too. Have a look at the test model structure and its test suites and test cases during these first steps.

Initializing the First Steps project


Preparation steps

Before starting work with the *Objecteering/UML Test Designer* module in a UML modeling project, you must first prepare the working environment, by creating a UML modeling project and selecting the UML Test Designer and Tests For Java/EJB modules.

Importing a model into the First Steps project

A UML modeling project, which will serve as the model for the First Steps, has to be imported into your UML modeling project. This is done by activating the "*Tools/Import...*" menu. The UML modeling project that you are going to import should contain at least one package containing a few classes. You can also import a demonstration project from the "*Tests for Java/EJB*" tab of the properties editor, as shown in the previous section of this user guide.

Selecting the UML Test Designer and Tests For Java/EJB modules into your project

Launch the Objecteering/UML Modeler tool for your UML modeling project. Next, open the window used to select modules, by clicking on the  "UML modeling project modules" icon, and carry out the steps illustrated in Figure 3-2.

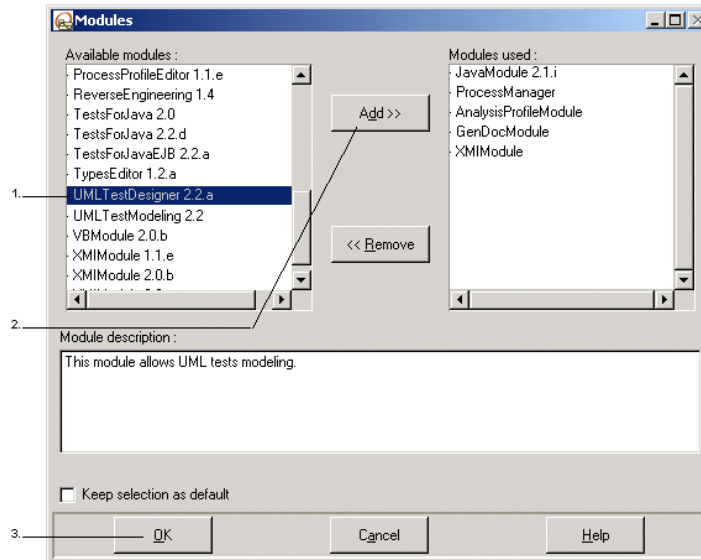



Figure 3-2. Selecting the *Objecteering/UML Test Designer* module

Steps:

- 1 - Select the *Objecteering/UML Test Designer* module from the list of available modules on the left-hand side of the window.
- 2 - Click on the "Add" button. The *UML Test Designer* module then appears in the right-hand "Modules used" list.
- 3 - Click on "OK" to confirm. If the "Keep selection as default" tickbox is checked, the *UML Test Designer* module will automatically be available during future Objecteering/UML sessions.

Configuring the UML Test Designer and Tests For Java/EJB modules

We are now going to open the module configuration window and enter the complete path for the editor which will be used to visualize files containing package and class metrics.

The module configuration window can be opened either by clicking on the  "Modify module parameter configuration" icon, or through the "Tools/Modify configuration" menu.

Configuring the Objecteering/UML Test Designer module

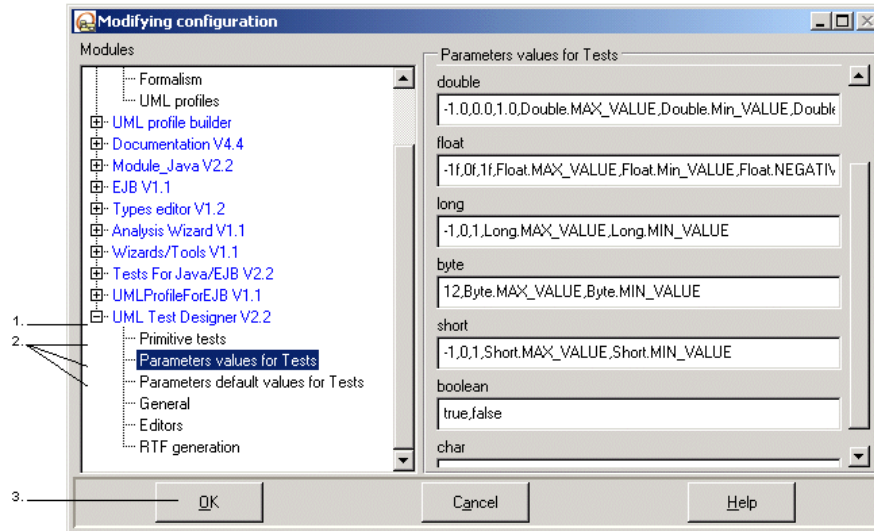


Figure 3-3. Editing the configuration of the *Objecteering/UML Test Designer* module

Steps:

- 1 - Click on the "External editor" section of the "UML Test Designer" parameter category.
- 2 - Define the fields for your project:
 - ◆ "Primitive tests": used to specify the target language used to execute tests. For the moment, you need the *Objecteering/Java* module to execute your tests in Java.
 - ◆ "Parameter values for tests": test cases can be generated for the methods whose parameter are all of primitive type. In this case, you have to specify for each type a set of values to be used in the test.
 - ◆ "Parameter default values for tests": for each primitive type, specify a default value to be used in the generated tests.
- 3 - Confirm by clicking on the "OK" button.

Configuring the Objecteering/Tests For Java/EJB module

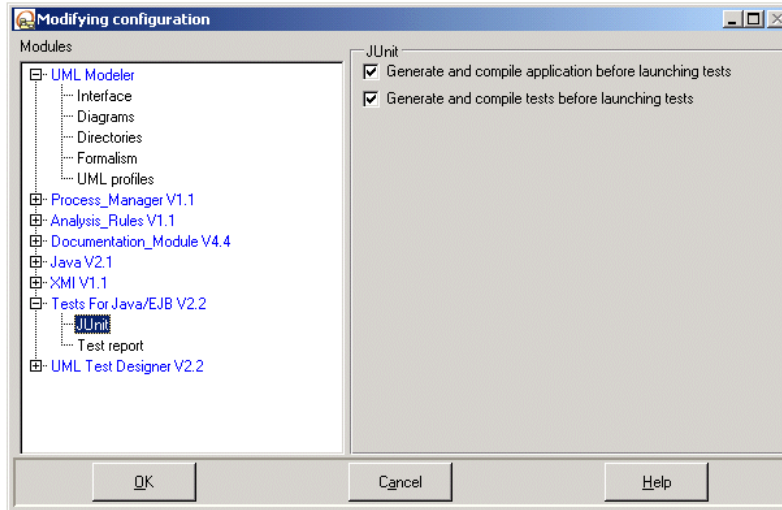


Figure 3-4. Editing the configuration of the *Objecteering/Tests For Java/EJB* module

Steps:

1 - Specify the "JUnit" options for your project:

- ◆ Select *"Generate and compile application before launching tests"*, if you have made a correction to your application model.
- ◆ Select *"Generate and compile tests before launching tests"*, if your test model has changed.

2 - Specify the "Test report" options for your project:

- ◆ A test report is automatically generated after each execution of the tests. In this field, you can select one of the two available XLS files for your test report.

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

Creating a UML Test Designer test suite

Introduction

To test your model, you will create one or several test suites. A "test" package will be created under the root of your UML modeling project. Each test suite corresponds to a package that is created in this "test" package.

Running the "UML Test Designer/Create a Test Suite" command

From the package you wish to test, run the "Create a Test Suite" command, as shown in Figure 3-5.

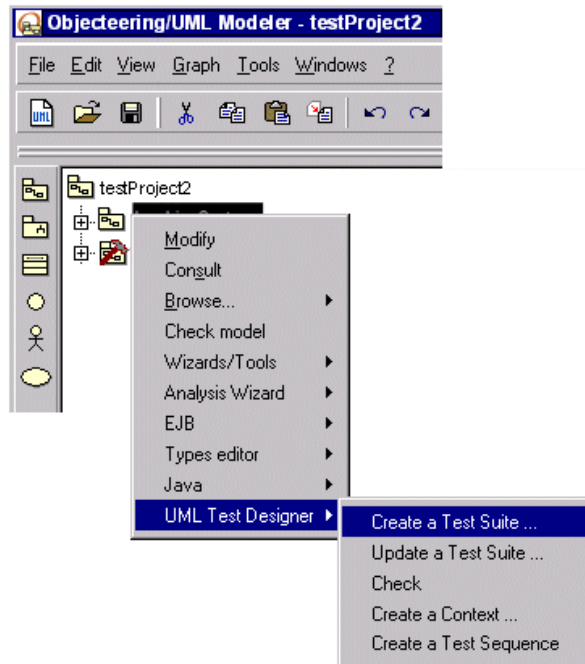


Figure 3-5. Creating a Test Suite

Steps:

- 1 - Right-click on the package for which you wish to create a test suite. The context menu then appears.
- 2 - Run the "Create a Test Suite" command.

Note: The purpose of this command is to define a test suite associated to a package. A test suite can be created on each package.

Naming your test suite

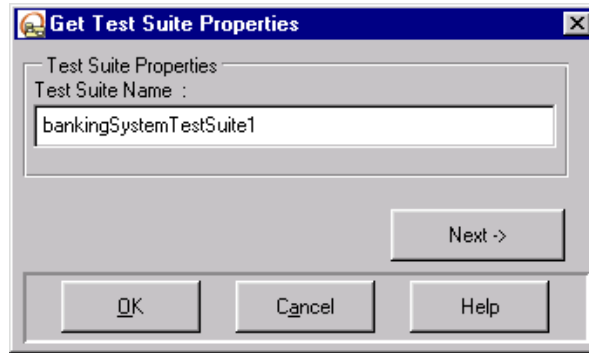


Figure 3-6. Naming your test suite

Enter a name for your test suite and click on the "Next ->" button. The window shown in Figure 3-7 then appears.

Selecting classes to be tested

Testable classes are those classes contained in your tested package.

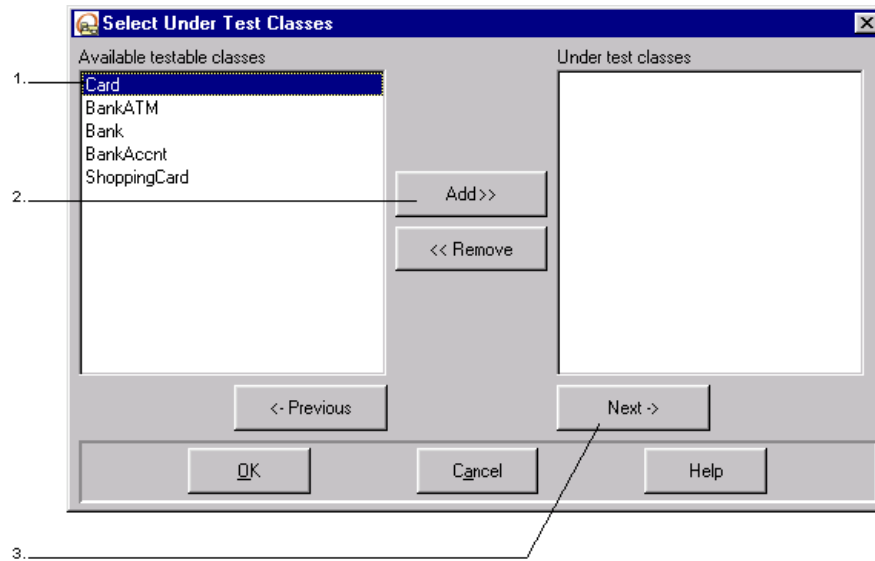


Figure 3-7. Selecting classes to be tested

Steps:

- 1 - Select a class.
- 2 - Click on the "Add" button.
- 3 - When all the classes you wish to test have been selected, click on "Next".

Selecting classes to be stubbed

Stubbable classes are:

- ◆ those classes of your tested package, without the "*under test classes*"
- ◆ those classes accessible from your tested package

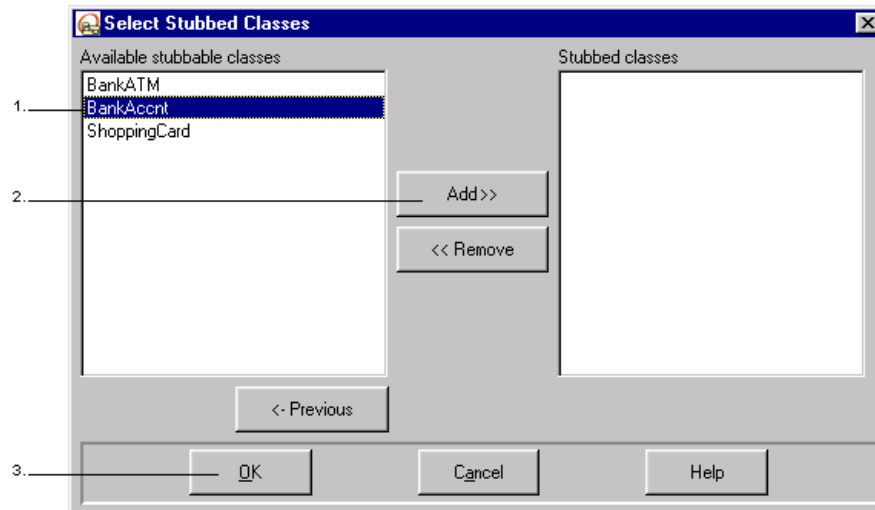


Figure 3-8. Selecting classes to be stubbed

Steps:

- 1 - Select a class.
- 2 - Click on the "Add" button.
- 3 - When all the classes you wish to stub have been selected, click on "OK" to create the test suite.

Result

The result of these operations is a structure like the one below (Figure 3-9).

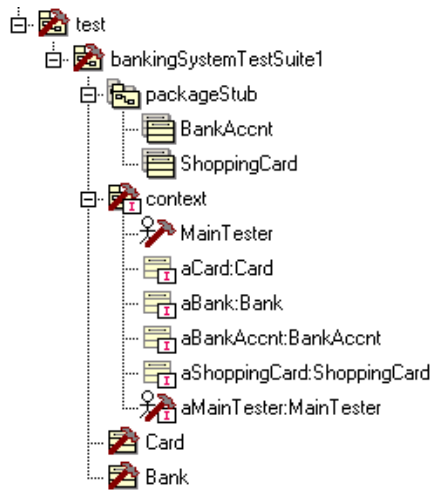


Figure 3-9. The structure of the test model

Description of the newly created test suite

A test suite package has been created in the "test" package.

The "test" package:

- ◆ is created under the UML model root
- ◆ is stereotyped "*OrganizationTestSuite*"
- ◆ contains all the project's test suite packages

Chapter 3: Objecteering/UML Test Designer First Steps

The "*test suite*" package:

- ◆ is created in the "*test*" package
- ◆ is stereotyped "*OrganizationTestSuite*"

and contains:

- ◆ a copy of all the "*under test classes*"
- ◆ a "*context*" package (see the description below)
- ◆ a "*PackageStub*" package, which contains a skeleton for each class to be stubbed
- ◆ a description note
- ◆ a summary note
- ◆ a "*testCase*" note

The "*context*" package:

- ◆ is created in the "*test suite*" package
- ◆ is stereotyped "*TestContext*"

and contains:

- ◆ an instance of each under test class
 - ◆ an instance of each stubbed class
 - ◆ a sequence diagram stereotyped "*Test*"
 - ◆ a summary note
 - ◆ a description note
 - ◆ a "*setUp*" note
-

Defining tests in the test suite

Introduction

Once your test suite has been created, you have to define tests in the test suite package. To get an overview of the test definition possibilities, we recommend that you have a look at the demonstration project, which can be imported from your root package.

Defining stubbed classes

Each stubbed class in your test suite has to be checked and completed, in order to be compilable. Instances of these stubbed classes can also be used in the context package.

