Specification

# *CodeCover*
# Glass Box Testing Tool

Student Project A "OST-WeST"
University of Stuttgart

Version: 1.1-dev

Last changed on May 28, 2008 (SVN Revision 23)

# Version History

| Date | Version | Author | Modifications |
|---|---|---|---|
| 11.01.2007 | 0.1 | Stefan Franke | - Chapter files and master document file |
| 16.01.2007 | 0.2 | Stefan Franke | - ui: Eclipse Plug-in and images within |
| 17.01.2007 | 0.3 | Michael Starzmann<br>Christoph Müller | - Headwords following the corresponding chapter in the analysis-notes<br>- nr: Keywords taken over by the analysis notes<br>- fr: The foreword for the functional requirements |
| 18.01.2007 | 0.4 | Stefan Franke | - ui: Rewrite source code highlighting<br>- Reorder chapters<br>- Rename some sections<br>- ui: Added source code highlighting for COBOL |
| 19.01.2007 | 0.5 | Christoph Müller | - Document structure changed<br>- fr: Use case pictures imported<br>- fr: Foreword, fr: actors, fr: general arrangements |
| 20.01.2007 | 0.6 | Johannes Langauf<br>Christoph Müller | - nf: Expand some keywords to complete sentences<br>- nf: Find new NFRs<br>- fr: Configuration<br>- fr: Use case description<br>- fr: Language support |
| 21.01.2007 | 0.7 | Christoph Müller<br>Stefan Franke | - fr: Use case description<br>- ui: Session view, Coverage view and Launching |
| 23.01.2007 | 0.8 | Christoph Müller<br>Stefan Franke<br>Michael Starzmann | - Correction after specification meeting<br>- fr: Use case description of measure coverage<br>- fr: General functional requirements<br>- fr: Reports<br>- ui: Configuration dialogs |
| 24.01.2007 | 0.9 | Michael Starzmann<br>Christoph Müller | - in: Introduction<br>- fr: Use case description<br>- fr: Coverage Criteria |
| 25.01.2007 | 0.10 | Michael Starzmann<br>Stefan Franke | - ui: Package and file selection to ... states<br>- fr: Coverage measurement improved<br>- in: Introduction |

| Date | Version | Author | Modifications |
|---|---|---|---|
| 26.01.2007 | 0.11 | Christoph Müller<br>Stefan Franke | - Correction after specification meeting<br>- fr: New use case instrument instrumentable items<br>- ui: Configuration sections<br>- ui: Source code highlighting |
| 27.01.2007 | 0.12 | Christoph Müller<br>Stefan Franke<br>Michael Starzmann | - fr: Use case description of administrate sessions<br>- ui: Instrumentation subsection<br>- ui: Solved todos<br>- ui: first draft for the Batch interface<br>- fr: Release, folders, files |
| 28.01.2007 | 0.13 | Christoph Müller<br>Stefan Franke<br>Johannes Langauf | - fr: Batch interface<br>- ui,fr: Correction after internal review<br>- nr: improve and fill out most non-functional requirements |
| 29.01.2007 | 0.14 | Christoph Müller | - fr: Solve todos, use case diagrams, folder structure<br>- Correction after QA meeting<br>- small spell check |
| 30.01.2007 | 0.15 | Christoph Müller<br>Stefan Franke<br>Johannes Langauf | - fr: New use cases: analyse coverage log, export session<br>- ui: New Context menu<br>- ui: Import, Export, Report<br>- ui: Small adaption at figures<br>- nf: Correction after specification review<br>- nf: Extensibility, performance requirements, program examples |
| 31.01.2007 | 0.16 | Christoph Müller | - Correction after Igor's big bang |
| 31.01.2007 | 1.0 | Igor Podolskiy | Declaring version 1.0, ready for review |
| 08.02.2007 | 1.1-dev-1 | Stefan Franke<br>Michael Starzmann | - Correction after specification review |
| 09.02.2007 | 1.1-dev-2 | Christoph Müller | - Correction after specification review |
| 10.02.2007 | 1.1-dev-3 | Christoph Müller | - Correction after specification review: Bugs 47, 90, 52, 57, 56, 58, 60, 62, 63, 64, 65, 39, 40, 41, 42, 44, 46, 50, 51, 52, 54, 34 |
| 11.02.2007 | 1.1-dev-4 | Johannes Langauf<br>Christoph Müller<br>Stefan Franke | - Correction after specification review: Bugs 48, 35, 32, 91, 16 |
| 12.02.2007 | 1.1-dev-5 | Johannes Langauf<br>Christoph Müller | - Correction after specification review: Bug 22<br>- new batch commands |

| Date | Version | Author | Modifications |
|---|---|---|---|
| 13.02.2007 | 1.1-dev-6 | Stefan Franke<br>Christoph Müller | - moved the glossary to specification document<br>- added links to glossary entries<br>- Bug 7: Work flow<br>- Bugs 6, 21, 28, 88, 89, 91 |
| 14.02.2007 | 1.1-dev-7 | Stefan Franke<br>Christoph Müller<br>Michael Starzmann | - Bug 45 |
| 16.02.2007 | 1.1-dev-8 | Stefan Franke<br>Christoph Müller | - Bug 45 |
| 11.05.2007 | 1.1-dev-9 | Stefan Franke<br>Christoph Müller | - Bugs 98, 99, 100 |
| 15.06.2007 | 1.1-dev-10 | Christoph Müller | - fr: 2.9 JUnit integration |
| 17.06.2007 | 1.1-dev-11 | Stefan Franke | - ui: 3.13 Boolean Analyzer |
| 18.06.2007 | 1.1-dev-12 | Christoph Müller | - fr: 2.7.6 coverage log file name<br>- fr: 2.4.6 instrument supports a charset<br>- fr: 2.4.7 analyze supports a charset<br>- fr: 2.4.5 Instrumenter-info |
| 19.06.2007 | 1.1-dev-13 | Christoph Müller | - fr: 2.4.6 instrument has `--copy-uninstrumented` |
| 29.06.2007 | 1.1-dev-14 | Christoph Müller | - fr: 2.4.6 instrument has include, exclude |
| 19.09.2007 | 1.1-dev-15 | Johannes Langauf | - ui: 3.14 Hot-Path: make outstanding decisions, update for configureable colors<br>- fr: remove PDF-Report support |
| 31.10.2007 | 1.1-dev-16 | Tilmann Scheller | general update of specification |
| 28.05.2008 | 1.1-dev-17 | Christoph Müller | - fr: `return` and `break` are basic statements too |

# Contents

# 1 Introduction

## 1.1 Project overview

*CodeCover* stands for **g**lass **b**ox **t**esting **t**ool. It measures the code coverage↗ of a running program and will be as independent as possible of the programming language of the covered program.

Characteristics of *CodeCover*:

- *CodeCover* runs at least on Linux and Windows,
- *CodeCover* can measure code coverage for programs written in Java and COBOL.
- *CodeCover* is extensible to measure code coverage for further programming languages as well.
- *CodeCover* measures multiple code coverage criteria and is extensible to further ones.
- *CodeCover* provides functionality to create reports of the measured code coverage in HTML↗-files.
- *CodeCover* is an Eclipse[1] plug-in with a graphical user interface, but also provides a command line interface for use without Eclipse.

To understand the functional requirements specified in this document, a visual overview of the work flow is shown in figure 1.1.

Several steps and intermediate results exist for the whole of the coverage measurement process. The ellipses stand for processing, the rectangles stand for intermediate results or final results.

The process starts with the instrumentation↗ of code files↗. A MAST↗ is produced in addition to the instrumented code files. The MAST↗ is stored with the code files↗ in a session container↗. After the compilation and execution of all the instrumented code files of the SUT↗, a coverage log↗ with the raw coverage results is produced.

During the analysis phase, the coverage log is processed to obtain a test session↗ with test cases↗. They contain all the processed coverage results. These information are

[1] `http://www.eclipse.org/`

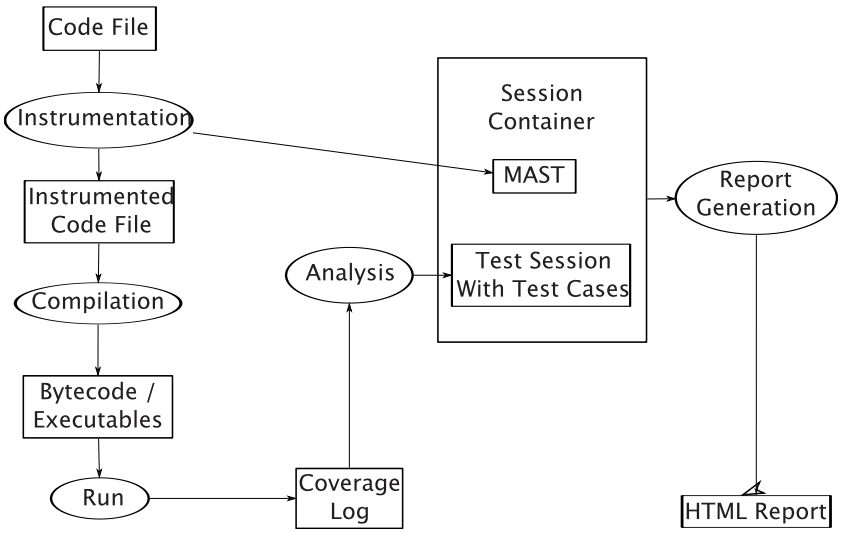


Figure 1.1: Work flow of the software

added to the session container↗.

Using the information of the MAST↗ and the test sessions↗, *CodeCover* can generate a HTML↗ report.

## 1.2 About this document

This document specifies all requirements the software has to fulfill and all interfaces to users or other programs. The design of the software will be written based upon this document. This document is the common ground between the customer and the developers↗. Therefore, it's important that both, customers and developers, pay attention to the quality of this document and keep it current.

## 1.3 Addressed audience

This document is addressed to

- the customer who ordered the software
- the project manager controlling the work
- the designers writing the software design

- the quality assurance division creating test cases↗ for the software
- the developers implementing the design
- future developers maintaining and extending the software
- interested users of the software
- students of upcoming student projects

## 1.4 Conventions for this document

A glossary is shipped together with this specification↗. It contains basic definitions and allows clear statements in this document because it prevents ambiguity. Therefore words mentioned in the glossary are used often and are not explicitly defined in this specification but in the glossary.

The term "software" is used for *CodeCover*. Code examples and file names are written in the `typewriter style`. Labels and names of graphical user interface components are written in SMALL CAPS. If necessary, examples are used and placeholders are enclosed by percentage signs: %placeholder%. Furthermore, glossary entries are marked with the symbol ↗, but only at the first occurrence in a section.

## 1.5 Authors

In the following table the contact persons per section are named.

| **Section** | **Author** | **E-mail** |
|---|---|---|
| Introduction | Michael Starzmann | starzmml@studi.informatik.uni-stuttgart.de |
| Functional requirements (2.1 – 2.5) | Christoph Müller | muellecr@studi.informatik.uni-stuttgart.de |
| Functional requirements (2.6 – 2.8) | Michael Starzmann | starzmml@studi.informatik.uni-stuttgart.de |
| Graphical user interface | Stefan Franke | frankesn@studi.informatik.uni-stuttgart.de |
| Non-functional requirements | Johannes Langauf | langaujs@studi.informatik.uni-stuttgart.de |

# 2 Functional requirements

## 2.1 Test sessions and test cases

The software produces test sessions↗ which contain test cases↗. A session container↗ stores a number of test sessions which each refer to a code base↗. Each test session has to have a unique name within a session container. It is not specified how the session containers are stored (XML, internal database, ...) but it should be decided in the software design phase.

Test cases are used to subdivide a test session. Test cases contain the results of the coverage measurement over a period of time during the execution of the SUT↗. This can either be the whole of the SUT run, in this case the test session contains only a single test case, or a smaller period of time.

The end of a test case should be explicitly declared, so that it is clear where a test case begins and ends.

The test cases do not overlap. In consequence, the start of a new test case enforces the end of the previous test case. A test case in a test session is uniquely defined by its (test case's) name. If a test case with the same name is started several times, all respective results of the coverage measurement are associated with the same test case.

Test cases will be defined by JUnit or by the user during the SUT execution using a dialog box.

In addition to that, the software provides an integrated test case notification mechanism.

To use the test case notification mechanism in Java, a small JAR file containing a `Protocol` class can be added to the SUT's class path. This class has the following methods the user can call anywhere in the code of the SUT to create test cases:

```
//defining the start of a named and described test case:
public static void startTestCase(String name, String comment)
//Alternatively defining the start of named test case:
public static void startTestCase(String name)
//Defining the end of the last test case:
public static void endTestCase(String name)
```

```
//Alternatively defining the end of the last without using the name:
public static void endTestCase()
```

The name of the `Protocol` class and the names of the methods are not normative. They are examples used to describe the mechanism and can be adapted by the software design at will.

If there is not a valid `Protocol` call in the code files of the SUT, the software will create one anonymous test case with the name `unnamed test case` for the full test session results. But if there are defined test cases, only the coverage occurring during test cases is measured.

Test sessions and test cases are related to a specific version of the code files, the code base. The software can only highlight the results of a coverage run of a code file (see section 3.8) if the code file is equal to the code file used for the instrumentation.

Test sessions depending on the same code base can be merged. This means that all test cases contained in two or more different test sessions are copied into a new test session. If test cases have the same name, they are renamed to `test case name (session name 1)`, `test case name (session name 2)`. Also, two or more test cases of one test session can be merged to a new test case.

## 2.2 Actors

To describe the use cases in the following section 2.3, actors must be defined. These actors are the users of the software. It is assumed that these users are software developers[1] or testers and thus have experience with software tools. The actor model is shown in figure 2.1.

As the software partly integrates with the Eclipse IDE, one type of actor is the *Eclipse user*. He has experience in using Eclipse and has worked with plug-ins before. He wants the plug-in interface to be intuitive and expects a similar behavior he is used to from other plug-ins. He is accustomed to control every software feature out of Eclipse and does not want to have to open external programs.

The *shell user* is the user who controls the software using the system shell. He has used Windows or Linux shells before as well as other command line programs and is used to their respective characteristics. He wants a well-written reference manual and on line

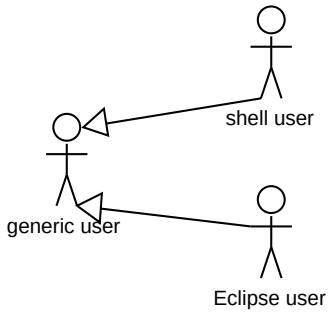


Figure 2.1: Actors

help with subcommand and option listing.

In the following use case models often a *generic user* is used, because Eclipse and shell users participate in the same use cases. In these cases, both are generalized to the *generic user*.

## 2.3 Use case description

### 2.3.1 Preface

In the rest of this section the functional specification↗ is described. To clarify the functional requirements, a use case analysis is used. There is no common understanding of the purpose of use cases. On the one hand, they are used to show the key functions of the specified software – the key functions a customer wants to be implemented. On the other hand, they are used to describe the sequence of actions clearly. For the following use case description both aspects are needed.

For the description of the key functions, key use cases are applied. They summarize smaller use cases and allow an overview. These use cases are defined in section 2.3.3. Besides the key use cases, standard use cases are employed to describe the functions in detail.

### 2.3.2 Predefinitions

#### 2.3.2.1 Use case descriptions

Each standard use case comes with a use case description. The use case description consists of the following items:

- Actor
- Preconditions
- Regular sequence
- Other sequences
- Postconditions
- Possible exceptions

The *actor* is the person performing the use case. The actors are described in section 2.2.

The *preconditions* describe the circumstances needed to start the use case. This can be a state of the software, an open dialog box or the successful termination of another use case.

The *regular sequence* is the description of the normal steps of the use case. It states how the actor interacts with the software, which input is made and which feedback is returned by the software. It is assumed that the sequence is successfully finished without errors.

If there are short cuts or small modifications possible for the regular sequence, they are described in *other sequences* too. Also the cancellation of a use case belongs to this section.

The *postconditions* describe the state of the software and – if affected – data after the regular sequence is successfully finished. If there are other possible sequences, the *postconditions* describe the state of the software after each of these sequences.

The *possible exceptions* are used to specify sequences where errors occur. They can be caused by mistakes of the actor, file system errors or other states which cause the sequences to be aborted.

Beginning with the section 2.3.2.2 general assumptions are described that hold true for every use case. An explicit statement$^{\nearrow}$ in the use case description can override these

assumptions for a particular use case.

#### 2.3.2.2 General preconditions

Considering the Eclipse user, Eclipse must be started. The plug-in must be installed correctly and must not be disabled. A Eclipse project↗ must be opened.

#### 2.3.2.3 General other sequences

For the use cases of the Eclipse user it is assumed, that every use case which includes interaction with a dialog, can be stopped immediately by clicking the Cancel button in the dialog.

There are often several ways to open a dialog or to execute a command in Eclipse. In most cases only one way is described for simplicity. Some are listed here, because they are explicitly used:

- the Eclipse dialog PROPERTIES can be opened using the context menu of the dialog or the menu PROJECT
- the context menu of a code file↗ can be opened in the PACKAGE EXPLORER and in the NAVIGATOR
- the context menu can be opened using a right click or the Context Menu Key on the keyboard

#### 2.3.2.4 General postconditions

If it is stated that something – e.g. a test session↗ – is *saved* the changes are stored persistently in the related session container↗.

#### 2.3.2.5 General possible exceptions

The general behavior of the software in abnormal situations is described in the section 4.7.

If a session container could not be updated or created – e.g. due to lack of access permissions or low disk space – an error message is shown to inform the actor.

### 2.3.3 Key use cases

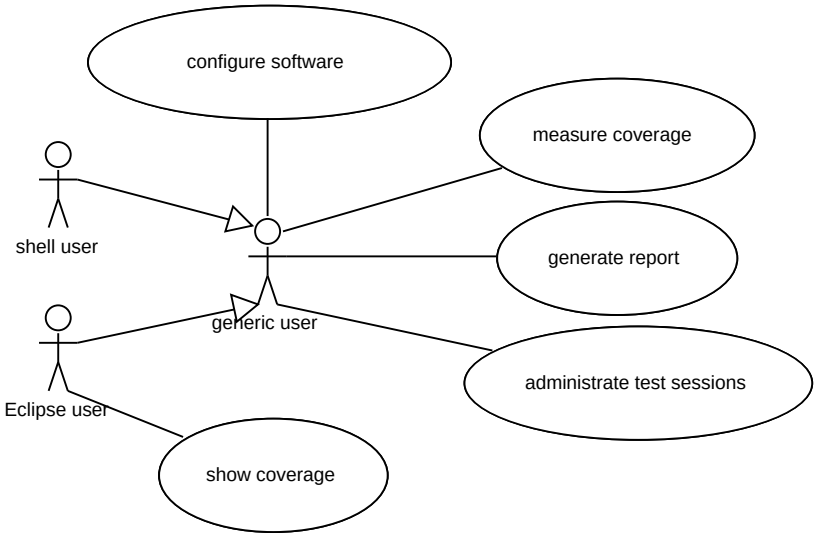


Figure 2.2: Key use cases

This diagram shows the key use cases of the software. As introduced in section 2.2, there is an Eclipse user and a shell user. Both are specialized from the generic user.

