

JUnit Functional Specification Document - Promotion F

June 16, 2005

Contents

1 Overview	3
2 Test code specification	4
2.1 Primary test-classes	4
2.1.1 Name	4
2.1.2 Base class	4
2.1.3 Contents	4
2.1.4 Method order	4
2.1.5 File	4
2.2 Primary test-cases	4
2.2.1 Name	4
2.2.2 Signature	5
2.2.3 Test-Suite	5
3 Editor	6
3.1 Creational Shortcuts	6
3.1.1 Create test-case for current method	6
3.1.2 Create/update test-class for current class	7
3.2 Navigational Shortcuts	9
3.2.1 Goto test-case	9
3.2.2 Open test-class	9
3.2.3 Goto source method	10
3.3 Compilation & running	10
3.3.1 Run tests for the current source class	10
3.3.2 Run all tests (in the background)	11
4 Projects View (contextual menu)	12
4.1 Creation	12
4.1.1 Create/Update tests for existing sources	12
4.1.2 Create/Update test-suite	12
4.1.3 Open Test Class	13
4.2 Running	13
4.2.1 Run tests for the selected source class	13

5	Main Menu	15
5.1	Creation	15
5.1.1	New Test for Existing class	15
5.1.2	New empty test-class	15
5.1.3	New test-suite	15
5.2	Running	16
5.3	Editor Actions	16
6	Test results panel	17
6.1	Overview	17
6.2	JUnit console output	17
6.3	Integration	18
6.4	Requirements	18
7	Dialog Specifications	19
7.1	New test-class	19
7.2	New test-classes	20
7.3	New empty test-class	21
7.4	New test-suite	21
7.5	Edit test-suite	22

1 Overview

This document specifies the functional properties of JUnit support in Promotion-F of the NetBeans IDE. It is based on several other documents:

- Proposed new features:
<http://junit.netbeans.org/proposed-new-features.html>
An older document comparing JUnit support in various IDEs and proposing some new features for NB. Never realized.
- Development directions:
http://junit.netbeans.org/junit_dev_directions.html
This document is the logical continuation of the *JUnit - proposed new features* document. It outlines future development direction and proposes new features that are being considered for mid-term implementation.
- Use-case specification:
<http://junit.netbeans.org/promof-usecases.html>
The document analysis the main missing use-cases for a JUnit support in NetBeans.
- Design-space analysis:
http://junit.netbeans.org/JUnit_concept_promof.sxw
This document summarizes the high-level concept for JUnit support in NetBeans IDE, starting Promotion F. It analyses the way tests are (or should be) worked with and proposes methods to support it.

To ensure consistency, this document contains also features already to be found in the current version. However, the present descriptions may differ from the existing ones in subtle details.

The document begins with Section 2 containing the definition of the minimal properties all test-classes and test-methods generated and handled by the module must conform to.

In the following sections, the module's integration into the editor (Section 3), projects view (Section 4) and the main menu (Section 5) is specified. Each of the features is assigned a priority from the range "must have", "should have", "nice to have". Features that are, in some form, already present in the current version are marked as "existing feature". In addition, for each of the features the matching use-case from the `promof-usecases.html` document is indicated.

Section 7 gives a functional specification of dialogs referred to in the rest of the text.

2 Test code specification

This section specifies the minimal properties all test-classes and test-methods must conform to. The specification intentionally leaves many details unspecified in order to account for manual changes in the generated code. The junit support must work correctly with all sources conforming to this specification and account for all possibilities within its bounds.

2.1 Primary test-classes

2.1.1 Name

For a class named C, the name of the corresponding primary test class is C+”Test”. The test class is located in the same package as the source class. For nested classes, the primary test class is located in the corresponding primary test class of its immediate container.

Example: the name of the primary test-class of the source class
”org.netbeans.modules.TestUtil.Data” is
”org.netbeans.modules.TestUtilTest.DataTest”.

2.1.2 Base class

The test-class extends `junit.framework.TestCase`, directly or indirectly.

2.1.3 Contents

The test class contains a method `public static TestSuite suite()` returning a new `TestSuite`

2.1.4 Method order

The order of methods (test-cases) within test-classes is unspecified.

2.1.5 File

The class is defined in a file that has name derived from the source class file by inserting ”Test” just in front of ”.java”. For example, a primary test-class of a source class defined in ”Source.java” is located in ”SourceTest.java”.