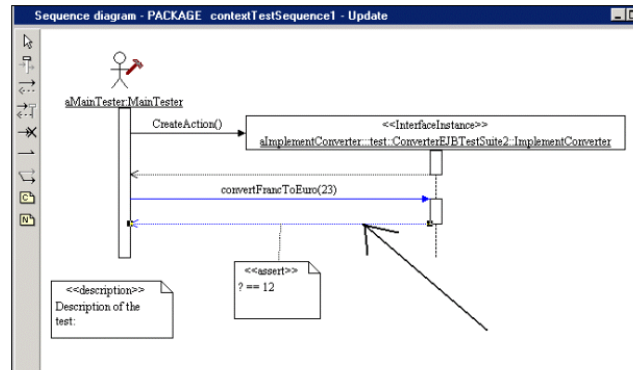
Defining test methods with sequence diagrams on the context package

The context package contains an instance of each class under test of the test suite.

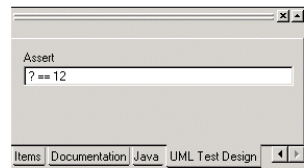
These instances may be used in a sequence diagram to define a test method.

For each element of the test sequence diagram, you can use the "UML Test Designer" tab of the properties editor. Simply select the item in the diagram and specify its content in the tab.

Example: Select a return message on your test sequence (1) and specify the content of the assert note in the properties editor (2). Only one assert note per message is allowed.



(1)



(2)

Figure 3-10. Defining test methods in sequence diagrams

Control structures: loop

Graphically, a sequence diagram loop is presented as a self invocation of the method for(a;b;c) on the Main Tester.

The values for a, b and c are defined in the target list of the method (Figure 3-11).

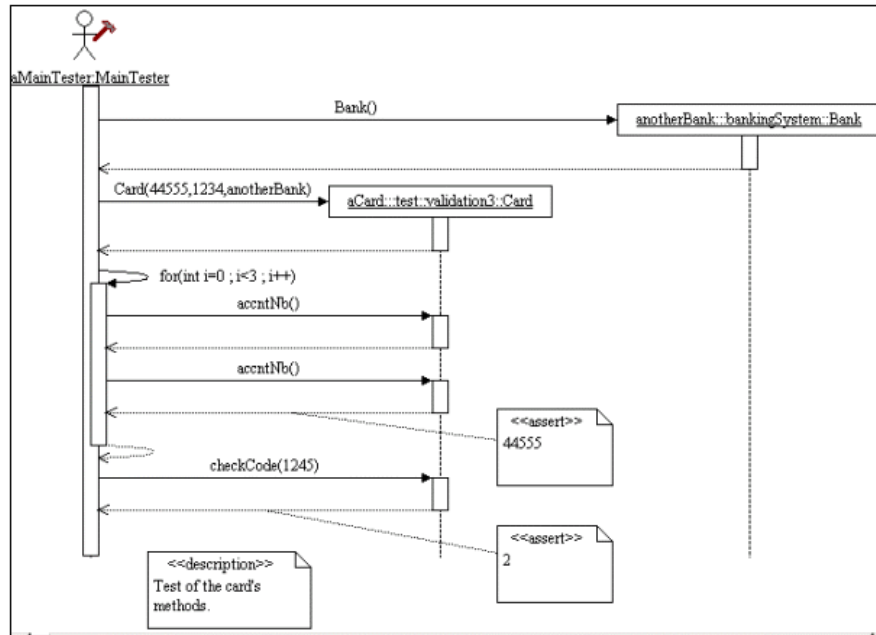


Figure 3-11. The loop control structure

Control structures: if () then {} else

This structure is represented with the self invocation of the "choice" method on the Main Tester. A guard is used to specify the condition. An instance creation is not allowed in a choice construction.

The following example (Figure 3-12) corresponds to:

```
if (x>2) then CallAction()  
else if (x<2) then CallAction2()  
else CallAction3
```

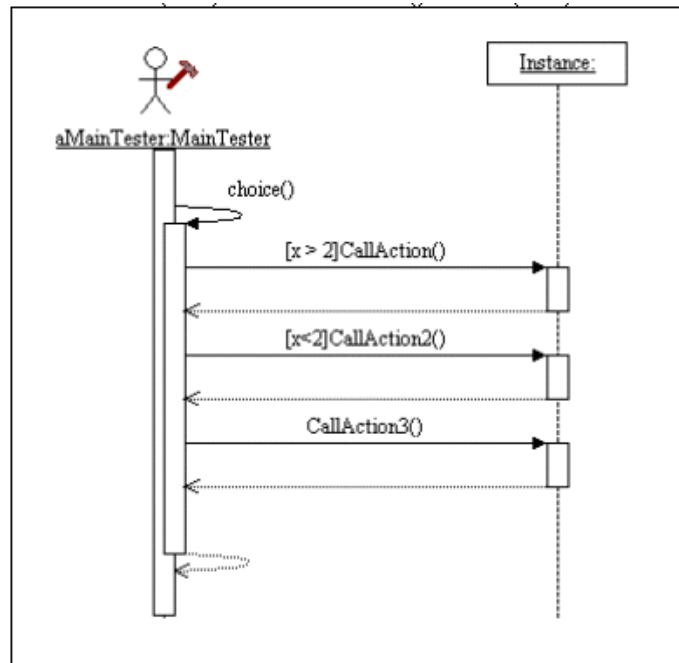


Figure 3-12. The if () then {} else control structure

Creating an instance

The instances of the context are automatically declared in the generated code. For example, "*ClassA myInstance*".

There are two different ways of creating the instance:

- ◆ in the "*setUp*" note of the context (Figure 3-13). For example, "*myInstance = new ClassA (parameter1, parameter2)*".

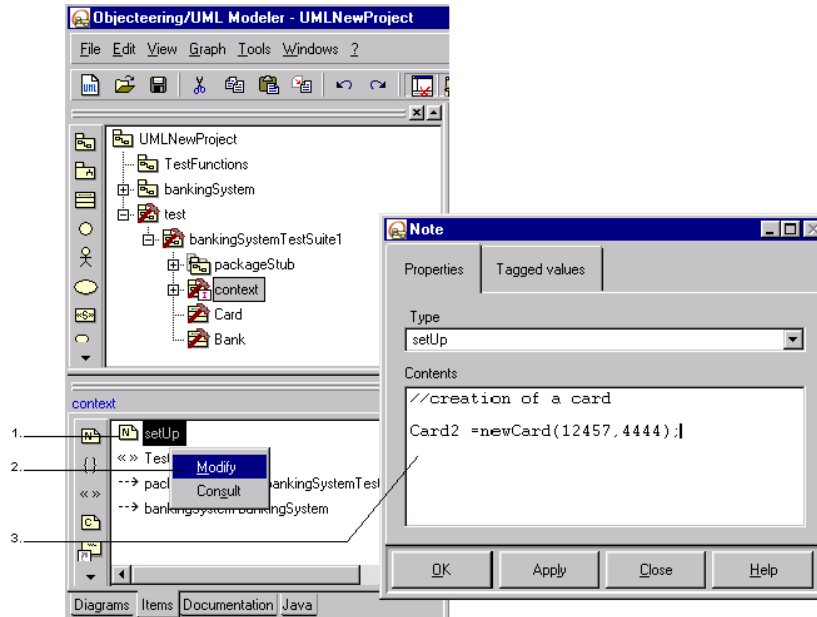


Figure 3-13. The context's "*setUp*" note is used to create an instance

Steps:

- 1 - Right-click on the context's "*setUp*" note in the "*Items*" tab of the properties editor.
- 2 - Run the "*Modify*" command from the context menu which then appears.
- 3 - Enter the contents shown above.

- ◆ in the sequence diagram, with a creation message sent to the "myInstance" instance

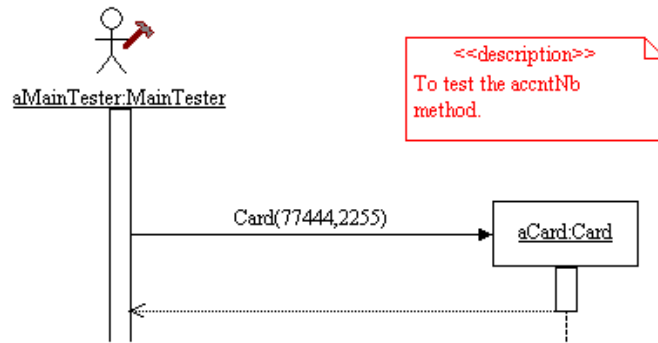


Figure 3-14. Creating instances in a sequence diagram

Running the test in the sequence diagram

The test is realized with an "assert" note on a return message. Syntax is as follows:

```
? == myValue
```

"myValue" can be an instance of the diagram or a primitive-type value. In case the type of "myValue" is different from the type of the returned value, a CAST is generated.

It is also possible to get the return value of a message with this "assert" note, using the following syntax:

```
myValue = ?
```

If "myValue" is an instance of the context, it will be updated. If not, a local variable will be created.

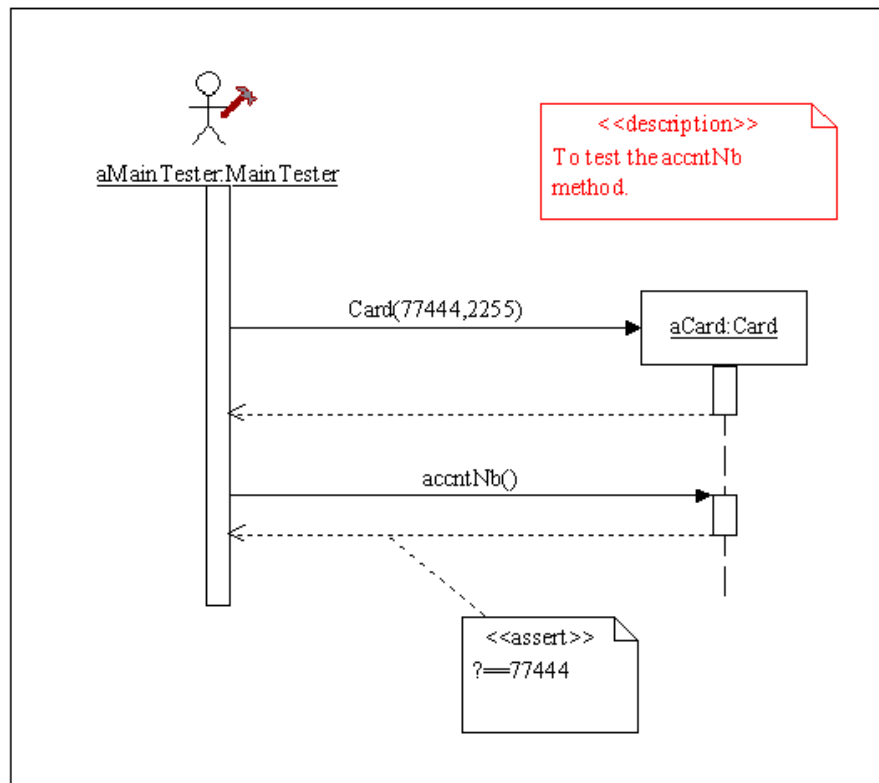


Figure 3-15. A simple test method

Defining Java test methods with "testCase" notes on the test suite package

For Java developers, it is also possible to write Java code in a "testCase" note of the test suite package. In this case, the developer need only write the body of his test method.

A test is made with the following possibilities:

- ◆ assert (boolean expression)
- ◆ assertEquals (object1, object2)

For further information, please refer to the "*How to write a test method in Java code*" section in chapter 4 of this user guide.

Creating new contexts

A test suite can have several different contexts. To create a new context, run the "*Create a context*" command on a test suite.

Creating new sequence diagrams

A context may have several sequence diagrams (in other words, several test methods). To add a test sequence, run the "*Create a test sequence*" command on the context package.

Creating new test case notes

A test suite package can have several test case notes (in other words, several test methods). As many "testCase" description notes as desired can be added to a test suite.

Maintaining your test model

The generation of a test model and of code from the model of your application have not modified your application model.

However, if you have modified your application model, your test model has to be updated. To do this, run the "*UML Test Designer/Update a Test Suite*" command from your test suites (as shown in Figure 3-16).

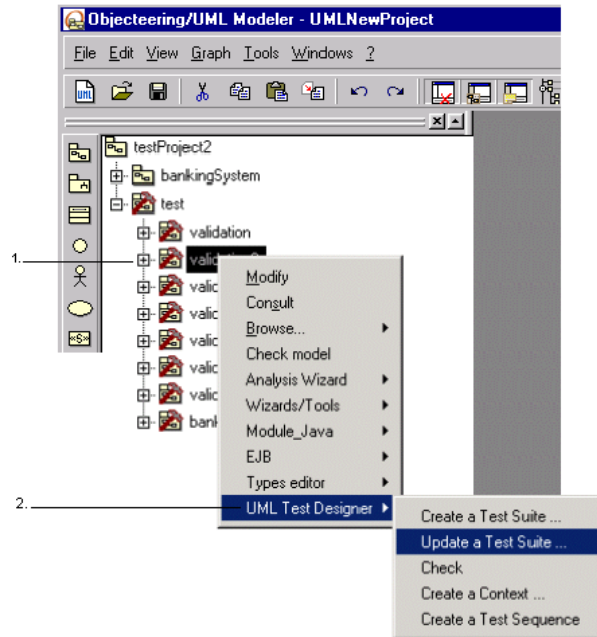


Figure 3-16. Updating a test suite belonging to your test model

Steps:

- 1 - Right-click on the test suite you wish to update.
- 2 - Run the "*UML Test Designer/Update a test suite*" command.

If you wish to check the global consistency between your application and your test model, use the "*UML Test Designer/Check*" command (as shown in Figure 3-17).

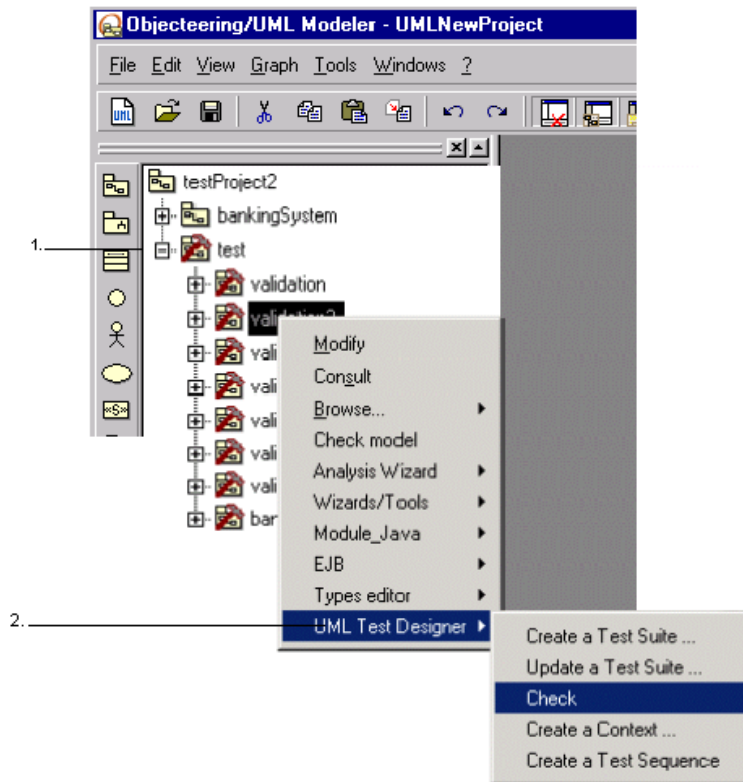


Figure 3-17. Checking the test model

Steps:

- 1 - In the explorer, right-click on the package for which you created a test suite.
- 2 - Run the "*UML Test Designer/Check*" command.

Chapter 3: Objecteering/UML Test Designer First Steps

These two commands can also be launched from the properties editor (as shown in Figure 3-18 below).

