Design

CodeCover
Glass Box Testing Tool

Student Project A "OST-WeST"
University of Stuttgart

Version: 1.2

Last changed on May 28, 2008 (SVN Revision 24)

Contents

1 Introduction

1.1 Project overview

CodeCover is a glass box testing tool to measure the coverage of a running program. It will be as independent as possible of the programming language of the covered program.

1.2 About this document

This document contains the necessary information to create a detailed design obeying the requirements documented in the specification. It describes the behaviour of the software and the artifacts which are created by it. The architecture is also defined by this document, as well as the data structures used to store the data pertaining to the operation of the software. Details like the structure of classes and the design of methods have to be determined in the detailed design document.

1.3 Addressed audience

This document is addressed to

- the customer who ordered the software
- the project manager controlling the work
- 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 Authors

In the following table authors of this document are named.

Author	E-mail
Robert Hanussek	hanussrt@studi.informatik.uni-stuttgart.de
Steffen Kieß	kiesssn@studi.informatik.uni-stuttgart.de
Tilmann Scheller	schellrt@studi.informatik.uni-stuttgart.de
Markus Wittlinger	wittlims@studi.informatik.uni-stuttgart.de

1.5 Notation

1.5.1 Identifiers

`FooBar` denotes a class named `FooBar`.

`FooBar` denotes an abstract class named `FooBar`.

`FooBar` denotes an instantiated object of class `FooBar`.

`FooBar` denotes an interface class named `FooBar`.

`FooBar` denotes an object of a class which implementes the interface named `FooBar`.

`fooBar(...)` denotes a method named `fooBar` which has parameters (which are not listed). Methods don't have a special color because they can be easily identified by the trailing brackets.

`fooBar()` denotes a method named `fooBar` which doesn't have any parameters.

`fooBar(...)` denotes a static method (which has parameters).

`fooBar` denotes a field named `fooBar`.

`foo.bar` denotes a package called `bar` which resides in a package called `foo`. Packages don't have a special color because they can be identified easily since they are the only items which contain a dot and are lower case.

2 General overview

2.1 Process chain

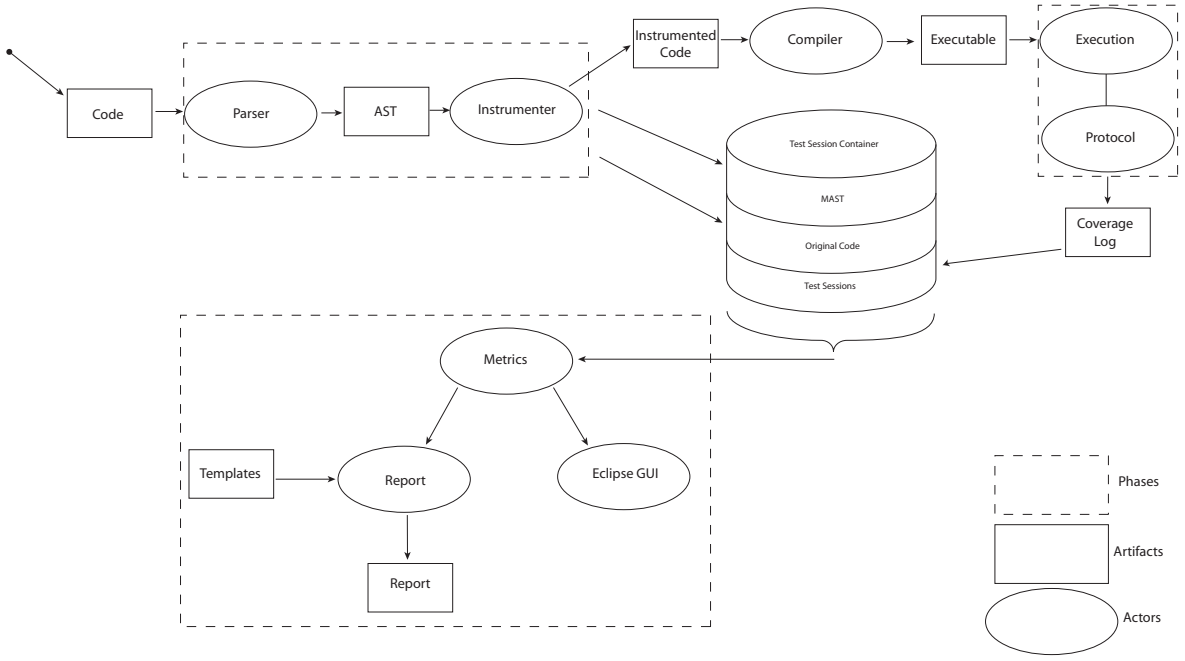

Figure 2.1: Process chain

This section describes the process of *CodeCover* in detail. The whole process consists of three phases: instrumentation, execution and reporting. Each phase includes actions which are performed and which create certain artifacts. The arrows in figure 2.1 denote the data flow between the actors.

The process starts in the instrumentation phase and the only required input is a set of valid code files (can be compiled without getting an error message). The instrumenter consists of a parser and a part which performs the actual instrumentation of the source code. The parser is generated automatically from a javacc grammar with the help of the Java Tree Builder (JTB). This needs to be done only once, e.g. when the instrumenter is built.

This parser creates an abstract syntax tree (AST) representation of the source code file. The actual instrumentation is performed by traversing this AST and adding additional

code to create an instrumented version of the code. During this traversal the static information about the code is collected and the modified abstract syntax tree (MAST) is created. 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 effect on control flow. The MAST is language independent. Every element in the MAST also contains a reference to its location in the corresponding source code file. Both the original code and the generated MAST are saved in the Test Session Container.

After the code is instrumented, executable code can be generated by a compiler. This has to be done by the user if the CLI is used. The compiling is initiated and controlled by *CodeCover* when using the Eclipse Plugin. But compilers are not part of *CodeCover*. The required compilers must be installed separately.

Then the instrumented code can be executed and the coverage measurements will be recorded into the coverage log separated by test cases. The coverage log is saved in the Test Session Container. It holds the information on the number of executions of a coverable item in case of statement coverage, branch coverage and loop coverage or the assignments which were assigned to boolean terms in case of condition coverage.

Now the Test Session Container contains all information which is needed to calculate coverage metrics:

- the MAST which is needed to connect the results of the coverage measurement in the coverage log to specific elements in the code
- the original code which is needed to generate reports with code highlighting
- the coverage log which contains the coverage measurements

Thus the coverage metrics can be calculated for report generation or to be output to the Eclipse GUI.

For report generation the structure of the code is retrieved from the MAST. Since the information in the MAST is independent from programming language, the implementation of report generation is also independent from the programming language. Report generation happens based on test cases and a template which defines the design and output format (e.g. HTML) of a report. By specifing test cases the coverage measurements, the report is based on, can be limited. Since the coverage log contains the coverage metrics separated by test cases, these metrics can be calculated and used in the report.

2.2 Adding support for new programming languages

In order to add support for a new programming language a new instrumenter needs to be written. If a javacc grammar is available for the language the parser can be easily generated with JTB. Then a component which performs the actual instrumentation needs to be written, which adds the necessary instrumentation code and creates the MAST. A component for logging needs to be supplied too. Everything else of *CodeCover* is not language dependent and requires no further action.

3 Component overview

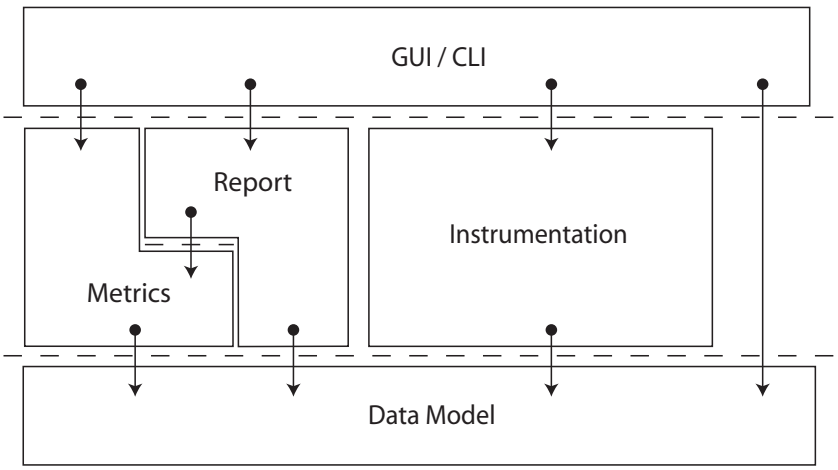

Figure 3.1: Component overview

As can be seen in Figure 3.1, the software consists of five general components, which are arrayed in a hierarchical fashion. The arrows leading from one component to another symbolize an *accessing* relationship, which runs from top to bottom. The `Data Model` is situated at the bottom, since it acts as the foundation for all other components. On top of the data model are the components `Instrumentation`, `Report` and `Metrics`, with the last one also being accessible by the `Report` component. The topmost layer is the user interface, be it a graphical user interface (`GUI`), or a commandline interface (`CLI`).

3.1 Data model

This component contains all the packages and classes that are needed to hold the data generated by the instrumentation itself and the data resulting from the execution of an instrumented SUT. Every other component has access to the data model.

The following classes are used to represent the AST. The data will be stored in XML files. A description of the format of these XML files can be found in the detailed design document.

3.1.1 MetaDataObject

`MetaDataObject` is implemented by all classes to which meta data can be associated.