2.2 Primary test-cases

2.2.1 Name

For a source method named M, the name of the corresponding primary test case is ”test” + `upcaseFirst(M)`, where `upcaseFirst` up-cases the first character of its argument. The test-case is located in the primary test class of the container of the source method.

Example: the primary test-case of the source method ”org.netbeans.A.B.method”, where ”A” and ”B” are class names, ”method” is a method name, is ”org.netbeans.ATest.BTest.testMethod”

2.2.2 Signature

The method is `public void` with no arguments and the `throws` declaration containing every exception declared to be thrown by the source method.

2.2.3 Test-Suite

Test-suites follow the specification of test-cases.

3 Editor

3.1 Creational Shortcuts

3.1.1 Create test-case for current method

Priority Should have

Matching use-case I a. Creating a test-case for a method

Summary While editing some method within a source class, the user presses a keyboard shortcut. The corresponding primary test class is opened, a new primary test-case for the current method is created within it, and cursor placed in it.

Body If the method doesn't exist, an empty skeleton of the test-case is generated from the method's declaration. Given a declaration of a non-static source method

```
retType m(type1 arg1, ..., typen argn) throws ex1, ..., exm,
```

the skeleton is generated by the following grammar from the `TestCaseBody` symbol.

```
TestCaseBody ::= InitBlock MethodCall Assertion
InitBlock    ::= type1 arg1 = default ValueType1;
               ...
               typen argn = default ValueTypen;
               clsName instance = default Instance Value;
               [retType expResult = default ValueRetType];retType≠void
MethodCall   ::= [retType result =];retType≠void instance.m (arg1, ..., argn);
Assertion    ::= [assertEquals(expResult, result)]retType≠void
```

For static methods, the generated skeleton is different in that the `instance` variable is not used:

```
TestCaseBody ::= InitBlock MethodCall Assertion
InitBlock    ::= type1 arg1 = default ValueType1;
               ...
               typen argn = default ValueTypen;
               [retType expResult = default ValueRetType];retType≠void
MethodCall   ::= [retType result =];retType≠void clsName.m (arg1, ..., argn);
Assertion    ::= [assertEquals(expResult, result)]retType≠void
```

The default argument values are 0 (appropriately typed) for numeric types, `true` for `boolean`, ' ' (the space character) for characters, "" (an empty string) for `java.lang.String` and `null` for other reference types.

In tests for non-static methods, the instance value is initialized by a call of public or package-private no-argument constructor, if any, or with `null` if no such constructor exists in the tested class.

For readability reasons, the skeleton uses argument names declared in the source for local variables holding their values in the test. However, if the source

method declaration doesn't specify all argument names or some of the arguments are named "instance", "result" or "expResult", other unique names must be used for the local variables. A sequence such as `arg1, arg2, ...` with the colliding indexes skipped should be fine. The constraint is that `arg1, ..., argn, "instance", "result", "expResult"` are distinct.

The block of code computing the expected result, the actual result and comparing the results is separated with an empty line.

Example:

The skeleton generated for source method

```
public boolean length(String s, int l)
```

in class `MySampleClass` with a public no-argument constructor would be

```
String s = "";
int l = 0;
MySampleClass instance = new MySampleClass();

boolean expResult = true;
boolean result = instance.length(s, l);
assertEquals(expResult, result);
```

The generation of bodies can be disabled by `....TODO-HOW....`. In that case, the body is:

```
fail("The test case is empty.");
```

Source Comments The following two features can be switched on/off by an option:

- i) a JavaDoc comment, something like "Test of ... method, of class ..." is generated above the method.
- ii) the generated source is commented with text hints that suggest further modifications of the skeleton.

Exceptions If the source method declares throwing of at least one uncaught exception (i.e. exception that is neither `java.lang.RuntimeException` nor its subclass), the generated test method will declare throwing of `java.lang.Exception`. This will ensure that the test method is compilable and that any uncaught exceptions are propagated to the JUnit framework which will report them as errors.