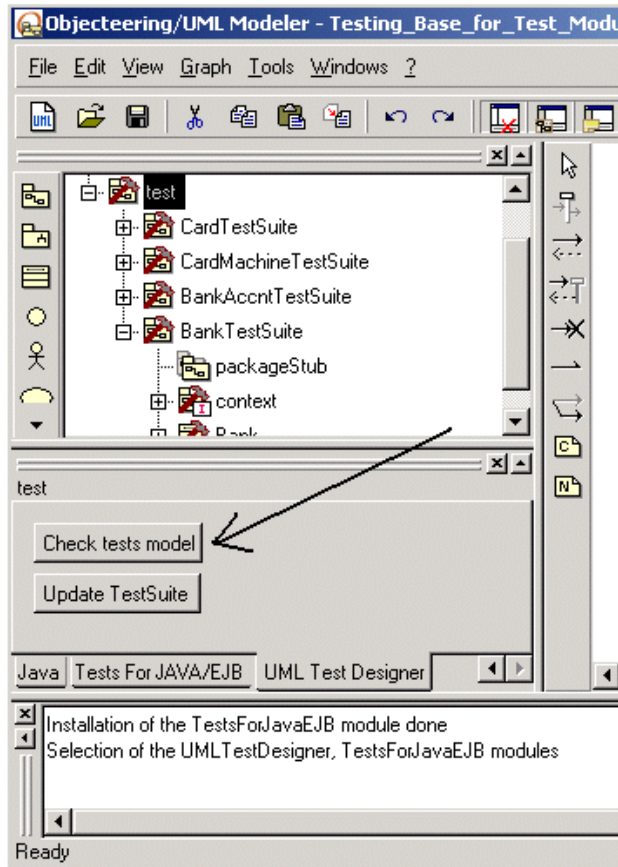


Figure 3-18. Running commands from the properties editor

Creating a Tests For Java/EJB generation work product

Overview

The *UML Test Designer* module is used to create your test model, whilst the *Tests For Java/EJB* module is used to generate the Java code of the tests.

In Objecteering/UML, the *Tests For Java/EJB* generation work product provides the commands for generating code, compiling, launching JUnit and generating documentation. It can also be used to manage the files that have been produced. Thus, if you destroy the generation work product, you will also destroy the files produced.

Creating a Tests For Java/EJB generation work product

The first step is the creation of a *Tests For Java/EJB* generation work product on the "test" package selected in the Objecteering/UML explorer.

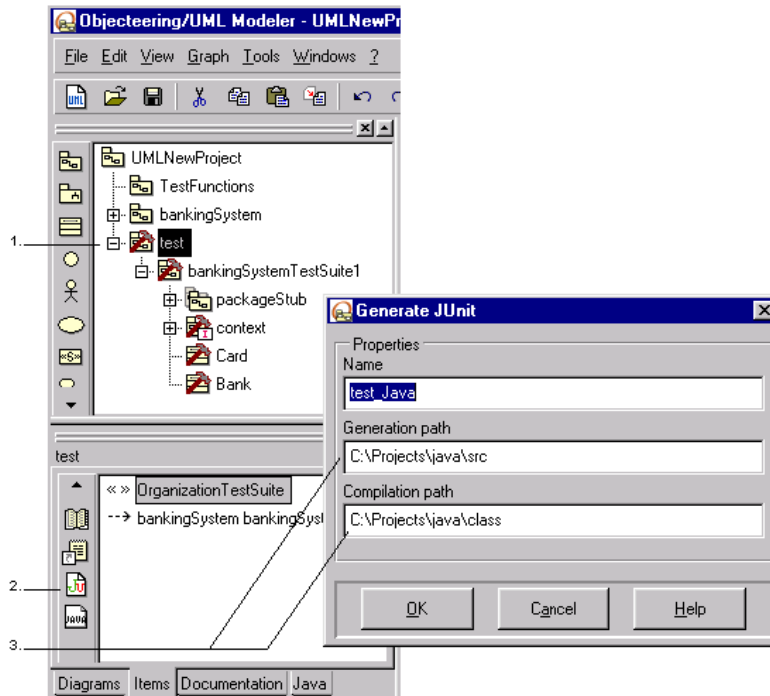



Figure 3-19. Creating the *Tests For Java/EJB* generation work product on the "test" package

Steps:

- 1 - Select the "test" package in the Objecteering/UML explorer.
- 2 - Click on the  "Tests For Java/EJB work product" icon in the "Items" tab of the properties editor.
- 3 - Specify the generation and compilation path for generated code.

Continue by creating a Java generation work product for each "PackageStub".

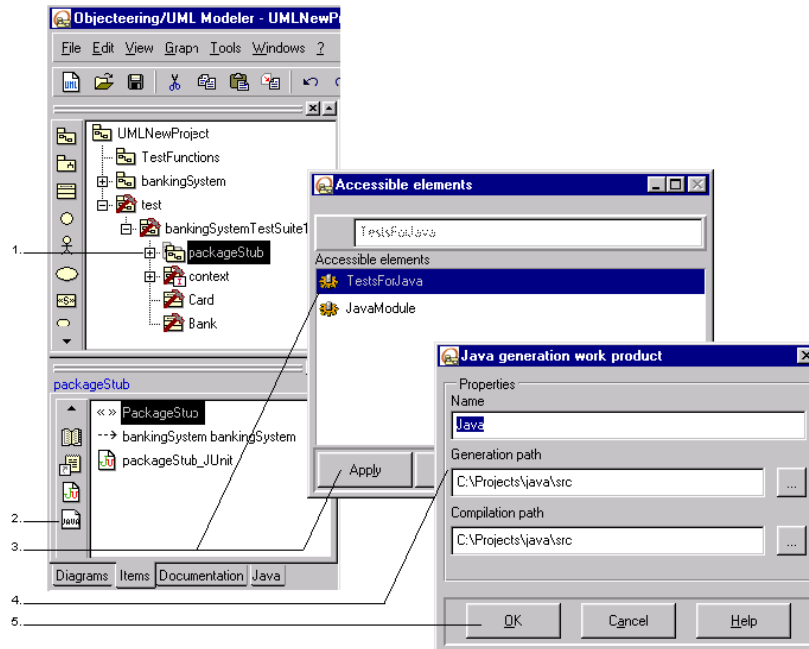



Figure 3-20. Creating the Java generation work product on the "PackageStub"

Steps:

- 1 - Select the "PackageStub" in the Objecteering/UML explorer.
- 2 - Click on the  "Java generation work product" icon in the "Items" tab of the properties editor.
- 3 - Select the "Tests For Java" module and then click on "Apply".
- 4 - Specify the generation and compilation path for generated code.
- 5 - Click on "OK" to confirm.

Generating and compiling test code

Generating and compiling code

Once the generation work products have been created, you are then able to generate and compile your tests' Java code.

First carry out the steps shown in Figure 3-21.

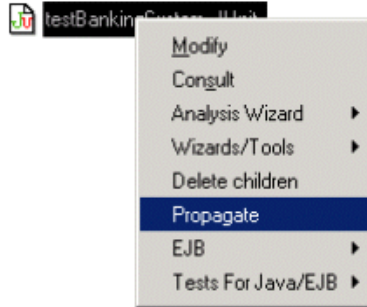


Figure 3-21. The "Propagate" command

Steps:

- 1 - Select the "test" package in the explorer.
- 2 - Run the "Propagate" command on the package's JUnit generation work product in the properties editor.

Continue by following the steps shown in Figure 3-22.

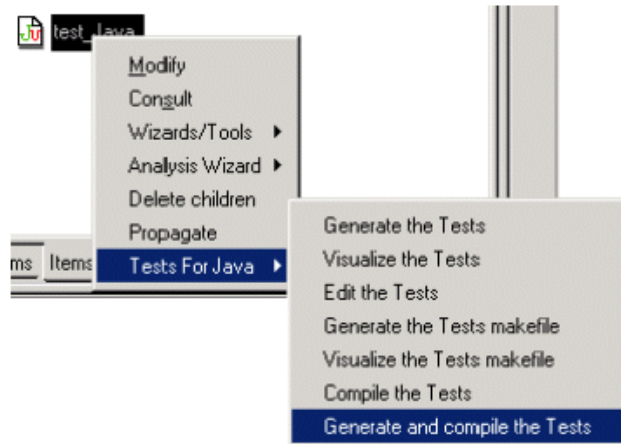


Figure 3-22. Generating and compiling

Steps:

- 1 - Run the "*Tests For Java/EJB//Generate and compile the tests*" command from the JUnit generation work product.

Note: If compilation fails, use the "*Analyze the compilation*" command.

Chapter 3: Objecteering/UML Test Designer First Steps

It is also possible to generate and compile tests from a "testSuite" or a "context" package.

These commands are accessed in the "Tests for Java/EBJ" tab in the properties editor (Figure 3-23).

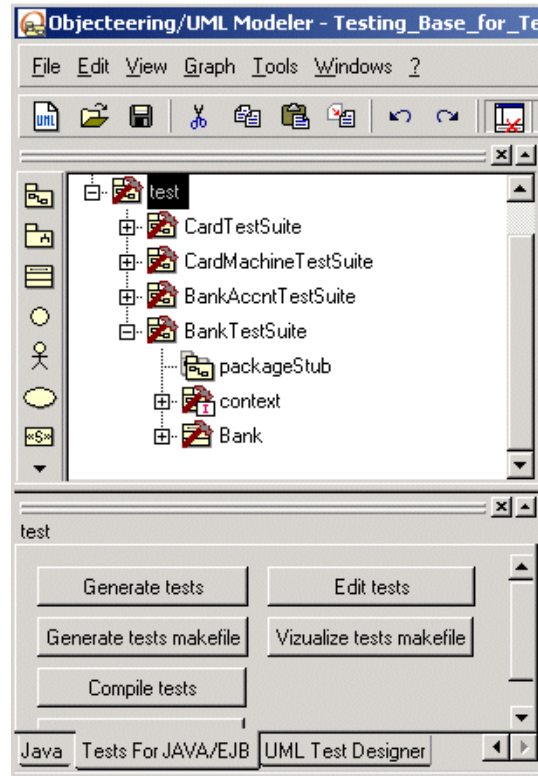


Figure 3-23. Commands available in the "Tests for Java/EJB" tab of the properties editor

Generated Tests For Java/EJB code

To run this command, you need to have already generated the Java code of your application.

When the "*Generate*" command is run on the selected work product, a Java type file is generated for the package referenced by the *Tests For Java* work product.

The file generated on a context package corresponds to a JUnit TestCase. The generated class contains test methods named "*testxxx()*", which are executed when the test runner is launched.

The file generated on a test suite package corresponds to a JUnit TestSuite. The generated class:

- ◆ contains test methods named "*testxxx()*" defined in the test case notes of the test suite package
 - ◆ includes all the TestCases generated on the "*context*" sub-packages of the test suite.
-

Running tests with JUnit

If your tests have been successfully compiled, you can now launch the JUnit Test Runner from the "test" package, or from a "testSuite" package or "context" package (as shown in Figure 3-24).

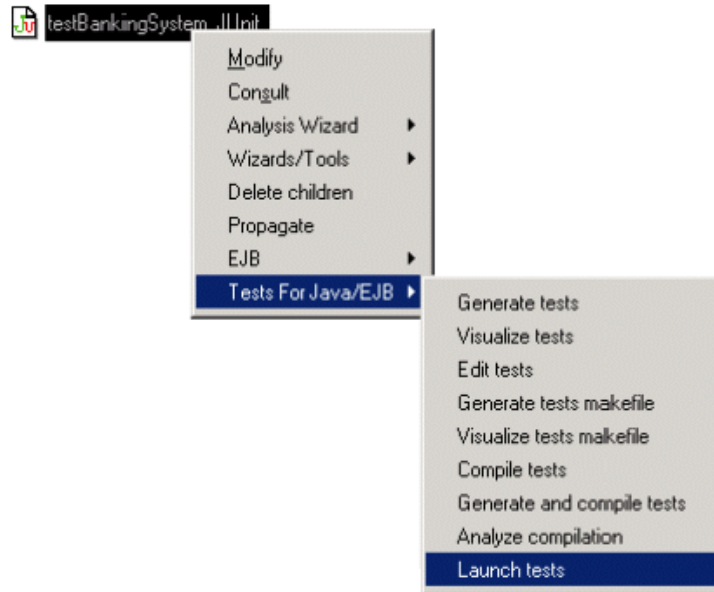


Figure 3-24. Launching the JUnit Test Runner

Steps:

- 1 - Select the generation work product in the "Items" tab of the properties editor.
- 2 - Run the "Tests For Java/Launch tests" command.

Commands can also be accessed from the properties editor (Figure 3-25).

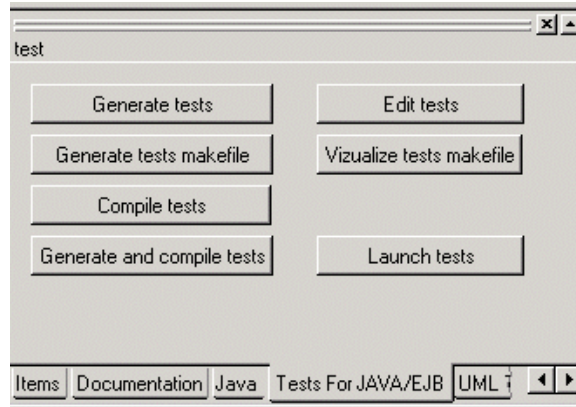


Figure 3-25. Accessing commands through the properties editor

All the test suites will then be executed (Figure 3-26).

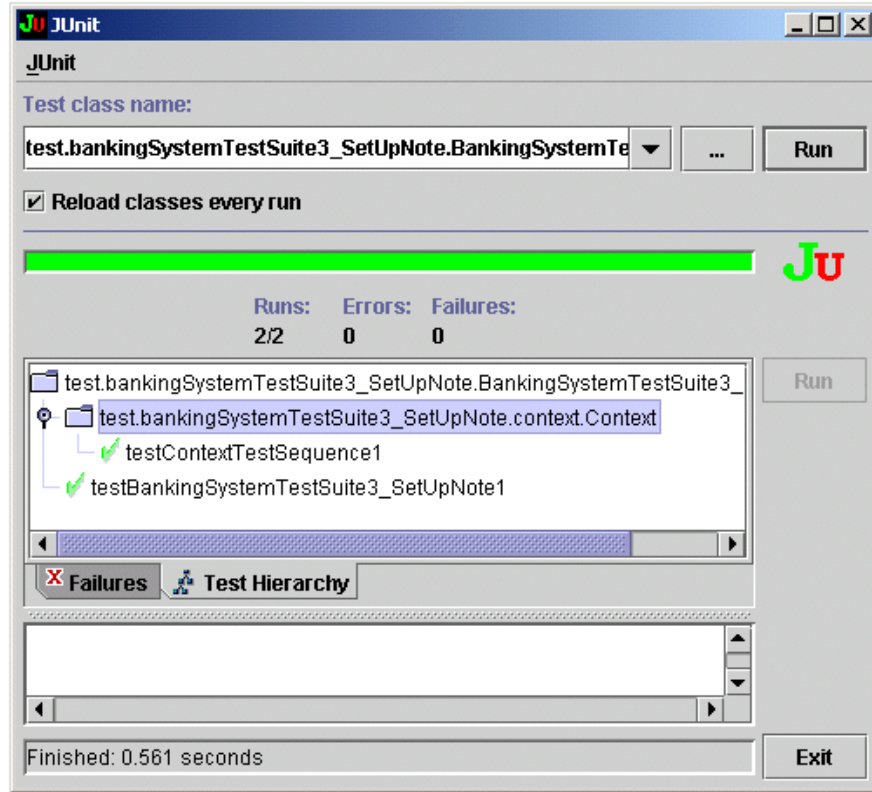


Figure 3-26. JUnit Test Runner

Generation work product commands

All the commands for generating, editing, compiling and executing *Tests For Java/EJB* are grouped together in the *Tests For Java/EJB* context menu (as shown in Figure 3-27).

