Combining source code and test coverage changes in pull requests

Version of January 19, 2015

Sebastiaan Oosterwaal

Combining source code and test coverage changes in pull requests

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Sebastiaan Oosterwaal
born in Rotterdam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Combining source code and test coverage changes in pull requests

Author:	Sebastiaan Oosterwaal
Student id:	1509322
Email:	`sebastiaan.oosterwaal@gmail.com`

Abstract

With the increasing popularity of open-source version control platforms like GitHub, developers get more feedback on the changes they make. A common tool in version control systems is the highlighted difference view of the source code. This enables reviewers to quickly view the changes made. A missing feature is comparison of the test coverage. Since more and more projects develop a test suite to ensure quality, it is worth looking into how a change affects the test coverage of the project.

In this thesis, a tool called *Operias* is proposed and implemented. Operias is a tool which accepts two versions of a project and calculates the difference in both source code and in statement coverage. The tool generates a HTML site, which developers can use to quickly review the changes. It is possible to run Operias as a service and to connect it to GitHub. This way, a report is automatically generated for every pull request that is opened. This helps reviewers to determine whether to accept the pull request or not.

The main contributions of this thesis is the generic description of how the tool Operias works and the actual implementation of the tool. The tool is available on GitHub, and with the tool we conducted a real life evaluation with three open-source projects on GitHub to determine the usefulness of the tool. We found that in many cases Operias can help developers and reviewers during the review process.

Preface

This master thesis was written as part of the Master Computer Science I followed at Delft University of Technology. During this last part of my study, Arie van Deursen and Roberta Coelho were a great help to accomplish this thesis. I want to thank both of them for their support and feedback, but also for their input and ideas on the subject of this thesis. I also want to thank my girlfriend for her support and help during the course of the thesis.

Sebastiaan Oosterwaal
Delft, the Netherlands
January 19, 2015

Contents

List of Figures

Chapter 1

Introduction

Code review and software testing have been key topics for many years. According to the IEEE standard, software testing is the process of analyzing a software item to detect the difference between existing and required conditions and to evaluate the features of the software item [4]. In other words, software testing is a useful way to show the presence of bugs in a program [12]. Throughout the years, many different testing strategies have been developed, each with different criteria, such as statement coverage and branch coverage [38]. Many modern IDEs (Integrated Development Environment) have numerous extensions and tools to calculate coverage metrics for several structural criteria. This makes the analysis of test coverage much easier.

Code review consists of manual inspection of the code done by someone else than the author. This is usually complements systematic testing and is considered a valuable way to reduce software defects and improve the overall quality [5, 6]. Both researchers and developers agree that code review is important and an effective way to ensure quality and share awareness and knowledge [29]. Several big organizations already use code review tools [8]. Most of these tools show the difference in source code in a side-by-side view where developers are able to share their knowledge and comments [29].

Both testing of the software and inspecting the code is part of the co-evolution of software. Software must evolve over time in order to stay useful [22]. Newly added features should be tested as soon as possible to ensure that they work properly [31]. Also, any change to existing code requires a retest of that code, since these changes could potentially invalidate the test suite [25].

In general, any change in the software should include a change in the test code. This co-evolution of the test suite is important. If such co-evolution takes place synchronously, reviews can address changes in both in source code and test coverage. Not every projects test suite co-evolves synchronously, some also use phased testing approaches [36, 37].

In order to review the co-evolution of the project, a review tool should show the highlighted diff changes of the source code and the change in test coverage. Until now, review tools only show the highlighted diff, but a combination with test coverage analysis has not yet been investigated [8].

It is therefore interesting to know in what way software testing and traditional code reviewing techniques can be used to construct a tool which merges the information on source code changes and test coverage analysis based on two versions of a project. Since the evolution of software is best tracked with a version control system, such a tool must be able to use such system to automatically analyze a new version of the code. This way, the co-evolution of the source code and test suite can be tracked during the review process.