Other Details

- i) If the test-class exists, it is not created, just opened and brought to top
- ii) If the test-method exists, the cursor is placed into it (first line). It is not modified.
- iii) If the cursor is not within any method in the source, nothing happens

3.1.2 Create/update test-class for current class

Priority Must have

Matching use-case I b. Creating a test-class for a class

Summary While editing some source class, the user presses a keyboard short-cut to create a new test-class for the current class. A dialog for selection of the contents of the test-class is displayed. After the dialog is finished, the corresponding primary test-class is created, opened and brought to top.

Detailed dialog behavior The dialog behaves as the New test-class dialog (7.1) invoked on the source file of the current class and current class. See also section *Nested classes* below for more details. The corresponding test-class is created if it doesn't exist and filled according to the dialog output after the dialog finishes. See *Results* for details.

Results After the dialog finishes, its output, a `NewTestClassModel` (see 7.1), is processed as follows.

- i) A corresponding primary test-file is opened (or created if it doesn't exist) (see 2.1.5). The test-file is created in the source-root specified `TestClassModel.TestClass.SourceRootID`.
- ii) The test-file is initialized with a standard comment header and a package statement. It is filled with nested classes and methods that satisfy the following properties with respect to `TestClassModel.Tests.ClassTree`.
 - (a) It contains a primary test-class of every class contained in `Tests.ClassTree.Class`. Such primary test-classes have `setUp` and `tearDown` methods if `NewTestClassModel.ClassParams.SetUp` or `NewTestClassModel.ClassParams.TearDown`, respectively, are `true`. For new test-classes, these methods are empty.
 - (b) It contains a primary test-case of every test method contained in `NewTestClassModel.Tests...MethodS.MethodID`. If the test-cases didn't exist before, they are created as defined by the Create test-case action (3.1.1). The parameter `Bodies` in `TestClassModel.ClassParams` specifies if the default test-case body is generated in the test-case. See section 3.1.1 for description of these options.
 - (c) All test-cases present in the test-sources but not present in the `NewTestClassModel.Tests` tree are removed from the test-file.

The created class is opened in the editor and its node selected in the projects view (if possible).

Method order If the class was originally empty, the situation is simple. Methods generated in the test-class are sorted according to their order in the source class. However, users can reorder the methods or even add their own among the generated ones and thus the problem of adding new methods to an existing source is not simple. In general, one wants the following to hold:

- i) User changes are preserved – both in terms of order and contents. That is, if the user reorders their test-cases and adds some new ones among them, the changes are preserved by any automatic generation action.
- ii) The order is deterministic. That is, it is not acceptable that the same action on the same source produces two different results at different times.
- iii) The order is natural (in a sense) and predictable. This most natural order of test-cases is that of the corresponding source methods. However, in the presence of user edits and the previous requirements, the order cannot be always preserved. Nevertheless, it should be preserved as closely as possible. More formally, the generated ordering should have the minimal number of inversions among all orderings consistent with the above constraints.

Nested classes If the cursor is within some nested class, all necessary container classes are created up to the root of the source file. The container classes are created minimal by default, the dialog should allow to override the default selection.

3.2 Navigational Shortcuts

3.2.1 Goto test-case

Priority Must have

Matching use-case II a. Change of source, update of test

Summary While editing some method within a source class, the user presses a keyboard shortcut. The corresponding primary test class is opened and the cursor placed within the primary test-case for the current method.

Details

- i) If the test-class or test-case don't exist, a message offering their creation is shown. If the user chooses to create the test-case, the action behaves as the Create test-case for current method action (3.1.1). If the user chooses not to create the test-case, nothing happens.
- ii) If the test-class is already opened, it is brought to top and cursor placed into the corresponding primary test-case.
- iii) The shortcut of this action is the same as that of the Goto source method action (3.2.3). As the situations in which these two actions are valid don't overlap and the functionality is dual, this is a valid requirement.
- iv) If multiple corresponding test-classes with the corresponding test-method exist for the source method, in multiple test-roots necessarily, the first such a test-method is opened.

If multiple corresponding test-classes in multiple test-roots exist, but only one of them contains the corresponding test-case, the test-class with the test-case is opened as if the other ones didn't exist.

3.2.2 Open test-class

Priority Must have

Matching use-case II a. Change of source, update of test

Summary While editing some source class, the user presses a keyboard shortcut. The corresponding primary test class is opened and brought to top.