The key use cases summarize a block of functionality and can be subdivided into smaller standard use cases. These are:

- Measure coverage (see section 2.3.4)
- Show coverage (see section 2.3.5)
- Administrate test sessions (see section 2.3.6)
- Generate report (see section 2.3.7)
- Configure software (see section 2.5)

### 2.3.4 Measure coverage

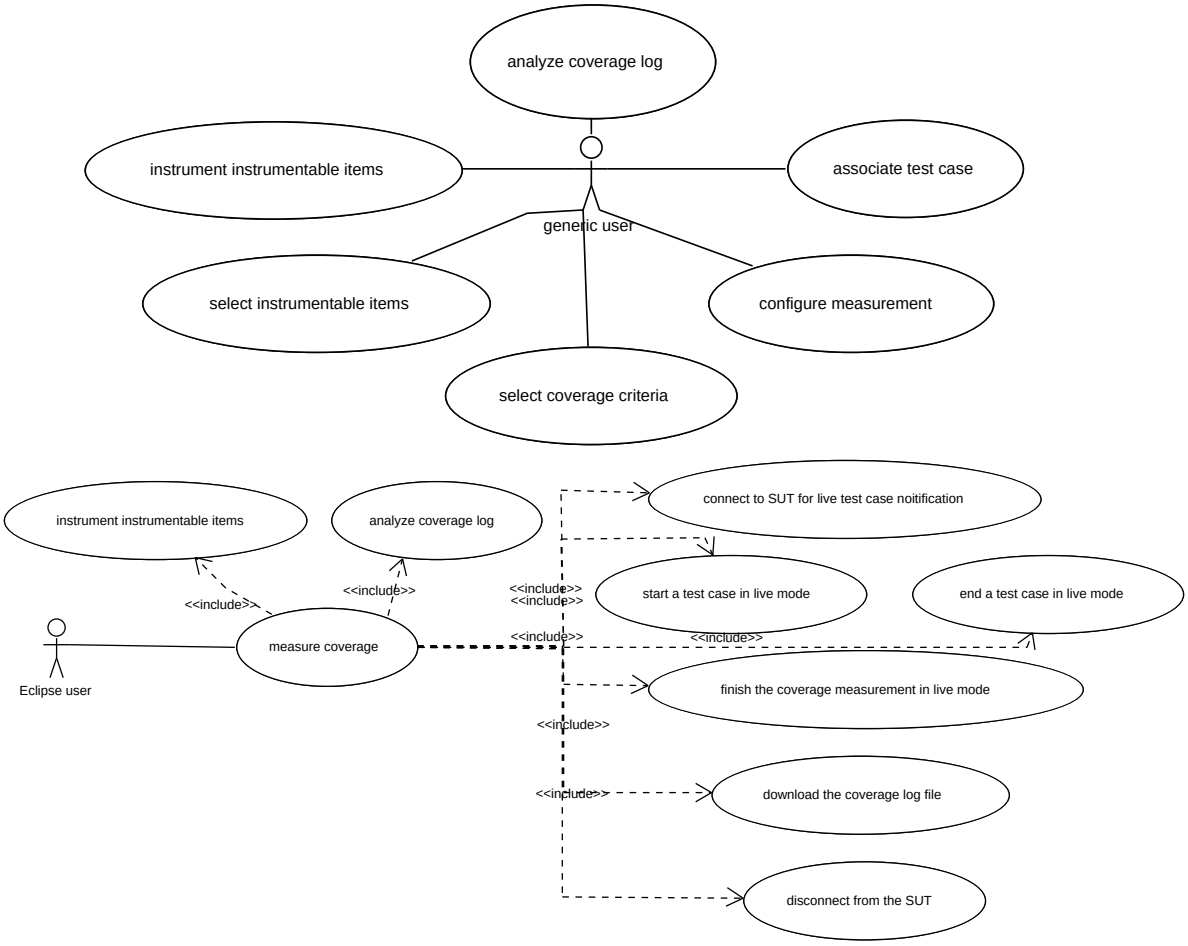


Figure 2.3: Use cases related to measuring coverage

#### 2.3.4.1 Use case: select instrumentable items

##### 2.3.4.1.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.4.1.2 Preconditions

The actor has opened an Eclipse project containing at least one instrumentable item↗.

##### 2.3.4.1.3 Regular sequence

The actor selects one or more instrumentable items – e.g. in the Package Explorer – and clicks on the check box menu item Use For Coverage Measurement in the

context menu.

To deselect instrumentable items, the actor repeats the described procedure and clicks on the context menu item Use For Coverage Measurement again.

#### 2.3.4.1.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.1.5 Postconditions

Selecting or deselecting an instrumentable item has an recursive effect on all its sub items: for example, selecting a package causes all its sub packages and types to be selected, too. The same applies for the deselecting. If an instrumentable item had been selected before and a parent item is selected later, the originally selected item remains selected.

In the Package Explorer, the icons of the selected instrumentable items change to Used For Coverage Measurement state.

If the actor has deselected instrumentable items, the icon changes to the normal state. (see section 3.1)

#### 2.3.4.1.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

### 2.3.4.2 Use case: select coverage criteria

#### 2.3.4.2.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.2.2 Preconditions

If the actor has not changed the coverage criteria of a project, all coverage criteria are selected.

#### 2.3.4.2.3 Regular sequence

The actor opens the Project Properties dialog for the particular project. Then he clicks on the item *CodeCover* . Here the actor can select which coverage criteria he wants to measure. To add a criterion for measurement, he activates the related check box. Deactivating a check box means that the corresponding coverage criterion↗ will not be measured. It is not possible to deselect all check boxes and apply the changes.

After the actor has made his choice, he clicks on the button OK. The dialog PROPERTIES closes.

#### 2.3.4.2.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.2.5 Postconditions

At least one coverage criterion is selected for the edited Eclipse project. The software saves this selection. In addition to that, the software checks whether the already instrumented instrumentable items↗ must be reinstrumented for the new selection of coverage criteria.

#### 2.3.4.2.6 Possible exceptions

If the actor has deselected all check boxes the dialog prohibits the click on OK.

### 2.3.4.3 Use case: instrument instrumentable items

#### 2.3.4.3.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.3.2 Preconditions

There are no special preconditions needed for this use case.

#### 2.3.4.3.3 Regular sequence

The actor uses the menu PROJECT and the menu item INSTRUMENT PROJECT... to explicitly instrument the selected instrumentable items. A dialog opens and asks the actor to enter the target path for instrumented code files↗. The actor puts in a valid path he has write access to. With a click on the button INSTRUMENT he starts the instrumentation↗ process.

While this process is running, a progress bar appears to inform the actor about the progress of the instrumentation process. The Eclipse integrated progress bar is used for this purpose.

#### 2.3.4.3.4 Other sequences

This use case is implicitly started by the use case *measure coverage* (see section 2.3.4.4). In this case, the default target folder of the project for instrumented code files is used

and the dialog is not displayed (see section 2.7.1).

If there are no instrumentable items selected for coverage measurement, the software opens a dialog box to ask the actor, whether he wants to instrument every instrumentable item or wants to cancel.

#### 2.3.4.3.5 Postconditions

If the use case is explicitly started by the user, a new code base↗ is created having the date and time of the end of the instrumentation process. A MAST↗ is created of the source files. A new session container↗ is created, containing the code base and the MAST. The session container is stored. In the Test Sessions view the new code base is selected. It has got no test session. All code files which are Used For Coverage Measurement are instrumented and stored at the given target path. All other source files are just copied.

The same procedure is used, if this use case is implicitly started by the use case *measure coverage*. The only exception is made, if no changes were made at the source files of the project since the last start of *measure coverage* for this project and the selection of the code files Used For Coverage Measurement has not changed. In this case, the last selected code base can be used again.

#### 2.3.4.3.6 Possible exceptions

If the instrumented code files could not be written – e.g. due to lack of access permissions or low disk space – an error message is shown to inform the actor.

### 2.3.4.4 Use case: measure coverage

#### 2.3.4.4.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.4.2 Preconditions

At least one coverage criterion is activated for measurement. There is an entry point↗ in the current project.

#### 2.3.4.4.3 Regular sequence

The actor navigates to the entry point for which he wants to start the coverage measurement. He clicks on the Coverage Button (see figure 3.3) and at the appearing menu on the button Coverage As. . . , Java-Application.

If no code file of this project has been changed since the last start of this use case for the same project and the selection of the code files USED FOR COVERAGE MEASUREMENT has not changed, no instrumentation is needed. Otherwise, the software will implicitly start the use case *instrument instrumentable items* (see section 2.3.4.3) and a new code base will be created.

After having instrumented all the required files, the software rebuilds the instrumented code files↗ and starts the SUT↗ using the selected entry point.

After the instrumented project has terminated, the software proceeds with the measurement calculation of the covered elements.

#### 2.3.4.4.4 Other sequences

If the actor has not selected any instrumentable item for coverage measurement, the software opens a dialog box to ask the actor, whether he wants to instrument every instrumentable item or wants to cancel.

If the entry point has been used for coverage measurement before, the list of the COVERAGE BUTTON contains this entry so that the actor can use this entry directly instead of using the buttons COVERAGE AS. . . , JAVA-APPLICATION again.

Additionally, the COVERAGE dialog (see section 3.3) contains entries for coverage measurements used in past. The actor can use this dialog to start a coverage measurement too. He selects the entry in the entry list on the left and clicks on the button COVERAGE.

#### 2.3.4.4.5 Postconditions

The result of the coverage measurement run is a coverage log↗. The use case *analyze coverage log* (see section 2.3.4.6) is implicitly started for this coverage log. This use case produces a test session for the measurement.

The software has saved the results of the coverage measurement in a new test session. This test session has the name *New test session* and a number as suffix if needed for uniqueness. The TEST SESSIONS view (see figure 3.5) contains the new test session which is automatically selected. All test cases of the new test session are shown in the list of test cases. They are all automatically activated.

If there had not been at least one instrumentable item selected for coverage measurement and the software had selected all after request, then the state of them changes to USED FOR COVERAGE MEASUREMENT and the PACKAGE EXPLORER updates their icons.

#### 2.3.4.4.6 Possible exceptions

If there are errors in the process, the process will be canceled and an error message will be shown. Possible errors might be:

- I/O errors while instrumenting
- compile errors
- errors starting the entry point
- access permissions or low disk space when writing the coverage log↗

### 2.3.4.5 Use case: associate test case

#### 2.3.4.5.1 Purpose

The actor wants the software to start a named test case, when the control flow passes a specific line in a code file, e.g. the actor has written a test script which calls several methods of several test classes. Test cases should be defined for each of these method calls.

#### 2.3.4.5.2 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.5.3 Preconditions

There is a code file in an open Eclipse project which contains the code the actor wants to add test case notifications to.

#### 2.3.4.5.4 Regular sequence

The actor navigates to the code file and positions the cursor before the line of the code file where the test case should start. Then he uses the menu items Source, *CodeCover* Test Case Notification, Start Test Case With Name And Comment (see section **??**). The software adds an import declaration in the file and adds a new code line at the position of the cursor:

```
Protocol.startTestCase("%NAME%", "%COMMENT%");
```

The actor changes `"%NAME%"` and `"%COMMENT%"` to the name and the comment of the test case. The software is ordered to start a new test case when this method is called in the coverage measurement.

To define the end of a test case, the actor uses the menu items SOURCE, *CodeCover* TEST CASE NOTIFICATION, END TEST CASE WITH NAME (see section **??**). The software adds a new code line:

```
Protocol.endTestCase("%NAME%");
```

The actor changes the `"%NAME%"` to the name of the test case started before and saves the file.

#### 2.3.4.5.5 Other sequences

If the actor wants multiple test cases, he uses this procedure at different lines of the code file. He can also associate test cases in other code files.

If the actor only wants to have one test case for the whole test script, he must not add a special statement anywhere. The software then treats the whole measurement as one test case. (see section 2.1)

There are also other test case notification forms possible that have the same effect. These are described in section 2.1. The start of a test case implies the end of the prior test case.

If the actor is more advanced, he can write the statement into the source code on his own. In this case he must add the JAR containing the `Protocol` class to the class path of the related Eclipse project too.

#### 2.3.4.5.6 Post conditions

The software or the actor has added the JAR containing the `Protocol` class to the class path of the Eclipse project. The code file is prepared for measurement with test case association.

#### 2.3.4.5.7 Possible exceptions

There are no special possible exceptions to be considered for this use case.

### 2.3.4.6 Use case: analyze coverage log

#### 2.3.4.6.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.6.2 Preconditions

The actor has used the use case *instrument instrumentable items* and has run the compiled SUT on his own and without Eclipse support. In the consequence there is a coverage log related to an Eclipse project.

If the actor has instrumented the code files out of Eclipse, the actor has to import the session container↗ with the corresponding code base↗ first (see section 2.3.6.2).

Anyway, there is a code base loaded in Eclipse and the source files of this code base were compiled and executed. A coverage log↗ file was created.

#### 2.3.4.6.3 Regular sequence

The actor activates the Test Sessions view (see figure 3.5). In the view, he clicks on the button Import Coverage Log. The import dialog for session containers opens.

In the dialog the actor specifies the coverage log. Moreover he must state a name and can type a comment for the test session that will be created. After that, he clicks on Finish. The dialog closes. (see figure **??**)

#### 2.3.4.6.4 Other sequences

The dialog for importing a coverage log↗ file can also be opened by using the default Eclipse import dialog. For example this can be opened using the menu File and the item Import... There the actor selects the item *CodeCover* Coverage Log in the group Other and clicks on Next.

#### 2.3.4.6.5 Post conditions

The software processes the given coverage log and creates a new test session with the given name and comment. The test session is assigned to the corresponding code base. The test session is saved in the session container↗.

In the new test session is selected in the Test Sessions view (see figure 3.5). All its test cases are shown in the list of the test cases and are selected.

#### 2.3.4.6.6 Possible exceptions

If there is no code base↗ in Eclipse, the new coverage log belongs to, an error message is shown.

If the test session is not related to the Eclipse project of the code base, the code highlighting won't be possible (see section 2.3.5.3) but the coverage results can be examined

(see section 2.3.5.2).

If there are errors processing the coverage log, the process is interrupted and an error message is shown.

#### 2.3.4.7 Use case: configure measurement

The following use cases contains the configuration of the measurement behavior. It is described in section 2.5.

#### 2.3.4.8 Use case: connect to an SUT for live test case notification

##### 2.3.4.8.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.4.8.2 Preconditions

The instrumented SUT has been configured to support the live test case notification and is running. The Test Case Notification view is open.

##### 2.3.4.8.3 Regular sequence

The actor enters the host name and port to connect to and clicks the Connect button.

##### 2.3.4.8.4 Other sequences

There are no other sequences possible for this use case.

##### 2.3.4.8.5 Postconditions

The view is connected to the running SUT.

##### 2.3.4.8.6 Possible exceptions

If there was a network error, an error message is shown.

If the CodeCover MBean is not available on the server, the client waits until a CodeCover MBean is registered; all operations except disconnecting will stay disable until a CodeCover MBean is registered on the Server.

#### 2.3.4.9 Use case: start a new test case in live mode

##### 2.3.4.9.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.9.2 Preconditions

A connection to a running SUT is established (see use case 2.3.4.8). The Test Case Notification view is open. The coverage measurement in the current SUT run is not finished.

#### 2.3.4.9.3 Regular sequence

The actor enters a test case name in the text field of the view and clicks the Start button.

#### 2.3.4.9.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.9.5 Postconditions

The test case has been started.

#### 2.3.4.9.6 Possible exceptions

If there was a network error, an error message is shown.

### 2.3.4.10 Use case: end the current test case in live mode

#### 2.3.4.10.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.10.2 Preconditions

A connection to a running SUT is established (see use case 2.3.4.8). The Test Case Notification view is open. A test case is started.

#### 2.3.4.10.3 Regular sequence

The actor clicks the End button in the Test Case Notification view.

#### 2.3.4.10.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.10.5 Postconditions

The test case has been ended.

#### 2.3.4.10.6 Possible exceptions

If there was a network error, an error message is shown.

### 2.3.4.11 Use case: finish coverage measurement in live mode

#### 2.3.4.11.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.11.2 Preconditions

A connection to a running SUT is established (see use case 2.3.4.8). The Test Case Notification view is open.

#### 2.3.4.11.3 Regular sequence

The actor clicks the Finished button in the Test Case Notification view.

#### 2.3.4.11.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.11.5 Postconditions

The measurement has been finished.

#### 2.3.4.11.6 Possible exceptions

If there was a network error, an error message is shown.

### 2.3.4.12 Use case: download the coverage log file

#### 2.3.4.12.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.12.2 Preconditions

A connection to a running SUT is established (see use case 2.3.4.8). The Test Case Notification view is open. The coverage measurement has been finished.

#### 2.3.4.12.3 Regular sequence

The actor clicks the Download button in the Test Case Notification view.

#### 2.3.4.12.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.12.5 Postconditions

The coverage log file is downloaded and has the same base name as the file written on the system running the SUT and the same location as if the execution was local and triggered by *CodeCover* Eclipse plug-in.

#### 2.3.4.12.6 Possible exceptions

If there was a network or file access error, an error message is shown.

### 2.3.4.13 Use case: disconnect from the SUT in live mode

#### 2.3.4.13.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.4.13.2 Preconditions

A connection to a running SUT is established (see use case 2.3.4.8). The Test Case Notification view is open.

#### 2.3.4.13.3 Regular sequence

The actor clicks the Disconnect button in the Test Case Notification view.

#### 2.3.4.13.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.4.13.5 Postconditions

The connection to the SUT is closed.

#### 2.3.4.13.6 Possible exceptions

If there was a network error, a warning is shown.

### 2.3.5 Show coverage

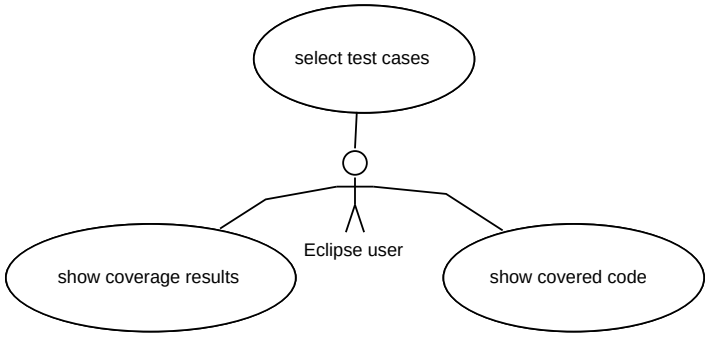


Figure 2.4: Use cases related to showing coverage

#### 2.3.5.1 Use case: select test cases

##### 2.3.5.1.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.5.1.2 Preconditions

There is at least one test session in Eclipse which has at least one test case.

##### 2.3.5.1.3 Regular sequence

The actor opens the Test Sessions view (see figure 3.5). Then he selects the related Code base.

Now the actor can activates all test cases he wants to view the coverage results of, using the Activated check boxes. The rest of the test cases' check boxes have to be deactivated.

##### 2.3.5.1.4 Other sequences

The actor can select test cases of different test sessions. To select all test cases of a test session, the actor uses the Activated check box of the specific test session.

There are some other sequences possible to activate test cases – e.g. using the context menu in the Test Sessions view (see section 3.5).

#### 2.3.5.1.5 Postconditions

The source code highlighting and the coverage view are refreshed if needed, based on the results of the selected test cases.

#### 2.3.5.1.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

### 2.3.5.2 Use case: show coverage measurement

#### 2.3.5.2.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.5.2.2 Preconditions

The actor has selected test cases of test sessions belonging to the same code base (see use case *select test cases*, section 2.3.5.1).

#### 2.3.5.2.3 Regular sequence

The actor activates the COVERAGE view of the plug-in (see figure 3.4). At this view he has an overview of all instrumented instrumentable items in a hierarchical order. He can expand an item of the hierarchy to examine the coverage results of its sub items.

The result columns show the measured results of the coverage by criterion. Only the criteria that are measured are shown in this view.

To order the lines of the tree table ascending or descending, the actor clicks respectively clicks twice on the specific column header. The lines of items are then sorted within their parent item in the tree table.

#### 2.3.5.2.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.5.2.5 Postconditions

There are no possible post conditions of this use case.

#### 2.3.5.2.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

#### 2.3.5.3 Use case: show covered code

##### 2.3.5.3.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.5.3.2 Preconditions

The actor has selected test cases of test sessions belonging to the same code base (see use case *select test cases*, section 2.3.5.1). The code base of the test sessions selected is still the current one, which means, no code file has been changed since the instrumentation of the code files.

##### 2.3.5.3.3 Regular sequence

The actor navigates to the code file in which the coverage results are to be displayed by source code highlighting. Then he opens the code file in an editor.

##### 2.3.5.3.4 Other sequences

There are no other sequences possible for this use case.

##### 2.3.5.3.5 Postconditions

The software highlights the elements of the code according to results of the measurement of the selected coverage criteria. The highlighting rules are specified in section 3.8 in detail.

##### 2.3.5.3.6 Possible exceptions

If a code file has changed since the coverage run of the test session, the highlighting can not be shown. Therefore the software has to check, if the code file has the same content as the code file used for the related coverage run.

If the actor changes a code file, the highlighting is not possible anymore. If he revokes his changes, the software shows the highlighting again.