It has only one method (`getMetaData()`) which can be used to get the `MetaData`.

Instances of this class can be passed to the methods `setObjectMetaData(...)` and `getObjectMetaData(...)` of the class `TestCase`.

3.1.2 MetaData

A `MetaData` represents the meta data associated to a AST element, e.g. coverage data linked to certain elements of the AST.

This class is used internally.

3.1.3 CoverableItem

A `CoverableItem` represents a coverable item which can be covered in a test case.

Instances of this class can be passed to the method `getCoverageCount(...)` of the class `TestCase`.

3.1.4 Locatable

A `Locatable` is an object with one or multiple locations.

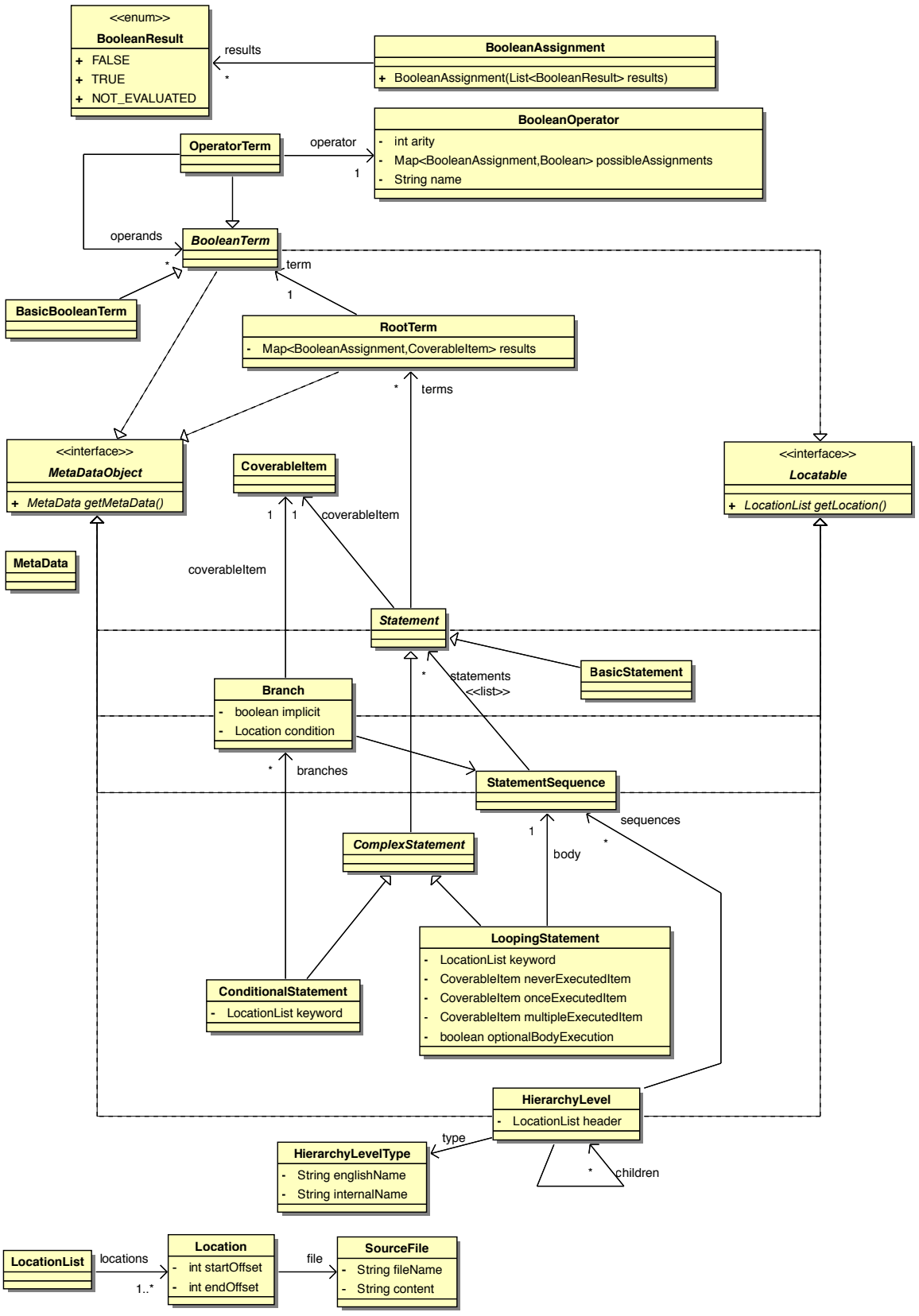

Figure 3.2: `org.codecover.model.ast`

The method `getLocation()` can be used to get this locations.

3.1.5 LocationList

A `LocationList` is a list of `Location`s.

This is necessary since some AST elements can have mulitple locations (e.g. a `partial class` in C#).

3.1.6 Location

A `Location` is a segment in a code file.

It is given by its `startOffset` (which is the offset of the first `char` belonging to the location), its `endOffset` (which is the offset of the first `char` no longer belonging to the location), and the file which contains this location.

3.1.7 SourceFile

A `SourceFile` represents a source file.

It contains the `fileName` and the `content` of the file.

3.1.8 BooleanResult

A `BooleanResult` is an `enum` which describes the result of the evaluation of a boolean term.

The value `NOT_EVALUATED` is used if a subterm was not evaluated, e.g. because of the short circuit behaviour of an operator.

3.1.9 BooleanAssignment

A `BooleanAssignment` assigns every basic boolean term of a boolean term an `BooleanResult`.

3.1.10 BooleanTerm

A BooleanTerm represents a boolean term which ist constructed of basic boolean terms and boolean operators.

A BooleanTerm can be a BasicBooleanTerm or a OperatorTerm.

3.1.11 BasicBooleanTerm

A BasicBooleanTerm represents a basic boolean term which is considered atomic.

3.1.12 OperatorTerm

A OperatorTerm represents a boolean term which consists of an operator connect zero, one or more BooleanTerms.

It contains a reference to the BooleanOperator used and the list of the operands (which are BooleanTerms).

3.1.13 BooleanOperator

A BooleanOperator is a function with a given arity arity which maps a arity-tuple of boolean values to a boolean value.

The object contains the arity, a map which maps the assignments to the result and a name.

3.1.14 RootTerm

A RootTerm is a boolean term which is not a part of another boolean term.

A RootTerm consists of a BooleanTerm and a CoverableItem for every possible assignment of this term.

3.1.15 Statement

A Statement is a basic or a complex statement.

Every Statement either has the type BasicStatement or on of the types derived of ComplexStatement, ConditionalStatement and LoopingStatement.

A Statement has a list of RootTerms which appear in the statement and a CoverableItem which will be covered when the Statement is executed.

3.1.16 BasicStatement

A BasicStatement is a statement which contains no other statements.

3.1.17 ComplexStatement

A ComplexStatement is a statement which can contain other statements and is either a ConditionalStatement or a LoopingStatement.

3.1.18 ConditionalStatement

A ConditionalStatement is a statement where the control flow splits up into a number of Branches.

The ConditionalStatement consists of these Branches and of the LocationList of the keyword of the statement (for the purpose of coloring the source code).

3.1.19 Branch

A Branch is a branch which can be taken in a conditonal statement.

It consists of a StatementSequence, a CoverableItem which is covered when the branch is executed, a flag which says that this branch does not appear explicitly in the source code (e.g. a `else` branch when there is no `else` keyword for a `if` statement or the `default` branch of a `select` statement where there is no `default:` block) and optionally

the LocationList of the conditon whether this branch is taken (for the purpose of coloring the source code).

3.1.20 LoopingStatement

A LoopingStatement is a statement which has a body which can be executed a number of times not known at compile time.

It has a StatementSequence representing the body, the LocationList of the keyword of the statement (for the purpose of coloring the source code), a boolean flag indicating whether the body also can be executed zero times and three CoverableItems covered when the body is executed zero times, one time or more often.

3.1.21 StatementSequence

A StatementSequence is a list of Statements.

3.1.22 HierarchyLevel

A HierarchyLevel is a program object which can contain other HierarchyLevels or StatementSequences, e.g. Java packages, files, classes and functions.

3.1.23 HierarchyLevelType

A HierarchyLevelType represents the type of a HierarchyLevel.

It contains an English name which can be used in texts shown to the user (this name might be e.g. “package” oder “class”) and an internal name which can be used e.g. for choosing a icon for the HierarchyLevel.