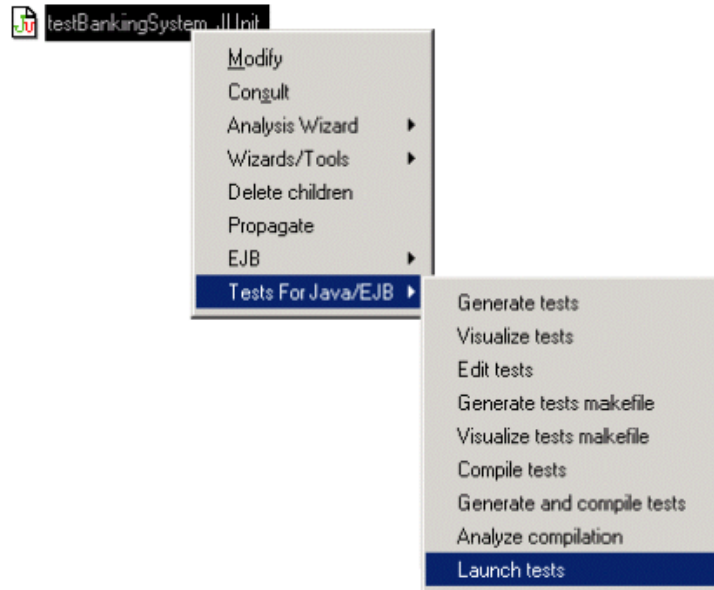


Figure 3-27. The *Tests For Java/EJB* generation work product context menu

These commands are also accessible from the properties editor (Figure 3-28).

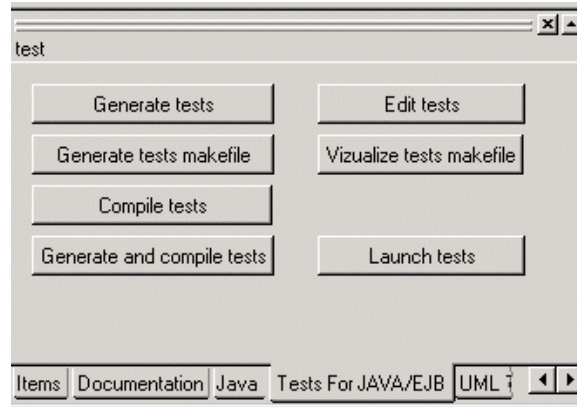


Figure 3-28. Commands accessible from the properties editor

Services available

The ... menu command	called from...	is used to...
Generate tests	a " <i>context</i> " package	generate the Java code of a test case.
Generate tests	a " <i>test suite</i> " package	generate the Java code of a test suite.
Generate tests	the " <i>test</i> " package	generate the Java code of the whole test suite.
Visualize tests		visualize the generated Java code.
Edit tests		edit and modify the generated Java code.
Generate tests makefile		generate the makefile for the java files.
Visualize tests makefile		
Compile tests		
Analyze the compilation		
Launch tests	a " <i>context</i> " package	launch JUnit TestRunner to execute all the test methods of the context and display the results.
Launch tests	a " <i>test suite</i> " package	launch JUnit TestRunner to execute all the test methods of the test suite and display the results.
Launch tests	the " <i>test</i> " package	launch JUnit TestRunner to execute all the test methods of the test suites and display the results.
Generate and compile tests		

Examples

Visualizing generated Tests For Java/EJB code

Generated *Tests For Java/EJB* code can also be visualized. The "*Visualize*" command can only be launched on a work product to which a ".java" file is associated, in other words, on a work product of a package for which code has been generated.

To visualize generated *Tests For Java/EJB* code, simply click on the generation work product in question using the right mouse button, and run the "*Tests For Java/EJB/Visualize*" command from the context menu which then appears. This command opens a window contained the generated code. It is not possible to modify the code directly in this window.

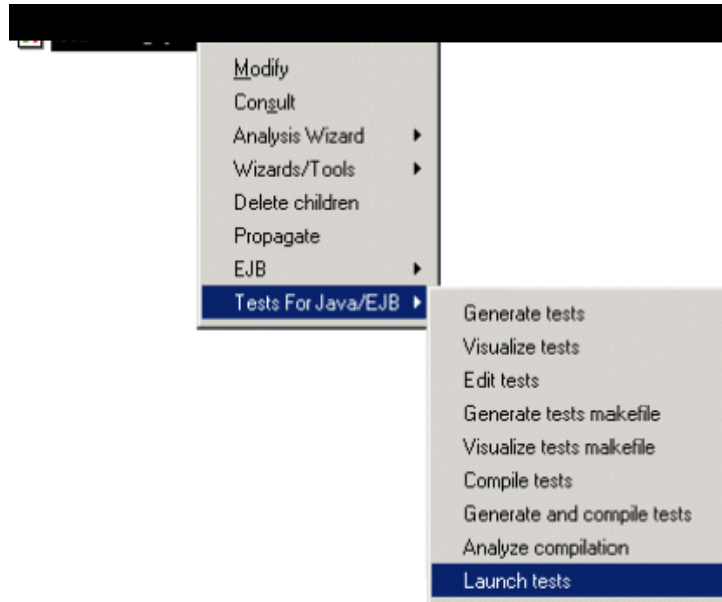


Figure 3-29. Visualizing generated code

Editing generated code

The *Tests For Java/EJB* code generated for the element in question can be edited using the editor chosen, as shown in Figure 3-30. Zones represented between markers can be modified, and modifications are directly incorporated into the model when the editor is closed.

```
package test;

import test.bankApplicationTestSuite1_insert.*;
import test.TestWithStubs.*;
import test.TestWithStubs2.*;
import test.aPackageSuite.*;
import test.test.*;

//START OF MODIFIABLE ZONE@OBJID@67095@3482059184:1782@T
/**
 * This is the description note of the test project.
 * This text is describing the test suites
 */
//END OF MODIFIABLE ZONE@OBJID@67095@3482059184:1782@E
public class Test extends junit.framework.TestCase
{
    public Test (String name)
    {
        super(name);
    }

    public static junit.framework.Test suite()
    {
```

Figure 3-30. Editing the code generated on the "test" package

Chapter 4: Frequently asked questions
(FAQ)

How do I test abstract class implementations?

Introduction

When different classes implement the same abstract class, it may be possible to define a generic scenario which will be played out with all those classes' instances.

Note: In order to help test abstract classes, we strongly recommend that you have a look at the "First Steps" chapter of this user guide.

User interactions in the test model

The first step is to create a test suite with the abstract class.

The abstract class has to be selected during the test suite creation. An abstract class implementation is created in the "Test Suite". Instances of this class may be used in the "Sequence diagram" of the test.

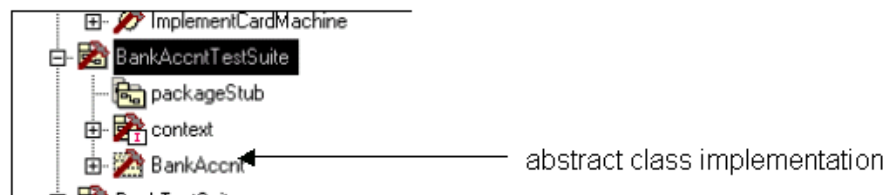


Figure 4-1. Abstract class implementation

Chapter 4: Frequently asked questions (FAQ)

Next, we have to define the test.

Create one or several sequence diagrams with an instance of the abstract class implementation.

In the context package, create an instance of a concrete child class of the abstract class. The test will be played out with this instance.

If you have several children classes, you can create an instance for each class. This means you will have one test for each class.

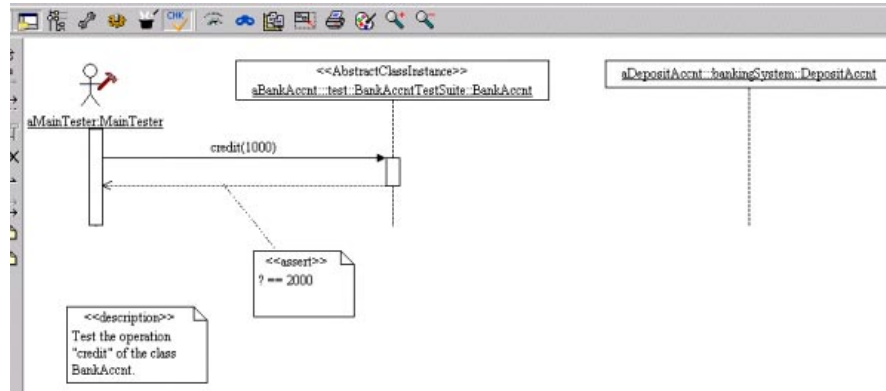


Figure 4-2. Creating sequence diagrams

Finally, we are now going to initialize the context.

The generated test sequence is a template defined for a generic instance. To execute it with different concrete instances, proceed as follows:

- ◆ add instances of the classes which inherit from the tested abstract class
- ◆ initialize these instances in the "setUp" note of the context.

For example:

```
aDepositAcnt = new DepositAcnt(1111,1000,false);  
aBankBook = new BankBook(1111,1000,3);  
anotherBankBook = new BankBook(2233,1001,3);
```

How do I test interface implementations?

Introduction

When different classes realize the same interface, it may be possible to define a test scenario to be applied to each class.

Note: In order to help test interfaces we highly recommend that you have a look at the "First Steps" chapter provided in this user guide.

User interactions in the test model

The first step is to create a test suite with the interface.

Firstly, you have to select an interface during the test suite creation.

An implementation class of the interface is created in the "Test Suite". This class will be used in the "Sequence Diagram" of the test.

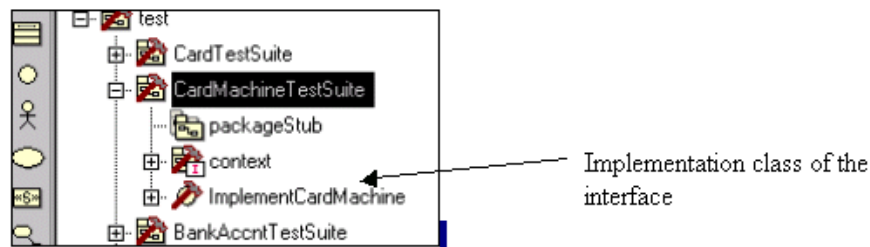


Figure 4-3. Implementation class of the interface

Chapter 4: Frequently asked questions (FAQ)

Next, we have to define the test.

Create a diagram sequence and use the instance of the implementation class of the interface to carry out your test.

In the context package, create instances of a concrete implementation class of the interface. The test scenario will be played out with this instance.

Those instances have to be instantiated in the "setUp" note of the context.

If you have several implementation classes, you can create an instance for each class. This means you will have one test for each class.

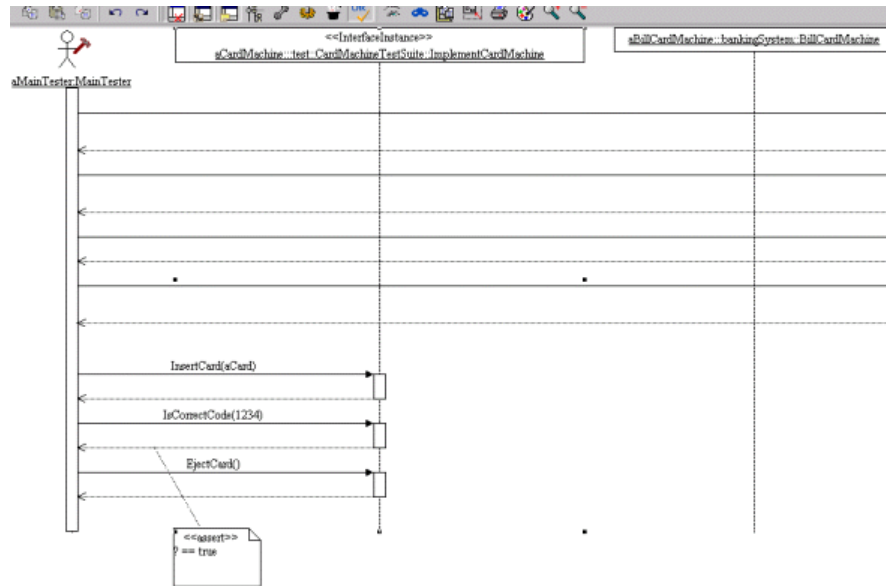


Figure 4-4. Creating sequence diagrams

Finally, we are now going to initialize the context.

The generated test sequence is a template defined for a generic instance. To execute it with different concrete instances, proceed as follows:

- ◆ add instances of the classes which realizes the tested interface
- ◆ initialize those instances in the setup note of the context

For example:

```
aBankATM = new BankATM();  
aBillCardMachine = new BillCardMachine();  
anotherBillCardMachine =new BillCardMachine();
```

How do I generate tests for functions with primitive parameters?

Introduction

A test scenario can be automatically generated for the functions of a class whose parameters are all of primitive type.

Note: In order to help generate tests, we strongly recommend you have a look at the "*First Steps*" chapter provided in this user guide.

User interactions in the test model

Start by configuring the *UML Test Designer* module. The following fields have to be defined:

- ◆ "Parameter values for tests": Specify the values used for each type in the test scenario (Figure 8-5).

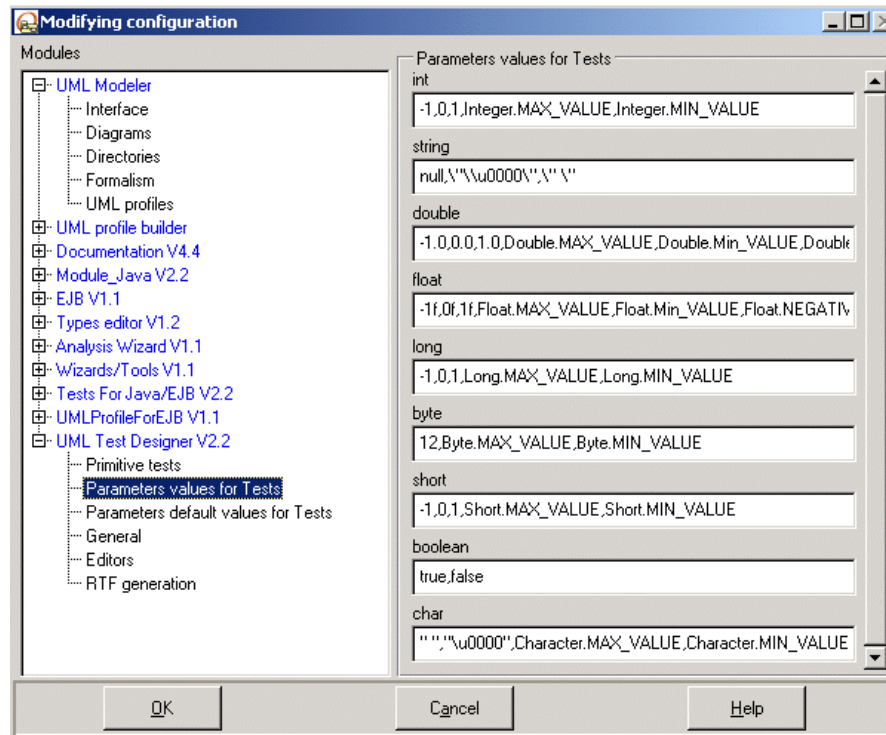


Figure 4-5. Configuring parameter values for tests

Chapter 4: Frequently asked questions (FAQ)

- ◆ "Parameter default values for tests": Specify the default values used for each type in the test scenario.