### 2.3.6 Administrate test sessions

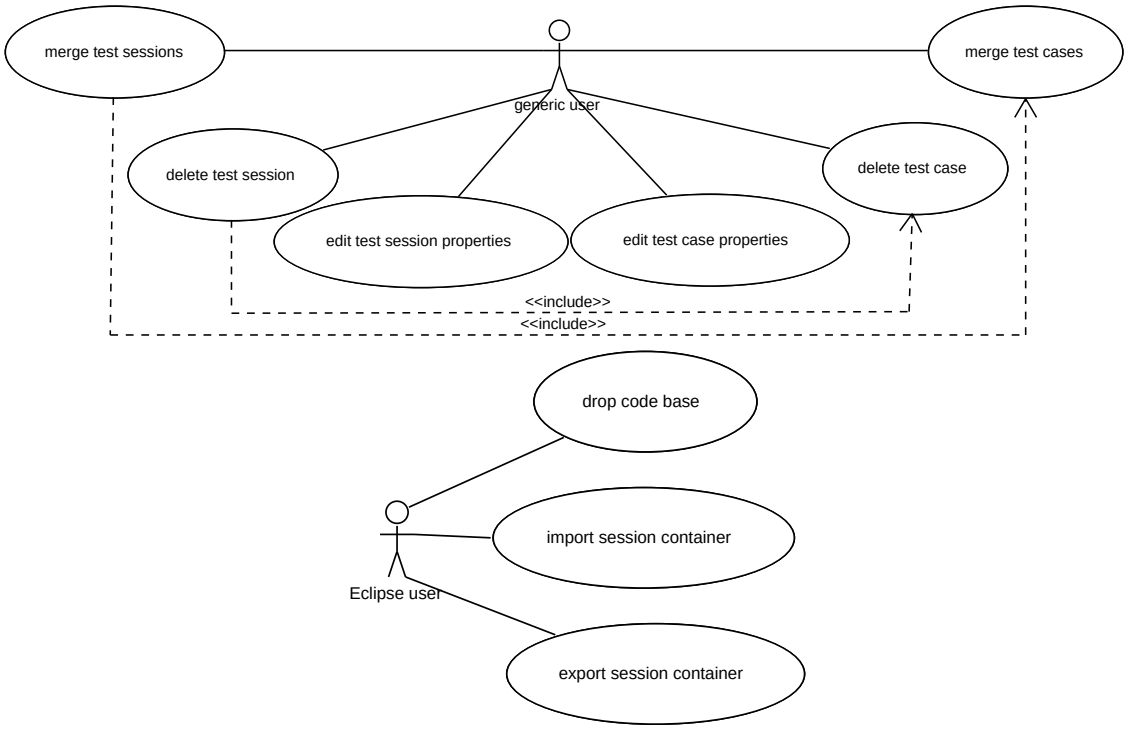


Figure 2.5: Use cases related to administrating test sessions

#### 2.3.6.1 Preface

These use cases are based on other use cases described before. By measuring the coverage, the software creates a test session containing associated test cases. They are the basis of analysis and can be edited in several ways. The test sessions and test cases can be merged and their properties can be altered. Test cases can be deleted from a test session and whole test sessions with all included test cases can be deleted.

To use the Eclipse plug-in and the batch interface side by side, import and export functionality is supported by the Eclipse plug-in. General definitions regarding test sessions↗ and test cases are made in the section 2.1.

#### 2.3.6.2 Use case: import session container

##### 2.3.6.2.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.2.2 Preconditions

There must be at least one session container⟋ in the file system. This can be either created by a coverage measurement in Eclipse or using the batch mode (see section 2.4).

#### 2.3.6.2.3 Regular sequence

The actor activates the TEST SESSIONS view (see figure 3.5). In the view, he clicks on the button IMPORT TEST SESSION. The import dialog for session containers opens (see figure **??**). In this dialog, the actor specifies the path to the session container and selects the related Eclipse project. Finally the actor clicks on the button FINISH. The dialog closes.

#### 2.3.6.2.4 Other sequences

The dialog for importing a session container can also be opened by using the default Eclipse import dialog. For example this can be opened using the menu FILE and the item IMPORT.... There the actor selects the item *CodeCover* SESSION CONTAINER in the group OTHER and clicks on NEXT.

If the code base of the session container is not related to an Eclipse project, the actor needn't select a project.

#### 2.3.6.2.5 Postconditions

The code base of the session container is imported into Eclipse.

If the session container has got test session(s), they are imported too. In this case, one of the imported test sessions is selected in the TEST SESSIONS view (see figure 3.5). All its test cases are shown in the list of the test cases and are activated.

All information needed to use the session container in Eclipse for future are saved.

#### 2.3.6.2.6 Possible exceptions

If the code base⟋ of the session container is not related to the specified project, the code highlighting won't be possible (see section 2.3.5.3) but the coverage results can be examined (see section 2.3.5.2).

If there are errors loading the session container, the process is interrupted and an error message is shown.

#### 2.3.6.3 Use case: export session container

##### 2.3.6.3.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.6.3.2 Preconditions

There must be at least one test session in Eclipse.

##### 2.3.6.3.3 Regular sequence

The actor selects the code base and its test sessions he wants to export in the TEST SESSIONS view (see figure 3.5) and clicks on the button EXPORT in the tool bar.

The export dialog opens (see figure 3.9). The selected code base and the selected test sessions are preselected in the dialog, but the actor can also select more test sessions. He changes the TYPE to *CodeCover* SESSION CONTAINER, chooses a destination and clicks on FINISH. The dialog closes.

##### 2.3.6.3.4 Other sequences

The dialog for exporting a test session can also be opened by using the default Eclipse export dialog. This dialog can for example be opened using the menu FILE and the item EXPORT.... In the selection dialog the actor selects the item *CodeCover* COVERAGE RESULT EXPORT in the group OTHER and clicks on NEXT.

##### 2.3.6.3.5 Postconditions

A session container is created at the specified destination. It contains the code base and all selected test session.

##### 2.3.6.3.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

#### 2.3.6.4 Use case: drop code base

##### 2.3.6.4.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

##### 2.3.6.4.2 Preconditions

There must be at least one code base in Eclipse.

#### 2.3.6.4.3 Regular sequence

The actor opens the Test Sessions view (see figure 3.5), selects the code base↗ and clicks on the button Drop Code Base. A dialog opens, requesting the actor if he wants to drop the selected code base out of Eclipse or if he wants to deleted the related session container↗ too. The actor clicks on the button Drop. The dialog closes.

#### 2.3.6.4.4 Other sequences

If the actor wants to drop the code base out of Eclipse and delete the related code base too, he clicks on Delete.

#### 2.3.6.4.5 Postconditions

The selected code base and all depending test sessions and test cases are removed from the Test Sessions view.

If the actor has chosen Delete, the session container of the code base is deleted in the file system too.

#### 2.3.6.4.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

### 2.3.6.5 Use case: merge test sessions

#### 2.3.6.5.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.5.2 Preconditions

There must be at least two test sessions in Eclipse.

#### 2.3.6.5.3 Regular sequence

The actor activates the Test Sessions view (see figure 3.5). In this view, he selects the test session, he want to merge and clicks on the button Merge. The dialog Test Session Properties opens (similar to figure 3.6).

The actor puts in the name and the comment of the merged test session and clicks on the button OK. The dialog closes.

#### 2.3.6.5.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.6.5.5 Postconditions

A new test session with the specified name is created. All test case information from the test sessions selected for merge are copied into the new test session. The new test session is saved.

If some test cases have the same name, they are renamed to `test case name (session name 1)`, `test case name (session name 2)`.

The new test session appears in the list of the Test Sessions view. The new session and all its test cases are activated.

#### 2.3.6.5.6 Possible exceptions

If the actor has selected less than two test sessions, the dialog prohibits the click on the button Merge.

### 2.3.6.6 Use case: edit test session properties

#### 2.3.6.6.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.6.2 Preconditions

There must be at least one test session in Eclipse.

#### 2.3.6.6.3 Regular sequence

The actor activates the Test Sessions view (see figure 3.5), selects a test session and clicks on the button Properties. The dialog Test Session Properties opens (similar to figure 3.6).

The actor changes the name and/or the comment of the selected test session. After he has finished editing the properties, he clicks on the button OK. The dialog closes.

#### 2.3.6.6.4 Other sequences

There are no other sequences possible for this use case.

#### 2.3.6.6.5 Postconditions

The new name and the new comment of the test session are saved. The name of the test session changes in the list.

#### 2.3.6.6.6 Possible exceptions

The new name of the test session can not equal to a name of another test session in the session container. The dialog does not allow to save a duplicate name.

### 2.3.6.7 Use case: delete test session

#### 2.3.6.7.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.7.2 Preconditions

There must be at least one test session in Eclipse.

#### 2.3.6.7.3 Regular sequence

The actor opens the Test Sessions view (see figure 3.5), selects the related code base⁄ and the test session. Then he clicks on the button Delete.

#### 2.3.6.7.4 Other sequences

If the actor wants to drop more than one test session at once, he selects these test cases and clicks on the button Delete. A dialog opens, requesting the actor if he really wants to delete the selected test sessions. The actor clicks on the button Delete. The dialog closes.

#### 2.3.6.7.5 Postconditions

The selected test sessions are removed from the session container and from the Test Sessions view.

#### 2.3.6.7.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

### 2.3.6.8 Use case: merge test cases

#### 2.3.6.8.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.8.2 Preconditions

There must be a test session in Eclipse that contains at least two test cases that fit the merging criteria stated in section 2.1.

#### 2.3.6.8.3 Regular sequence

The actor opens the TEST SESSIONS view (see figure 3.5), selects the related code base↗, the test session and the test cases. Then he clicks on the button MERGE. The dialog TEST CASE PROPERTIES opens (see figure 3.6).

The actor puts in the name and the comment of the merged test case and clicks on the button OK. The dialog closes.

#### 2.3.6.8.4 Other sequences

The actor can also use the item MERGE in the context menu of the test cases.

#### 2.3.6.8.5 Postconditions

A new test case with the specified name is created. All information from the selected test cases are copied into the new test case. The test case is saved.

The new test case appears in list of the TEST SESSIONS view.

#### 2.3.6.8.6 Possible exceptions

If the actor has selected less than two test cases, the button MERGE TEST CASES is deactivated.

### 2.3.6.9 Use case: edit test case properties

#### 2.3.6.9.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.9.2 Preconditions

There must be at least one test case in a test session in Eclipse.

#### 2.3.6.9.3 Regular sequence

The actor opens the TEST SESSIONS view (see figure 3.5), selects the related code base↗, the test session and the test case. Then he clicks on the button PROPERTIES. The dialog TEST CASE PROPERTIES opens (see figure 3.6).

The actor changes the name and/or the comment of the selected test case. After he has finished editing the properties, he clicks on the button OK. The dialog closes.

#### 2.3.6.9.4 Other sequences

The actor can also use the menu item PROPERTIES in the test case's context menu in the TEST SESSIONS view.

#### 2.3.6.9.5 Postconditions

The new name and the new comment of the test case are saved. The name of the test case is updated in the list.

#### 2.3.6.9.6 Possible exceptions

If the actor has not selected exactly one test case, the button TEST CASE PROPERTIES is deactivated.

The new name of the test case can not equal to a name of another test case in the test session. The dialog does not allow to save a duplicate name.

### 2.3.6.10 Use case: delete test case

#### 2.3.6.10.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

#### 2.3.6.10.2 Preconditions

There must be at least one test case in a test session in Eclipse.

#### 2.3.6.10.3 Regular sequence

The actor opens the TEST SESSIONS view (see figure 3.5), selects the related code base↗, the test session and the test case. Then he clicks on the button DELETE. A dialog opens, requesting the actor if he really wants to delete the selected test cases. The actor clicks on the button DELETE. The dialog closes.

#### 2.3.6.10.4 Other sequences

The actor can also use the menu item DELETE in the test case's context menu in the TEST SESSIONS view.

#### 2.3.6.10.5 Postconditions

The selected test cases are removed from the list in the TEST SESSIONS view. The test cases are removed from the test session in the session container.

#### 2.3.6.10.6 Possible exceptions

There are no special possible exceptions to be considered for this use case.

## 2.3.7 Use case: generate report

### 2.3.7.1 Actor

The actor of this use case is the Eclipse user (see section 2.2).

### 2.3.7.2 Preconditions

There must be at least one test session in Eclipse.

### 2.3.7.3 Regular sequence

The actor selects a code base[/] and a set of test sessions in Test Sessions view (see figure 3.5). Then he clicks on the button Export. The Export dialog opens (see figure 3.9).

The current code base and the selected test sessions are preselected in the dialog, but the actor can also select another code base or other test sessions. He changes the Type to Report, chooses a destination and clicks on Next. The report dialog opens (see figure 3.10).

The actor clicks on the button Finish. The dialog closes.

### 2.3.7.4 Other sequences

The dialog for generating a report can also be opened by using the default Eclipse export dialog. For example, this can be done using the menu File and the item Export... . There the actor selects the item *CodeCover* Coverage Result Export in the group Other and clicks on Next.

### 2.3.7.5 Post conditions

A report is generated and stored in the chosen directory. The contents and appearance of the generated report is defined in section 2.6.

#### 2.3.7.6 Possible exceptions

If the report files could not be written – e.g. due to lack of access permissions or low disk space – an error message is shown to inform the actor.

## 2.4 Batch interface

### 2.4.1 Preface

To describe the use cases for the actor *shell user*, the shell commands are described. It is recommended that the executable `codecover` is contained in the `PATH` variable of the operating system.

The software can be run in the shell by calling either `codecover %command% [%options%]` or `codecover %option%`. All available commands are described in the rest of this section. Section 2.4.3 contains an overview about all supported commands.

If the actor wants to use `spaces` in arguments – e.g. file names – he must take care that the shell he is using parses the file name as one argument. For example he might enclose the file name using quotation marks: `"my file name.sql"`.

If the actor has called the software with an unsupported command, the software stops immediately and prints an error message:
`Command not supported. Use "codecover --help" for a command overview.`

If the actor has called the software with a supported command but a wrong option or syntactically wrong parameters, the software stops immediately and prints an error message like this:
`Wrong argument usage. Use "codecover help %command%" for options description.`

### 2.4.2 General options

Using the software without a command some general options are supported. These options can be used by: `codecover %option%`.

| Option | Explanation |
|---|---|
| --help\|-h | A help page containing this command overview. Has the same effect like `codecover help`. |
| --version\|-V | Prints out the version of the software. |

Table 2.1: General batch options

### 2.4.3 Command overview

For almost every command, either a long or a short version can be used. These commands can be used by `codecover %command% [%options%]`.

| Command | Description |
|---|---|
| instrumenter-info\|ii | Information of all available instrumenters |
| instrument\|in | Instrumentation↗ of code files↗ |
| analyze\|an | Analysis of a coverage run to create a test session |
| report\|re | Generating a report from a test session |
| info | Showing the information of a session container↗ and contained test sessions and test cases |
| merge-sessions\|ms | Merging two or more test sessions |
| alter-session\|as | Altering test session information |
| copy-sessions\|cs | Copy test sessions from one session container to another |
| remove-sessions\|rs | Removing test sessions from a session container |
| merge-test-cases\|mc | Merging two or more test cases |
| alter-test-case\|at | Editing test case information |
| remove-test-cases\|rt | Removing test cases from a session container |
| help\|h | A help page containing this command overview or an option and parameter overview a given command. |

Table 2.2: Batch command overview

### 2.4.4 Global command options

For every command a set of options is supported. In addition to specific command options, all commands support some global parameterless options. They are not required but change the behavior of the software when used. These commands can be used by `codecover %command% [%options%]`.

| **Option** | **Explanation** |
|---|---|
| --verbose\|-v | Orders the software to print more information as usual. For example this can be a description of the actions being done. This option is the opposite of --quiet. |
| --quiet\|-q | Orders the software not to print information to the shell. This option is the opposite to --verbose. |
| --pretend\|-p | Orders the software not to perform any actions affecting the data persistently but to print information about what the software would do instead. Using --pretend the actor can make sure that his command has the correct syntax and would be successfully executed. |
| --help\|-h | Prints an option and parameter overview of the given command. Has the same effect like `codecover help %command%`. |

Table 2.3: General batch options

### 2.4.5 Instrumenter-info

This command allows the actor to get information of the instrumenters, that are available by the software. Using this command, the actor can get to know, which instrumenter fits to his programming language and which additional options are supported.

This command is run by:

```
codecover (instrumenter-info|ii) [options]
```

| **Option** | | **Parameter and description** | **Default** |
|---|---|---|---|
| **Short** | **Long** | | if omitted |
| -l | --language | the name of the programming language | all |

| Option | | Parameter and description | Default |
|---|---|---|---|
| **Short** | **Long** | | if omitted |

Table 2.4: Options for command instrumenter-info

### 2.4.6 Instrument

This command instruments a set of code files. To select the code files, which should be instrumented, a `root-directory` must be specified. All code files must be located under this directory. For this reason a `default package` would be the best choice.

For a more detailed selection, include patterns can be used. These patterns allow wildcards and are adopted from the apache ant project[2] (see the pattern description[3]). The actor can specify more than one include pattern, to select the source files for instrumentation. The same way exclude patterns can be specified.

A file will be instrumented, if its relative path matches at least one include pattern, if it has the correct extension for the stated programming language and its relative path matches no exclude pattern.

This command is run by:

```
codecover (instrument|in) [options]
```

| Option | | Parameter and description | Required / Default |
|---|---|---|---|
| **Short** | **Long** | | |
| -r | --root-directory | the root directory of the source files | • |
| -l | --language | the name of the programming language | • |
| -d | --destination | the destination directory for the instrumented files | • |
| -c | --container | the new session container | • |
| -I | --instrumenter | the unique key of the instrumenter to use | |
| -a | --charset | the character encoding of the source files | system default |
| -i | --include | a relative include pattern; this argument can occur more than one time | all files |

[2] `http://ant.apache.org/`
[3] `http://ant.apache.org/manual/dirtasks.html`

| **Option** | | **Parameter and description** | **Required / Default** |
|---|---|---|---|
| **Short** | **Long** | | |
| -f | `--includes-file` | a file containing a list of relative include patterns separated by new line | |
| -e | `--exclude` | a relative exclude pattern; this argument can occur more than one time | |
| -x | `--excludes-file` | a file containing a list of relative exclude patterns separated by new line | |
| -o | `--criterion` | one of (`all`, `st`, `br`, `co`, `lo`); this argument can occur more than one time – once for every criterion | `all` |
| -u | `--copy-uninstrumented` | advices the software to copy all files of the root-directory, that were not instrumented, to the destination | disabled |
| -D | `--directive` | arguments of the style `key=value` to enable special features of the instrumenter; the instrumenter-info command should print out a list of directives, an instrumenter supports | |

Table 2.5: Options for command instrument

The arguments of option `criteria` stand for:

| **Criteria abbreviation** | **Explanation** |
|---|---|
| `all` | all criteria |
| `st` | statement coverage↗ |
| `br` | branch coverage↗ |
| `co` | condition coverage↗ |
| `lo` | loop coverage↗ |

Table 2.6: Explanation of the criteria abbreviations

An example call of the command `instrument` is:

```
codecover instrument --root-directory "C:\my files\project 1\" --include
```

```
"de\foo\pak1\**" --include "de\foo\pak2\dot*.java" --exclude "**\*Test.java"
-d "C:\my files\instrumented\" --language java --criterion st --criterion br
--copy-uninstrumented --container session-container-file.xml
```

The option `--copy-uninstrumented` can be used to get a replica of a source directory including resource files like images or configuration files.

If there is more than one instrumenter available for the specified programming language, the software aborts this instrumentation attempt. An error message is printed out like

> There is more than one instrumenter available for %programming language%. Please use the command `codecover instrumenter-info` to get to know the unique key of the instrumenter you prefer. Than use the option `instrumenter` for this command to exactly specify the instrumenter by its unique key.

Along with the instrumented code files, the instrumentation process produces a session container↗ containing the code base↗ and the MAST↗. The new code base↗ is has the date and time of the end of the instrumentation process. The code base is stored.

### 2.4.7 Analyze

This command is run by:

```
codecover (analyze|an) [options]
```

This command needs the session container produced by the instrumentation process and the coverage log↗ produced by the executed instrumented and compiled program.

| **Option** | | **Parameter and description** | **Required / Default** |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container the coverage data should be added to | • |
| -g | --coverage-log | the coverage log produced by the executed program | • |
| -n | --name | the name of the new test session containing the coverage results | • |
| -m | --comment | a comment describing the test session | empty |

| Option | | Parameter and description | Required / Default |
|---|---|---|---|
| **Short** | **Long** | | |
| -a | --charset | the character encoding of the coverage log file | system default |

Table 2.7: Options for command analyze

If the target session container does not exist, an empty session container is created at the specified target.

### 2.4.8 Report

This command is run by:

```
codecover (report|re) [options]
```

This command requires a test session, produced by the command `analyze` and a template file for the report generation.

| Option | | Parameter and description | Required |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | the name of the test session for the report | • |
| -p | --template | the template file containing transformation descriptions | • |
| -d | --destination | the destination for the report | • |

Table 2.8: Options for command report

The generated report file will be a HTML↗ file. The HTML file and a subdirectory containing other sources will be created.