3.1.24 Example

To further explain the data model the AST representation of a Java code example is shown.

```
public class TestClass {
  public static void main(String[] args) {
    if (something || nothing) {
      foo();
    } else {
      bar();
    }

    while (bar()) {
      i++;
      j++;
    }
  }
}
```

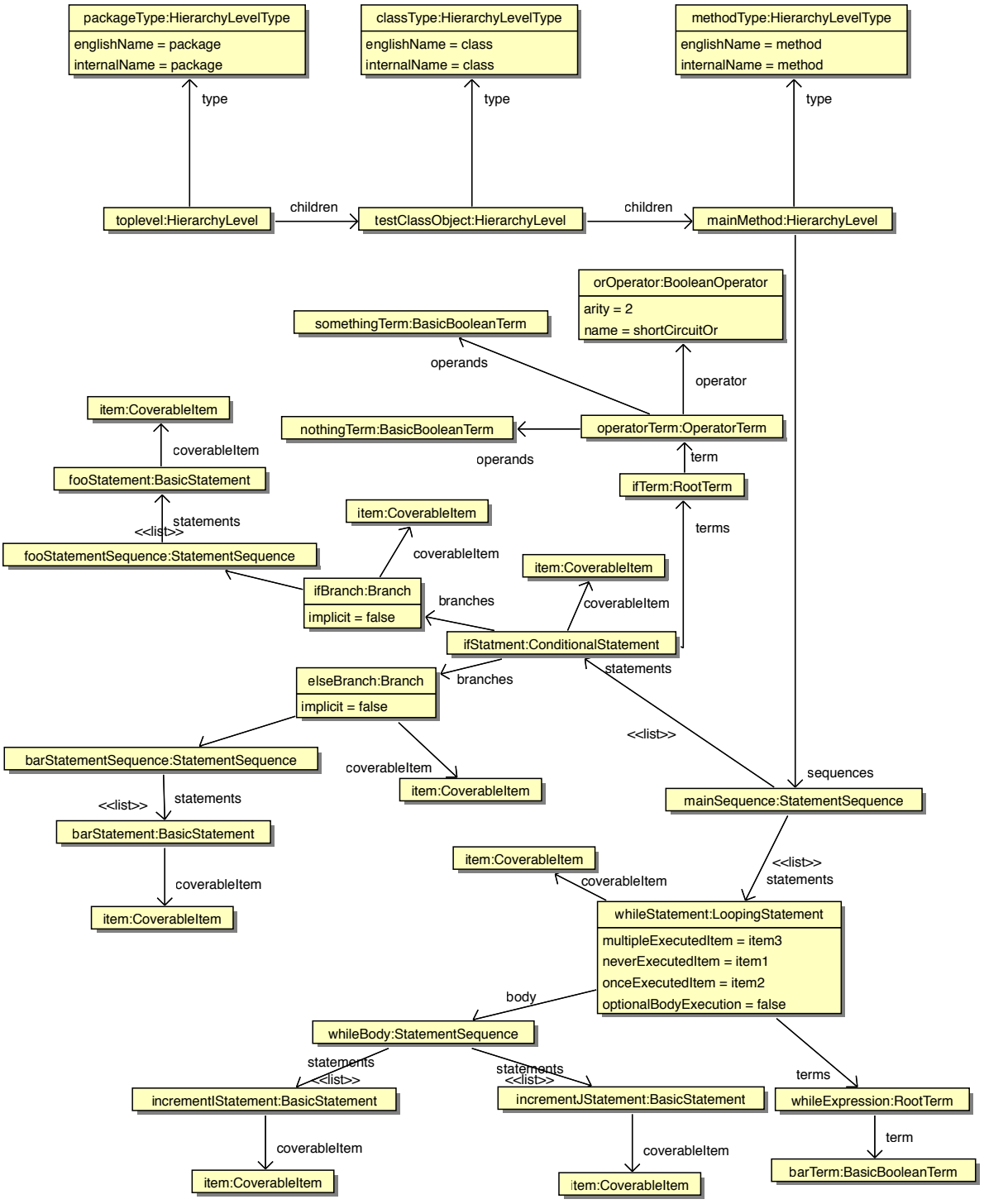

Figure 3.3: Created object graph

3.2 Instrumentation

3.2.1 Overview

This component includes all the packages and classes which deal with the instrumentation of source files. It includes all classes necessary to configure an instrumenter according to the users needs (e.g. select specific coverage criteria). All instrumenters are integrated using interfaces. A new instrumenter for a certain programming language must support these interfaces. The implementation will include instrumenters for Java 1.5 and COBOL-85. The instrumentation component is modeled in greater detail in the detailed design document.

3.2.2 Instrumentation approach

There are four coverage criteria: statement, branch, condition and loop coverage. The instrumentation for each criterion is independent. For that reason, the instrumentation approach is described separately using a pseudo programming language. All coverage data is captured using counters. These counters can be variables, fields or arrays. In the following examples, we use variables in the form `counter1` to represent the counters. All counters are stored persistently in the coverage log file, when a test case ends. The specification of the coverage log file can be found in appendix B on page 49.

3.2.2.1 Statement coverage

In Java, the statement coverage has the semantic, that a statement is covered, if the programm flow **starts** to execute the statement. For this reason, a simple instrumentation before each statement is sufficient. The following example shows how statements are instrumented:

```
                                    counter1 := counter1 + 1
<statement1>                        <statement1>

                                    counter2 := counter2 + 1
<statement2>                        <statement2>

                                    counter3 := counter3 + 1
<statement3>                        <statement3>
```

In addition to the instrumentation of simple statements, looping and conditional statements are instrumented specially, see loop respectively branch coverage.

3.2.2.2 Branch coverage

Branches are created by if, switch or similar statements. For each branch a counter is introduced. If a branch is omitted, like an else-branch, it is added with an appropriate counter.

```
if <condition> then                 if <condition> then
  <statement sequence>                counter1 := counter1 + 1
end if                                <statement sequence>
                                    else
                                      counter2 := counter2 + 1
                                    end if

if <condition> then                 if <condition> then
  <statement sequence>                counter3 := counter3 + 1
else                                  <statement sequence>
  <statement sequence>              else
end if                                counter4 := counter4 + 1
                                      <statement sequence>
                                    end if
```

```
switch <variable>
  case <value>
    <statement sequence>
  case <value>
    <statement sequence>
end switch
```

```
switch <variable>
  case <value>
    counter5 := counter5 + 1
    <statement sequence>
  case <value>
    counter6 := counter6 + 1
    <statement sequence>
  default
    counter7 := counter7 + 1
end switch
```

```
switch <variable>
  case <value>
    <statement sequence>
  case <value>
    <statement sequence>
  default
    <statement sequence>
end switch
```

```
switch <variable>
  case <value>
    counter8 := counter8 + 1
    <statement sequence>
  case <value>
    counter9 := counter9 + 1
    <statement sequence>
  default
    counter10 := counter10 + 1
    <statement sequence>
end switch
```

3.2.2.3 Condition coverage

The condition coverage instrumentation approach is highly dependent on the programming language. For example, short-circuit operators or side effects have to be considered. For programming languages without such characteristics the straight forward method which is shown here may be used.

```
if <conditionA> AND
   <conditionB> then
  <statement sequence>
end if
```

```
if <conditionA> then
  if <conditionB> then
    counter11 := counter11 + 1
  else
    counter10 := counter10 + 1
  end if
else
  if <conditionB> then
    counter01 := counter01 + 1
  else
    counter00 := counter00 + 1
  end if
end if
if <conditionA> AND
   <conditionB> then
  <statement sequence>
end if
```

For programming languages with side effects and short-circuit operators another approach is needed. See appendix A on page 44 for details.

3.2.2.4 Loop coverage

For each looping statement an auxiliary variable and three counter variables are introduced. The auxiliary variable is incremented for each run of the loop body. After the loop, the counter variable which corresponds to the value of the auxiliary variable is incremented.

```
while <condition> do
  <statement sequence>
end while
```

```
auxiliary := 0
while <condition> do
  auxiliary := auxiliary + 1
  <statement sequence>
end while
switch auxiliary
  case 0
    counter0 := counter0 + 1
  case 1
    counter1 := counter1 + 1
  default
    counter2 := counter2 + 1
end switch
```

3.3 Metrics

This component contains the classes necessary to impose metrics on the data model. Furthermore all classes needed for the introduction of new metrics are provided as well. The implementation will include coverage metrics such as statement coverage, branch coverage, conditional coverage and loop coverage.

3.4 Report

The report component consists of report generators whereas each of them is responsible for the creation of an output format, e.g. hierarchical HTML. A specific report is generated based on a template which specifies the report generator needed for report generation. The report component chooses the report generator which is able to process the template and dispatches report generation to the chosen report generator. That means for every report generator and thus for every output format exists a different template format.

An alternative to this design would have been to have only one specific template format for all types of output formats. But since output formats differ extensively in their struc-