Details

- i) If the test class doesn't exist, a dialog message offering its creation is shown. If the user chooses to create the test-class, the action behaves as the Create/update test-class for current class action (3.1.2).
- ii) If the test class is opened, it is brought to top.
- iii) If multiple corresponding test-classes exist for the source class in multiple source roots, a dialog with choice of test-roots is displayed and the action continues with the chosen one.

3.2.3 Goto source method

Priority Must have

Matching use-case II b. Writing tests, checking code

Summary While editing some primary test-case, the user presses a keyboard shortcut. The corresponding source-class is opened and cursor placed into the corresponding source method.

Details

- i) If the source class doesn't exist or the test-case is not primary, nothing happens
- ii) If the test-case is a primary test-case but the corresponding source method doesn't exist, the source class is opened and brought to top.
- iii) The shortcut of this action is the same as that of the Goto test-case action (3.2.1). As the situations in which these two actions are valid don't overlap and the functionality is dual, this is a valid requirement.
- iv) If multiple corresponding source-classes with the source-method exist for the test-method in multiple source roots, the first such a source-method is opened.

If multiple corresponding source-classes exist but only one of them contains the corresponding source method, it is opened as if the other source-classes didn't exist.

3.3 Compilation & running

3.3.1 Run tests for the current source class

Priority Must have

Matching use-case III a. Change of source, update of test

Summary While editing some source class, the user presses a keyboard shortcut. The test corresponding to the source class is run.

Details

- i) If the corresponding test-class doesn't exist, a message box is displayed offering to create a new test-class.
- ii) When the tests are run, the test result window (pane) is shown.
- iii) If multiple test-classes for the source-class exist in multiple test-roots, an informational message with choice of test-roots is displayed.

3.3.2 Run all tests (in the background)

Priority Must have

Matching use-case III a. Changing source, running test-cases

Summary While anywhere in the editor, the user presses a shortcut. All tests in the project are run.

Details

- i) When the tests are run, the test result window (pane) is shown to collect all results.
- ii) The tests are run in the background, that is, it is possible to continue working while the tests are running.

4 Projects View (contextual menu)

4.1 Creation

4.1.1 Create/Update tests for existing sources

Priority Must have, existing feature.

Summary The user selects several source class nodes in the projects view. Using the contextual menu or shortcut an action is invoked to create tests for the selected sources. A dialog appears for specification of the details.

Dialog specification If the selection contains exactly one testable class, the dialog behaves as the New test-class dialog (7.1). If the selection contains more than one testable classes, the dialog behaves as the New test-classes dialog (7.2). If there is no testable class in the selection, an appropriate information message appears.

Results The generated results of the action are the same as of the Create/update test-class for current class action (3.1.2) invoked on each class. Moreover, if specified so by the dialog, test-suites are generated for each collection of newly generated tests belonging to the same java package.

Other Details

- i) The action is enabled if and only if the selection contains just source files or source classes. The action skips all non-testable classes.
- ii) Operation on source files is translated to operation on all contained source classes.
- iii) The action works only for source nodes from a single source-root. If sources from multiple source-roots are selected, an informational message appears informing the user that “Sources from multiple source-roots were selected but it is not supported to create test for sources from multiple source-roots” (or similar) and the action terminates.
- iv) If the creation results in a single class, the class is opened in the editor. All created classes should become selected in the projects view (if possible).
- v) The action works for source files from different source-roots normally. All tests are created to the same test-root of choice (specified in the dialog).

4.1.2 Create/Update test-suite

Priority Must have, existing feature.

Summary The user selects several test nodes in the projects view. Using the contextual menu or shortcut the user invokes an action to create a new test-suite for the selected tests. A dialog appears for specification of the details.

Results One test-suite is created/updated, even if the test-cases from multiple packages are selected. The created test-suite is put to the root of the smallest directory subtree covering all selected tests. The created/updated test-suite is opened in the editor and highlighted in the projects view (if possible, probably not).

Details

- i) The action is enabled if and only if the selection contains just test-cases (or source files containing them, see below). A test-case is a test-class or test-suite – a class deriving `junit.framework.TestCase`.
- ii) Operation on source files is translated to operation on all contained test-cases, if there are any. It skips all non-test-cases in the source.
- iii) The action works only for source nodes from a single source-root. It is not possible to create a test-suite that contains tests from more source-roots. If test-cases from multiple test-roots are selected, an informational message appears informing the user that “Sources from multiple test-roots were selected but it is not supported to create test-suites for sources from multiple source-roots” (or similar) and the action terminates.