### 2.4.9 Info

This command is run by:

```
codecover info [options]
```

This command shows information about a session container.

| Option | | Parameter and description | Required |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container | • |
| -s | --session | the name of a test session | |
| -T | --test-cases | showing test case information | |

Table 2.9: Options for command info

If no options are used, the program puts out a list of all sessions ordered by code base. The output can look like this:

```
user@rechner ~ >codecover info --container main.xml
codecover session container: "main.xml"

code bases and test sessions:
code base ID | session      | date       | time
-------------------------------------------------
12           |              | 21.10.2006 | 17:23:00
             | GUI          | 22.10.2006 | 20:14:03
             | Performance  | 22.10.2006 | 20:14:50
-------------------------------------------------
14           |              | 22.10.2006 | 21:12:00
             | Model I      | 22.10.2006 | 22:16:41
-------------------------------------------------
15           |              | 23.10.2006 | 08:11:00
             | Model II     | 24.10.2006 | 06:43:00
```

If the argument `--test-cases` is set, additionally to every session all test cases are put out.

If the test session name is set, the output is reduced just for the indicated test session. The output can look like this:

```
user@rechner ~ >codecover info --container main.xml --session "GUI test"
--test-cases
codecover session container: "main.xml"
session name:    GUI test
session comment: some clicks in the menu
session date:    22.10.2006
session time:    20:12:01

test cases:
name         | date       | time
-------------------------------------
menu file    | 22.10.2006 | 20:14:03
menu edit    | 22.10.2006 | 20:14:50
menu options | 22.10.2006 | 20:16:41
menu view    | 22.10.2006 | 20:17:13
menu help    | 22.10.2006 | 20:19:37
```

### 2.4.10 Merge-sessions

This command is run by:

```
codecover (merge-sessions|ms) [options]
```

With this command the actor can merge two or more test sessions in a session container↗ into a new test session (see section 2.1).

| **Option** | | **Parameter and description** | **Required / Default** |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | a name of a test session participating at the merging; this argument can occur more than one time – once for every participant | • |
| -R | --remove-old-test-sessions | indicates, whether or not the test sessions, that were merged, are removed after merging | |

| Option | | Parameter and description | Required / Default |
|---|---|---|---|
| **Short** | **Long** | | |
| -n | --name | the name of the merged test session | • |
| -m | --comment | a comment describing the merged test session | empty |

Table 2.10: Options for command merge-sessions

### 2.4.11 Alter-session

This command is run by:

```
codecover (alter-session|as) [options]
```

With this command the actor can change the information of a test session.

| Option | | Parameter and description | Required / Default |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | the old name of the test session | • |
| -n | --name | a new name of the test session | name not altered |
| -m | --comment | a new comment describing the test session | comment not altered |

Table 2.11: Options for command alter-session

### 2.4.12 Copy-sessions

This command is run by:

```
codecover (copy-sessions|cs) [options]
```

With this command the actor can copy one or more test sessions from a session container to another.

| Option | | Parameter and description | Required |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the source session container | • |
| -s | --session | a name of a test session participating at the copy; this argument can occur more than one time – once for every participant | • |
| -d | --destination | the destination session container | • |

Table 2.12: Options for command copy-sessions

If the destination session container does not exist, a copy of the source session container containing only the defined sessions is created at the specified destination.

### 2.4.13 Remove-session

This command is run by:

```
codecover (remove-sessions|rs) [options]
```

With this command the actor can remove one or more test sessions and their test cases from a session container.

| Option | | Parameter and description | Required |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to remove from | • |
| -s | --session | the name of the test session to be removed; this argument can occur more than one time – once for every test session | • |

Table 2.13: Options for command remove-sessions

### 2.4.14 Merge-test-cases

```
codecover (merge-test-cases|mt) [options]
```

With this command the actor can merge two or more test cases into one test case. These test cases must be in one test session and must fit the merging criteria stated in section 2.1.

| **Option** | | **Parameter and description** | **Required / Default** |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | the name of the test session | • |
| -t | --test-case | a name of a test case participating at the merging; this argument can occur more than one time – once for every participant | • |
| -R | --remove-old-test-cases | indicates, whether or not the test cases, that were merged, are removed after merging | |
| -n | --name | the name of the merged test case | • |
| -m | --comment | a comment describing the merged test case | empty |

Table 2.14: Options for command merge-test-cases

### 2.4.15 Alter-test-case

```
codecover (alter-test-case|at) [options]
```

With this command the actor can change a test case (see section 2.1).

| **Option** | | **Parameter and description** | **Required / Default** |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | the name of the test session | • |
| -t | --test-case | the old name of the test case | • |
| -n | --name | the new name of the test case | new name ignored |
| -m | --comment | a new comment describing the test case | new comment ignored |

Table 2.15: Options for command alter-test-case

### 2.4.16 Remove-test-cases

This command is run by:

```
codecover (remove-test-cases|rt) [options]
```

With this command the actor can remove one or more test cases of a test session from a session container.

| Option | | Parameter and description | Required |
|---|---|---|---|
| **Short** | **Long** | | |
| -c | --container | the session container to use | • |
| -s | --session | the name of the test session | • |
| -t | --test-case | the name of the test case to be removed; this argument can occur more than one time – once for every test case | • |

Table 2.16: Options for command remove-test-cases

### 2.4.17 Help

```
codecover (help|h) [%command%]
```

Using the command `help` without using an optional command, the program prints a help page containing the command overview (see section 2.4.3).

If a command is given, the program prints an option and parameter overview of the given command. Has the same effect like `codecover` %command% `--help`.

## 2.5 Configuration

### 2.5.1 Overview

Eclipse creates files containing preferences and other information at the run time. These files are stored in the default folders defined for Eclipse plug-ins. They can be separated into global preferences and project wide properties. Files to store are:

- preferences for the plug-in
- properties for every Eclipse project↗
- stored session containers with test sessions
- instrumented code files↗
- compiled instrumented code files

There are several options which control the behavior and the appearance of the software. For the Eclipse user, there is the dialog PREFERENCES for Eclipse wide configuration and PROPERTIES for project wide configuration. (see section 2.5.2)

For the batch mode there is a default configuration in the release jar, which can not be altered by the shell user but can be overwritten for each batch run by setting options on the command line.

The report style is configured by template XML files (see section 2.6), which can be processed using the batch mode.

### 2.5.2 Configure Eclipse plug-in

To configure the behavior and the general appearance of the Eclipse plug-in, a configuration file is stored. The Eclipse default mechanism for storing plug-in preferences is used. The target folder will be a sub folder of the `.metadata` folder in the Eclipse workspace.

The Eclipse user can use the Eclipse dialog PREFERENCES to edit the Eclipse-wide plug-in preferences of the software. The Eclipse user clicks on the menu WINDOW and the entry PREFERENCES.... In the Eclipse configuration dialog, there is an entry *CodeCover* – the configuration section of the software. (see section 3.9)

On the corresponding dialog page, the user can configure the following preferences:

| **Configurable property** | **Available options** |
|---|---|
| colors of the source code highlighting | for each code element – covered, partly covered and not covered – one color from a color chooser and an enable button |

Table 2.17: Configurable properties of Eclipse

To configure project properties, the actor uses the Eclipse dialog PROPERTIES (see sec-

tion 3.10). These preferences are also stored using common Eclipse preference methods for plug-ins.

In the dialog PROPERTIES there is a section *CodeCover* where following properties can be configured:

| **Configurable property** | **Available options** |
|---|---|
| coverage criteria | a not empty multiple selection out of statement coverage↗, branch coverage↗, condition coverage↗, loop coverage↗ |

Table 2.18: Configurable properties of an Eclipse project

## 2.6 Report

### 2.6.1 HTML

#### 2.6.1.1 Overview

The hierarchic HTML↗ report consists of a set of HTML files placed into a directory tree. The HTML files contain the results of the coverage measurement.

There are three different types of HTML files: code pages, selection pages and title pages. These types are in a hierarchical order: the top-most page is the title page, followed by a number of selection pages. The bottom-most page type is the code page. Depending on the programming language the SUT↗ is written in, the depth of this structure may vary.

Each page type contains a lexicographically ordered list of elements with the coverage results measured for this element. Results for each coverage criterion↗ are always shown in two columns: in the first column, the number of covered items (e.g. branches) and the total number of items is written, separated by a slash, and in the second column the percentage of coverage is written with a colored bar graph visualizing this percentage.

#### 2.6.1.2 Title page

Each report has exactly one title page named `index.html` that shows a summary of the measured coverage of the whole project↗ (as far as it was instrumented).

This summary is followed by a list of metrics, number of instrumented packages, classes and methods.

The next element of the title page is a list as described in subsection *Overview*. It contains the top-most structural elements the programming language of the SUT provides. Each of these elements is a link to a file on the next deeper level. If the language only has two hierarchical levels, that file is a code page, otherwise it is a selection page.

In the following the title page shows an overview of the test cases↗. If JUnit test cases were used, the test case overview is enriched with these information. Here is an example of this overview:

| | |
|---|---|
| **Number of test cases** | 8 |
| **Number of JUnit test cases** | 6 |
| **Number of failures** | 3 |
| **Number of errors** | 1 |

Table 2.19: Draft of the test case overview at the report title page

| Date | Test case name | Comments |
|---|---|---|
| 2007-03-13 15:43:02 | GUI test 1 | |
| 2007-03-13 18:09:18 | GUI test 2 | |
| 2007-03-13 21:55:57 | Black box test | |
| 2007-03-13 23:01:20 | tests.MoneyTest.testMoney1 | |
| 2007-03-13 23:01:22 | tests.MoneyTest.testMoney2 | **failure**<br>AssertionFailedError at MoneyTest.java:23 |
| 2007-03-13 23:06:28 | tests.PersonTest | tests.PersonTest.testSetName<br>tests.PersonTest.testSetSalary<br>**error**<br>ArithmeticException at Person.java:45 |
| 2007-03-13 23:06:29 | tests.DatabaseTest | tests.DatabaseTest.testLoad<br>tests.DatabaseTest.testStore<br>**failure**<br>AssertionFailedError at DatabaseTest.java:57<br>tests.DatabaseTest.testCommit |

| | | |
|---|---|---|
| 2007-03-13 23:06:44 | tests.FileTest | tests.FileTest.testImport<br>**error**<br>AssertionFailedError at FileTest.java:21<br>tests.FileTest.testExport |

Table 2.20: Draft of the test case table at the report title page

The first three test cases were captured out of JUnit. For the fourth and the fifth test case (tests.MoneyTest.testMoney) test methods of JUnit test cases were used as test cases. The other test cases are equal to the JUnit test cases. To allow a more detailed inspection, their test methods are show too. In the column *Failures and Errors* JUnit failures and errors are listed for JUnit test cases and test methods.

#### 2.6.1.3 Selection page

Each selection page belongs to one element of the SUT at one hierarchical level. In Java, e.g., one selection page could belong to the package *Package A*. A selection page starts with a link to the file belonging to the next-higher level, which can be the title page or another selection page, and the overview of the element's name and coverage results measured for it. The rest of the page is a list as described above with the structural elements on the next deeper level. Each of these elements is linked to a file of the next deeper level. It the level of the current page is last but one, the linked file is a code page, otherwise it is a selection page of the next deeper level.

#### 2.6.1.4 Code page

Each code page starts with a link to the file belonging to the next higher level, which can be the title page or a selection page, the name of the current element and the overview of the coverage results for the current element (e.g. a class in Java).

At the bottom of the page is the source code belonging to this element. If condition coverage↗ is activated, a table with the covered boolean expressions is written after each condition.

Between the code and the overview stands a list as described above with the deepest structural elements of the programming language provides (e.g. methods in Java). Each item of the list provides a link to an anchor in the corresponding line in the code.

#### 2.6.1.5 Implementation for Java and COBOL

For Java software, the title page lists the packages of the project↗ and links to selection pages. Each selection page belongs to a package. This selection pages link to code pages. Each code page belongs to a class. In each code page is a list of the methods in the corresponding class.

For COBOL software, the title page lists the sections of the program. There are no selection pages. Each entry of the list on the title page links to one code page. The code pages contain no list, because there are no next-deeper elements than sections.

## 2.7 Instrumentation, types of coverage and measurement

### 2.7.1 Instrumentation process

The instrumentation↗ process uses instrumentable files as input, adds additional counters at all code elements of interest and saves the instrumented code file↗ in a given target folder. Where these counters are placed and how they are used must be clarified in the software design.

When using Eclipse for the instrumentation process, all instrumented code files of a Eclipse project are persistently saved in a sub folder of the plug-in's properties folder of the project. Moreover the compiled instrumented code files are stored in a bin folder too.

Saving these already instrumented and compiled files allows Eclipse to execute an entry point↗ of the Eclipse project and using an already created code base↗. So not every instrumentation process creates a new code base↗.

Code files that are not Used For Coverage Measurement (see section 2.3.4.1) are compiled into a folder, where the compiled instrumented files are stored as well. Moreover, non-code files of the original source folder are copied too. This is done to emulate the original bin folder when measuring the coverage. If these files are too big, meaning there is insufficient free disk space, the standard message for insufficient disk space is shown in Eclipse (if the Actor uses the Eclipse plug-in) or an error message is written in the console to informs the user about this problem asking him to resolve it.

This can also happen if the space is not enough to compile the instrumented source files or to write the report.

### 2.7.2 Statement coverage

*Statement coverage* is defined in the glossary shipped with this specification↗. Also *basic statement*↗ is defined there generically.

For Java *basic statement* is according to the Java Grammar[4]:

- return [Expression] ;
- throw Expression ;
- StatementExpression ;
- break [Identifier] ;
- continue [Identifier] ;

In COBOL, everything that is (according to the COBOL grammar for JavaCC[5]) matched by `void Statement()` except `void IfStatement()` and `void PerformStatement()` is counted as a statement.

In general, statement coverage is defined as a percentage that is calculated as follows for the instrumented part of the SUT:

$$\frac{\text{number of covered basic statements}}{\text{total number of basic statements}}$$

### 2.7.3 Branch coverage

*Branch coverage* is defined in the glossary shipped with this specification.

If the programming language of the SUT supports exception handling, the branches implied by the *possible* exceptions are excluded from the branch coverage↗ calculations but explicit TRY-CATCH BLOCKS are treated as branches: one for running without an exception and one for every catch statement.

Branch coverage is defined as a percentage that is calculated as follows for the instru-

[4] `http://java.sun.com/docs/books/jls/third_edition/html/syntax.html`
[5] `http://mapage.noos.fr/~bpinon/cobol.jj`

mented part of the SUT:

$$\frac{\text{number of covered branches}}{\text{total number of branches}}$$

## 2.7.4 Condition coverage

### 2.7.4.1 General view

*Condition coverage* and *strict condition coverage*↗ are defined in the glossary shipped with this specification.

The software uses the strict condition coverage but it is intended that other condition coverage criteria can be adapted with small effort (see section 4.11).

In general, strict condition coverage is defined as a percentage that is calculated as follows for the instrumented part of the SUT:

$$\frac{\text{number of covered basic boolean terms}}{\text{total number of basic boolean terms}}$$

There are some characteristics handling condition coverage for the Java and COBOL programming language. They are considered in the next sections.

### 2.7.4.2 Short-circuit operators

Some languages, e.g. Java, provide so called short-circuit boolean operators: the operands are only evaluated as far as they could affect the result of the whole expression. For example, as `||` is the short-circuit logical OR operator in Java, if `A` in `(A || B)` is true, `B` is not evaluated at all. To cover `A`, it must be once false while `B` is false, and once true while the value of `B` does not matter since it is not evaluated.

If the normal logical OR operator `|` was used, `(A | B)`, `B` would be required to stay false in both cases for `A` to be covered.

### 2.7.4.3 Java ternary operator

There are two different cases in handling the Java ternary operator `(A ? B : C)` in conditional expressions.

If the operator is used as a boolean term in a conditional expression, as in

```
if (x > 5 ? isA() : y == 7) {...},
```

`A`, `B` and `C` are considered *separate* coverable items. The use in the conditional expression implies that `B` and `C` are boolean expressions themselves. The coverage is determined based on the same criteria as described above: the whole expression must change if the covered basic boolean term is changed, while the other boolean terms stay the same, as far as they are evaluated. That is:

- to cover `A`, `B` has to be the opposite of `C`, otherwise the change of A would not affect the whole expression,
- to cover `B`, `A` must stay true, `C` may have any value since is not evaluated and `B` must evaluate both to true and false,
- to cover `C`, `A` must stay false, `B` may have any value since it is not evaluated and `C` must evaluate both to true and false.

In all other cases, the ternary operator does not affect the condition coverage. For example, in

```
if (i == (foo ? 2 : 3)) {..},
```

the expression `i == (foo ? 2 : 3)` is a *single* basic boolean term.

#### 2.7.4.4 COBOL boolean abbreviations

COBOL provides abbreviations in boolean expressions, e.g. `IF A = 3 OR = 7`. These abbreviations are converted to their long form, in the example `IF A = 3 OR A = 7`, and checked the usual way: to achieve full strict condition coverage, A = 3, A = 4 and A = 7 would be sufficient.

### 2.7.5 Loop coverage

*Loop coverage* is defined in the glossary shipped with this specification.

Loop coverage does not consider elements of the source code as coverable items↗ but the number of times the loop body is entered. The coverable items are:

- loop body is not entered
- loop body is entered once, but not repeated
- loop body is repeated more than one time

Looping statements like do-while cannot be bypassed and have only two possible coverable items.

In general, loop coverage is define as a percentage that is calculated as follows for the instrumented part of the SUT:

$$\frac{\text{number of covered coverable items}}{\text{total number of coverable items}}$$

### 2.7.6 Coverage measurement

The process of the coverage measurement needs the instrumented code files↗. They are compiled together with the uninstrumented code files. When the instrumented SUT is executed, a coverage log↗ is produced. This log contains counters for all instrumented statements and code elements.

The name of the log file can have one of the following formats:

`coverage-log.clf`

or

`coverage-log-yyyy-MM-dd-HH-mm-ss-SSS.clf`
The date and time refer to the start of the coverage measurement. An example is `coverage-log-2007-06-07-09-48-12.clf`.

If a file with the given name still exists, it is not overwritten, but the name of the new file is extended by `(1)`, `(2)` and so on. The instrumenter can make the instrumented SUT to support additional parameters that the tester can specify a path of the coverage log file or enable overwriting – e.g. environment variables or system properties for Java.

### 2.7.7 Coverage analysis

The coverage log↗ file is processed in the analysis period. For this purpose, the Eclipse user can use the *analyze coverage log* use case (see section 2.3.4.6) or the complete instrumentation↗, execution and analysis use case (*measure coverage*, section 2.3.7). The shell user can use the command `codecover analyze` (see section 2.4.7).

The result of the analysis process is a test session, that could be used for report generation (see section 2.3.7) or coverage analysis (see section 2.3.5.2).

## 2.8 Language support

The software can show all texts used in the Eclipse plug-in as well as in the reports in any language of which all needed characters are part of the Unicode standard. All releases will be delivered in German and English localizations. The default Eclipse language support for plug-ins is used.

The Eclipse plug-in tries to use the language Eclipse uses, taking English if it can't find an appropriate language setting. The language used in reports, on the other hand, results from the template files which not only define the layout but also set every string used in the report outputted by *CodeCover*.

The batch interface is English only.

## 2.9 JUnit integration

### 2.9.1 Preface

The term *test case* can be mixed up in this section. For this reason we distinguish the terms *JUnit test case* and test case$^{\nearrow}$ in the understanding of this software.

The phrase *a test case failed* will be used in the meaning that a test case had an unexpected behaviour or causes an error.

### 2.9.2 Basic concepts

Till now, only the manually added method calls can set the start and the end of a test case:

```
Protocol.startTestCase("JUnit Test 1")
```

This rudimentary test case notification mechanism for Java will be enhanced. Therefore JUnit[6] will be integrated in the software. Thereby the software is informed about the start and the end of JUnit test cases while the instrumented SUT is running.

[6] `http://www.junit.org`

It must be configurable whether to use the JUnit test cases or the test methods as test cases in the understanding of the software. This must be decidable for each JUnit test run. The test cases of the SUT need not to be instrumented or changed for this feature.

To support these features, the software must log static information of the JUnit test cases and observe its run. This includes for each test case:

- the name of the related JUnit test case class
- the names of the test methods of the JUnit test case
- whether the test methods of the JUnit test case failed or not
- if a test method failed, which failure respectively error was the reason
- the date and time of the execution
- the belonging code coverage⟋ results

If a JUnit test case is used as a test case, the test case has to store all test methods of the JUnit test case. The related test case is marked as failed if at least one test method hast failed.

If the test methods of a JUnit test case are used as test cases, the test case name must have an unique name to identify the test method.

All the data collected must be stored in the coverage log⟋ file, cause this is the only result of the coverage measurement phase.

### 2.9.3 Compatibility

The approach must be compatible to:

- the JUnit 3.8.x family
- the JUnit 4.x family
- the Eclipse plug-in family: `org.junit_3.8.x`
- the Eclipse plug-in family: `org.junit4_4.x`

This means, that the JUnit integration works with all four families and allows the features described above.