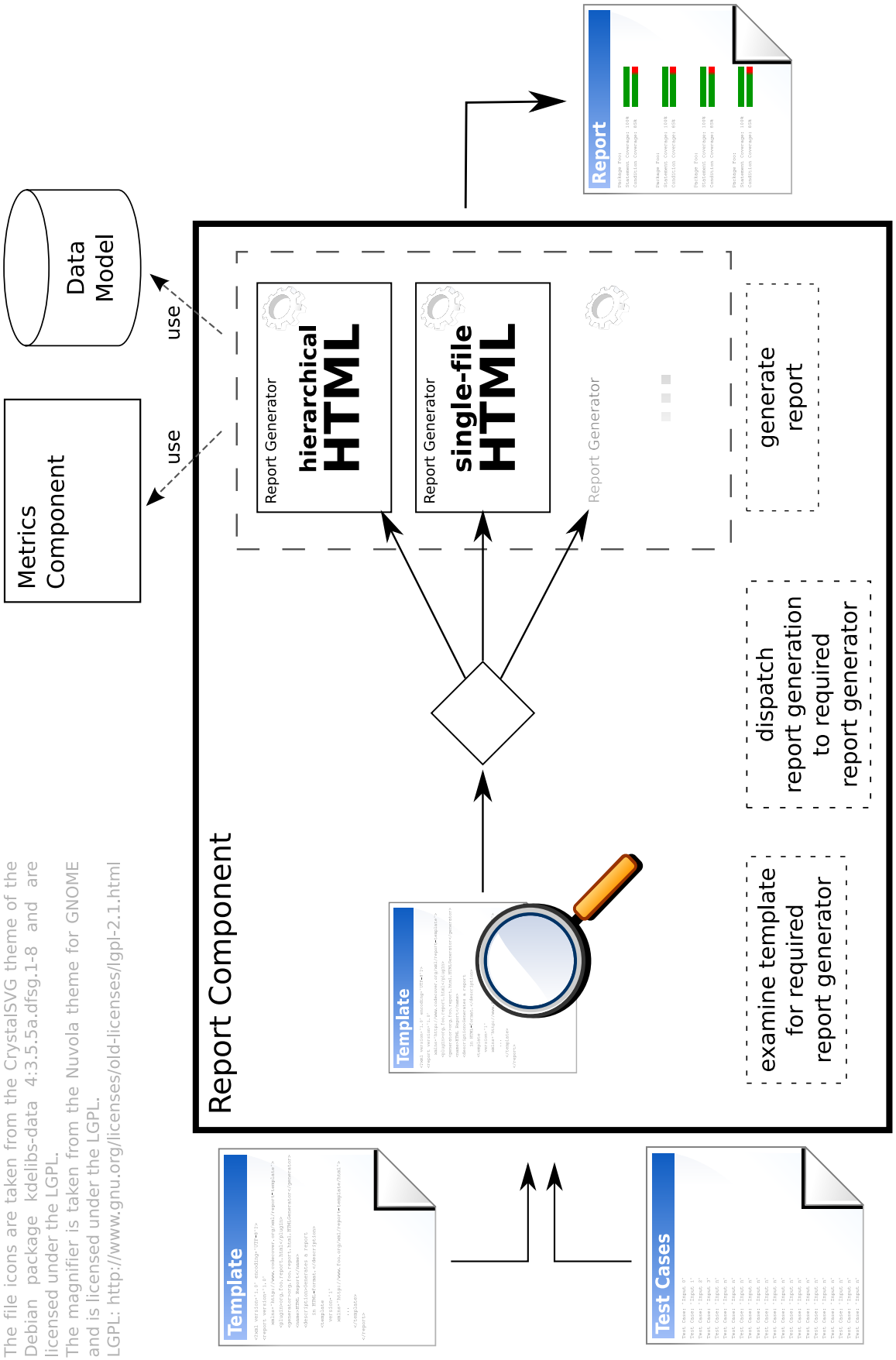

Figure 3.4: Report generation

ture and don't share many similarities the adjustements that could be set in templates would be very limited.

Another alternative would have been to generate an intermediate report which is transformed into the desired output format and document structure which are specified by templates. This templates, of course, would be specific to output format. The advantage of this alternative is that the generation of the raw data, which is saved in the intermediate report, is seperated from the generation of a specific output format. The problem is that the a powerful intermediate format has to be defined since the whole report component is based on it. And a mechanism would have to be implemented which transforms the intermediate report into the desired output format. But this is a difficult task which would consume too much time. An intermediate format which is already defined is DocBook. The DocBook report could be translated to the desired output format by using XSLT. But this means that templates would have to be defined in XSLT which is very complex and wouldn't comply with the requirement to allow easy adjustements to the design of reports.

Another alternative would have been to use an already existing report engine. But due to the lack of report engines which can produce hierarchically linked HTML reports (see 3.4.5) this is not an option.

The report component needs the following input to generate a report:

- the test cases which results are described by the report
- the template which specifies the design and output format of the report
- the structure of the code
- the coverage metrics for each code element (e.g. package, class)
- the original code

The *test cases* are part of the data model and handed over to the report component when a report is requested.

The *template* is a file in XML-format and is given to the report component when a report is requested. Templates are natural language specific. Although templates have a general structure (described in 3.4.3), their detailed specification differs for each report generator.

The *structure of the code* is needed to structure the report according to the code's

structure. Thus the report component needs access to the data model which models the code's structure.

To calculate the *coverage metrics*, the metrics component is used. It receives test cases and a structural code element as input and then calculates the metrics by investigating the coverage log which is part of the data model.

The *original code* is needed for code listings in reports.

3.4.1 Programming language independency

Since the report component builds on the data model, which is language independent, the report component is language independent, too. The data model is hierarchically structured by `HierarchicalLevel`s which build a tree. This tree structure can be used to map the code's structure on the structure of a report, that is the tree structure of the code builds the document structure of the report.

The data model identifies *methods* for every programming language and SUT. The report component can use this information to place code listings with highlighting in the correct hierarchical level of the report's structure. Moreover this information can be used to decide which is the deepest level of the data model which will be mapped onto an own hierarchical level in the report. In case of the HTML report (see 3.4.5) this means that it is possible to find out which structural code elements (namely methods) are described by code pages.

In Java the above mentioned *methods*, of course, identify methods. In COBOL *methods* identify sections.

3.4.2 Natural language independency

Names of language specific types are saved for each structural code element. This makes it possible to output these language specific names (e.g., package, class, method in Java) to the report. The language specific names are saved in english. To generate a report in a different language, a template has to be created which translates the names. Thus templates are (natural) language specific.