4.1.3 Open Test Class

Priority Must have, existing feature.

Summary The user selects several source files in the projects view. Using the contextual menu or shortcut the user invokes an action to open the corresponding test files for the selected sources.

Details

- i) The action is enabled if and only if some source nodes¹ and only source nodes are selected. For a non-source-file source node, the action is considered to operate on the containing source-file node.
- ii) If some of the test-files don't exist, an informative message is displayed after the existing tests are opened, offering to create the missing tests. If the user selects “yes, create them”, the action behaves as if the Create/Update tests for existing sources (4.1.1) was invoked on the nodes with missing tests².
- iii) The action works for source files from different source-roots.
- iv) If multiple corresponding test-classes exist for the source-class, all of them are opened.

4.2 Running

4.2.1 Run tests for the selected source class

Priority Should-have

¹source files or source classes

²On the nodes originally selected, not the covering source-file nodes.

Matching use-case II a. Change of source, update of test

Summary User selects a source-class in the projects view and runs the corresponding test-case for it by selecting the corresponding menu item in the context menu.

Details

- i) The action behaves exactly as the equivalent action bound to keyboard shortcut in editor (3.3.1)
- ii) The item is enabled if and only if the source class has a primary test class.
- iii) When the tests are run, the test result window (pane) is shown.
- iv) If multiple corresponding test-classes exist for the source-class, in multiple source roots necessarily, a dialog with choice of test-roots is displayed.

5 Main Menu

5.1 Creation

5.1.1 New Test for Existing class

Priority Must have, existing feature.

Summary Using the standard NetBeans new-from-template mechanism, the user creates a new test-class for some existing class. In the wizard, they choose the class to create the test for, the methods to create test-cases for, and other details.

Details

- i) The wizard contents behaves as the New test-class dialog (7.1) with the selection of class to test initially empty and editable. When the selection is changed, the contents is updated as per the dialog definition.
- ii) The result is the same as of the Create/update test-class for current class action (3.1.2).
- iii) The template name is “Test for Existing Class”

5.1.2 New empty test-class

Priority Must have, existing feature.

Summary Using the standard NetBeans new-from-template mechanism, the user creates a new empty test-class.

Details

- i) The wizard contents behaves as the New empty test-class dialog (7.3).
- ii) The result is an empty test-class as specified by the dialog.
- iii) The template name is “Empty Test”.

5.1.3 New test-suite

Priority Must have, existing feature.

Summary Using the standard NetBeans new-from-template mechanism, the user creates a new test-suite and selects its contents.

Details

- i) The wizard contents behaves as the New test-suite dialog (7.4).
- ii) The result is an empty test-suite as specified by the dialog.
- iii) The template name is “Test Suite”.

5.2 Running

Running from the main menu is handled by the projects infrastructure; menu items: Run/Test {project name}, Run/Run File/Test {file name}.

5.3 Editor Actions

All actions available by shortcuts from the editor should be mapped to the main menu for greater discoverability. Namely Creational Shortcuts (3.1) and Navigational Shortcuts (3.2).

6 Test results panel

6.1 Overview

Currently, tests are being run through the project ant script. The output of the run is displayed in the *Output console*, in the same place other build results are displayed. The output text is parsed and annotated with hyperlinks to source code, where recognized and the source exists. The output from the junit ant task however contains more information that is currently, without any highlighting, being lost in the clutter of stack traces and other text.

Thus, a more user friendly panel is needed that would capture and parse the output of the ant junit task and present it in a user friendly manner.

6.2 JUnit console output

While tests are run, no console output is provided except output generated by the test-cases directly to standard output or error (`System.out`, `System.err`). After the tests are finished a summary – test results – are printed. See an example bellow.

```
init:
deps-jar:
compile:
Created dir: /home/or141057/tmp/a2/build/test/classes
Compiling 1 source file to /home/or141057/tmp/a2/build/test/classes
compile-test:
Created dir: /home/or141057/tmp/a2/build/test/results
>>> some output from testMain <<<
>>> some error from testMain <<<
>>> some output from testNewMethod <<<
>>> some error from testNewMethod <<<
Testsuite: a2.MainTest
Tests run: 2, Failures: 1, Errors: 0, Time elapsed: 0.006 sec

----- Standard Output -----
>>> some output from testMain <<<
>>> some output from testNewMethod <<<
-----
----- Standard Error -----
>>> some error from testMain <<<
>>> some error from testNewMethod <<<
-----

Testcase: testMain(a2.MainTest): FAILED
The test case is empty.
junit.framework.AssertionFailedError: The test case is empty.
    at a2.MainTest.testMain(MainTest.java:39)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcc...
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM...
```

```

Test a2.MainTest FAILED
test-report:
/home/or141057/tmp/a2/nbproject/build-impl.xml:406: Some tests fai...
BUILD FAILED (total time: 0 seconds)