The implementation of the JUnit integration must be compatible to the Java version of the supported JUnit family – e.g. the implementation of the support for JUnit 3.8.x must be compatible to Java 1.4. For this reason the software supports test case execution for older systems, that still rely on Java 1.4.

### 2.9.4 Report

The additional JUnit test case information are added to the report. See section 2.6.1.2.

## 2.10 ANT integration

### 2.10.1 Preface

*CodeCover* provides an Apache ANT[7] integration.

The *CodeCover* ANT integration will provide a `codecover` command which will have subcommands as its content, i.e. an example will look like:

```
<target name="foo">
  <codecover>
    <subcommand1 param1="bar" param2="foobar" />
    <subcommand2>
      <someElement param="42" />
    </subcommand2>
  </codecover>
</target>
```

### 2.10.2 Subcommand overview

The following subcommands are available:

| **Target** | **Description** |
|---|---|
| load | load a session container |

[7] `http://ant.apache.org/`

| Target | Description |
|---|---|
| save | save a session container |
| createContainer | creates a new container |
| instrument | Instrumentation↗ of code files↗ |
| analyze | Analysis of a coverage run to create a test session |
| report | Generating a report from a test session |
| mergeSessions | Merging two or more test sessions |
| alterSession | Altering test session information |
| copySessions | Copy test sessions from one session container to another |
| removeSessions | Removing test sessions from a session container |
| mergeTestCases | Merging two or more test cases |
| alterTestCase | Editing test case information |
| removeTestCases | Removing test cases from a session container |

### 2.10.3 load

This subcommand loads a session container. The loaded session container then can be used in future subcommands.

Syntax:

```
<load containerId="..." filename="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | An ID assigned to the loaded container. This ID can later be used to reference this container. |
| `filename` | • | The file to load. If `filename` is a relative filename, it will be interpreted relative to the project's basedir. |

### 2.10.4 save

This subcommand saves a session container.

Syntax:

```
<save containerId="..." filename="..." override="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container which will be saved. |
| `filename` | • | The name of the new file. If `filename` is a relative filename, it will be interpreted relative to the project's basedir. |
| `override` | | If this attribute is true, an existing file will be overridden. Defaults to true. |

### 2.10.5 createContainer

This subcommand creates a new test session container using the static information from another container.

Syntax:

```
<createContainer oldContainerId="..." newContainerId="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `oldContainerId` | • | The ID of a container. |
| `newContainerId` | • | An ID assigned to the newly created container. This ID can later be used to reference this container. |

### 2.10.6 instrument

This subcommand instruments source code.

Syntax:

```
<instrument containerId="..." language="..." instrumenter="..."
    destination="..." charset="..." copyUninstrumented="..." override="...">
  <source ...>
    ...
  </source>
```

```
  <criteria>
    <criterion name="..." />
    <criterion name="..." />
    ...
  </criteria>
</instrument>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | An ID assigned to the newly created container. This ID can later be used to reference this container. |
| `language` | • | The name of the programming language. |
| `instrumenter` | • | The full name of the instrumenter to use. **TODO:** This appears in the batch specification; it however is not implemented. What is this for? Is this option really required? Yes! |
| `destination` | • | The destination directory for the instrumented files. If `destination` is a relative filename, it will be interpreted relative to the project's basedir. |
| `charset` | | The character encoding of the source files. If none is given, the system default will be used. |
| `copyUninstrumented` | | If this attribute is true, all non-instrumented files in the root directory will be copied to the destination. Defaults to false. |
| `override` | | If this attribute is true, existing files will be overridden. Defaults to true. |

| Element | Required | Description |
|---|---|---|
| `source` | • | A fileset[8] pointing to the files to instrument. The root directory of the fileset has to point to the root directory of the source files. |
| `criteria` | | A list of criteria to be used for instrumentation. If this element isn't given, all criteria will be used. |

[8] `http://ant.apache.org/manual/CoreTypes/fileset.html`

### 2.10.7 analyze

This subcommand takes the information from a coverage log and writes it into a session container.

Syntax:

```
<analyze containerId="..." coverageLog="..."
    name="..." comment="..." charset="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container in which the information will be written. |
| `coverageLog` | • | The coverage log file. If `coverageLog` is a relative filename, it will be interpreted relative to the project's basedir. |
| `name` | • | The name of the new test session. |
| `comment` | | The comment for the new test session. If none is given, the comment is empty. |
| `charset` | | The character encoding of the coverage log. If none is given, the system default will be used. |

### 2.10.8 report

This command creates a report containing information about test cases.

Syntax:

```
<report containerId="..." destination="..."
    template="..." override="...">
  <testCases>
    <testSession name="...">
      <testCase pattern=".*" />
    </testSession>
    <testSession pattern="foo.*bar">
```

```
      <testCase pattern=".*" />
    </testSession>
    <testSession name="...">
      <testCase name="..." />
      <testCase name="..." />
      <testCase name="..." />
    </testSession>
  </testCases>
</report>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container in which the information will be written. |
| `destination` | • | The destination file for the report. If `destination` is a relative filename, it will be interpreted relative to the project's basedir. |
| `template` | • | The template file. If `template` is a relative filename, it will be interpreted relative to the project's basedir. |
| `override` | | If this attribute is true, existing files will be overridden. Defaults to true. |

| Element | Required | Description |
|---|---|---|
| `testCases` | • | The test cases to use in the report. This element is described in 2.10.17 |

## 2.10.9 mergeSessions

This command merges multiple sessions in a session container into one session.

Syntax:

```
<mergeSessions containerId="..." name="..." comment="..."
    removeOldSessions="...">
  <testSessions>
```

```
    <testSession name="..." />
    <testSession pattern="foo.*bar" />
  </testSessions>
</mergeSessions>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container of the used container. |
| `name` | • | The name of the new test session. |
| `comment` | | The comment for the new test session. If none is given, the comment is empty. |
| `removeOldSessions` | | If this attribute is true, the original test sessions will be removed. Defaults to false. |

| Element | Required | Description |
|---|---|---|
| `testSessions` | • | The test sessions to merge. This element is described in 2.10.16 |

### 2.10.10 alterSession

This command modifies the name and/or the comment of a test session.

Syntax:

```
<alterSession containerId="..." session="..." name="..." comment="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container of the used container. |
| `session` | • | The old name of the test session. |
| `name` | | The new name of the test session. If none is given, the name will not be changed. |
| `comment` | | The new comment for the test session. If none is given, the comment will not be changed. |

### 2.10.11 copySessions

This command will copy session from one session container into another.

Syntax:

```
<copySessions sourceContainerId="..." destinationContainerId="..."
    removeOldSessions="...">
  <testSessions>
    <testSession name="..." />
    <testSession pattern="foo.*bar" />
  </testSessions>
</copySessions>
```

| Attribute | Required | Description |
|---|---|---|
| `sourceContainerId` | • | The ID of the container of copy from. |
| `destinationContainerId` | • | The ID of the container to copy to. |

| Element | Required | Description |
|---|---|---|
| `testSessions` | • | The test sessions to copy. This element is described in 2.10.16 |

### 2.10.12 removeSessions

This command remoces sessions from a session container.

Syntax:

```
<removeSessions containerId="...">
  <testSessions>
    <testSession name="..." />
    <testSession pattern="foo.*bar" />
  </testSessions>
</removeSessions>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container to remove sessions from. |

| Element | Required | Description |
|---|---|---|
| `testSessions` | • | The test sessions to remove. This element is described in 2.10.16 |

### 2.10.13 mergeTestCases

This command merges multiple test cases in a session into one test case.

Syntax:

```
<mergeTestCases containerId="..." name="..." comment="..."
    removeOldTestCases="...">
  <testCases>
    <testSession name="...">
      <testCase pattern="foo.*bar" />
      <testCase name="..." />
      <testCase name="..." />
      <testCase name="..." />
    </testSession>
  </testCases>
</mergeTestCases>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container of the used container. |
| `name` | • | The name of the new test case. |
| `comment` | | The comment for the new test case. If none is given, the comment is empty. |
| `removeOldTestCases` | | If this attribute is true, the original test cases will be removed. Defaults to false. |

| Element | Required | Description |
|---|---|---|
| `testCases` | • | The test cases to merge. This element is described in 2.10.17. The list has to contain exactly one test session. This session will also contain the new test case. |

### 2.10.14 alterTestCase

This command modifies the name and/or the comment of a test case.

Syntax:

```
<alterTestCase containerId="..." session="..." testCase="..." name="..."
    comment="..." />
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container of the used container. |
| `session` | • | The name of the test session. |
| `testCase` | • | The old name of the test case. |
| `name` | | The new name of the test case. If none is given, the name will not be changed. |
| `comment` | | The new comment for the test case. If none is given, the comment will not be changed. |

### 2.10.15 removeTestCases

This command remoces sessions from a session container.

Syntax:

```
<removeTestCases containerId="...">
  <testCases>
    <testSession name="...">
      <testCase pattern=".*" />
    </testSession>
    <testSession pattern="foo.*bar">
```

```
      <testCase pattern=".*" />
    </testSession>
    <testSession name="...">
      <testCase name="..." />
      <testCase name="..." />
      <testCase name="..." />
    </testSession>
  </testCases>
</removeTestCases>
```

| Attribute | Required | Description |
|---|---|---|
| `containerId` | • | The ID of a container to remove test cases from. |

| Element | Required | Description |
|---|---|---|
| `testCases` | • | The test cases to remove. This element is described in 2.10.17 |

### 2.10.16 List of test sessions

A `testSessions` element contains a list of `testSession` elements. Each `testSession` element has either a `name` attribute, which gives the exact name of the test session it matches, or a `pattern` attribute, which gives a Java regular expression which test sessions names it will match.

So e.g.

```
<testSessions>
  <testSession name="42" />
  <testSession name="23" />
  <testSession pattern="foo.*bar" />
</testSessions>
```

will match the session with the name “42”, the session with the name “23” and any session starting with “foo” and ending with “bar”.

### 2.10.17 List of test cases

A `testCases` element contains a list of `testSession` elements (as described above), however here each `testSession` element contains a list of `testCase` elements. Each `testCase` element has either a `name` attribute, which gives the exact name of the test case it matches, or a `pattern` attribute, which gives a Java regular expression which test cases names it will match.

So e.g.

```
<testCases>
  <testSession name="foo">
    <testCase pattern=".*" />
  </testSession>
  <testSession pattern="foo.*bar">
    <testCase pattern=".*" />
  </testSession>
  <testSession name="42">
    <testCase name="1" />
    <testCase name="2" />
    <testCase name="3" />
  </testSession>
</testCases>
```

will contain all test cases from the test session “foo”, all test cases from test sessions which names start with “foo” and end with “bar” and the test cases “1”, “2” and “3” from the test session “42”.

### 2.10.18 Examples

#### 2.10.18.1 Instrumentation

```
<codecover>
  <instrument containerId="container" language="java"
     destination="instrumented" charset="utf-8" copyUninstrumented="true">
    <source dir="src">
```

```
      <include name="**/*.java" />
    </source>
    <criteria>
      <criterion name="st" />
      <criterion name="br" />
    </criteria>
  </instrument>
  <save containerId="container" filename="container.xml" />
</codecover>
```

This will instrument all java files in the directory `src`, write the result into `instrumented` and use statement and branch coverage. The resulting test session container will be written into `container.xml`.

#### 2.10.18.2 Analysis

```
<codecover>
  <load containerId="container" filename="container.xml" />
  <analyze containerId="container" coverageLog="coverage.log"
      name="New Test Session" />
  <save containerId="container" filename="container.xml" />
</codecover>
```

This will write the content of `coverage.log` into `container.xml` into a new test session called “New Test Session”.

#### 2.10.18.3 Reporting

```
<codecover>
  <load containerId="container" filename="container.xml" />
  <report containerId="container" destination="report.html"
      template="HTML_Report_hierarchic.xml">
    <testCases>
      <testSession pattern=".*">
        <testCase pattern=".*" />
      </testSession>
    </testCases>
```