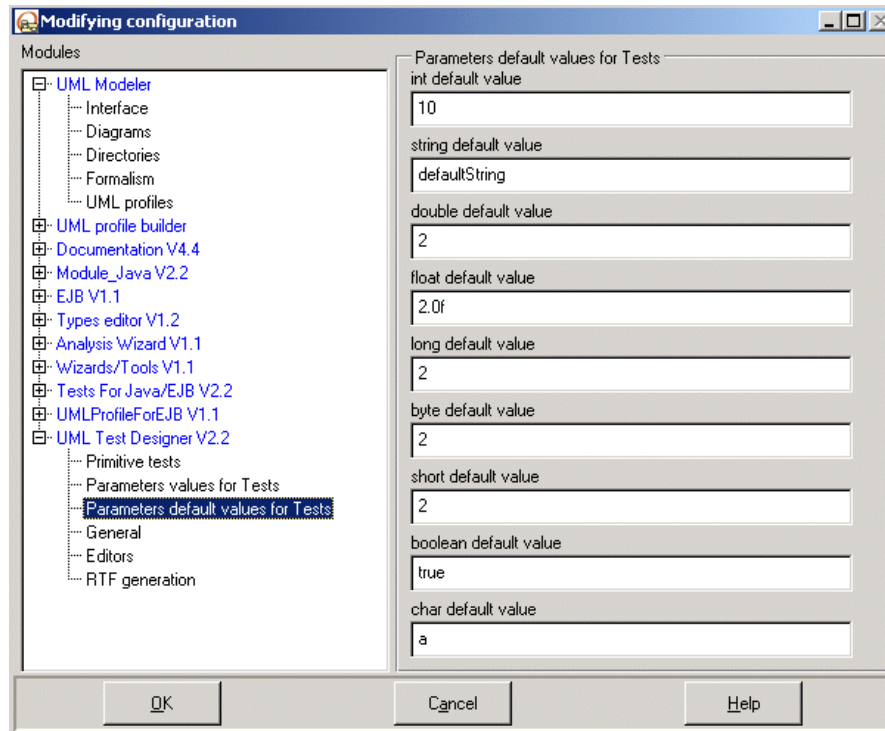


Figure 4-6. Configuring parameter default values for tests

Next, launch the "*Generate primitive tests*" command from the *UML Test Designer* menu on the class under test. If the class owns one or several operations with only primitive parameters, a test scenario is created with different calls to these operations.

Continue by checking the instance creation. An instance of the tested class is used in the test sequence diagram. You have to specify the instance creation in the sequence diagram or in the "*setUp*" note of the context.

Finally, generate the code. The testable methods are called with different sets of parameters and the corresponding code can be generated.

How do I add a new context to a test suite?

Introduction

Several different contexts can be defined in a test suite. This makes it possible to separate your test methods, and use different instances.

The "Create a context" command

To create a context, carry out the steps illustrated in Figure 4-7.

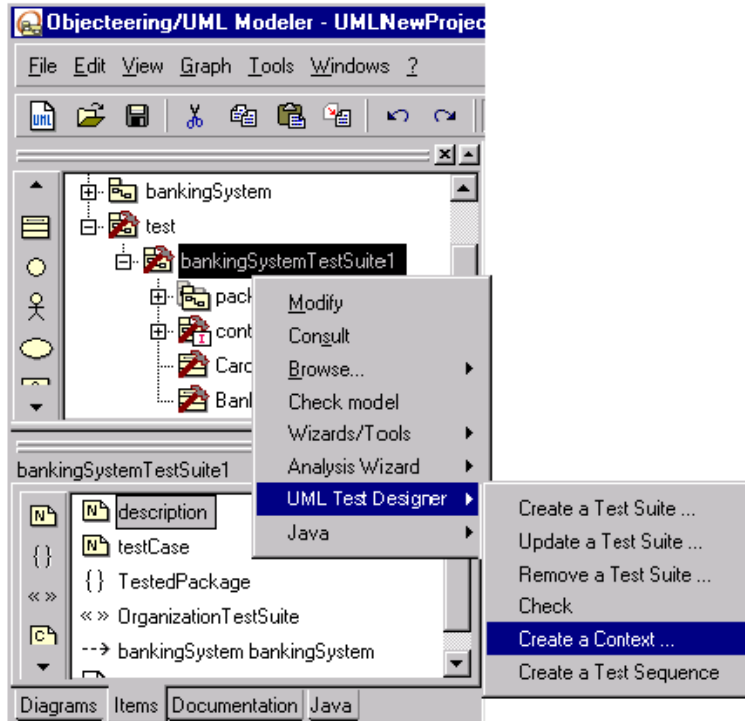


Figure 4-7. The "Create a context" command

Steps:

- 1 - Right-click on a test suite in the Objecteering/UML explorer.
- 2 - Run the "UML Test Designer/Create a context" command.

Chapter 4: Frequently asked questions (FAQ)

You can also select a test suite and launch the command from the property box button (as shown in Figure 4-8).

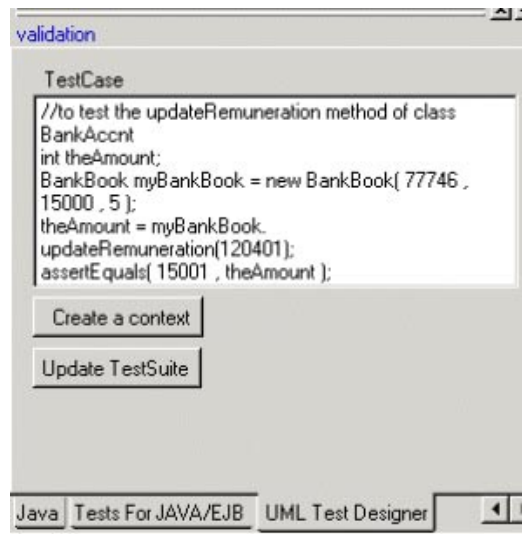


Figure 4-8. Launching the command from the properties editor.

A new context package is then created, inside which you can define tests.

How do I add a new test sequence to a context?

Introduction

Several test sequences can be defined within a context. Each sequence will be translated into a JUnit test method.

The "Create a test sequence" command

To create a test sequence, carry out the steps illustrated in Figure 4-9.

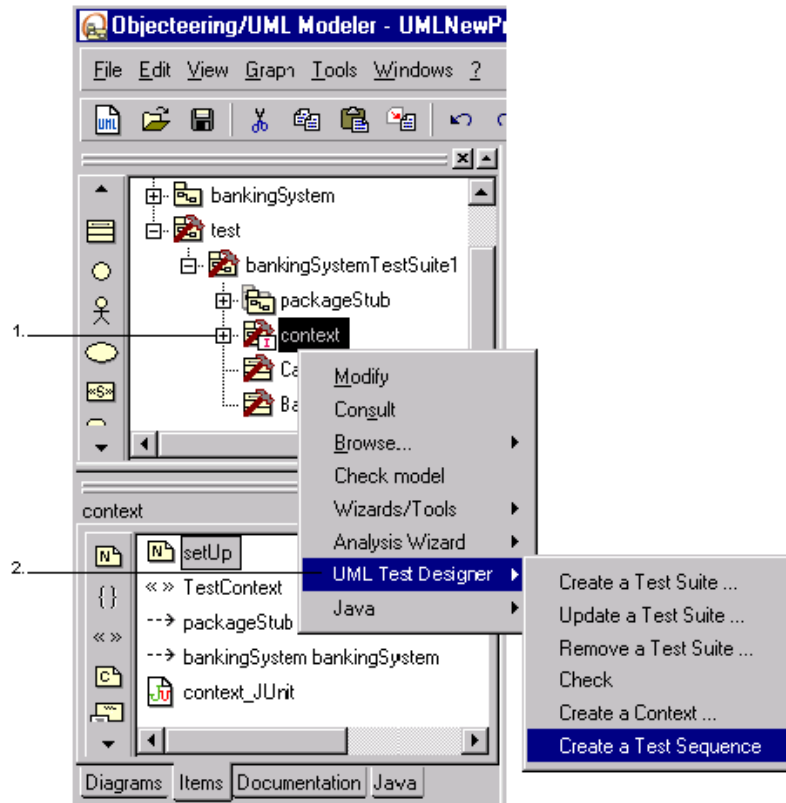


Figure 4-9. The "Create a test sequence" command

Steps:

- 1 - Right-click on a context package in the Objectteering/UML explorer.
- 2 - Run the "UML Test Designer/Create a test sequence" command.

Continue by naming the test sequence diagram (Figure 4-10).

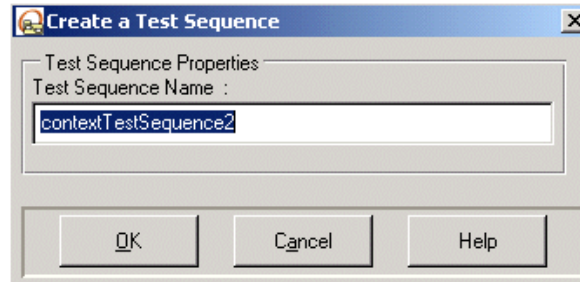


Figure 4-10. Naming the test sequence

Chapter 4: Frequently asked questions (FAQ)

A new test sequence is automatically created, with a default structure in which you can define your test (as shown in Figure 4-11).

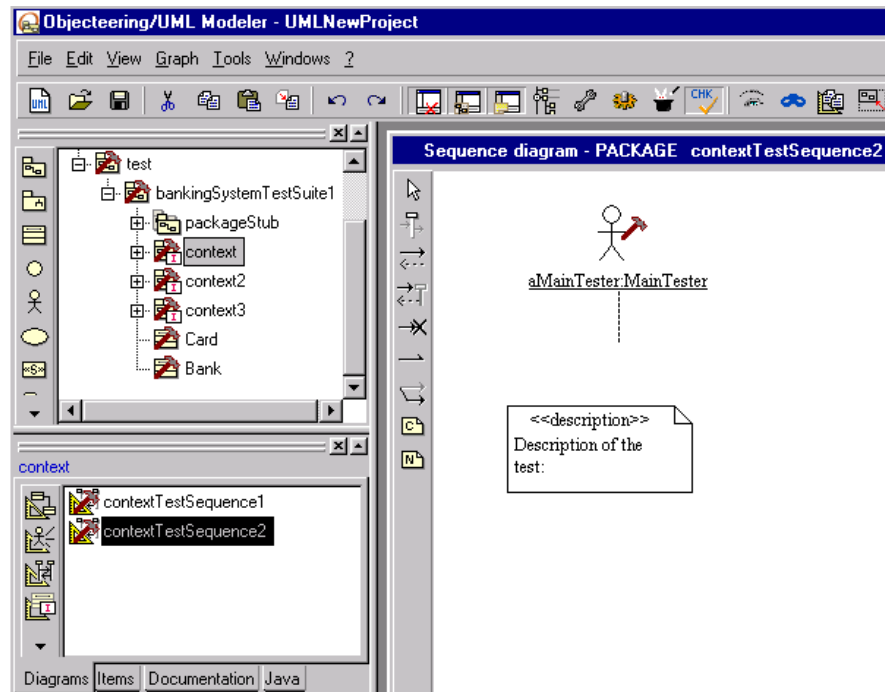


Figure 4-11. The default structure created for the new test sequence

How do I write a test method in Java code?

Introduction

It can be useful to use Java code instead of a sequence diagram to define a test method.

User operations in the test model

To write a test method in Java code, carry out the following steps:

- 1 - Create a "*testCase*" note on a test suite. Indeed, the Java code has to be written in a "*testCase*" note on a test suite. To define several methods in a test suite, simply add notes stereotyped <<testCase>>.
- 2 - Write the method body. The code written in this note is inserted into a test method during code generation. This code must contain a JUnit assertion. This insertion is carried out either using "*assert (boolean)*" or with "*assertEquals(Object1, Object2)*".

Example

The contents of the "testCase" note are as follows:

```
//to test the updateRemuneration method of class BankAccnt
int theAmount;
BankAccnt myBankAccnt = new BankAccnt( 77746 , 15000 );
theAmount = myBankAccnt.updateRemuneration(120401);
assertEquals( 15001 , theAmount );
```

The generated test method is as follows:

```
public void testBankingSystemTestSuite7_testCase1()
{
//BeginIdTxt.....T/5HG3/CLM4LL1:SD1
//to test the updateRemuneration method of class BankAccnt
int theAmount;
BankAccnt myBankAccnt = new BankAccnt( 77746 , 15000 );
theAmount = myBankAccnt.updateRemuneration(120401);
assertEquals( 15001 , theAmount );
//EndIdTxt.....E/5HG3/CLM4LL1:SD1
}
```

The corresponding code will be generated from the test suite JUnit generation work product.

How do I define a `Setup()` called before each test method?

Introduction

A context package can own many test methods. Each of these methods uses the same instances of the context.

JUnit provides the user with the possibility of defining a "`setUp()`" method which is called before each of the context's test methods.

This method's body can be written in the context's "`setUp`" note.

User operations in the test model

To define a "`Setup()`" called before each test method, carry out the following operations:

- 1 - Edit the "`setUp`" note of the selected context.
- 2 - Write the method body. The code may be used to initialize instances of the context. It is not necessary to declare the instance, as this is automatically done in the generated code.

Example

The contents of the "setUp()" note are as follows:

```
//called before each test method  
Card2 =new Card(12457,4444);
```

The generated "setUp()" method is as follows:

```
protected void setUp()  
{  
//BeginIdTxt.....T/SCG2/4EHPGN1:3L  
4  
//called before each test method  
Card2 =new Card(12457,4444);  
//EndIdTxt.....E/SCG2/4EHPGN1:3L  
4  
}
```

The corresponding code will be generated from the context.

The "tearDown" method

It is also possible to define a "tearDown" method with a "tearDown" note defined on a context. The corresponding method will be called after each test method.

How do I test a message with an assert note?

Introduction

Inside a test sequence diagram, tests are made on return messages with an "assert" note.

User operations

To test a message with an "assert" note, carry out the following steps:

- 1 - Add a note with the <<assert>> stereotype on a return message.
- 2 - Complete this note using the syntax described in the paragraph below.