```

The structure of the test results appears to be specified by the following grammar. Note that only failed test-cases are reported.

```

TestResults      ::= SuiteResult*
SuiteResult      ::= SuiteHeader nl nl
                  SuiteSummary nl
                  TestcaseResult* nl
SuiteHeader      ::= 'TestSuite: ' String
SuiteSummary     ::= 'Tests run: ' Int
                  ', Failures: ' Int
                  ', Errors: ' Int
                  ', Time elapsed: ' Float ' sec'
TestcaseResult   ::= TestcaseHeader nl TestcaseOutput nl
                  Stacktrace? nl
TestcaseHeader   ::= 'Testcase: ' String
                  ': ' TestcaseStatus nl
TestcaseOutput   ::= MultilineString
Stacktrace       ::= MultilineString
TestcaseStatus   ::= 'FAILED' | 'Caused an ERROR'

```

Note, that the output could be interlaced with output of other tasks running in parallel. However, in the typical situation it won't be interlaced and the leading terminals of the productions of `SuiteHeader`, `SuiteSummary` and `TestcaseHeader` form reliable recovery points.

6.3 Integration

The output panel is a part of the project infrastructure for building (based on ant) and it is not possible to be replaced. It has very limited features. The only possibility to highlight text in it is by hyperlinks to URLs. Thus, instead of replacing it, an additional *test results panel* is to be shown with the outputs panel and provide an alternative view on the results.

The test results panel is hidden (or in the background) while the test run is in progress (in order to allow the user to see compilation and other messages). As soon as the ant task finishes with any tests run (i.e. it didn't fail in the compilation phase), the Test results panel is brought to top and allows convenient navigation of the test results.

6.4 Requirements

- i) The panel must present the information about failures as described above in a comprehensible form.
- ii) It must allow easy navigation to the places of failure in sources.

7 Dialog Specifications

This section contains the functional description of dialogs referred to in the text. The dialogs are described from the functional side only, that is, every modal dialog is a function from *inputs* to *outputs*. The inputs are transformed to the *model*, upon which the UI operates. That is, UI is a function from model to model. When the UI finishes, the model is transformed to outputs.

The description here concerns description of inputs, model and outputs, and the transformations $\text{inputs} \rightarrow \text{model}$ and $\text{model} \rightarrow \text{outputs}$.

To achieve a sufficient level of precision, we describe models with a context free grammar in the CNF form. It is a matter of implementation to realize the terminals and nonterminals. The realization needn't, of course, follow the given grammar rigidly. It is intended just a specification of the information flow between the program and the dialogs.

Note that certain nonterminals, namely those ending with "ID", are left unspecified. That's because the internal structure of these nonterminals is not important however, the ID's are potentially distinct and therefore we couldn't have used terminals. Just assume that there is some internal structure that has to be preserved by the dialogs, which is utilized inside the implementation, which is described informally in other sections of this specification.

7.1 New test-class

Inputs A single *source file* and a *source class*.

Model The model is two trees of class and method nodes, together with some other parameters (described below). Each node consists of an identification of a class or method. One of the class trees models sources, the other one the corresponding tests.

The two trees specify the desired configuration of sources and tests. The model is rather general thus allowing almost arbitrary UI for specification of tests. For example, to specify that a certain test-method should be created, the corresponding method should be added to the test tree. To specify, that all test methods for a certain source class should be created, the test tree must contain all tests methods corresponding to all source methods in the source tree.

```
ClassTree      ::= ( Class ClassBodyTree )
ClassBodyTree ::= ( ClassTree | MethodS )*
MethodS       ::= ( MethodID )
Boolean       ::= true | false
Class         ::= ( SourceRootID PackageID ClassID )
```

The `ClassID` and `MethodID` nonterminals are intentionally left unspecified as an implementation detail.