```
  </report>
</codecover>
```

This will create a report in `report.html` containing all test cases from the test session container `container.xml` using the template in `HTML_Report_hierarchic.xml`.

## 2.11 Live Test Case Notification

The live test case notification feature provides a way to manually define test case borders during the execution of the instrumented SUT without manually modifying either the test cases or the SUT before instrumenting or compiling. Additionally, this feature can be used to automatically download the created coverage log files from a remote SUT, for example, a web application.

Live test case notification is available for Java SUTs only.

The communication between the instrumented SUT and *CodeCover* is carried out using the Java Management Extensions (JMX)[9] remote protocol over Java RMI over TCP. As JMX is generally protocol-independent, support for other protocols can be added in the future; this is, however, beyond the scope of the *CodeCover* project.

### 2.11.1 Basic principle of operation

JMX uses a client/server model to enable management and monitoring of a Java application. The target Java application (the SUT) acts as a server. As long as the application is being executed, one or more clients can connect to it, query property values and initiate operations exposed by the server. The communication model and infrastructure is defined in the JMX specification and implemented in most major JRE Version 5 or later distributions as well as in both open source and commercial products and libraries. The target application only needs to provide its specific management functionality by creating objects that adhere to a particular interface and naming convention (called *MBeans* in JMX terminology) and registering them with a *MBean Server* which is provided by the JMX implementation.

[9] `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`

During the instrumentation process of the SUT, a MBean class is added to the SUT along with the standard measurement classes. When the SUT starts and the coverage logging facilities are initialized, an object of this class is created and registered with the MBean Server. If the SUT was started in a JVM which supports remote JMX, operations and properties of the *CodeCover* MBean are automatically exported to the network interface. At the time of this writing, remote JMX is not enabled by default but can be simply enabled by setting special system properties at startup of the JVM. The procedure of enabling remote JMX on major JREs is to be documented in the user's manual.

The *CodeCover* client, which is a part of the *CodeCover* Eclipse plug-in, connects to the JMX server in the SUT's JVM and executes operations on the MBean as requested by the Eclipse user. These operations trigger test case notification information to be written into the coverage log file.

It is assumed that only one client accesses the instrumented SUT via remote JMX. Concurrent accesses are not supported as they wouldn't have any meaningful semantics.

### 2.11.2 MBean Interface

The MBean exports the following properties and operations to remote clients:

1. An operation to start a test case with a name and to end a test case. These operations correspond to methods described in section 2.1.
2. An operation which instructs the measurement classes to finish the coverage logging and close the coverage log file.
3. A read-only property that contains the file name of the current coverage log file.
4. An operation which allows the client to download the contents of the current coverage log file.

### 2.11.3 Application life cycle issues

#### 2.11.3.1 Java SE applications

In standard Java SE applications, the MBean is initialized together with the rest of the coverage logging classes.

#### 2.11.3.2 Web applications

The MBean must be registered with the MBean Server when the web application is initialized and unregistered when the application shuts down. Since the application startup and shutdown times are not necessarily identical with the application container's ones, some interaction with the application container is necessary to get the notifications on startup and shutdown.

This is done by installing a *context listener* into the web application context. Context listeners are part of the servlet specification[10] since version 2.3. *CodeCover* provides a generic context listener class which registers and unregisters the required MBean.

The user is only required to add a generic context listener declaration to the deployment descriptor (`web.xml`) of the SUT. This process is described in the user's manual.

Moreover, as the MBean is initialized with the application and a MBean Server might be initialized together with the container, a connected client is required to listen for MBean registration and deregistration events on the MBean Server and behave accordingly to the current MBean status.

[10] `http://java.sun.com/products/servlet/index.jsp`

# 3 Graphical User Interface

## 3.1 Package and file states

The instrumentable items↗ (e.g. packages or source code files↗) are presented in one of two states: *normal* and *used for coverage measurement*. The following table shows the icons for different instrumentable items in these states.

| Object | normal | used for coverage measurement |
|---|---|---|
| Package |  |  |
| Java file |  |  |

The user can change the state of an instrumentable item by selecting the Use For Coverage Measurement menu item in the context menu of an instrumentable item (e.g. in the JDT's Package Explorer or the generic Navigator). Use for Coverage Measurement is a check box menu item so that a second selection of this menu item removes the instrumentable item from coverage measurement.

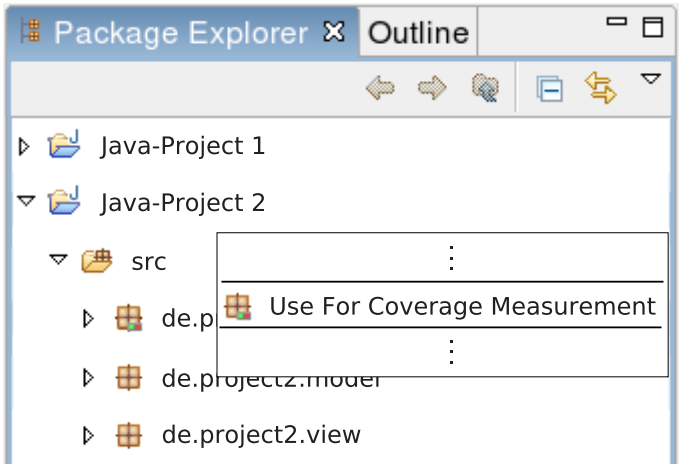


Figure 3.1: Package selection

## 3.2 Instrumentation

The INSTRUMENT PROJECT... item is added to the PROJECT menu. This command opens the dialog window which is shown in figure 3.2.

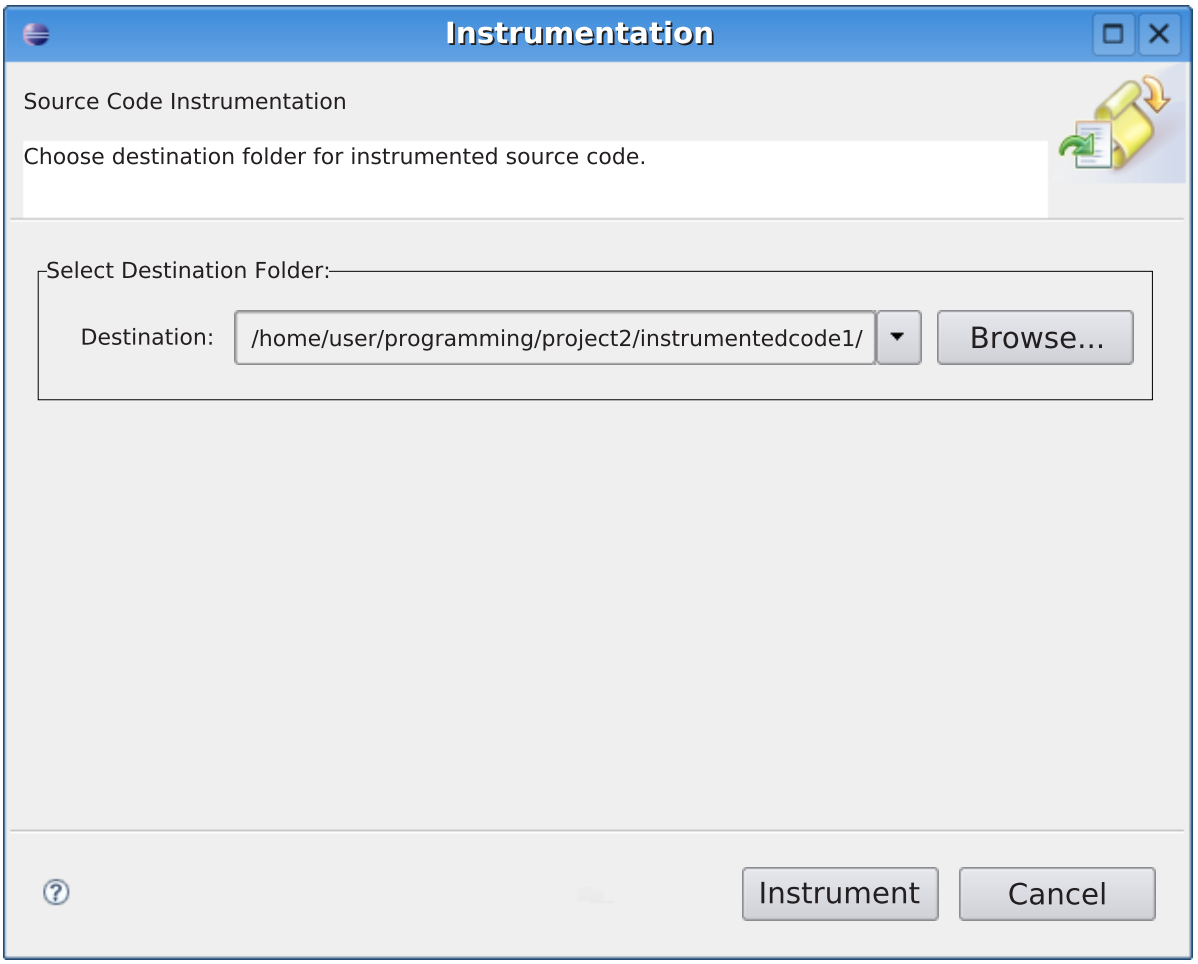


Figure 3.2: Instrumentation dialog

## 3.3 Launching

The software adds a new launch mode to the Eclipse workbench. This mode is called Coverage mode and works exactly like the existing Run and Debug modes. Figure 3.3 shows the pull down menu of the COVERAGE BUTTON on the tool bar. The menu items COVERAGE HISTORY, COVERAGE AS and COVERAGE... are added to the COVERAGE

menu.

The COVERAGE... option opens the COVERAGE dialog which is similar to the Run dialog, except that the RUN button in that dialog is called COVERAGE.

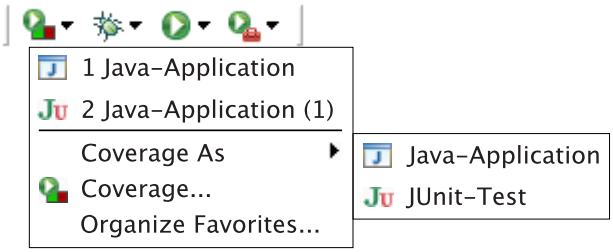


Figure 3.3: Coverage button

## 3.4 Coverage view

The coverage results are presented in the COVERAGE view. It displays the results of statement↗, branch, condition and loop coverage↗ per project↗, package, class (including interfaces and enums) and method. This view is shown in figure 3.4.

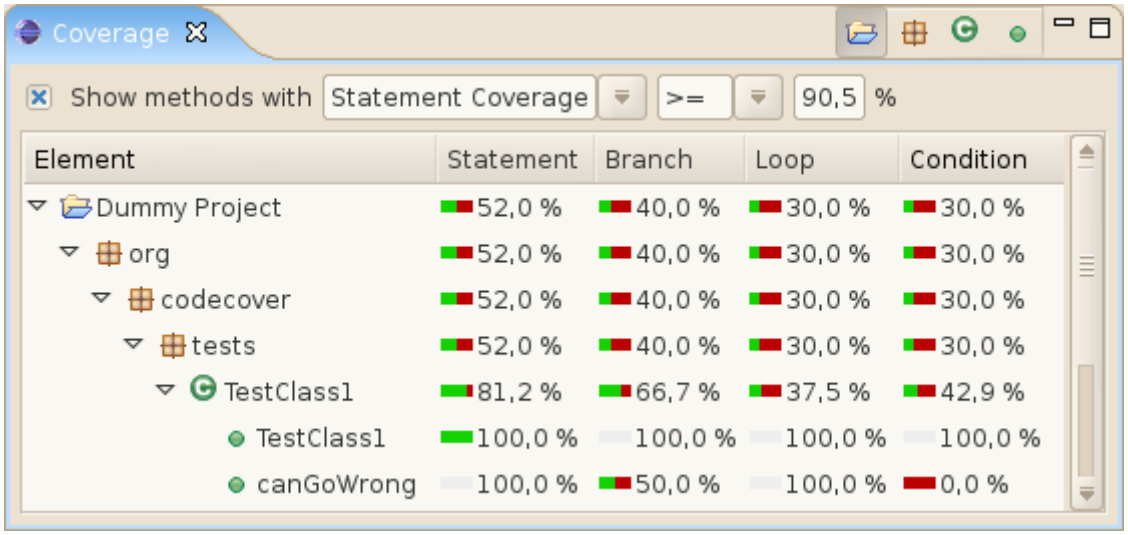


Figure 3.4: Coverage view

A grey bar is shown left of a coverage result if there are no coverable items↗ of the associated coverage criterion↗ in the associated ELEMENT.

The check box in the upper-left corner of the view enables a simple filter if checked. If the filter is activated only methods with a coverage result smaller, smaller or equal, greater, greater or equal than a given percentage are shown in the coverage view. The coverage criterion to compare with can be selected in the most left combo box in the view. The operator to compare the coverage result with can be selected in the combo box right of the combo box of the coverage criteria. The percentage to compare with can be entered in the field right of the combo box of the comparison operators.

The tool bar items in the upper-right part of the view provide the possibility to select Java elements shown as root entries in the element column. Possible root entries are projects, packages, classes (including interfaces, enums and annotations) and methods.

## 3.5 Test sessions view

The Test Sessions view lists the test sessions↗ of the selected session container↗. Figure 3.5 shows an example of this view. A session container can be chosen using the combo box. The session container list entries display the date and time along with the associated project↗.

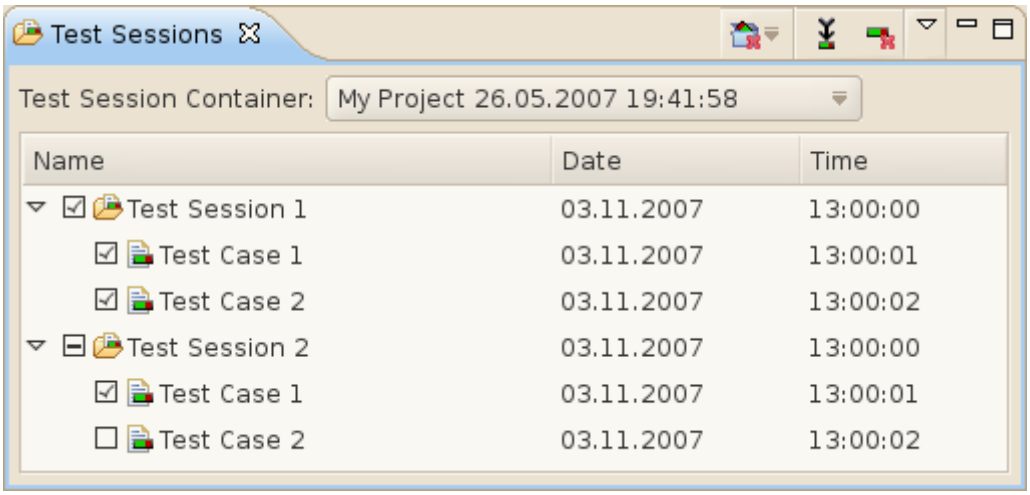


Figure 3.5: Test Sessions view

The check box in front of each row determines whether a Test Element is activated. The coverage results of activated Test Elements are visualized by the views of the plugin (e.g., Coverage view) and the source code highlighting. The visualizations are

automatically refreshed as the user activates or deactivates particular Test Elements. By default, all Test Elements are activated.

The activation or deactivation of a test session activates or deactivates all test cases of this test session. For test sessions the check box has an additional state, partly activated. This state is visualized by the crossed out check box in figure 3.5 and means that at least one test case of a selected test session is deactivated.

The information about the set of active (visualized) Test Elements is stored; that is, selecting a new session container does not change the set of the active Test Elements of the previous session container.

The tool bar items represent following commands (from left to right):

- Delete Test Session Container
  - Delete Multiple Containers
- Merge
- Delete

Delete Test Session Container deletes the active session container. Prior to the actual deletion the user has to confirm his intent to do so. By using the drop-down menu of the Delete Session Container item, the user can reveal the item for the deletion of multiple session containers: Delete Multiple Containers. It opens a dialog that prompts the user to select the session containers to delete and performs the deletion after the user confirmed the selection. The item Merge allows the user to merge the selected test elements. On selection of this item a dialog pops up which prompts the user to select the type of test element (test cases or test sessions) to merge and prompts the user to enter a name and comment for the merged test element. Furthermore the user is able to review and change the set of test elements to merge. The Delete item deletes the selected test elements, whereas the user is asked for confirmation before the deletion is performed.

The Test Elements have a context menu which contains following items:

- Select All
- Activate All
- Deactivate All
- Delete

- Properties

The item SELECT ALL selects all test elements of the active session container to be able to delete them all at once for example. ACTIVATE ALL and DEACTIVATE ALL activate or respectively deactivate all test elements of the active session container. The item DELETE evokes the same action as described above. The PROPERTIES item opens a dialog which allows the user to edit the properties of the selected TEST ELEMENT. The TEST CASE PROPERTIES dialog is shown in figure 3.6. It is possible to change the name of the selected TEST ELEMENT. Furthermore, a multi line comment may be entered.

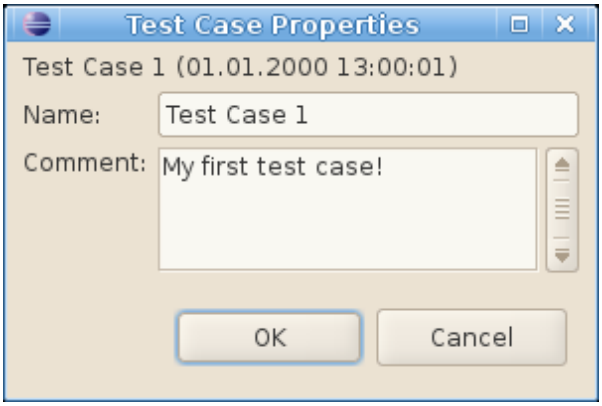


Figure 3.6: Test case properties

## 3.6 Import

The software extends the standard Eclipse IMPORT interface by adding two entries to group *CodeCover* , TEST SESSION CONTAINER and *CodeCover* COVERAGE LOG. Figure 3.7 shows the correspondent IMPORT wizard page. This dialog allows the user to select a session container↗ and a project↗ into which the session container will be imported. The file extensions used in the following dialogs are examples.

Selecting *CodeCover* COVERAGE LOG proceeds with the wizard page shown in figure 3.8. This import operation requires the coverage log↗ file, created while running the instrumented program. Moreover the user has to select the session container the coverage log will be imported to and enter a name and comment for the new test session that will contain the data of the coverage log.

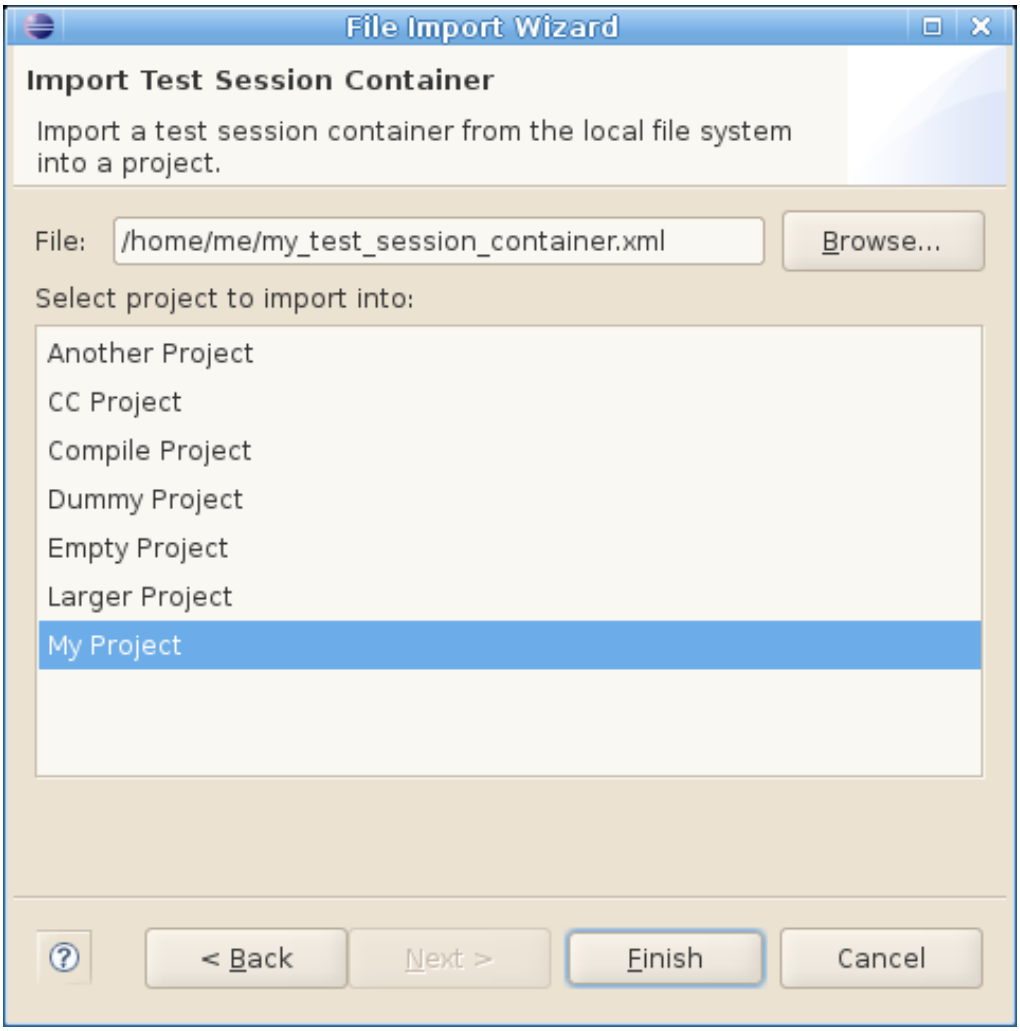


Figure 3.7: Import Test Session Container

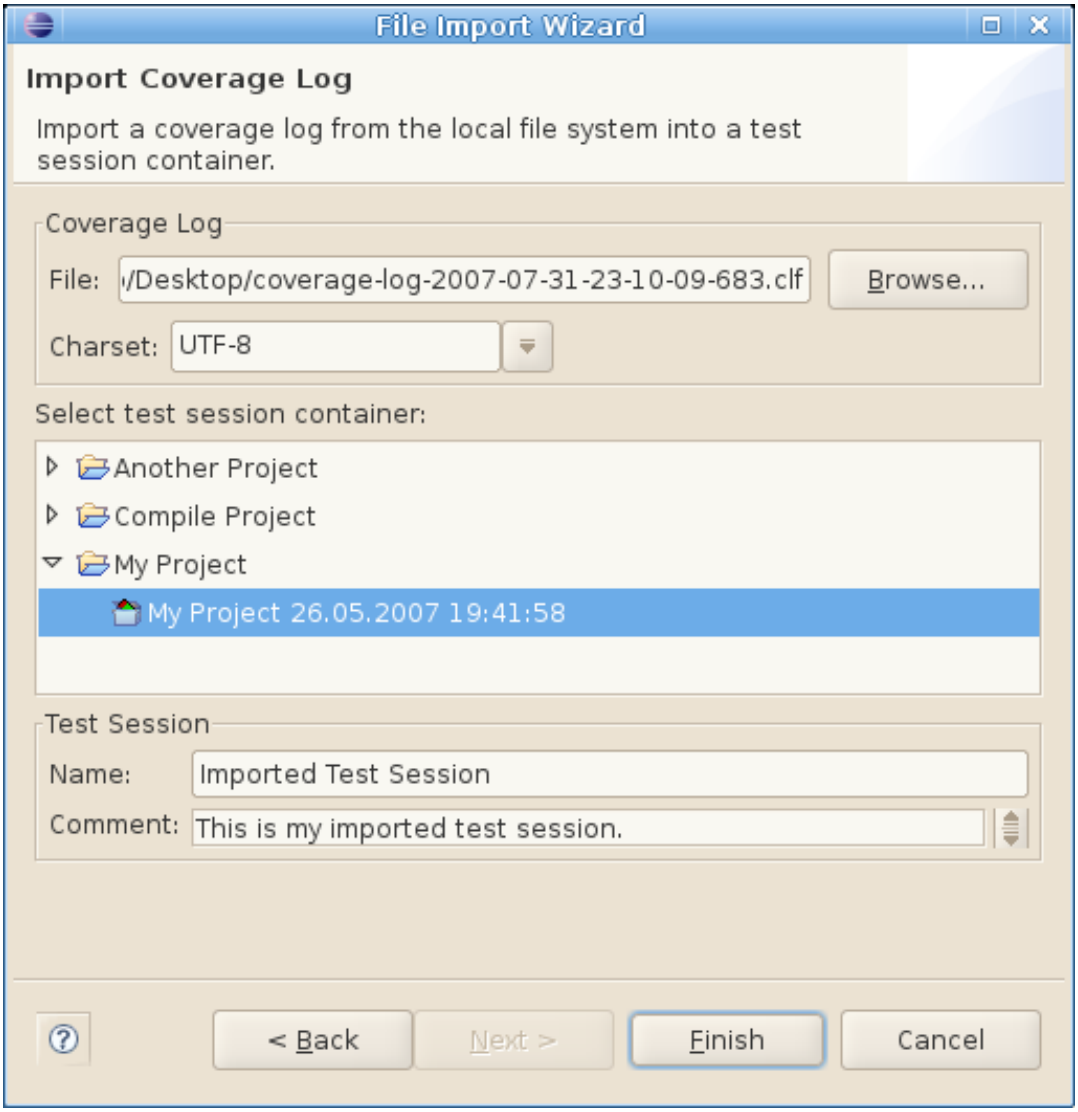


Figure 3.8: Import Coverage Log

## 3.7 Export

The item *CodeCover* Coverage Result Export is added to the Other group of standard Eclipse Export dialog. The respective wizard page is shown in figure 3.9. The dialog contains the list of all available test sessions↗ in the selected code base. The list Available Test Sessions allows multiple selections. The code base can be chosen from the combo box. By default, the last code base or the code base which is used in the Test Sessions view is preselected.

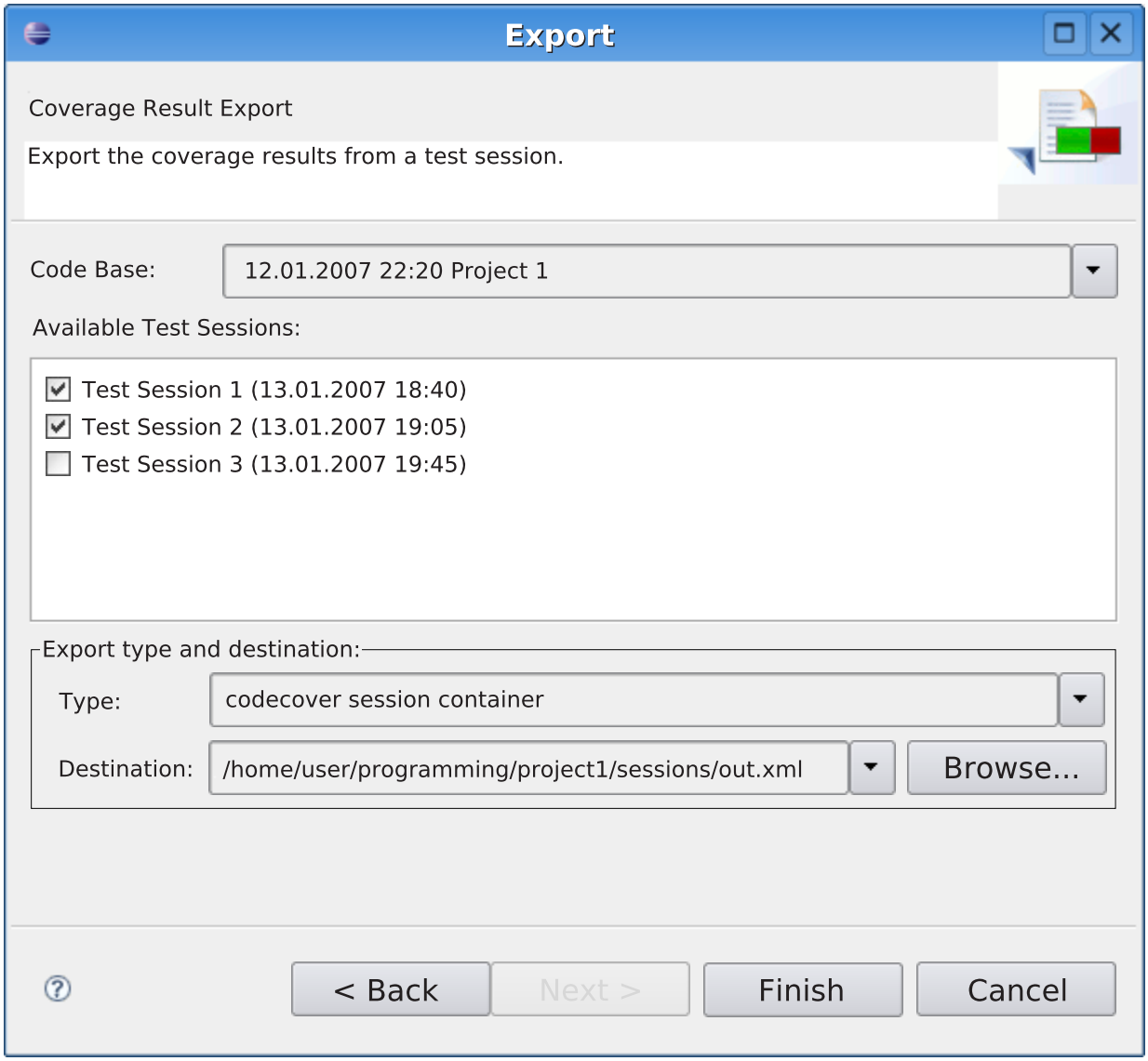


Figure 3.9: Export Test Session

Possible export types are *CodeCover* Session Container and *Report*. The report type has an extra wizard page which allows the user to select a report template (figure 3.10).

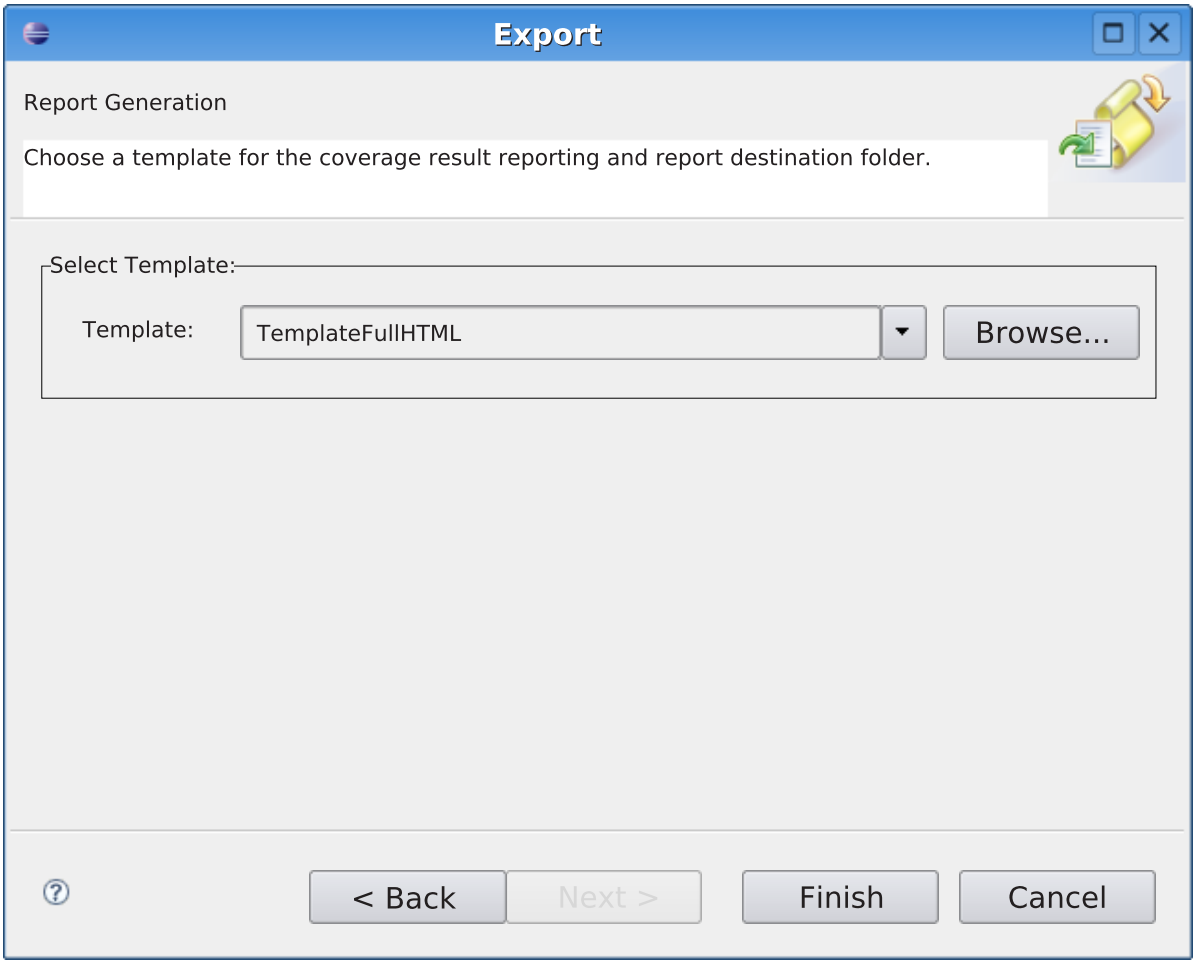


Figure 3.10: Report dialog

## 3.8 Source code highlighting

### 3.8.1 General

This section describes the visualization of coverage results by highlighting the source code of the SUT↗. There are three different colors for displaying the state of a particular part of code. The default color scheme uses green for “covered”, yellow for “partly covered”

and red for "not covered". All these colors are configurable (see section 3.9). Throughout this section, the default colors are used to explain the details of the highlighting.

Statement coverage is shown by highlighting the statements. To display branch coverage, the keywords of conditional statements are highlighted. Condition coverage is represented by coloring each basic term of a condition with green or red. For loop coverage, the keywords of looping statements are highlighted. All examples in this section show source code highlighting with all coverage criteria.

## 3.8.2 Java

### 3.8.2.1 Basic statements

Basic statements are completely highlighted either green or red, for covered and not covered statements, respectively.

### 3.8.2.2 Conditional statements

#### 3.8.2.2.1 If-then-else statements

The following list describes the meaning of the background color of the `if` keyword:

- **Green:** Both branches are covered.
- **Yellow:** Only the then-branch is covered.
- **Red:** Only the else-branch is covered.

Figure 3.11 illustrates the different possibilities of if-then coverage. If the branching statement is not evaluated because it is not executed, it is also highlighted with red color.

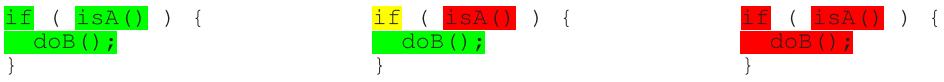


Figure 3.11: If-then statements

The background color of `else` keyword has the following meanings:

- **Green:** The else-branch is covered.
- **Red:** The else-branch is not covered.

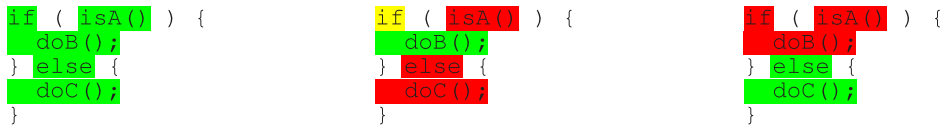


Figure 3.12: If-then-else statements

Example highlighting of if-then-else statements is shown in the figure 3.12. The left picture represents full statement, branch and condition coverage↗. In the middle picture only the then-branch and in the right one only the else-branch is covered.

An if-then-else statement can be nested in another if-then-else statement. In that case the same color scheme as described above is applied. Figure 3.13 shows some examples for nested if-then-else statements.


Figure 3.13: Nested if-then-else statements

#### 3.8.2.2.2 Switch statements

The following list describes the highlighting scheme for the `switch` keyword:

- **Green:** All cases are covered.
- **Yellow:** Some cases are covered, but at least one case is not covered.
- **Red:** No case is covered.

If the default section of the `switch` statement is not covered the `switch` keyword is highlighted as partly covered (yellow) as well. The highlighting is independent from the fact that the default section can be omitted. Figure 3.14 shows a sample highlighted `switch` statement with a `default` section.

For every case the keyword `case` and the `constant` are highlighted the following:

- **Green:** The case is covered.
- **Red:** The case is not covered.

If the default section is not omitted, the keyword `default` is highlighted the same way as cases.