Syntax

To compare the returned object with another object:

```
? == aValue
```

or

```
?==aValue
```

To test a boolean expression:

```
? == false
```

or

```
? == true
```

To compare reals:

```
? == aValue , delta
```

Example 1

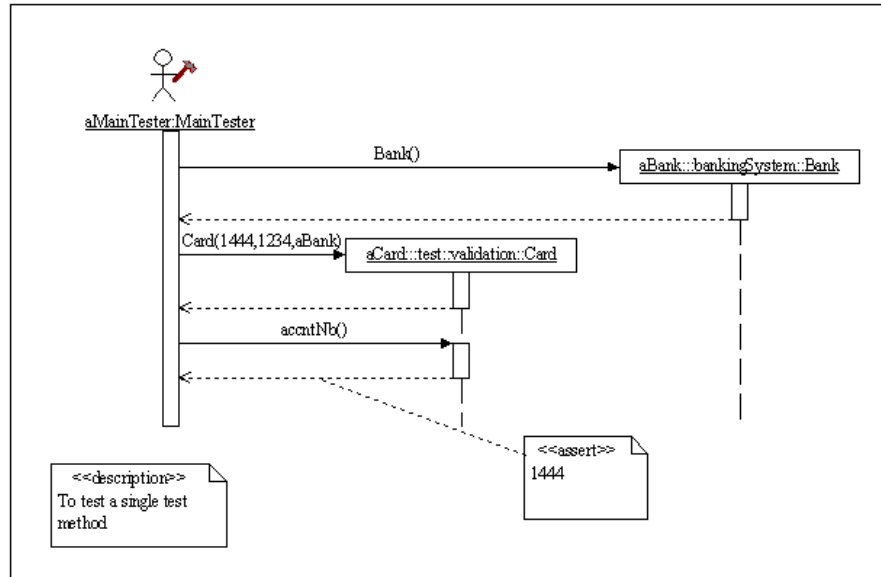


Figure 4-12. First example

Example 2

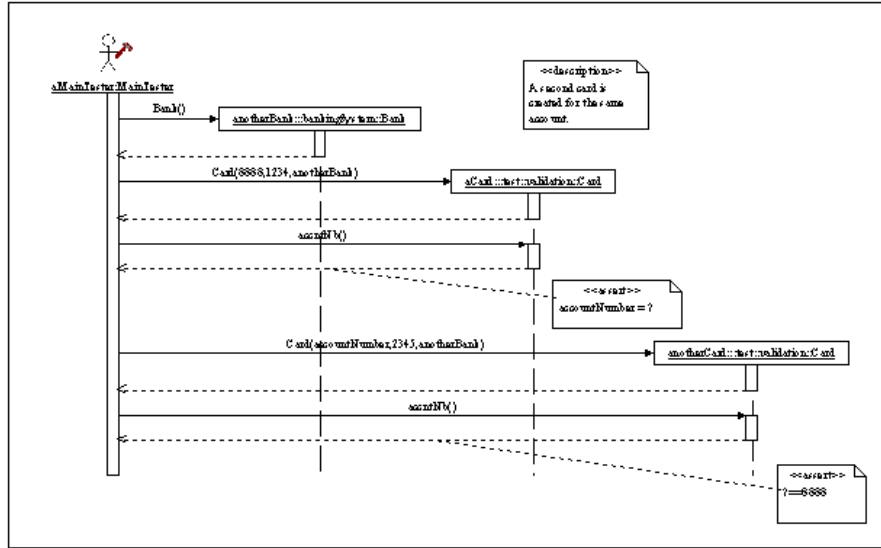


Figure 4-13. Second example

How do I get a returned value with an assert note?

Introduction

It is also possible to instantiate or create a variable with an *"assert"* note.

Syntax

To get the return value of a method invocation, an *"assert"* note is created on the return message, with the following syntax:

```
myValue = ?
```

Interpretation

If *"myValue"* is the name of one of the context's instances, it will be instantiated using the following code:

```
myValue = theDiagramInstance.method(param, ...)
```

If *"myValue"* is unknown in the diagram, a local variable is created with the following code:

```
myValueType myValue = the DiagramInstance.method(param, ...)
```

Example

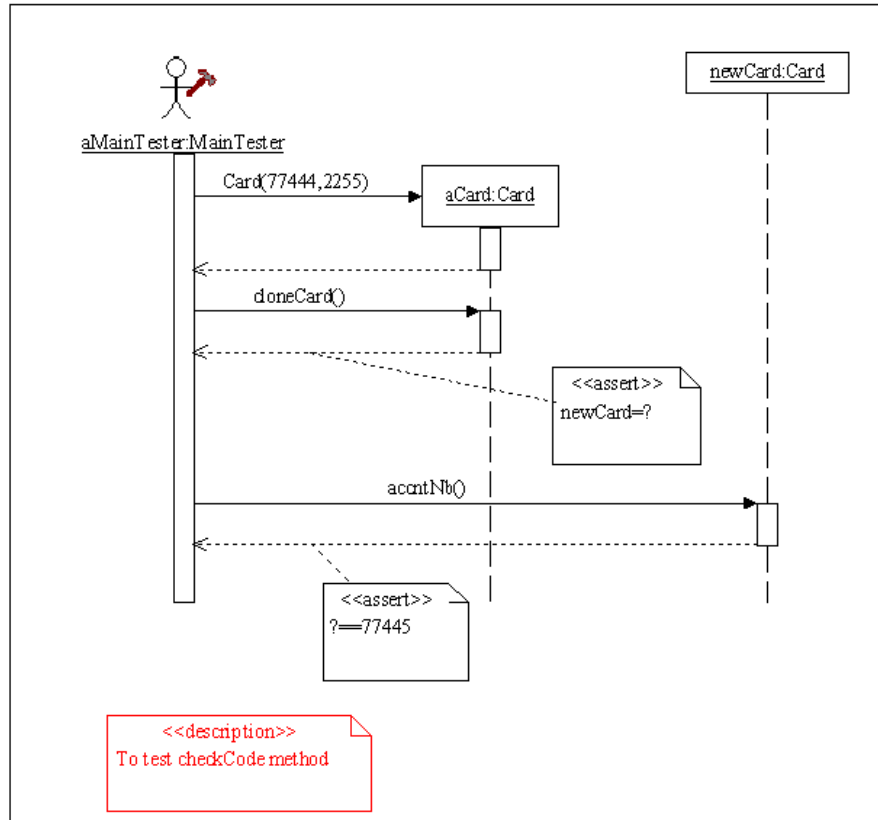


Figure 4-14. Example of getting a return value with an "assert" note

In this example, the "`newCard=?`" assert note will be translated into:

```
newCard = ( bankingSystem.Card ) aCard.cloneCard();
```

How do I insert an assertion into the test?

Introduction

It also possible to add an assertion in the test sequence.

User interactions

- 1 - Add a note with the stereotype <<assert>> on a returned message. Only one “assert” note is allowed on a returned message.
- 2 - Complete it using the following syntax.

Syntax

The content of the note has to be a boolean expression, “*myInstance.Name == 12*” for example and is written in target code.

How do I insert target code into the test?

Introduction

It may be useful to add directly source code in a test sequence when the interactions are hard to represent in a UML sequence diagram.

User interactions

- 1 - Add a note with the stereotype <<javaTargetCode>> on a returned message. Only one note (“assert” or “javaTargetCode”) is allowed on a returned message.
- 2 - Complete it with Java code.

Test execution

This code is inserted in the generated file after the returned message assertions.

How do I generate and compile a new test suite?

The "*UML Test Designer/Create a test suite*" command is used from the application model to create new test suites. To make test suites executable, carry out the following steps:

- 1 - From the test model JUnit work product in the "*Items*" tab of the properties editor, run the "*Propagate*" command.

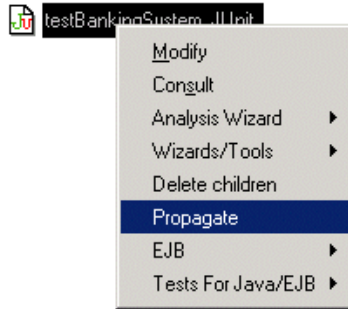


Figure 4-15. The "*Propagate*" command

- 2 - Create a Java generation work product on the test suite stub package with the "*Tests For Java*" element.
 - 3 - Launch the "*Tests For Java/Generate and compile*" from the test package JUnit work product. This command can also be launched from the test suite package.
-

How do I generate and compile a new context?

Introduction

The "*UML Test Designer/Create a context*" command is used from a test suite to create new contexts. To make these executable, carry out the following steps:

- 1 - From the test model JUnit work product in the "*Items*" tab of the properties editor, run the "*Propagate*" command.
 - 2 - Launch the "*Tests For Java/Generate and compile*" from the test package JUnit work product. This command can also be launched from the context package.
-

How do I test an EJB?

Introduction

It is possible to test an EJB in a real J2EE server environment using the *Objecteering/Tests For Java/EJB* module.

The EJB selected for tests could be a stateful, a stateless or entity one.

Note: In order to help test EJBs, we strongly recommend that you have a look at the "*First steps*" chapter in this user guide.

User interactions in the test model

The first step is to create a test suite for the EJB.

Once the Stateless EJB has been generated, compiled and deployed, select the EJB sub-system and create a test suite (Figure 4-16).

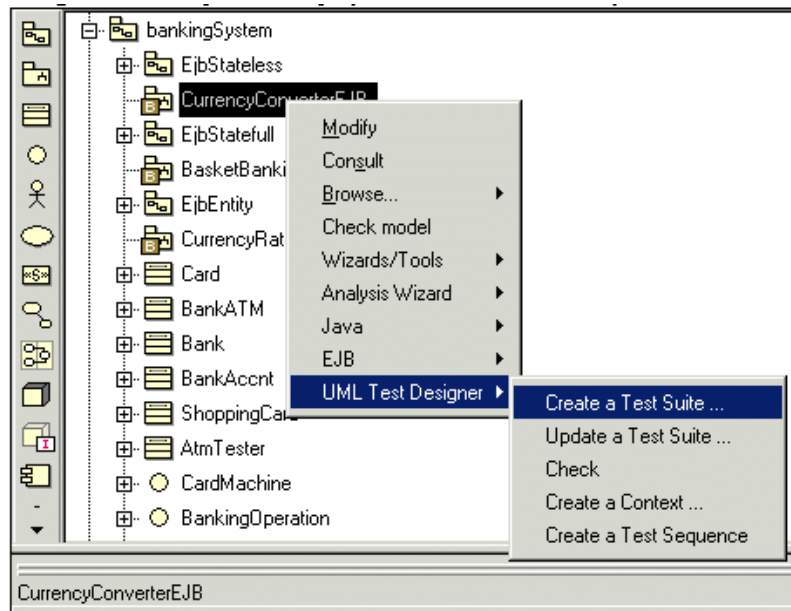


Figure 4-16. Creating a test suite

A dialog box is open to enter the test suite name and to select the class to be tested.

Warning: It is not possible to select stubs.

Chapter 4: Frequently asked questions (FAQ)

Continue by defining the test for the EJB.

Select the created test suite (Figure 4-17).

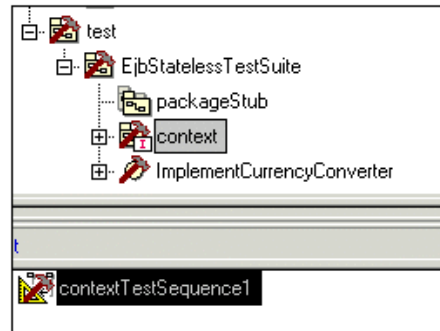


Figure 4-17. Selecting the test suite

Next, create one or several sequence diagrams on its context package (Figure 4-18).

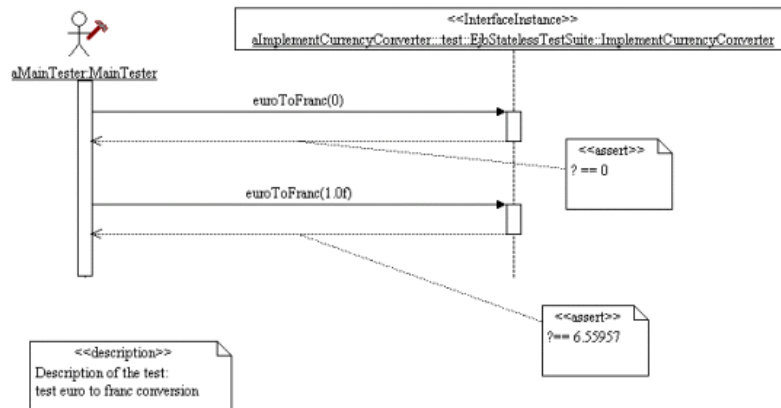


Figure 4-18. Creating sequence diagrams

Finally, build and launch EJB tests.

Create a JUnit work product at the root of test suite package and generate and compile the tests.

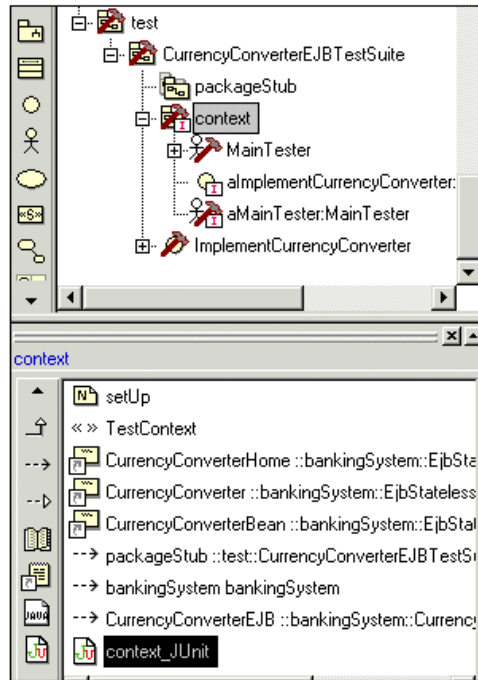


Figure 4-19. JUnit generation work product

Launch the tests. The JUnit control panel is launched and the results of tests are displayed.

Note: Your server must be running to enable *Objecteering/Tests For Java/EJB* to connect to it. The EJB must be deployed before running the test.

Chapter 5: Principles of
Objecteering/UML Test
Designer

Selecting and parameterizing the Objecteering/UML Test Designer module

Selecting the Objecteering/UML Test Designer module

Before being able to use the *Objecteering/UML Test Designer* module in your UML modeling project, it must first be selected. This operation is detailed in the "Selecting modules in the current UML modeling project" section in chapter 3 of the *Objecteering/Introduction* user guide.

Parameterizing the Objecteering/UML Test Designer module

The *Objecteering/UML Test Designer* and *Objecteering/Tests For Java* modules provide various ways of customizing the *Tests For Java* generator, in order to adapt it to your specific programming style.

- ◆ Parameterization through general *UML Test Designer* module parameters (for further information, please refer to chapter 7, "Parameterization" of this user guide).
 - ◆ Parameterization using the *UML Profile Builder* tool. Code generation can be adapted by redefining the J methods in charge of producing zones of *Tests For Java* code. Through *Objecteering/UML Profile Builder*, you can define your own tagged values, documents and items, as well as rules for checking, generating and validating (for further information, please refer to the *Objecteering/UML Profile Builder* user guide).
-

Work products

The *Objectteering/Tests For Java/EJB* module is used to generate *Tests For Java/EJB* code from a UML model created in Objectteering/UML.

Before generating *Tests For Java/EJB* code, a work product must be created.

Work products represent elements external to Objectteering/UML, which are delivered externally or which are used by other tools. These can be deliverables, such as documentation, source code (C++, Java, etc), database SQL schemas or any other element produced by Objectteering/UML. Work products can be accessed in the "Items" tab of the properties editor. They maintain consistency with related external elements (typically files), and provide services specific to the target and the external element, such as "Generate" or "Visualize".

Work products follow the project's composition structure (project/package/class/logic). If you create a work product on a package during the first generation, a work product will be created for each component (sub-package or class) that can have such a work product. If components are added after the last generation, subsequent generation will create work products on the new components that can have such work products. In this way, these work products will maintain consistency with the model.

For further information on work products, please refer to chapter 6, "Functions of the *Objectteering/UML Test Designer* module", of this user guide.

Tagged values, notes and stereotypes

Objecteering/UML is a multi-target workshop used to model a large quantity of application elements, whatever the language used.

However, during the technical designing and programming phases, certain implementation details expressed in the target language have to be specified and added to the model. This information, used to complete the model before generation, is entered using:

- ◆ *Tagged values*, which provide implementation rules for the generator
- ◆ *Notes*, which correspond to the zones inserted directly into the generated code
- ◆ *Stereotypes*, which provide implementation rules for the generator, different to those of a non-stereotyped element

Tagged values, stereotypes and notes specific to *UML Test Designer* can be added to a model element only if the *Objecteering/UML Test Designer* module has been selected. For further information on tagged values, notes and stereotypes, please refer to the related sections in chapter 6 of this user guide.

Consistency checks

Objectteering/UML consistency checks

The Objectteering/UML CASE tool provides over 200 consistency checks in real time, used to guarantee the quality and coherence of the model produced. The advantages of these consistency checks are clear:

- ◆ Model consistency is checked when elements are entered, thus ensuring that inconsistent names or elements are not entered.
- ◆ Model consistency is maintained when elements are modified, thus allowing the user to avoid having to manually update all instances of the modified element.

For further information on consistency checks, please refer to the "*Consistency checks*" section in chapter 1 of the *Objectteering/UML Modeler* user guide.