The other parameters are defined as follows.

```
ClassParams   ::= ( SetUp TearDown Bodies Hints Suites )
SetUp         ::= Boolean
TearDown      ::= Boolean
Bodies        ::= Boolean
Suites        ::= Boolean
```

To wrap this section up, we define the model.

```
NewTestClassModel ::= ( Sources Tests ClassParams )  
Sources           ::= ( ClassTree+ )  
Tests            ::= ( ClassTree* )
```

Note that the model doesn't contain the parameters currently found in the Create Test Dialog, such as public, protected, package-private that serve to select methods simply by filter rather than by individual selection. We leave this as a detail of the UI that serves to simplify the selection of multiple methods from the UI point of view rather than a parameter of the model. The test creation code does not depend on the way individual methods and classes were selected for creation, only the final state for each method or class is important.

Outputs The output is the model, `NewTestClassModel`. The output model of the dialog represents the desired state of sources and tests as configured by the user. Thus, for each test-method that should be created, the output field `NewTestClassModel.Tests` contains a corresponding tests method.

Inputs→model

- i) The `Sources.ClassTrees` are constructed from all source classes in the source file by copying their class structure. The `Tests.ClassTrees` are constructed from the test-classes corresponding to the source classes by copying their structure.
- ii) All `ClassParams` are initially set to `true`. On subsequent invocations, the state is recovered from the last known state, `StoredClassParams` (see the Model→outputs section for more details).

Model→outputs The model→outputs is the identity (outputs are the model). In addition, `ClassParams` are remembered in a global variable to be used with next invocation. For further reference, we name this global variable `StoredClassParams`.

7.2 New test-classes

Inputs A list of source files.

Model

```
NewTestClassesModel ::= ( FilesList ClassParams )  
FilesList           ::= ( File* )  
File                ::= ( FileID Sources Tests )
```

Outputs `NewTestClassesModel`

Inputs→model

- i) `FilesList` is built in the obvious way from the files and classes within.
- ii) `Sources` and `Tests` for each `File` are built as for the New test-class dialog.
- iii) The parameters share the same global state, `StoredClassParams`.

model→outputs The identity.

7.3 New empty test-class

Inputs A package.

Model

```
EmptyClassModel ::= ( ClassID SetUp TearDown )
```

Outputs EmptyClassModel

Inputs→model

- i) The **ClassID** is constructed from the package and a new class name such that the resulting **ClassID** refers to a nonexistent class in the package.
- ii) The **SetUp** and **TearDown** states are initially true. In subsequent invocations, they are recovered from **StoredClassParams**.

Model→outputs Identity. The states of **SetUp** and **TearDown** are stored to **StoredClassParams**.

7.4 New test-suite

Inputs A source root, a package, a list of test-classes.

Model

```
TestSuiteModel ::= ( Class ClassListS SuiteParams )
ClassListS     ::= ( ClassS* )
ClassS        ::= ( Class Checked )
SuiteParams   ::= ( SetUp TearDown Hints )
```

Outputs TestSuiteModel

Inputs→model

- i) The **Class** is constructed from the source root and the package and a class name such that the resulting **Class** refers to a nonexistent class in the package.
- ii) The **ClassListS** is constructed from all test classes in the source-root (including test-suites). The states of the **ClassSs** are initially set to true for classes in the input list of test-classes, false otherwise.
- iii) The **SetUp**, **TearDown** and **Hints** states are initially true. In subsequent invocations, they are recovered from **StoredClassParams**.

Model→outputs Identity.

7.5 Edit test-suite

Inputs A test-suite class.

Model TestSuiteModel

Outputs TestSuiteModel

Inputs→model

- i) The **Class** refers to the test-suite class.
- ii) The **Classes** in **ClassListS** are constructed from all test classes in the package (including test-suites). The state of the **ClassSs** are set to **true** if the class is in the test-suite, **false** otherwise.

This requires an analysis of the source of the test-suite to figure out which classes are included in the suite. It is sufficient if the analysis recognizes classes added exactly in the way as generated. In other words, it needn't (cannot) recognize classes added to the suite by user code that doesn't follow the same source pattern.

Model→outputs Identity.