```
switch ( iValue ) {
  case 0: doA();
          break;
  case 1: doB();
          break;
  default: doC();
}
```

Figure 3.14: Switch-statement with some cases and a default section

#### 3.8.2.3 Looping statements

##### 3.8.2.3.1 General

The keywords of looping statements (`while`, `for` and `do-while`) are highlighted as specified in the following list. The loop coverage↗ is defined in section 2.7.5.

- **Green:** Loop body is covered (fulfill all requirements of loop coverage).
- **Yellow:** Loop body is partly covered (at least one requirement of loop coverage, but not all requirements).
- **Red:** Loop body is not covered at all.

##### 3.8.2.3.2 While loops

Figure 3.15 shows full, partly and not covered `while` loops (from left to right).

```
while ( isA() ) {
  doB();
}
```

```
while ( isA() ) {
  doB();
}
```

```
while ( isA() ) {
  doB();
}
```

Figure 3.15: Highlighting of while loops

##### 3.8.2.3.3 Do-while

Do-while loop is represented similar to the `while` loop; only the `while` keyword is highlighted. The colors are the same (see figure 3.16).

##### 3.8.2.3.4 For

The coverage results of `for` loops are visualized in the same way as those for `while` loops. Samples of highlighted `for` loops are shown in the figure 3.17.


Figure 3.16: Do-while statements

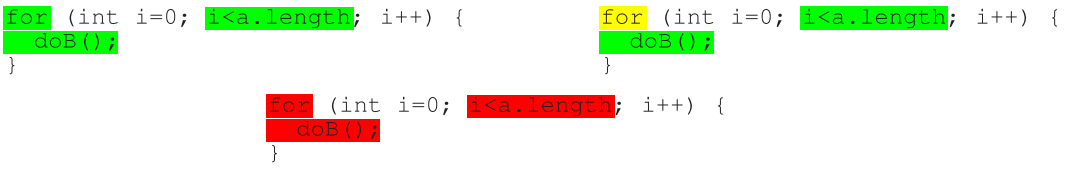


Figure 3.17: For statements

## 3.8.3 COBOL

### 3.8.3.1 Basic statements

Basic statements like `DISPLAY`, `ACCEPT` or `COMPUTE` are highlighted with green and red for covered and not covered statements, respectively.

### 3.8.3.2 Conditional statements

#### 3.8.3.2.1 If-then-else statements

The highlighting scheme for if-then-else statements is completely analogous to that of the corresponding statements in Java described above. Figure 3.18 shows the highlighting of an if-then statement in the COBOL programming language:

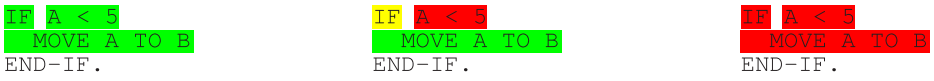


Figure 3.18: If-then statements in COBOL

#### 3.8.3.2.2 Evaluate statement

The `EVALUATE` statement is the counterpart of the Java `switch` statement. Therefore analogous highlighting rules are applied for this statement. Figure 3.19 shows an example of the `EVALUATE` statement highlighting:

```
EVALUATE VALUE
  WHEN "0" PERFORM A
  WHEN "1" PERFORM B
  WHEN OTHER PERFORM C
END-EVALUATE.
```

Figure 3.19: Evaluate statement

#### 3.8.3.3 Looping statements

##### 3.8.3.3.1 Perform

The `PERFORM` statement is overloaded and can be used as a basic statement↗ to jump to a particular paragraph (e.g. as in figure 3.19) as well as a loop statement. Figure 3.20 shows a while-equivalent statement and figure 3.21 a do-while-equivalent one. The highlighting rules for the while and do-while loops in Java apply accordingly.

```
PERFORM WITH TEST BEFORE UNTIL A = "10"
  DISPLAY A
  ADD 1 TO A
END-PERFORM.
```

Figure 3.20: Perform statement with test before