Removable consistency checks

However, the user may, in some modeling situations, prefer to have a certain degree of flexibility with regard to the consistency checks applied to his model. This can be the case, for example, during the preliminary phases of a project (analysis, specifications, and so on), when users can prefer to have freer, less restrictive use of the CASE tool. Similarly, when importing models from other CASE tools that do not necessarily employ the same level of consistency checks as Objectteering/UML, it can also be useful to be able to deactivate certain checks. For this reason, certain Objectteering/UML consistency checks are removable. For further information on removable consistency checks, please refer to the "*Removable consistency checks*" section in chapter 3 of the *Objectteering/UML Modeler* user guide.

The "UML Test Designer/Check" command

The "*UML Test Designer/Check*" command checks that:

- ◆ modifications made to the application model have been taken into account in the test model. If this is not the case, a call to the "*UML Test Designer/Update*" command is needed.
- ◆ the test model contains all the information required to execute a test suite correctly.

Chapter 6: Functions of the
Objecteering/UML Test
Designer module

Overview of Objecteering/UML Test Designer functions

Introduction

The *Objecteering/UML Test Designer* module provides the following functions to create and complete a test model:

- ◆ *Create a test suite*: This command is used from a package of your application model to select classes to test and classes to stub, in order to create a test suite package.
- ◆ *Update a test suite*: This command is used from a tested package to update one of its test suites after a modification.
- ◆ *Create a context*: This command is used from a test suite package to add a context package and its elements (instances, instance diagram, etc.).
- ◆ *Create a test sequence*: This command is used from a context package to add a sequence diagram to define a test.
- ◆ *Check*: This command is used from the test model to check consistency between the test suites and the application package.

The *Objecteering/Tests For Java/EJB* module provides the following functions, to generate code and execute tests with JUnit TestRunner:

- ◆ Generate tests
- ◆ Visualize tests
- ◆ Edit tests
- ◆ Generate tests makefile
- ◆ Compile tests
- ◆ Analyze the compilation
- ◆ Launch tests
- ◆ Generate and compile tests

Using tagged values, notes and stereotypes

Tagged values, notes and stereotypes are used in the test model to define the type of each element and provide useful information. They are described in the following sections of this chapter.

Tagged value types

The tagged values used by *Objecteering/UML Test Designer* are all generated by one of the commands, and cannot be modified by the user.

For example, the *{TestedPackage}* tagged value on a test suite package provides an identification of the package tested by the test suite.

Note types

Overview

Objecteering/UML note types are used to complete the UML test model with texts in *UML Test Designer* syntax.

Note types on a test suite package

The ... note type	is used to ...
summary	provide a summary of the package contents for documentation purposes.
description	provide a description of the package contents for documentation purposes.
testCase	insert the Java code of a test method.

Note types on a context package

The ... note type	is used to ...
setUp	instantiate the context before calling a test method. The contents of this note are inserted into a method called before each test method of the context.

Only one "setUp" note is allowed on a test package.

Stereotypes

Overview

Objectteering/UML Test Designer defines stereotypes, used to designate certain objects as being concerned by the generation of *Tests For Java* code.

The stereotypes used in *Objectteering/UML Test Designer* are generated automatically. However, it may be useful to know what they are used for.

Stereotypes on a package

The ... stereotype	is applied to ...
OrganizationTestSuite	a test suite package.
PackageStub	a package containing stubs.
TestContext	a context package.

Stereotypes on a class

The ... stereotype	is applied to ...
ClassUnderTest	a tested class in a test suite package.
ClassifierStub	a stubbed class in a test suite package.

Chapter 7: Customizing the
Objecteering/UML Test
Designer module

Defining module parameters

Introduction


The user can parameterize the following elements of *Objectteering/UML Test Designer*:

- ◆ Primitive tests
- ◆ Parameter values for tests
- ◆ Parameter default values for tests
- ◆ General
- ◆ Editors
- ◆ RTF generation

The following elements of Objectteering/Tests For Java/EJB can be parameterized by the user:

- ◆ JUnit
- ◆ Test report

Configuring the modules

The *Objectteering/UML Test Designer* and *Objectteering/Tests For Java* modules can be parameterized through the "Modifying configuration" window (as shown in Figures 7-1 and 7-2 respectively). This window is opened either by clicking on the  "Modify module parameter configuration" icon or through the "Tools/Modify configuration" menu.

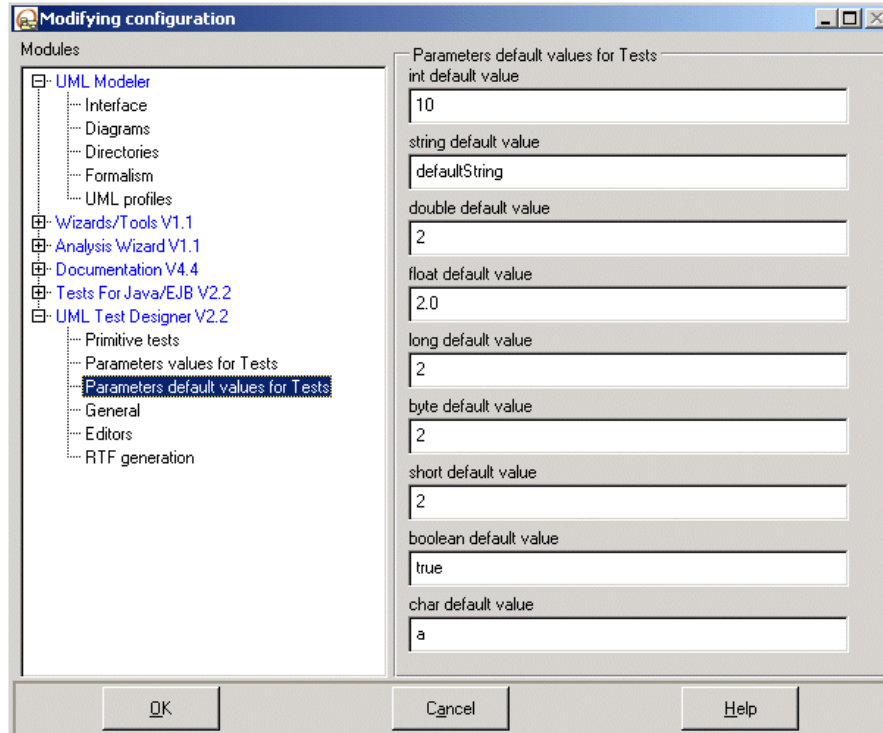


Figure 7-1. The "UML Test Designer" section of the "Modifying configuration" dialog box

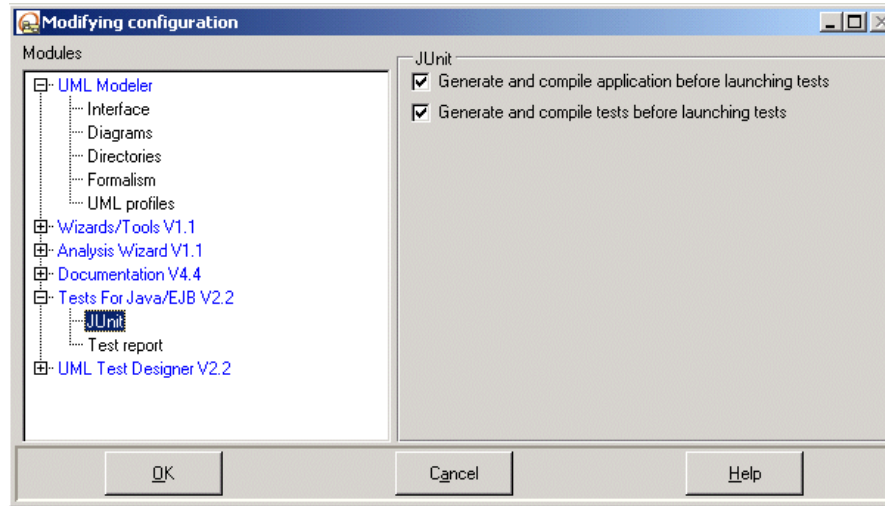


Figure 7-2. The "Tests For Java/EJB" section of the "Modifying configuration" dialog box

Sets of parameters

In the *UML Test Designer* sub-category of parameter, six sets of parameters are available.

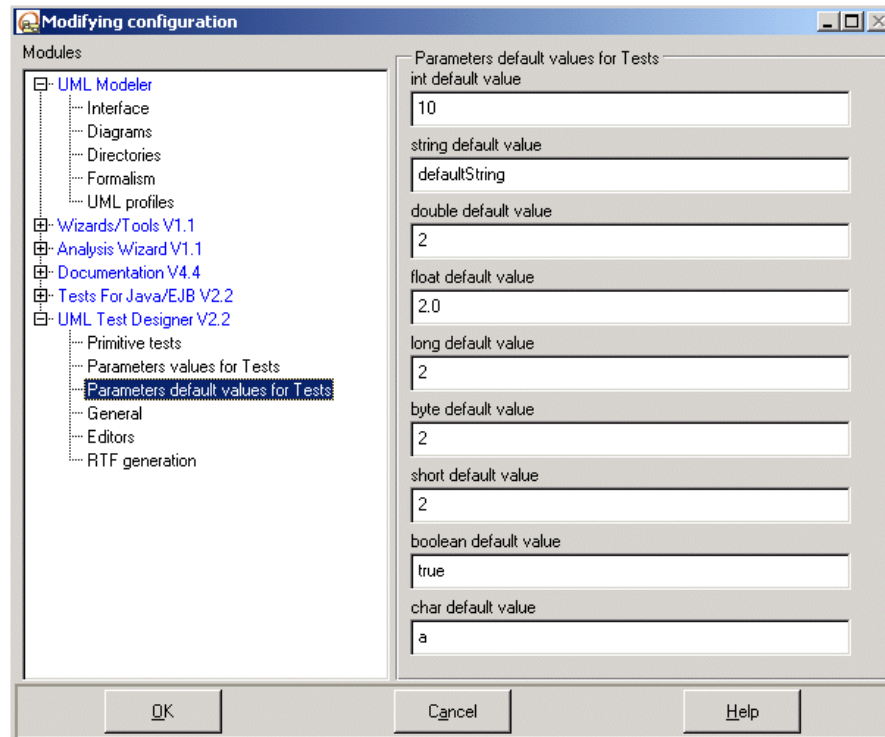


Figure 7-3. The "Parameter default values for tests" set of *UML Test Designer* parameters

In the *Tests For Java/EJB* sub-category of parameters, two sets of parameters are available.

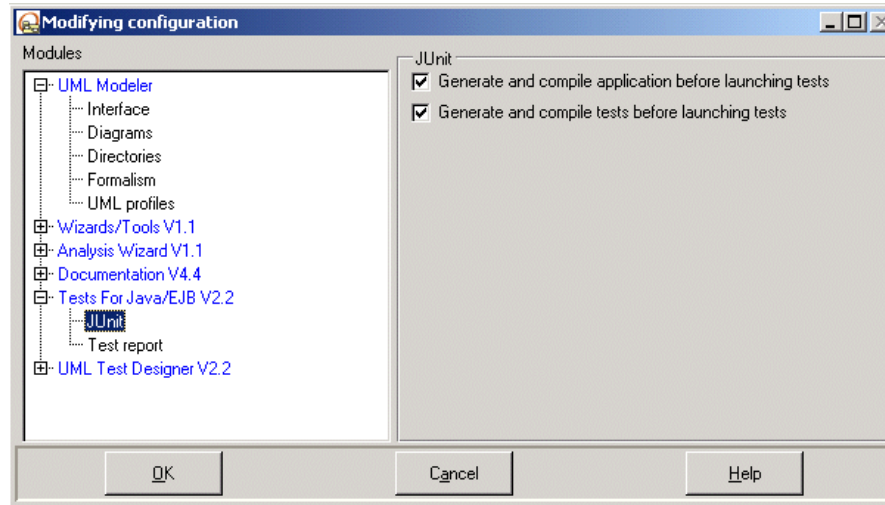


Figure 7-4. The "JUnit" set of UML Test Designer parameters

Chapter 8: UML Test Designer
documentation generation

Overview of UML Test Designer documentation generation

Two kinds of documentation can be generated on your tests:

- ◆ documentation of your test model
- ◆ the test report after execution of the tests with JUnit (only available with the *Tests For Java/EJB* module)


First, create a new document by selecting a test package (either the root test package or a test suite package in the Objectteering/UML explorer) and then clicking on the  "Create a document" icon in the "Items" tab of the properties editor (as shown in Figure 8-1).



Figure 8-1. Creating a new document

Chapter 8: UML Test Designer documentation generation

A window then appears. In this window, select the *UML Test Designer* module (Figure 8-2).

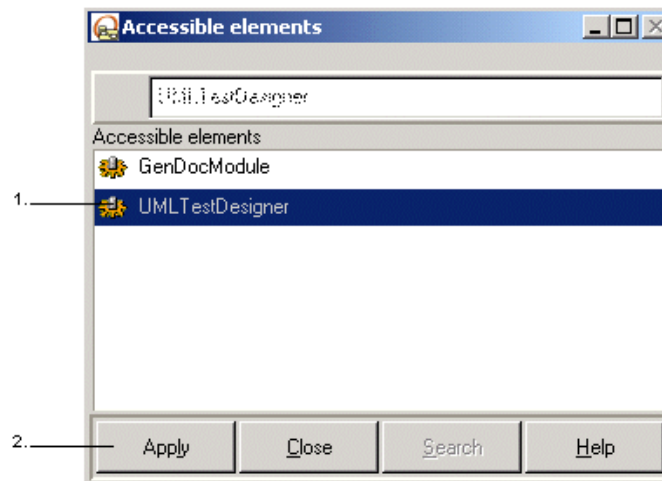


Figure 8-2. Selecting the *Objectteering/UML Test Designer* module

Steps:

- 1 - Click on the *UMLTestDesigner* module.
- 2 - Click on "Apply" to confirm your choice.

The "Documentation generation" window then appears (Figure 8-3).

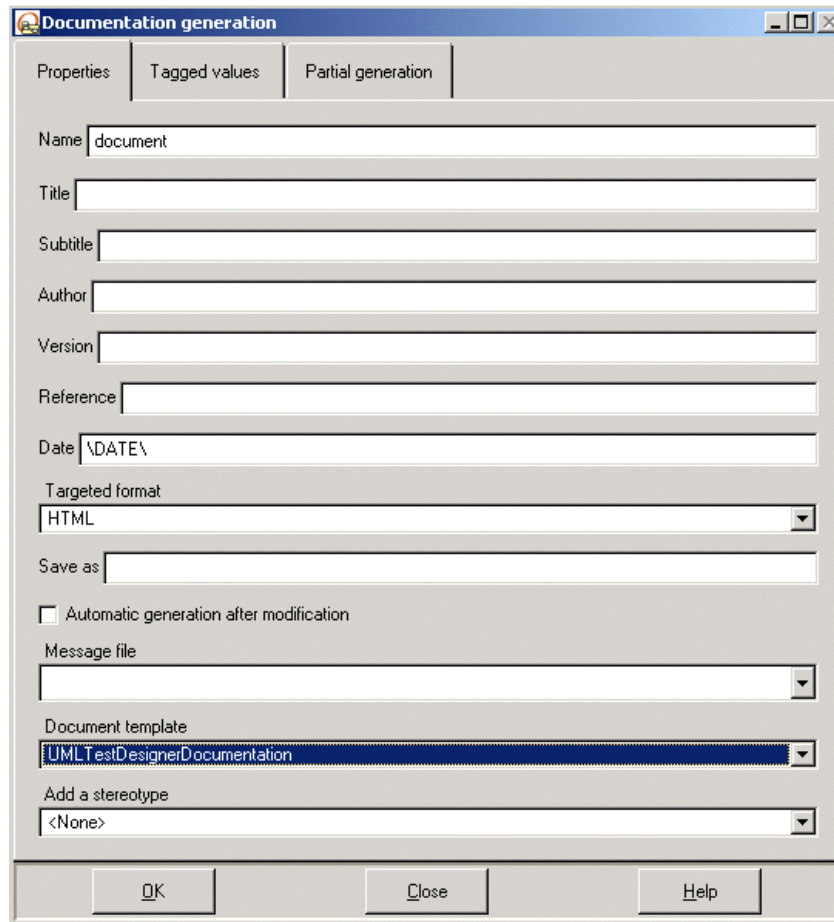


Figure 8-3. Creating documentation

Generating the test report of the executed tests

Once tests have been executed once, a test report is available. This test report is in an XML file available in the same location as your test model documentation or in the default directory if the test model documentation has not already been generated.