3.4.3 Template format

The format of templates is XML and the general structure is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<report version="1.0" xmlns="http://www.codecover.org/xml/report-template">
    <plugin>org.codecover.report.html</plugin>
    <generator>org.codecover.report.html.HTMLReportGenerator</generator>
    <name xml:lang="en">HTML Report (hierarchic)</name>
    <name xml:lang="de">HTML Report (hierarchisch)</name>
    <description xml:lang="en">
        Generates a hierarchical report in HTML-format.
    </description>
    <description xml:lang="de">
        Erstellt einen hierarchischen Report in HTML.
    </description>
    <template
      version="1"
      xmlns="http://www.codecover.org/xml/report-template/html-hierarchic">
    ...
    </template>
</report>
```

All of the above defined elements and attributes are required in a template (none is optional). It is recommended but not required to use UTF-8 as the character encoding.

The element `plugin` defines the name of the plugin which contains an extension which is the report generator which can process the template. The extension is identified by the name contained in the element `generator`. In the above example the plugin which contains the report generator is named `org.codecover.report.html` and the report generator's name is `org.codecover.report.html.HierarchicalHTMLReportGenerator`.

The attribute `version` of the root element `report` sets the version of the template format. The template format which is specified in this document has the version 1.0. The version is compared to the internal version number of the report component. The major version number is incremented if a change in the component happens that requires restructuring the template format so that it is incompatible with the component before

the change. If only minimal changes to the template format happen which don't affect the compatibility then the minor version number is incremented. This is the case if for example new elements are added to the template. An older version of the report component can't process these new elements but it still must be able to process the template and create a report. Changes to the structure inside the `template` element don't affect the version number because this structure is specific to the class which generates the report.

The namespace of the root element is defined in its attribute `xmlns` and must be `http://www.codecover.org/xml/report-template`.

The name of the template is set with the element `name` and a short description is set with the element `description`. The attribute `xml:lang` is used to indicate the language of the name and description. The values of the attribute are language identifiers as defined by RFC 4646[1]. The name and the description can be multiply defined with different language identifiers to supply the name and description in different languages. The Eclipse-Plugin can use this attribute to identify the name/description which language matches the one set in the GUI.

The attribute `version` of the element `template` sets the version of the template and is specific to the class which generates the report. It is recommended to use this version in the implementation of a report generating class to assure compatibility of the template and the class.

The content of the element `template` (indicated by `...` in the above example) is specific to the class which generates the report. Typically the element `template` contains a new level of subelements and these subelements contain CDATA-sections which contain the real templates with placeholders, see 3.4.5 for an example. The namespace of the element `template` is defined in its attribute `xmlns`. It must be unique and is recommended to be `http://www.codecover.org/xml/report-template/` plus a name of the output format of the template. For example the hierarchical HTML Report uses `html-hierarchic` as the name and the full namespace is
`http://www.codecover.org/xml/report-template/html-hierarchic`.

If the template is saved as a file in the filesystem the recommended filename is the name defined in element `name` whereas spaces are translated to underscores and potentially problematic characters like /, \ and other non-basic latin letters should be avoided in

[1]RFC 4646, Tags for the Identification of Languages: http://www.rfc-editor.org/rfc/rfc4646.txt

the filename. The recommended file extension is `xml`. The filename for the hierarchical HTML Report is `HTML_Report_hierarchic.xml` for example.

3.4.4 Extensibility

The report component can be extended to support further output formats by implementing a new report generator and specifing a new template format.

Since the report component is independent from programming language, no costumization is needed to support new programming languages.

3.4.5 Hierarchical HTML Report

3.4.5.1 Structure

The hierarchical HTML Report consists of three types of pages which are described in the specification. This document only describes how the hierarchical HTML report generator maps the structure of the data model onto the structure of the report.

Every report has exactly one title page. It lists the topmost elements of the hierarchical code structure. Every listed element is linked to its corresponding detail page. A detail page is either a selection page or a code page.

All elements above the level of methods (see 3.4.1) are described in selection pages. Selection pages describe their corresponding element and contain a list of the direct children of the corresponding element. Every listed element is linked to its corresponding detail page. Methods are described in code pages which are the deepest hierarchical level of the hierarchical HTML report.

In Java the only structural types above the level of methods are packages and classes which are described in selection pages.

In COBOL methods, which are described by code pages, are sections. COBOL-programs, which can be nested to arbitrary depth, are described by selection pages. If a program contains code without an enclosing section, it is described in an extra code page which is put on the same level (in the report) as the code pages of the sections of the program.

To avoid redundancy of code listings, the code of sub-programs is omitted in code pages. For example in the report of figure 3.5 the code page of section SB wouldn't neither

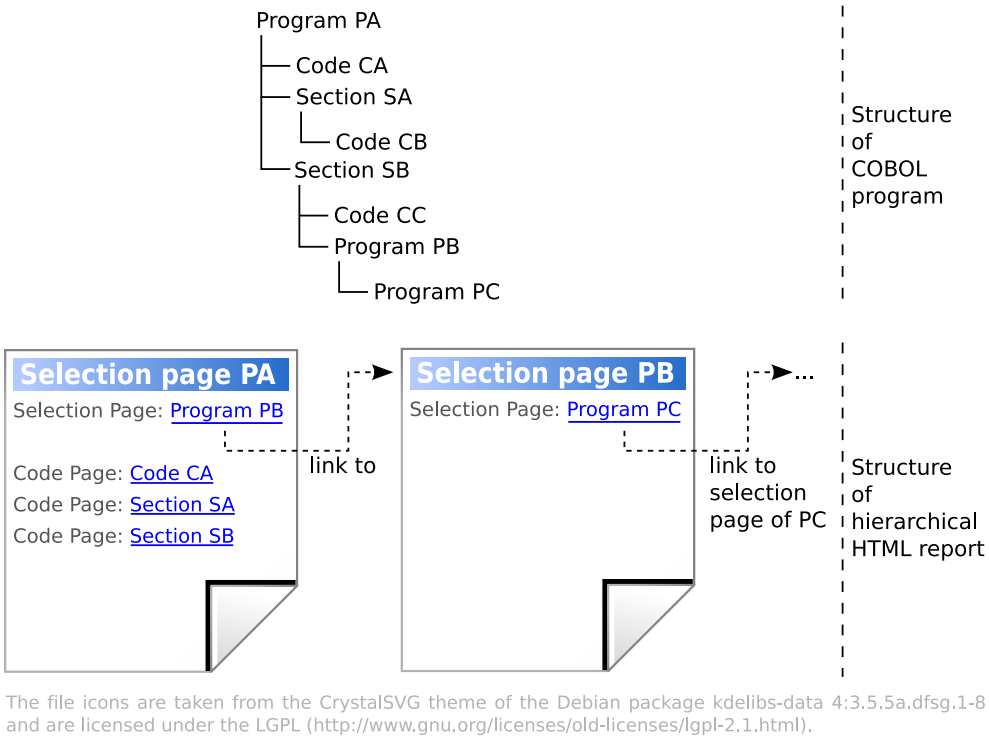

Figure 3.5: Structure of hierarchical HTML report of COBOL programs

contain the code of program PB nor PC. The selection page for program PC and the code pages of code CA, section SA and SB are omitted in figure 3.5 but they would exist in the real report of course.

3.4.5.2 Generation

Generating a report is basically done by traversing the tree structure of the data model and generating the corresponding selection or code page for each element of the tree. Every page contains metrics about its corresponding element. To calculate these metrics, the hierarchical HTML report generator uses the metrics component (see 3.3).

Apache Velocity[2] is used as the template engine. This works as follows:

1. An instance of the Apache Velocity template engine is created.
2. A so-called Context is filled with the data that will be inserted into the template HTML code for the title page. The inserted data is for example the achieved statement coverage for the whole SUT or the names of the top-level packages (if

[2]Apache Velocity: http://velocity.apache.org/

it's a report about a Java SUT). To fill the Context, the actual value and a key to this value are saved. The key of the value is the name of the placeholder in the template HTML code.

3. The template HTML code is *merged* with the Context, that is the placeholders are replaced with the real data.
4. Now a Context is created for each selection page. In case of Java it is a Context for each package and class. In case of COBOL it is a context for each program.
5. The Contexts are merged with the template HTML code for selection pages. This creates the selection pages of the report.
6. Now a Context is created for each code page, that is for each method (see 3.4.1).
7. The Contexts are merged with the template HTML code for code pages. This creates the code pages of the report.

3.4.5.3 Template format

The template format is extended as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<report version="1.0" xmlns="http://www.codecover.org/xml/report-template">
    <plugin>org.codecover.report.html</plugin>
    <generator>org.codecover.report.html.HierarchicalHTMLReportGenerator</generator>
    <name xml:lang="en">HTML Report (hierarchic)</name>
    <name xml:lang="de">HTML Report (hierarchisch)</name>
    <description xml:lang="en">
        Generates a hierarchical report in HTML-format.
    </description>
    <description xml:lang="de">
        Erstellt einen hierarchischen Report in HTML.
    </description>
    <template
      version="1"
      xmlns="http://www.codecover.org/xml/report-template/html-hierarchic">
        <language>de</language>
        <title-page><![CDATA[
            ...
```

```
        ]]></title-page>
        <selection-page><![CDATA[
            ...
        ]]></selection-page>
        <code-page><![CDATA[
            ...
        ]]></code-page>
        <text-file filename="style.css" content-type="text/css"><![CDATA[
            ...
        ]]></text-file>
        <resource filename="logo.png">...</resource>
    </template>
</report>
```

The filename of the template is `HTML_Report_hierarchic.xml`.

The version number defined in the attribute `version` of element `template` must be incremented if any changes to the HTML report generation happen that affect the structure or semantics of placeholders inside the element `template`.

The element `language` identifies the language of the generated report and is a two-letter code according to ISO 639-1.

There are three new elements which contain the template HTML code for title (`title-page`), selection (`selection-page`) and code pages (`code-page`). The HTML code of the template contains placeholders for the data which will be inserted in the template by Apache Velocity.

The two elements `text-file` and `resource` are used to provide additional text files and binary resources. The `text-file`-element contains the text (e.g., a CSS stylesheet) which is to be written into a text file with the content type specified in the attribute `content-type`. The `resource`-element contains the content of a binary file encoded in Base64[3]. Both elements have the attribute `filename` which denotes the path to the file relative to the output directory of the report.

[3]Base64 is specified in RFC 4648: ftp://ftp.rfc-editor.org/in-notes/rfc4648.txt

3.4.6 Single-file HTML Report

3.4.6.1 Structure

This type of report consists of only one HTML page. It is divided into an overview section and a code section. The overview section lists the units (e.g. classes, methods) of the SUT and their corresponding metrics. Moreover it contains some statistics on the structure of the SUT (e.g., the number of classes) and information about the test cases the report is about. The code listings of all source files are contained in the code section.

3.4.6.2 Generation

To generate the single-file HTML report nearly the same procedure as described in 3.4.5.2 is used. The only difference is that only one Context is created which contains the data to be inserted into the template of the file.

3.4.6.3 Template format

The template format is extended as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<report version="1.0" xmlns="http://www.codecover.org/xml/report-template">
    <plugin>org.codecover.report.html</plugin>
    <generator>org.codecover.report.html.SingleFileHTMLReportGenerator</generator>
    <name xml:lang="en">HTML Report (single-file)</name>
    <name xml:lang="de">HTML Report (eine Datei)</name>
    <description xml:lang="en">
        Generates a single-file report in HTML-format.
    </description>
    <description xml:lang="de">
        Erstellt einen Report in einer einzigen HTML-Datei.
    </description>
    <template
      version="1"
      xmlns="http://www.codecover.org/xml/report-template/html-single-file">
        <language>de</language>
        <html-page><![CDATA[
```