```
PERFORM WITH TEST AFTER UNTIL A = "10"
  DISPLAY A
  ADD 1 TO A
END-PERFORM.
```

Figure 3.21: Perform statement with test after

## 3.9 Preferences dialog

The *CodeCover* entry in the Preferences dialog contains Eclipse-wide options for configuring the software. This dialog page is shown in figure 3.22.

**TODO:** adopt this fully to standard dialog for annotations. Picture may be made when it's coded. The dialog provides a list of annoations to configure. For each of the four metrics there are three annotations: fully covered, partly covered, not covered. For each annotation the color can be selected using the standard mechanisms of Eclipse. A partly covered state is not produced by the default metrics for basic

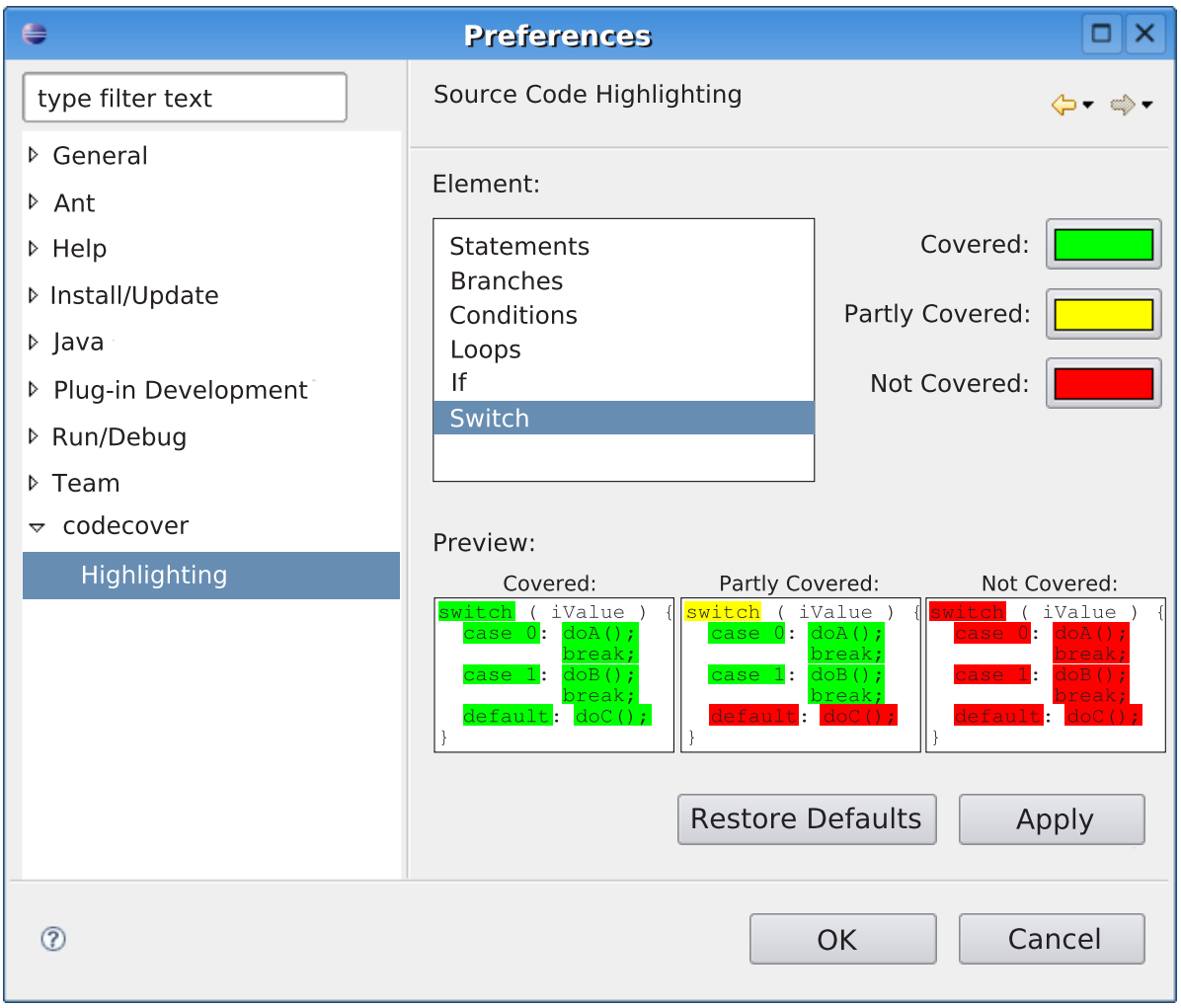


Figure 3.22: Preferences dialog

statements, basic boolean terms and branches. However as these may very well be produced by add on metrics there are also options to configure their color.

## 3.10 Project properties dialog

In the PROPERTIES dialog of a project↗ a *CodeCover* entry is added. On this page, the user can activate codecover for the project. If codecover is activated the selection of coverage criteria which are to be measured for the project is enabled too. Figure 3.23 shows this dialog page.

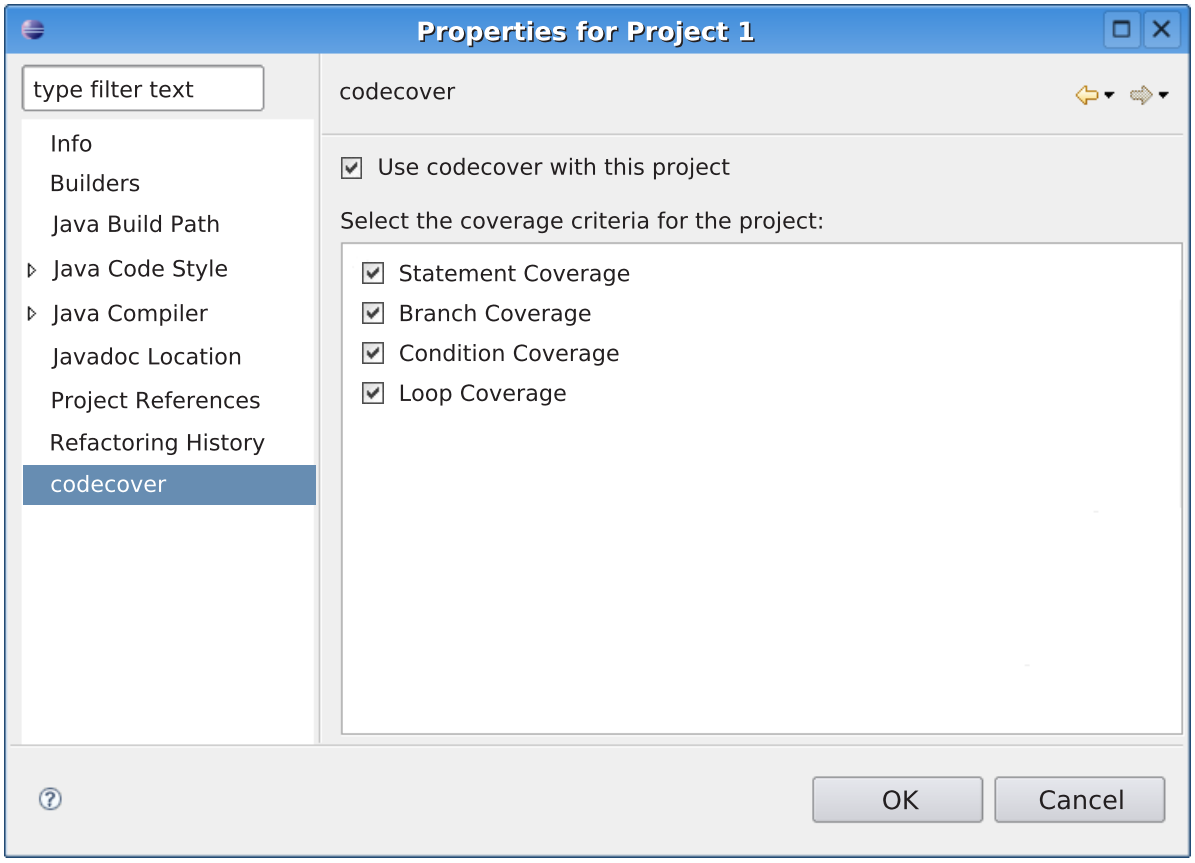


Figure 3.23: Project properties dialog

## 3.11 Correlation Matrix

### 3.11.1 Mathematic prelude

The Correlation Matrix shows the correlation between test cases of a single test-session-container. Every test case contains a set of coverable items, which represents those parts of the code, that were covered during an execution of the instrumented system under test. Those sets are defined as follows:

$$C_T := \{x | x \in T \wedge x \in CoverableItems \wedge x \in covered\} \tag{1}$$

Correlation between two test cases $T_1$, $T_2$ is then defined as:

$$K_{U,V} := \frac{\mid U \cap V \mid}{\mid V \mid} \tag{2}$$

with $U = C_{T_1}$ and $V = C_{T_2}$.

Using this definition of correlation a *partially ordered set* $R$ can be defined:

$$R = \{(U, V) \in C \times C : K_{U,V} = 1\} \tag{3}$$

In words this means, that two test cases $T_1$, $T_2$ are in relation $R$, if, and only if, $T_1$ contains all the coverable items $T_2$ does(or possibly more), which would make $T_2$ superfluous. This *partially ordered set* then allows to detect and establish subsumption chains, in which one test cases completely contains another and so forth.

Proof that $R$ is a *partially ordered set* requires to proof that it possesses the following three attributes:

1. reflexivity

$$(U, U) \in R \Leftrightarrow \frac{\mid U \cap U \mid}{\mid U \mid} = 1 \Leftrightarrow \frac{\mid U \mid}{\mid U \mid} = 1 \tag{4}$$

2. antisymmetry

$$(U, V) \in R \Leftrightarrow \frac{\mid U \cap V \mid}{\mid V \mid} = 1 \Leftrightarrow V \subseteq U \tag{5}$$

$$(V, U) \in R \Leftrightarrow \frac{\mid V \cap U \mid}{\mid U \mid} = 1 \Leftrightarrow U \subseteq V \tag{6}$$

$$\Rightarrow V \subseteq U \wedge U \subseteq V \Rightarrow U = V \tag{7}$$

3. transitivity

$$(U,V) \in R \Leftrightarrow \frac{\mid U \cap V \mid}{\mid V \mid} = 1 \Leftrightarrow V \subseteq U \tag{8}$$

$$(V,W) \in R \Leftrightarrow \frac{\mid V \cap W \mid}{\mid W \mid} = 1 \Leftrightarrow W \subseteq V \tag{9}$$

$$\Rightarrow W \subseteq V \wedge V \subseteq U \Rightarrow W \subseteq U \Leftrightarrow \frac{\mid U \cap W \mid}{\mid W \mid} = 1 \Leftrightarrow (U,W) \in R \tag{10}$$

with $T_1$, $T_2$ and $T_3$ being three test cases and $U = C_{T_1}$, $V = C_{T_2}$ and $W = C_{T_3}$.


Figure 3.24: Correlation View

## 3.11.2 Correlation View

Using the definitions from 3.11.1, the view in Eclipse is defined as shown in figure 3.24. A tree is located on the left of the view. This tree shows the subsumption chains of test cases defined in 3.11.1. All the children of a node in the tree are completely covered by the node itself.

The CORRELATION MATRIX is located on the right side of the view. It shows all the test cases, that were used in the calculation of the correlation. The meaning of the colors is

explained in the legend situated to the right of the matrix. The matrix is to be read from the left, e.g [test case left] covers [test case top] by [color value]%. A tooltip shown, when hovering above one entry of the matrix, contains the exact percentage of correlation, as well as the amount of total coverable items and shared coverable items.

The tool bar items represent following commands (from left to right):

- Export the currently displayed matrix data into a .csv file - This command exports the data of the matrix into a comma seperated file.
- Hide top-level tree items with no children - This command hides all the top level entries in the tree, which have no children, meaning they do not subsume any other test case;
- Choose and calculate correlation - This command selects the metric to be used in calculating the correlation, with the pull-down menu of the arrow and calculates the correlation with a push on the button.
- Show Legend - This command shows or hides the legend.
- Automatically calculate correlation - If this command is toggled on, the correlation is automatically recalculated, when the selection of test case is changed. It can be switched off to improve performance.

The test cases, that are to be used in the calculation, are selected using the Test sessions view. If the Automatically calculate correlation command is toggled on, every change in the selection causes the correlation view to update its contents. Since this can potentially be time-consuming, the Automatically calculate correlation command can be toggled off and the Choose and calculate correlation command can be used, after the desired selection was achieved.

## 3.12 Live Notification View

The Live Notification View is used to implement the Live Test Case Notification (2.11). Two textfields are used to enter the hostname and port. The name of the test case can be entered as well. Test cases can be started and stopped with two labeled buttons. The test session can be finished with another button. The log file can be retrieved with the “Download Coverage Log File” button. This also automatically stores the data of log file in the test session container.

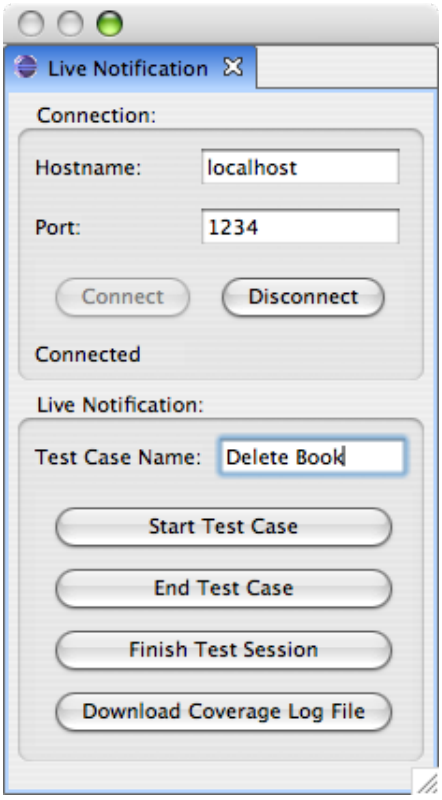


Figure 3.25: Live Notification

## 3.13 Boolean Analyzer

The Boolean Analyzer shows the boolean value of each basic boolean term, operator term and the root term according to evaluations of the condition which are recorded during the execution of the SUT. This data is presented in a table. The operators, operands and brackets define the columns and the evaluations are shown as rows in the table. In addition, a column for test cases is added. In that column one can see the test cases which covered the evaluation and the number of execution.

Two table values of a column may contain green background. This represents that the basic boolean term of that column is covered according to the strict condition coverage criterion. If a column of a basic boolean term does not contain any colored values then the term is not covered. Figure 3.26 shows the Boolean Analyzer.

There are two ways to select a condition in the Boolean Analyzer. First, there are combo-boxes for classes and conditions. The second way is to select the keyword of the condition in the source code, right-click, and select the ”Analyze Term” item in the

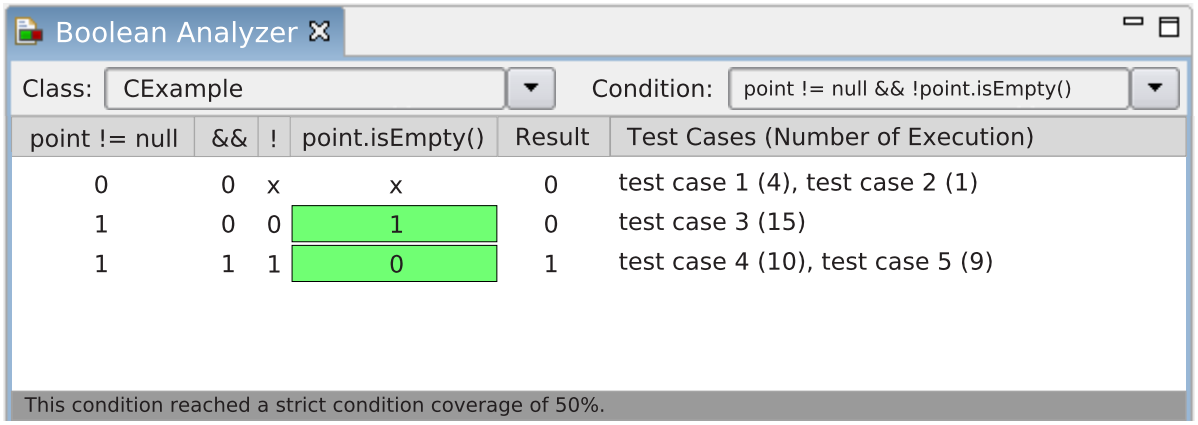


Figure 3.26: Boolean Analyzer

context menu, which will automatically select the condition in the Boolean Analyzer.

## 3.14 Hot-Path

The plugin displays line-wise execution counters corresponding to the currently selected test cases. The measured number of executions of lines is added to the VERTICAL RULER of the java text editor via annotations with a color code. When the cursor hovers over such a Hot-Path annotation a tooltip with the execution counter is shown.

If no basic statement is found in a line, then no color is shown for that line in the ruler. Otherwise the color corresponding to the most often executed basic statement of that line is shown.

The color code is a mapping from the execution count of one line ($e$) and the highest exection count within a source file ($e_{max}$) to the color to mark that line. It must be encapsulated in a single method to make it easy to change in the source code. The mapping is a linear blending between the colors given in the users preferences. The exect mapping is: $color(e, e_{max}) = color_{max} * (e/e_{max}) + color_{null}(1 - e/e_{max})$.

# 4 Non-functional requirements

## 4.1 Technologies and development environment

The following software is used for development:

- Java 5 SE
- Eclipse 3.3.x
- the customer's COBOL-85 grammar [11]
- Subversion 1.3.0 on the server for configuration management
- ArgoUML 0.22 for use case diagrams in the specification↗
- BOUML 2.21.5 or compatible for UML diagrams

The following technologies are used for development:

- LaTeX as document format
- UTF-8 for encoding text
- XML as the intermediate format for reports
- Java and COBOL-85 example programs

## 4.2 Requirements to the working environment

### 4.2.1 Software requirements

The following programs are required to use the software:

- Java version 5 or later
- Eclipse 3.3.x for all GUI functions
- PDF↗ viewer for documentation which supports at least PDF 1.4
- JUnit for automatic test case↗ recognition

[11] bruessel.informatik.uni-stuttgart.de:/home/export/bruessel/projects/stupro06/grammars/cobol.jj

- for COBOL support a COBOL-85 preprocessor to prepare code for instrumentation↗ and a compiler to compile the instrumented code

### 4.2.2 Hardware requirements

To support a wide installation base moderate hardware should be enough for using the software. Exact minimum requirements must be determined based on the final application.

The minimum hardware required is:

- 512 MiB [12] of RAM
- a CPU as powerful as an AMD Athlon with 1 GHz clock rate
- 100 MiB of free hard disk space for installation with Eclipse already installed, to store working data and for some session containers↗

The following hardware is recommended:

- 1 GiB of RAM
- a CPU as powerful as single core AMD Athlon with 2 GHz clock rate
- 60 MiB/s read and write sequential transfer rate measured at file system level
- 10 GiB of free hard disk space to store working data and some coverage results

## 4.3 Quantity requirements

Quantities that have no defined maximum are only limited by the resources of the PC the software runs on.

[12] 1 MiB = $2^{20}$ Bytes

### 4.3.1 Program examples

#### 4.3.1.1 Small program

Fred v1.3.5 RC2[13] is a small program. All functions of the software that dont' work with it are completely useless.

#### 4.3.1.2 Medium program

Tomcat 5.5.20[14] is a medium sized program. All functions of the software must work with it.

#### 4.3.1.3 Large program

Eclipse SDK 3.1.2[15] is a large program. Running functions on the large program is sufficient to show that they work for any project↗ that must be supported.

### 4.3.2 Timestamp

Timestamps have a resolution of one minute or finer. The software must use a state of the art representation of timestamps to support a sufficiently wide time period.

### 4.3.3 Text value

Text values are of arbitrary length and may contain any Unicode characters.

### 4.3.4 Test case

The name and the comment of a test case↗ are text values. The start time is a time stamp.

[13] `http://sourceforge.net/projects/fred`
[14] `http://tomcat.apache.org/download-55.cgi`
[15] `http://archive.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600/index.php`

### 4.3.5 Test session

An arbitrary number of test sessions↗ must be supported.

The name and the comment of a test session↗ are text values. The start time is a time stamp.

### 4.3.6 Code Base

An arbitrary number of code bases↗ must be supported. The date and time of the first instrumentation is a time stamp.

## 4.4 Performance requirements

All performance requirements must be met on the recommended hardware.

### 4.4.1 Batch processing

Building and instrumenting a large program (4.3.1.3) must be possible within 10 hours.

### 4.4.2 Eclipse plug-in

The following constraints must hold true for a medium program (4.3.1.2) and should hold true for a large program (4.3.1.3).

The Eclipse plug-in may not slow down Eclipse significantly.

Additionally the following response times must be met. They don't apply to the first call of a function. During the first call initialization routines may add a significant delay.

For simple GUI interaction 0.1 s response time is enough while 0.5 s is too slow. Selecting instrumentable items is such a simple GUI interaction.

For interaction resulting in complicated rebuilds of the UI components the code highlighting 5 s response time is enough while 30 s is too slow. During this time the Eclipse

UI may not respond. An example for such interaction is changing the source code annotations while they are displayed.

For loading and saving files as well as processing tasks like building the correlation matrix and generating a report there are no performance requirements. These tasks may not block the Eclipse UI.

## 4.5 Availability

There are no special availability requirements.

## 4.6 Security

There are no special security requirements.

## 4.7 Robustness and failure behavior

File types are detected using a syntax check for plain text files and at least 64 bit long magic numbers for binary files. That syntax check only has to detect wrong file types.

The software may show arbitrary behavior when it runs on broken hardware.

## 4.8 Usability

Any string displayed by the Eclipse plug-in is localizable. The default language is English.

A developer[⁄] can install the software easily. The installation procedure may only require unzip and Java. The software must be installable within 5 minutes by a developer after he has found the installation instructions and downloaded the necessary files.

All colors used for highlighting can be changed by the user.

Internal procedures must be invisible to the user as long as he doesn't want to change their behavior. Where applicable sensible defaults must be preset.

An English user's manual is required. Online help is not required.

The user interface must be intuitive to the point that on line help is not needed. Eclipse User Interface Guidelines version 2.1[16] must be followed for the Eclipse plug-in. Additionally per default the user is asked in a confirm dialog before a file with valuable user data is deleted or overwritten.

## 4.9 Portability

The software must run on all platforms Eclipse 3.2.x runs on.

The delivered instrumentation↗ procedure must be compatible with Java 2 version 1.4.x and Java 5 as well as preprocessed COBOL-85. Other languages must be implementable without changing the session container's↗ format.

## 4.10 Maintainability

The source code follows a style guide based on Sun's Code Conventions[17]. The style guide is described in a separate document.

All technical documents made specifically for creating and verifying the software are released to the public.

## 4.11 Extensibility

The software is written to be highly extensible and documented on a level easy to understand for their target audience. The documentation of the Eclipse plug-in is written for the Eclipse user, the documentation of the batch interface is written for the shell user (see Actors 2.2) and the documentation to extending the Software is written for maintenance engineers↗. Tutorials for maintenance engineers will show how to extend the software to other programming languages than COBOL and Java, how to add the collecting of metrics during the instrumentation↗ and how to implement further coverage criteria. The tutorials may assume good knowledge of Java, Grammars, JavaCC and the languages that must be supported by the changed software.

[16] `http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html`
[17] `http://java.sun.com/docs/codeconv/`

Reports can be customized using templates, which can be selected in Eclipse using a wizard. It must be easy to add new report formats like PDF↗ and DocBook XML↗, which can be transformed into many formats.

To ease external analysis and report generation all data of a test session↗, except for the boolean values sampled in conditional statements↗, must be available to easily implement an export to XML. The design team must decide if the session container↗ already has such a format, or if report generation already contains it as an intermediate step.

Other coverage criteria should be easy to add for maintenance engineers↗ as long as they don't depend on execution order. No change to the instrumentation↗, test case management and highlighting may be necessary to implement another condition coverage criterion.

It must be easy to add further analysis of a session container. All evaluation results of boolean expressions and counters must be easily accessible.

Based on the TestCaseNotifier class (see 2.1), functions to define test cases must be added: a live mode with a graphical dialog which allows the user to start and stop test cases during the SUT↗ execution and automatic test case recognition of JUnit test cases.

# List of Figures

# Glossary

**basic statement**

is a statement, that is not a looping statement↗ or a conditional statement↗. For Java the statements `return`, `throw`, `assert` are also excluded.

**branch coverage**

(synonym: decision coverage) is a coverage criterion↗. A coverable item is a branch of a conditional statement↗. For branch coverage, a coverable item is covered, if it is entered at least once.

**code base**

contains all the uninstrumented code files↗ of a specific version of the SUT↗.
A code base has a date and time of the first instrumentation↗. If a code base is instrumented from Eclipse, it has a relation to an Eclipse project.

**code coverage**

has two meanings:

1. is a measurement needed for a glass box test↗. There are different coverage criteria, each defining the coverable items↗ and how they are covered.
2. depends on a concrete coverage criterion↗ and is defined—only considering the instrumented part of the SUT—as the quotient of the covered coverable items and the total number of coverable items.

**code file**

is a file containing the whole or a part of the source code of the SUT. For example a `*.java` file in Java.

**condition coverage**

is a coverage criterion↗. Condition coverage defines the coverable items↗ as basic boolean terms↗ used in statements↗ which require a boolean expression that affects the control flow. There are different definitions of when such a basic boolean term is considered as covered—e.g. strict condition coverage↗.

**conditional statement**

is a statement↗ of a specific programming language. Conditional statements are statements creating branches in the control flow, e.g. `if` or `switch` in Java. It does not matter, if a *particular* usage of a conditional statement creates a branch (e.g. `if (true)`) or if branches of a *particular* usage are equal (e.g. `if (a) { }`). It is a conditional statement creating two branches nonetheless, because an `if` usually creates two branches.

On the other hand, though the result of some operators depends on a decision, an operator itself creates no branch in the control flow. An example for such an operator would be the `A ? B : C` operator, of which the result is determined by the value of A. Accounted by itself, such an operator only influences data, not the control flow. Therefore, it is no conditional statement.

**coverable item**

is the smallest unit that can be covered by a coverage criterion, e.g. a `then` branch in branch coverage.

**coverage criterion**

defines the coverable item↗ and under which condition they are covered. Some coverage criteria are:

- statement coverage↗
- branch coverage↗
- condition coverage↗
- loop coverage↗

**coverage log**

is a container for raw result data of a coverage run. It contains e.g. counters for all basic statements. This file must be processed afterwards to produce test sessions↗ and test cases↗.

**developer**

is a person who is able to write and compile programs in at least one programming language and can understand well documented programs. He is also experienced in both using his computer, especially with file system interaction, web browsing,

extracting archive files, applying patches and editing plain text.

### DocBook XML

is a XML based markup language for technical documentation.

### entry point

is the item, that is used to start the SUT↗. For Java, it is a class file containing the `main` method. For COBOL, it is the single code file↗.

### HTML

(abbreviation for: Hyper Text Markup Language) is the predominant markup language for the creation of web pages.

### instrumentable item

is an item of the SUT↗. An instrumentable item is a package, containing other instrumentable items, or a code file.

### instrumentation

is the process of adding extra code elements to a code file↗ in order to get information about the control flow of the running SUT. Instrumentation is used to measure the code coverage↗ of the code file.

### loop coverage

is a coverage criterion. The loop coverage defines several coverable items↗ for each looping statement↗:

- loop body is not entered
- loop body is entered once, but not repeated
- loop body is repeated more than one time

Looping statements like do-while cannot be bypassed and have only two possible coverable items.

### maintenance engineer

is a developer↗ who changes software. He understands technical English. He is able to work out technical details himself, if he is pointed to good documentation.

### MAST

(abbreviation for: More Abstract Syntax Tree) The MAST is a model of the source code containing only the elements of the source code which are necessary to calculate coverage criteria↗ e.g. statements↗, branches or boolean expressions which have an affect on control flow.
A MAST always refers to a specific code base↗.

### PDF

(abbreviation for: Portable Document Format) is an file format created and controlled by Adobe Systems, for representing two-dimensional documents in a device independent and resolution independent fixed-layout document format.

### project

is an Eclipse project that appears e.g. in the Package Explorer of Eclipse.

### session container

is a file in the file system. It contains:

- a code base↗
- a MAST↗ of this code base
- a number of test sessions↗ ($\geq 0$)

### specification

describes all functional and non-functional requirements the software has to fulfill.

### statement

is an element in a code file↗ that is the result of the statement production of the grammar of the corresponding programming language.

### statement coverage

is a coverage criterion↗. Statement coverage defines the coverable items↗ as basic statements↗. A coverable item is covered, if the basic statement is executed. For the Java programming language the execution of the statement has to start to set the statement covered. For COBOL, a basic statement is called covered, if it is executed and the program flow goes to the next statement.

**strict condition coverage**

is a kind of condition coverage↗. Strict condition coverage defines a basic boolean term as covered, if it is evaluated to both true and false and the change from true to false (or false to true) changes the result of the whole condition while every other basic boolean term of the condition remains constant or is not evaluated.

**SUT**

(abbreviation for: system under test) The system tested with the software.

**test case**

1. is the description of the input for a test with its expected output according to the specification↗.
2. is an element of a test session↗ containing a part or all of the code coverage↗ results for code files depending on a set of coverage criteria. Additional information which are stored with a test case are:
   - a name
   - date and time of measurement
   - a comment
   - the related test session

   If the test case is related to a JUnit test case or test method extra information are needed:
   - the names of the test methods of the JUnit test case
   - whether the test methods of the JUnit test case failed or not
   - if a test method failed, which failure respectively error was the reason

**test session**

is the result of a coverage measurement of the SUT by the software. It has:

- a name
- a date and a time of measurement
- a comment

and contains:

- a number of test cases$^{\nearrow}$ ($\geq 0$)
- the measurement results
- calculated coverage by instrumentable item$^{\nearrow}$
- a reference to a code base$^{\nearrow}$
- possibly a reference to a related Eclipse project$^{\nearrow}$