The XSL file associated with the XML file has to be specified at *Tests For Java/EJB* module configuration level.

Two kinds of XSL file are available in your Objecteering/UML module directory (Figure 8-4).

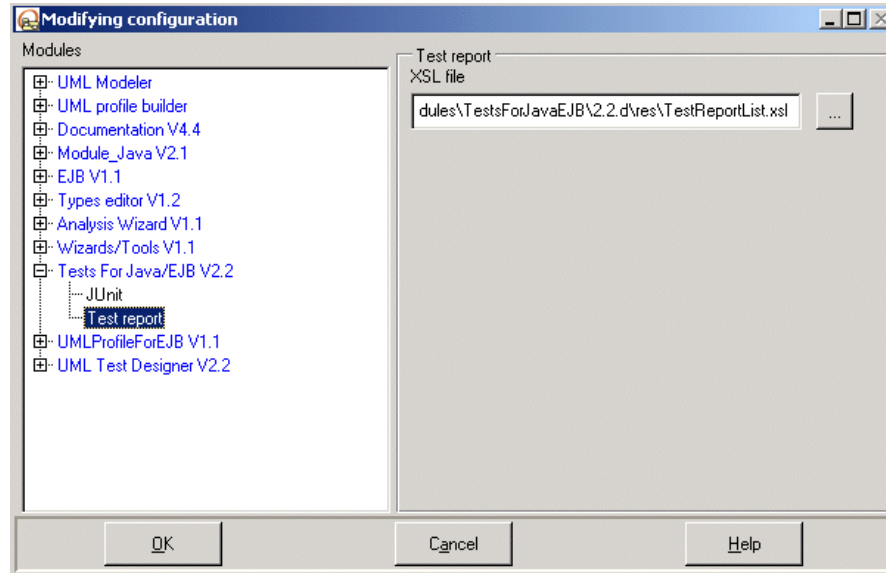


Figure 8-4. Test report module parameters

Functions of UML Test Designer documentation generation

Once the documentation generation work product has been created, you can generate and visualize documentation corresponding to your tests.

The "UML Test Designer/Generate test documentation" command

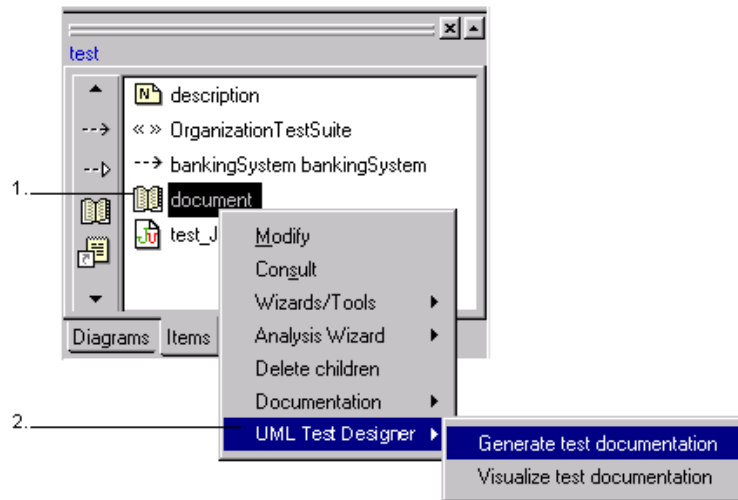


Figure 8-5. The "UML Test Designer/Generate test documentation" command

Steps:

- 1 - Select the documentation work product in the "Items" tab of the properties editor.
- 2 - Run the "UML Test Designer/Generate test documentation" command.

The "UML Test Designer/Visualize test documentation" command

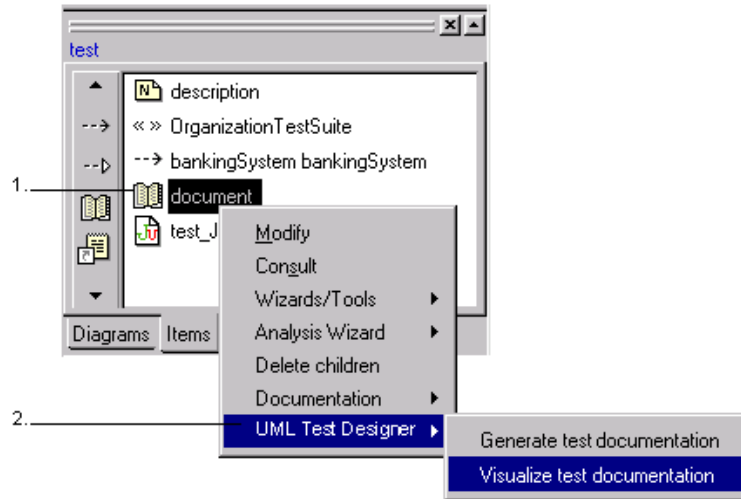


Figure 8-6. The "UML Test Designer/Visualize test documentation" command

Steps:

- 1 - Select the documentation work product in the "Items" tab of the properties editor.
 - 2 - Run the "UML Test Designer/Visualize test documentation" command.
-

Index

- "assert" note 3-23, 4-23, 4-26
- "description" note 6-5
- "setUp" method 4-22
- "setUp" note 1-18, 4-21, 6-5
- "summary" note 6-5
- "tearDown" method 4-22
- "tearDown" note 4-22
- "testCase" note 1-17, 3-25, 4-19, 6-5
- {TestedPackage} tagged value 1-17, 6-4
- <<assert>> stereotype 4-23
- <<ClassifierStub>> stereotype 6-6
- <<ClassUnderTest>> stereotype 6-6
- <<OrganizationTestSuite>> stereotype 6-6
- <<PackageStub>> stereotype 6-6
- <<testCase>> stereotype 4-19
- <<TestContext>> stereotype 6-6
- Adding a new context to a test suite 4-12
- Adding a new test sequence to a context 4-15
- Annotating the model 5-5
- Assert note 1-18, 3-18
- Class under test 1-18
- Command for compiling 3-29
- Command for generating code 3-29
- Command for generating documentation 3-29
- Command for launching JUnit 3-29
- Complete model-driven generation 1-4
- Configuring modules 7-4
- Consistency 1-4, 1-7
- Consistency checks 5-6
- Context package 1-18, 3-16, 3-41
- Contexts 3-25
- Creating a document work product 8-3
- Creating a test model 1-8
- Creating a UML modeling project 2-3
- Creating test suites 3-10
- Customizing the Tests For Java generator 5-3
- Defining a Setup() called before each test method 4-21
- Defining documents 5-3
- Defining items 5-3
- Defining rules 5-3
- Defining tagged values 5-3
- Defining test cases 1-3
- Defining test suites 1-3
- Developing applications
 - Creating a UML test model from your UML model 1-17
 - Defining tests in the generated test model 1-17
 - Generating and compiling tests 1-17
 - Launching the JUnit test runner 1-17
- Document work product 8-3, 8-7
- Documentation generation window 8-5
- Editing generated code 3-43
- Execution with JUnit 1-14
- FAQ
 - Adding a new context to a test suite 4-12
 - Adding a new test sequence to a context 4-15
 - Defining a Setup() called before each test method 4-21

- Generating and compiling a new context 4-31
- Generating and compiling a new test suite 4-30
- Generating tests for functions with primitive parameters 4-8
- Getting a returned value with an assert note 4-26
- Inserting an assertion into the test 4-28
- Inserting target code into the test 4-29
- Testing a message with an assert note 4-23
- Testing abstract class implementations 4-3
- Testing an EJB 4-32
- Testing interface implementations 4-5
- Writing a test method in Java code 4-19
- First Steps
 - Compiling 3-3
 - Creating a Tests For Java generation work product 3-3
 - Creating a Tests For Java/EJB generation work product 3-30
 - Creating a UML Test Designer test suite 3-3
 - Creating an instance 3-21
 - Creating new contexts 3-25
 - Creating new sequence diagrams 3-25
 - Creating new test case notes 3-25
 - Defining Java test methods 3-25
 - Defining stubbed classes 3-17
 - Defining test methods with sequence diagrams on the context package 3-18
 - Defining tests in the test suite 3-3
 - Description of the newly created test suite 3-15
 - Generating Tests For Java code 3-3, 3-35
 - Generation work product commands 3-39
 - Initializing the First Steps project 3-3
 - Maintaining your test model 3-3, 3-26
 - Naming your test suite 3-12
 - Running tests with JUnit 3-3
 - Running the "UML Test Designer/Create a Test Suite" command 3-11
 - Running the test in the sequence diagram 3-23
 - Selecting classes to be stubbed 3-14
 - Selecting classes to be tested 3-13
 - Sources 3-4
 - Tests For Java generation work product menus 3-3
 - Visualizing generated Tests For Java code 3-3
- Functions of the Objectteering/UML Test Designer module 1-4
 - Defining tests inside a test model 1-4
 - Generating a test model 1-4
 - Generating Java code to be executed 1-4
 - Selecting classes to be stubbed 1-4
 - Selecting classes to be tested 1-4
- General first steps 1-3, 3-3
- Generated Tests For Java/EJB code 3-35

- Generating a test model 1-3
- Generating and compiling a new context 4-31
- Generating documentation 8-3, 8-7
- Generating test documentation 1-16
- Generating tests for functions with primitive parameters 4-8
- Generating the test report of the executed tests 8-6
- Generating, compiling and executing tests 1-3
- Generation work product 3-29
- Getting a returned value with an assert note 4-26
- if () then {} else 3-20
- Initializing the First Steps project
 - Configuring modules 3-7
 - Configuring the Objecteering/Tests For Java/EJB module 3-9
 - Configuring the Objecteering/UML Test Designer module 3-8
 - Creating a UML modeling project 3-5
 - Importing a model into the First Steps project 3-5
 - Selecting modules 3-6
 - Selecting the UML Test Designer and Tests For Java modules 3-5
- Inserting an assertion into the test 4-28
- Inserting target code into the test 4-29
- Integrating JUnit into Objecteering/UML 1-7
- J methods 5-3
- Java generation work product 3-31, 4-30
- Java generation work product window 3-31
- Java notes 1-17
- JUnit assertion 4-19
- JUnit framework 1-6
- JUnit test runner 1-15
- JUnit TestCase 3-35
- JUnit TestRunner 3-41
- JUnit toolkit 2-3
- JUnit window 3-30
- JUnit work product 4-30
- loop 3-19
- Model consistency 5-6
- Module configuration window 3-7
- Note types on a context package 6-5
- Note types on a test suite package 6-5
- Notes 1-17, 5-5, 6-3
 - "assert" 3-23, 4-23, 4-26
 - "description" 6-5
 - "setUp" 1-18, 4-21, 6-5
 - "summary" 6-5
 - "tearDown" 4-22
 - "testCase" 1-17, 3-25, 6-5
- Objecteering/Administrating Objecteering Sites 1-3
- Objecteering/Introduction 2-3, 3-3, 5-3
- Objecteering/Java 1-3
- Objecteering/UML license 1-3, 2-3
- Objecteering/UML Modeler 1-3, 2-3, 5-6
- Objecteering/UML Profile Builder 5-3
- OBJING_PATH environment variable 2-3
- Organization test suite 1-18
- Parameterizing the Objecteering/UML Test Designer module 5-3

- Prerequisites for use of the Objecteering/UML Test Designer module 2-3
- Propagate command 4-30, 4-31
- Properties editor 3-30, 5-4, 8-3, 8-7
 - Items tab 3-30, 5-4, 8-3, 8-7
- Redefining J methods 5-3
- Removable consistency checks 5-6
- Selecting classes to be stubbed 1-10
- Selecting classes to be tested 1-9
- Selecting modules 3-6
- Selecting the module for the current UML modeling project 2-3
- Selecting the Objecteering/UML Test Designer module 5-3
- Sequence diagrams 1-17, 3-25
- Stereotypes 5-5, 6-3
 - <<ClassifierStub>> 6-6
 - <<ClassUnderTest>> 6-6
 - <<OrganizationTestSuite>> 6-6
 - <<PackageStub>> 6-6
 - <<TestContext>> 6-6
- Stereotypes on a class 6-6
- Stereotypes on a package 6-6
- Stub packages 1-11
- Stubbable classes 3-14
- Stubbed class 1-18, 3-17
- Tagged values 1-4, 1-17, 5-3, 5-5, 6-3
 - {TestedPackage} tagged value 1-17, 6-4
- Test 1-18
- Test case 1-18
- Test case note 1-18
- Test case notes 3-25
- Test definition
 - Overview 1-8
- Test documentation 1-4
- Test methods 3-35
 - "testxxx()" 3-35
- Test packages 3-10, 3-41
- Test report 1-4
- Test suite 1-18
- Test suite packages 3-15, 3-41
- Testable classes 3-13
- Tested package 1-18
- Testing a message with an assert note 4-23
- Testing abstract class implementations 4-3
- Testing an EJB 4-32
- Testing interface implementations 4-5
- Tests For Java commands
 - Analyze the compilation 3-33
 - Generate and compile 3-33, 4-30, 4-31
- Tests For Java generation work product 3-29, 3-35
- Tests For Java module parameters 7-3
- Tests For Java//EJB commands
 - Generate tests 3-41
- Tests For Java/EJB commands 3-29, 3-39
 - Analyze the compilation 3-41
 - Compile tests 3-41
 - Edit tests 3-41
 - Generate 3-35
 - Generate and compile tests 3-41
 - Generate tests makefile 3-41
 - Launch tests 3-41
 - Visualize tests 3-41
 - Visualize tests makefile 3-41
- Tests For Java/EJB functions 6-3

- Analyzing the compilation 6-3
- Compiling tests 6-3
- Editing tests 6-3
- Generating and compiling tests 6-3
- Generating tests 6-3
- Generating the tests makefile 6-3
- Launching tests 6-3
- Visualizing tests 6-3
- Tests For Java/EJB generation work product 5-4
- Tests For Java/EJB module parameters
 - JUnit 7-3
 - Test report 7-3
- UML Test Designer commands 3-11
 - Check 3-27, 5-6
 - Create a context 4-13, 4-31
 - Create a test sequence 4-16
 - Create a test suite 3-11, 4-30
 - Generate test documentation 8-7
 - Update 5-6
 - Update a test suite 3-26, 3-27
 - Visualize test documentation 8-8
- UML Test Designer first steps 1-3
- UML Test Designer functions 6-3
 - Checking 6-3
 - Creating a context 6-3
 - Creating a test sequence 6-3
 - Creating a test suite 6-3
 - Updating a test suite 6-3
- UML Test Designer module parameters 7-3
 - Editors 7-3
 - General 7-3
 - Parameter default values for tests 7-3
 - Parameter values for tests 7-3
 - Primitive tests 7-3
 - RTF generation 7-3
- Updating your test model 3-26
- Using tagged values, notes and stereotypes 6-3
- Using the Objectteering/UML Test Designer module 2-3
- Visualizing documentation 8-7
- Visualizing generated Tests For Java/EJB code 3-42
- Writing a test method in Java code 4-19