```
            ...
        ]]></html-page>
    </template>
</report>
```

The filename of the template is `HTML_Report_SingleFile.xml`.

The version number defined in the attribute `version` of element `template` must be incremented if any changes to the report generation happen that affect the structure or semantics of placeholders inside the element `template`.

The element `language` identifies the language of the generated report and is a two-letter code according to ISO 639-1.

The template HTML code is defined with the element `html-page`.

3.5 Batch

This component encompasses all the packages and classes necessary for the command line interface.

3.6 Eclipse

This component contains all the packages and classes which are related to the Eclipse plugin part of *CodeCover*.

3.6.1 Data management

The classes and interfaces mentioned in this section are contained in the package `org.codecover.eclipse.tscmanager` except for the class `TestSessionContainer` which is contained in the package `org.codecover.model`.

3.6.1.1 Terminology

The Eclipse plugin of *CodeCover* can handle multiple test session containers. Each test session container is associated with a specific project in Eclipse. Eclipse projects are

stored in the *workspace* of Eclipse. The files of the test session containers are stored in the *CodeCover-folder* of the associated project. The *CodeCover-folder* is a folder on the root level of each Eclipse project which contains test session containers. Thus the user just has to copy the folder of an Eclipse project to get a full backup of the project including his coverage measurements.

A Test session container which is associated with a currently *open* project is called a *known test session container*. The reason for this term is that files and folders which reside in closed projects aren't accessible and thus can't be "known" by the plugin.

The *active test session container* is the known test session container which contains the *active test cases* which are currently visualized in the plugin's views, e.g. the Coverage view and the Test Sessions view. The *active test cases* can be selected in the Test Sessions view, see the specification document for details.

A *test element* is either a test session or a test case.

3.6.1.2 TSContainerManager

The central component which handles the data management in the Eclipse plugin is the `TSContainerManager` which provides the following functions:

- Providing methods to select the active test session container and the active test cases
- Providing access to the currently active test session container (to add/delete/modify test elements)
- Providing the currently active test cases
- Providing a list of the currently known test session containers
- Providing methods to add and delete test session containers
- Notification of listeners (e.g., the views of the plugin) about changes
- Saving and loading the test session containers
- Saving and loading which test cases are active

3.6.1.2.1 Storing the test session containers

The implemented concept of handling multiple test session containers with the Eclipse plugin is simple. As mentioned before each test session container is associated with a

project and stored in the *CodeCover*-folder of this project. This means adding a test session container to a project is as simple as copying the file to the *CodeCover*-folder of the project.

On startup of the plugin the `TSContainerManager` scans all *CodeCover*-folders of all open projects and loads all files of the *CodeCover*-folders once to determine which test session containers they contain. Since holding all test session containers in memory would consume to much memory, only a representation of each found test session container is stored in the `TSContainerManager`. This representation is an object of class `TSContainerInfo` and stores the following information about the represented test session container:

- the path to the file of the test session container
- the associated project
- the creation date of the test session container
- the names of the active test cases of the test session container

The path is used as a unique ID for each test session container and makes it possible to distinguish known test session containers from each other.

The managed list of known test session containers can of course change during the execution of the plugin if:

- the user opens a project: the newly accessible test session containers are added to the list
- the user closes a project: the test session containers contained in (associated with) the project are removed from the list
- the user creates a test session container by running a coverage measurement: the test session container is added to the list
- the user imports a test session container into a project: the test session container is added to the list
- the user deletes a test session container: the test session container is removed from the list

The first two cases are detected by listening to changes in the workspace of Eclipse, which is done by `WorkspaceListener`, and then taking the appropriate action (adding/removing). The other three cases are detected by the `TSContainerManager` itself because adding

and deletion of test session containers are handled by the TSContainerManager.

3.6.1.2.2 Active test session container

As mentioned before it would be too resource-consuming to hold all known test session containers in memory, which is the reason why only the active test session container is actually represented by an object of class TestSessionContainer and can be modified. To avoid inconsistencies the TSContainerManager provides this TestSessionContainer as a compound together with the respective TSContainerInfo-representation and the active test cases in the form of an object of class ActiveTSContainerInfo.

Activating an other test session container is done by passing the TSContainerManager the TSContainerInfo-representation of the test session container to activate. The TSContainerManager then loads the test session container from its file into memory.

Since Eclipse is a multi-threaded application, the plugin has to take care of concurrent modifications. To apply changes to the active test session container one has to provide an ActiveTSContainerRunnable which is then passed to the TSContainerManager and synchronized with other modifications to the active test session container.

3.6.1.2.3 Active test cases

The active test cases can be selected by a component of the plugin by passing a set of test cases to the TSContainerManager. In the perspective of the TSContainerManager the active test cases are just a set of test cases of the currently active test session container, which are provided to the components of the plugin which visualize them.

To be able to ”remember” the active test cases of a test session container after an other test session container was activated, the names of the active test cases are stored in the TSContainerInfo-representations of the known test session containers as objects of class TestCaseInfo. When a test session container is activated the set of active test cases can be restored by reading the stored names (the TestCaseInfo objects) from the TSContainerInfo-representation. This information is also stored on disk by ActiveTestCasesStorage and ActiveTestCasesSaveParticipant to be able to ”remember” the active test cases after a restart of the plugin.

3.6.1.2.4 Listeners

There are two ways to access the information managed by the TSContainerManager. One way is to actively get the information by calling the methods of TSContainerManager. The passive approach is to register a component as a listener, which is then notified of

changes in the TSContainerManager. Listeners, which must implement the interface TSContainerManagerListener, are informed about changes of:

- the selection of the active test session container
- the active test session container itself (e.g., deletion of a test case)
- the selection of active test cases (of the active test session container)
- the list of known test session containers (i.e., a test session container was added / removed)

The listeners are handled by the TSContainerManagerListenerHandler.

3.6.1.2.5 Saving and loading

The easiest way of implementing the saving of test session containers, that is the process of writing them to disk, would have been to perform a save instantly after a change had been performed. Since saving is a resource-consuming operation, this approach isn't feasible. To achieve the goal of minimizing save operations, saving is only performed when an other test session container is activated or the associated project is closed. Since only the active test session container is kept in memory, the save operation can't be deferred any further (else the changes would be lost).

The actual operation of reading and writing the files of the test session containers is already implemented by the class TestSessionContainer and called by the class TSContainerStorage which handles loading and saving of test session containers.

For saving test session containers when a project is closed, a special treatment has to be undertaken because during the close-event (propagated by Eclipse) the workspace is locked for changes and thus saving, which implicates a modification of the workspace, can't be performed. Therefore the save operation is queued in the TSContainerSaveQueue until Eclipse requests the plugin to perform saving. This request can be detected by implementing a org.eclipse.core.resources.ISaveParticipant which is then registered to receive save requests from Eclipse. TSContainerManagerSaveParticipantHandler is the class which receives the requests and distributes them to all registered SaveParticipants. The two save participants are the TSContainerSaveParticipant and the ActiveTestCasesSaveParticipant. The former saves the active test session container if it changed since it was last saved and furthermore performes the save operations queued in the TSContainerSaveQueue. The latter saves which test cases are active (see 3.6.1.2.3) and which test session container is active.

3.6.2 Build and Run

This section covers the parts of the Eclipse plugin which are relevant for the build of the instrumented SUT, its execution and necessary post-processing.

3.6.2.1 Building the instrumented SUT

CodeCover doesn't do incremental instrumentation yet, so whenever a build is triggered by Eclipse a full build needs to be done. The build process of an instrumented SUT basically works like this: *CodeCover* participates in the build of the uninstrumented SUT with the help of `CodeCoverCompilationParticipant`, which allows *CodeCover* to perform additional actions before the actual build This includes searching for a test session container which matches the current code base, or creating a new one if necessary. As a second step the instrumentation of the sources is also performed there. The instrumented sources are placed in a project dependent location in the metadata section of the workspace. Last step is the compilation of the instrumented sources by invoking the Eclipse java compiler with the compiler settings of the project.

3.6.2.2 Running the instrumented SUT

The compiled instrumented SUT has been placed in a project dependent location in the workspace metadata section by the `CodeCoverCompilationParticipant`. In order to execute the instrumented object code, a redirection from the uninstrumented object to the instrumented object code needs to be performed. This is done with help of the `CodeCoverClasspathProvider` which adds another classpath entry to the top of the list of classpath entries which points to the directory where the instrumented object code is stored. Since the java runtime searches for classes in the order of the classpath entries, the instrumented classes will be found first, leading to the execution of the instrumented SUT. A `ClasspathProvider` is bound to a specific launch configuration. By using the `CodeCoverTab`, which is added to the list of tabs which are shown for a launch configuration, the user is able to set and unset the `CodeCoverClasspathProvider` for the opened launch configuration and thereby be able to quickly switch between runs of the instrumented and uninstrumented SUT.

3.6.2.3 Post-execution actions

After the termination of a SUT certain actions need to be performed. One important action is the import of the coverage log which resulted from the execution of the SUT. In order to track the execution of a SUT a `CodeCoverDebugListener` is registered with Eclipse. On SUT termination `CodeCoverDebugListener` receives a respective event. This event triggers functionality which searches for a test session container to whom the coverage log belongs to. In case the search is successful, the coverage log is imported into the test session container and the views are updated to show the new data.

3.6.3 Annotation

The classes and interfaces described in this section are contained in the package `org.codecover.eclipse.annotation`. Other classes are mentioned with their qualified names where they could be confused easily.

This package adds coverage highlighting of the active test cases to the default java editor of eclipse. This is done by attaching an implementation of `org.eclipse.jface.text.source.IAnnotationModel` to every java editor as it is opened. The model then queries the `org.codecover.report` to find out what information to annotate where. It adds an `org.eclipse.jface.text.source.Annotation` to the editor for each consecutive region of text to be highlighted in a certain way. The layout of these annotations is defined via `org.eclipse.ui.editors.markerAnnotationSpecification` .

3.6.3.1 Attaching the model

The `org.codecover.eclipse.CodeCoverPlugin` makes sure that an instance of `EditorTracker` exists while the plugin is running. `EditorTracker` registers listeners with Eclipse to get informed whenever a `org.eclipse.ui.IWorkbenchPart` is created. For every `org.eclipse.ui.texteditor.ITextEditor` that is opened it calls `CoverageAnnotationModel.attach(...)`, which adds a new instance of `CoverageAnnotationModel` to the editors `org.eclipse.jface.text.source.IAnnotationModelExtension`. From then on all annotations provided by this model will be displayed among the usual annotations that editor displays.

3.6.3.2 The annotation model

`CoverageAnnotationModel` implements an annotation model to serve `EclPositionedAnnotation`s to the editor. They are subclassed as required to show every kind of highlighting including hotpath. The model is also responsilbe for deciding when and what to annotate.

`CoverageAnnotationModel` listens for changes in the document and in the data model of *CodeCover*. On every change it uses `org.codecover.eclipse.utils.EclipseMastLinkage` to find the corresponding `SourceFile` to the `ICompilationUnit` displayed in the editor. If it finds a SourceFile and it's content is equal to the editor's content the file is annotated.

The `CoverageAnnotationModel.createAnnotations()` is responsible for querying the model what to annotate and creating all `EclPositionedAnnotation`s the user sees. To measure the coverage of source files it queries `org.codecover.highlighting.CodeHighlighting`. Currently it just converts the results of `annotateCoverage(...)` and `generateLineAnnotationsByFile()` into `EclPositionedAnnotation`s.

Don't confuse the annotations subclassing `org.eclipse.jface.text.source.Annotation` with those in `org.codecover.report.highlighting.annotation`. The *former*, prefixed with Ecl, are those you see in your Eclipse editor. The *latter* make up the source file oriented view of what characters were covered how, using a specific set of metrics and test cases.

3.6.3.3 Layout of annotations

In Eclipse every `org.eclipse.jface.text.source.Annotation` has a type string that is defined in `plugin.xml`. This type decides which layout is used to display the annotation in the editor.

There are two kinds of annotations displayed in Eclipse. The first is `EclCoverageAnnotation` which is used to show the user which elements are covered. The type of these annotations has the form `org.codecover.eclipse.annotation.*Coverage*Annotation`. At the first `*` the level of coverage is inserted (one of `full`, `partial` and `no`). At the second `*` the kind of coverage metric is inserted (one of `Other` – for user defined metrics – and `Branch`, `Conditional`, `Loop` and `Statement` – for the bundled metrics). All of these annotations have their default color layout defined in `plugin.xml` and are defined to be configurable individually in the Eclipse preferences dialog.

The second kind of annotation is `hotpath.EclLineExecutionAnnotation` which is used to show the hot path icons in the editor. It's type is `org.codecover.eclipse.annotation.lineExecutionAnnotation`. The class `LineExecutionImageProvider` generates the icons the user sees based on the attributes of `hotpath.EclLineExecutionAnnotation`. The mapping between annotation and color is implemented in `generateIcon(EclLineExecutionAnnotation)`.

3.7 Package overview

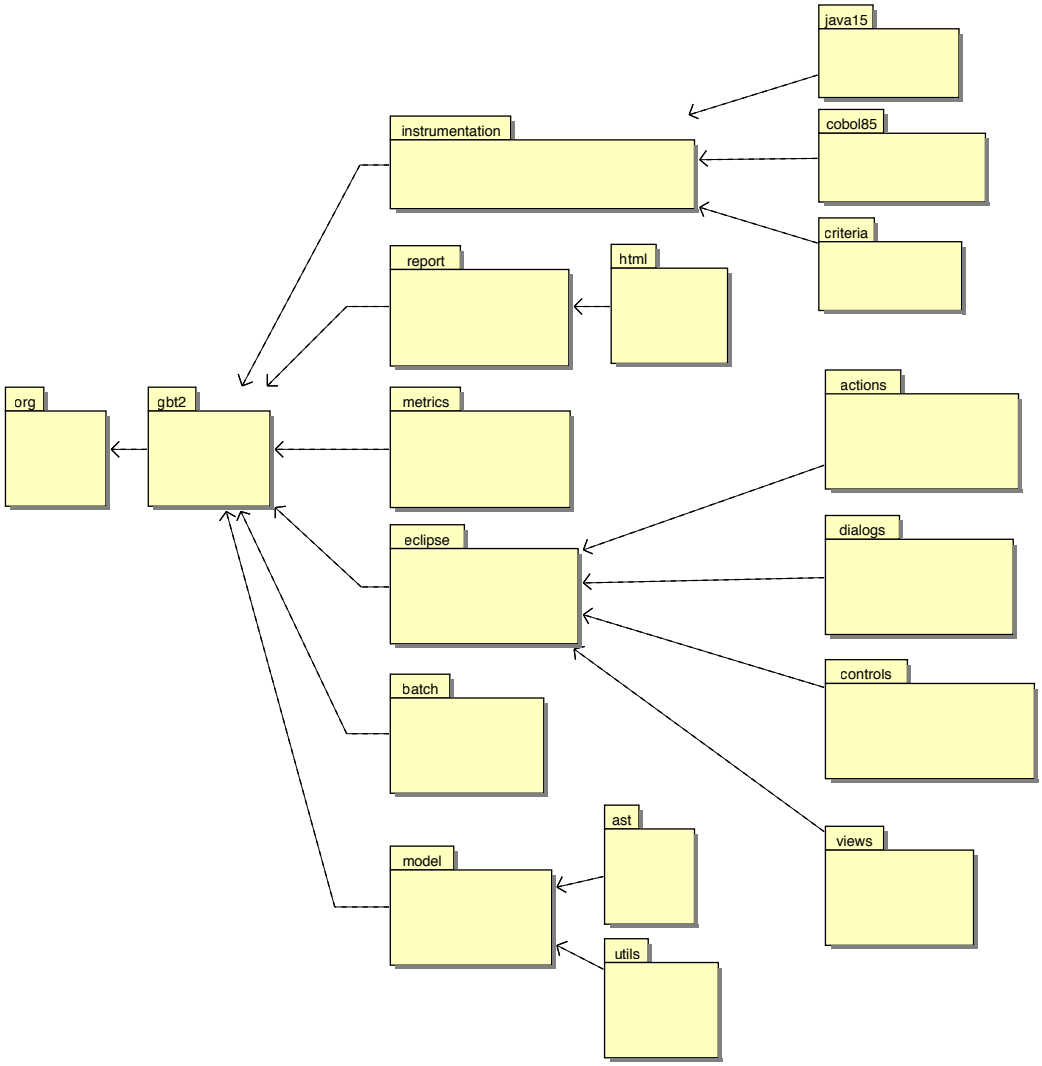

Figure 3.6: Package overview

This section provides an overview of the deployment plan of the software. The domain used for the development of the software is used as the common root for the packages, meaning all packages are in the `org.codecover` namespace. As seen in figure 3.6 the components from section 3 are represented as the six main packages. They are further refined through packages of their own, e.g. the packages for the java instrumentation (`java15`) and the COBOL instrumentation (`cobol85`).

List of Figures

A Formal Proof Of Conditional Coverage Instrumentation

A.1 Java Language Specification

We have created source code examples in Java and have instrumented them *by hand*. Especially the instrumentation for the conditional coverage is very tricky. Therefore we want to proof, that the semantic of the boolean terms is not effected and is equal to the instrumented boolean terms.

First there are quotations of the Java Language Specification—Third Edition[4].

> **§15.23**
> The **&&** operator is like &, but evaluates its right-hand operand only if the value of its left-hand operand is true. [..] At run time, the left-hand operand expression is evaluated first; if the resulting value is false, the value of the conditional-and expression is false and the right-hand operand expression is not evaluated. If the value of the left-hand operand is true, then the right-hand expression is evaluated; the resulting value becomes the value of the conditional-and expression.

> **§15.24**
> The || operator is like | , but evaluates its right-hand operand only if the value of its left-hand operand is false. [..] At run time, the left-hand operand expression is evaluated first; if the resulting value is true, the value of the conditional-or expression is true and the right-hand operand expression is not evaluated. If the value of the left-hand operand is false, then the right-hand expression is evaluated; the resulting value becomes the value of the conditional-or expression.

[4] `http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`

A.2 Predefinitions

Lets now discuss a number of functions that transform boolean terms. We use ϕ for the boolean environment, that assigns every boolean expression either true or false.

$$\begin{aligned} f_L(T) &:= (S \,\&\&\, T), \phi(S) = true \\ f_R(T) &:= (T \,\&\&\, U), \phi(U) = true \\ \phi(f_L(T)) &= \phi(f_R(T)) = \phi(T) \end{aligned} \tag{1}$$

For $f_L(T)$: according to §15.23, first the value of S is evaluated. Only if $\phi(S)$ is true, which is certain, then T is evaluated. To sum this up, $\phi(f_L(T))$ is true if, and only if, $\phi(T)$ is true.
For $f_R(T)$: according to §15.23, first the value of T is evaluated. Only if $\phi(T)$ is true, then U is evaluated (to true). Moreover $\phi(f_R(T))$ is true if, and only if, $\phi(T)$ is true.

$$\begin{aligned} g(T) := f_R \circ f_L(T) = f_R(f_L(T)) = ((S \,\&\&\, T) \,\&\&\, U), \phi(S) = \phi(U) = true \\ \phi(g(T)) \overset{\text{def. of } g}{=} \phi(f_R(f_L(T))) \overset{eq.\ (1)}{=} \phi(f_L(T)) \overset{eq.\ (1)}{=} \phi(T) \end{aligned} \tag{2}$$

According to equation (1), which is used twice, $\phi(g(T))$ is true, if, and only if, $\phi(T)$ is true.

$$\begin{aligned} h(T) := (T \,||\, V), \phi(V) = false \\ \phi(h(T)) = \phi(T) \end{aligned} \tag{3}$$

According to §15.22.4, the || operation evaluates the left operant (T) first. If it is true, the whole || expression is evaluated to true. Is the first operant (T) false, then the second operant (V) determines the result of the whole expression. So $\phi(h(T))$ is false if and only if $\phi(T)$ is false.

$$\begin{aligned} i(T) := h \circ f_R \circ f_L(T) = h(f_R(f_L(T))) \\ i(T) = ((S \,\&\&\, T) \,\&\&\, U) \,||\, V,\ \phi(S) = \phi(U) = true, \phi(V) = false \\ \phi(i(T)) \overset{\text{def. of } i}{=} \phi(h(f_R(f_L(T)))) \overset{eq.\ (3)}{=} \phi(f_R(f_L(T))) \overset{eq.\ (1)}{=} \phi(f_L(T)) \overset{eq.\ (1)}{=} \phi(T) \end{aligned} \tag{4}$$

According to equations (1) and (3) $\phi(i(T))$ is true, if, and only if, $\phi(T)$ is true.

Using Java syntax, we can consider some terms:

$$\begin{aligned} \text{INITIAL} &:= (((\text{intBitMask } = 0) == 0) \,||\, true) \\ \phi(\text{INITIAL}) &= true \\ \text{USAGE} &:= ((\text{intBitMask} \,|= 1 == 0) \,||\, true) \\ \phi(\text{USAGE}) &= true \\ \text{RESULT} &:= ((\text{intBitMask} \,|= 2 == 0) \,||\, true) \\ \phi(\text{RESULT}) &= true \end{aligned} \tag{5}$$

The evaluation of the left operands is true for the first example and false for the second and third example. But as §15.22.2 says, for the right operands always evaluating to true, ϕ(INITIAL), ϕ(USAGE) and ϕ(RESULT) are true too.

Last but not least we need a description of a Java method:

```
public boolean add(int intBitMask) {
    [..]
    return true;
}
```

$$\begin{aligned} \text{STORE-T} &:= (\text{add(intBitMask)} \,||\, true) \\ \phi(\text{STORE-T}) &= true \\ \text{STORE-F} &:= (\text{add(intBitMask)} \,\&\&\, false) \\ \phi(\text{STORE-F}) &= false \end{aligned} \tag{6}$$

The evaluation of `add(intBitMask)` is true for both equations. The usage of the $||\, true$ does not change this semantic. According to §15.22.4: if the first term is evaluated to true, the second term is not evaluated anymore and can not affect the evaluation. So ϕ(STORE-T) is known to be true.
The usage of the $\&\& \, false$ in the second term sets the whole term to false. According to §15.22.3: if the first term is evaluated to true, which is the case, the second term is considered. If and only if this term is true too, which is not the case, the whole expression is evaluate to true. So ϕ(STORE-F) is known to be false.

A.3 Consideration Of The Instrumentation I

CodeCover will instrument every condition of `if`, `while` and `for` statements. Every basic boolean term of these conditions is instrumented by using function g (see equation (2)). The terms S and U are replaced by USAGE and RESULT. Two examples will illustrate the instrumentation.

```
if (position == 0)
```

```
if ( (((intBitMask1 |= 1 == 0) || true) &&
      (position == 0)) &&
       ((intBitMask1 |= 2 == 0) || true))
```

```
if ((position == 0) ||
  list.isEmpty())
```

```
if ( ((((intBitMask2 |= 1 == 0) || true) &&
        (position == 0)) &&
         ((intBitMask2 |= 2 == 0) || true)) ||
       ((((intBitMask2 |= 4 == 0) || true) &&
        (list.isEmpty())) &&
         ((intBitMask2 |= 8 == 0) || true)))
```

As discussed after equation (2), the semantic of the basic boolean terms is not changed. In addition to that, the bit mask is used to get to know, whether an basic boolean term is evaluated or not. Moreover the result of the evaluation can be stored in the bit mask too.

A.4 Consideration Of The Instrumentation II

Unfortunately we need to add more instrumentation tags. This is needed because we have to store the full evaluation of the conditional terms within the `if`, `while` or `for` statements. Two examples illustrate these extensions.

```
if (position == 0)
```

```
if ( ((((((intBitMask1 = 0) == 0) || true) &&
        ((((intBitMask1 |= 1 == 0) || true) &&
          (position == 0)) &&
         ((intBitMask1 |= 2 == 0) || true))) &&
       (counter1.add(intBitMask1) || true)) ||
      (counter1.add(intBitMask1)) && false) )
```

```
if ((position == 0) ||
  list.isEmpty())
```

```
if ( ((((((intBitMask1 = 0) == 0) || true) &&
        (((((intBitMask2 |= 1 == 0) || true) &&
          (position == 0)) &&
         ((intBitMask2 |= 2 == 0) || true)) ||
        ((((intBitMask2 |= 4 == 0) || true) &&
          (list.isEmpty())) &&
         ((intBitMask2 |= 8 == 0) || true)))) &&
       (counter1.add(intBitMask1) || true)) ||
      (counter1.add(intBitMask1)) && false) )
```

The whole boolean expression will be instrumented using function i (see equation (4)). The terms S, U and V are replaced by INITIAL, STORE-T and STORE-F. So the semantic is not changed.
But what is this instrumentation doing? The INITIAL is needed to reset the bit mask to zero before starting to measure the evaluations of the basic boolean terms. The STORE-? method calls are needed to store the bit mask for every time, the whole expression has been evaluated. And only one of the STORE-? methods is evaluated.
Is the whole (expression) true, than the STORE-T is evaluated, as a result of the && operator. This evaluation does not change the semantic and the expression is true again. For this reason the STORE-F is not evaluated, because of the short circuit semantic of the || operator.
Is the whole (expression) false, than the STORE-T is not evaluated, because of the short circuit semantic of the && operator. But in this case the STORE-F is evaluated. For being false, the STORE-F causes the whole expression to be evaluated to false.
So in each case of the evaluation of (expression), only one STORE-? method is called.

by Christoph Müller

B Coverage log file specification

B.1 General description

The coverage log file contains all information of a test session. These are:

- the names of the test cases ≥ 1
- sections for different source files within a test case
- ID of a Coverable item and a counter value

The coverage log file is no XML file, cause this would be too much overhead. This file is a plain text file with a specific grammar. This grammar is presented in the following, using the "Extended Backus–Naur form" as a notation. Some production use Unicode[5] code points and ranges to define the valid characters.

The nonterminal `EOL` is standing for *end of line* and can be `CR (\r)`, `NL (\n)` or `CRNL (\r\n)`. The Literal `"EOF"` is standing for *end of file*. The nonterminal `NotEscaped-Character` is standing for any character of the character encoding, that is no `Escaped-Character` and no *control character*.

Nevertheless, the coverage log file can use any character encoding the *java.nio.Charset* supports (see Supported Encodings[6]).

B.2 EBNF grammar

```
CoverageLogFile = Comment TestCase {Comment | TestCase} {EOL} "EOF";

Comment = {CommentLine};
CommentLine = "//" ExtendedCharacter* EOL;

TestCase = TestSessionContainer StartTestCase [Section] EndTestCase;
StartTestCase = "START_TEST_CASE" " " TestCaseName [" " TimeStamp]
                [" " TestCaseComment] EOL;
```

[5] http://www.unicode.org/
[6] http://java.sun.com/j2se/1.5.0/docs/guide/intl/encoding.doc.html

```
TestCaseName = StringLiteral;
TimeStamp = IntegerLiteral;
TestCaseComment = StringLiteral;
EndTestCase = "END_TEST_CASE" " " TestCaseName [" " TimeStamp]
              [" " TestCaseComment] EOL;

TestSessionContainer = "TEST_SESSION_CONTAINER" " "
                       TestSessionContainerUID EOL;
TestSessionContainerUID = StringLiteral;

Section = (NamedSection {NamedSection}) | UnnamedSection;
NamedSection = StartSection Counter*;
StartSection = "START_SECTION" " " SectionName EOL;
SectionName = StringLiteral;
UnnamedSection = Counter Counter*;

Counter = CounterID " " IntegerLiteral EOL;
CounterID = Character Character* IntegerLiteral {"-" IntegerLiteral};

IntegerLiteral = Digit Digit*;
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

StringLiteral = '"' {ExtendedCharacter} '"';
Character = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
            "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
            "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" |
            "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
            "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
            "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
ExtendedCharacter = Digit | Character | EscapedCharacter |
                    NotEscapedCharacter;
EscapedCharacter = "\n" | "\t" | "\b" | "\r" | "\f" | "\\" | '\"' |
                   "\'";
NotEscapedCharacter = "U+0020" | "U+0021" | "U+0023" .. "U+0026" |
                      "U+0028" .. "U+005B" | "U+005D" .. "U+10FFFF";
```

```
EOL = "U+000D" | "U+000A" | "U+000D" "U+000A";
```

B.3 Example

```
// ////////////////////////////////
// Start Session
// 18.05.2007 19:54:52.797
// ////////////////////////////////
// 18.05.2007 19:36:17.704
TEST_SESSION_CONTAINER "4f97f9b3-9284-4d36-817b-a4bda7714540"
START_TEST_CASE "My Name is \"Test Case 1\"" 1179509777704
START_SECTION "org.codecover.CodeExample"
S2 1
S4 1
C1-1010 2
C1-1100 1
C2-10 2
C2-11 40
C3-10 2
C3-11 38
L1-2 1
L4-0 15
L4-1 8
L4-2 221
END_TEST_CASE "My Name is \"Test Case 1\"" 1179509778775
// 18.05.2007 19:36:18.775
TEST_SESSION_CONTAINER "4f97f9b3-9284-4d36-817b-a4bda7714540"
START_TEST_CASE "Second Class"
START_SECTION "org.codecover.CodeExample"
S20 1
S21 1
START_SECTION "org.codecover.SecondClassOfFile"
C1-10 1
C1-11 2
C2-10 1
C3-10 1
C6-11 1
C7-11 1
C8-1111 1
C21-10101010101010101010101010101010 30
C21-11101110111000000000000000000000 1
C22-10101010101010101010101010101011 30
C22-11100000000000000000000000000000 1
```

```
L4-2 1
END_TEST_CASE "Second Class"
```