1.1 Motivation

There are several tools available to show the difference in the source code and there are numerous to calculate the code coverage but there aren't many tools which try to combine both the source changes and coverage metrics together in one view. Two online available tools are Coveralls.io [21] and SonerQube [32]. These tools will be covered more extensively in chapter 2.

1.1.1 Coveralls.io

Coveralls.io [21] is a webbased application which analyzes the report created by Cobertura [2] by comparing the test coverage metrics to a previous report.

Coverall.io shows an overview with detailed coverage information. It shows the difference in coverage percentage and a view that shows how many lines a class has, how many were covered or missed and the average hits per line. It is easily seen which classes were edited or had changed coverage. A downside is that classes with only a small change in the hits per line are also marked as changed, which makes the overview messy.

While viewing the code of a class, we can see the current state of the coverage. In order to see what actually changed, the button *previous version* on the top of the page must be pressed to manually compare the source files and the changes in coverage.

Because checking these changes manually is often a hard thing to do, many developers would not prefer to use coveralls. The feature of coveralls which can be used in a review process, is the overview table on which the changed files are listed, but also here are sometimes files shown which are irrelevant to the pull request or commit.

1.1.2 SonarQube

SonarQube [32] is an open source product which can be used to support the manual inspection of a project. SonarQube supports multiple languages and can analyze code duplication, coverage, code complexity and more. In this work, only the coverage feature and the combination with the source changes of SonarQube is interesting to look into. Via the webbased application, it is possible to enter a demo site where the different kinds of reports SonarQube produces can be seen.

A view of the changed files between versions as in Coveralls.io can not be found in SonarQube, instead, after selecting a file, it only shows the current coverage information of the class. There are several ways to filter the changes. It is possible to show the lines to cover, branches to cover, uncovered lines and uncovered branches. It is also possible to also select a timeframe, but by default it shows the current state of the file. A time frame of 90 days, 30 days or since the last analysis can be selected. The biggest downside of

SonarQube is that a comparison overview page is missing and that it is impossible to show the difference between two versions of the software, but rather see the state of the code on previous date.

1.1.3 Proposed Approach

The proposed work in this thesis will consist of the construction of a tool, named Operias, which will combine test coverage changes and code changes into one single report. After a short evaluation of the two described tools, we can identify what the tool must be able to produce.

The tool can compare two versions of a software project. It will calculate the code changes and the changes in test coverage. Based on these changes a report is generated which can be used by developers and reviewers to inspect the changes of the new version.

Besides constructing such a tool, the tool will be evaluated by analyzing the reports of the tool based on real changes in open-source projects. To do the evaluation, Operias is extended to communicate with Github and can collect information on pull requests, which represents a change of the project which must be analyzed before it can be accepted.

After showing examples of interesting reports, a short study on the impact of pull request on test coverage is described. With this study, we want to determine whether or not Operias is a useful addition to the review process on Github.

1.2 Outline

The chapters of this thesis are structured as follows. Chapter 2 describes the basic information about code coverage and the used analysis tool Cobertura. Chapter 3 describes the requirements of the tool. After that, chapter 4 describes the implementation details of Operias. The extension to the source code management system Github is described in chapter 5. The evaluation of the tool and its results are described in chapter 6. Limitations of the tool and the future work is written in chapter 7. Finally, a conclusion is given in chapter 9.

Chapter 2

Background

This chapter provides background information about test coverage and the tool Cobertura, which is used to collect test coverage information. Also, more information about Coveralls.io and SonarQube is provided.

2.1 Test coverage

Nowadays, many software development teams test their code to increase the reliability and quality of their products. By testing your program you can show the presence of bugs, but never their absence [12]. According to Bezier, testing is simple, all you need to do is ”find a graph and cover it” [10]. To do this, a test criterion needs to be defined on which the program will be exercised. But what is a test criterion [14, 15]? A test criterion defines what constitutes an adequate test [38]. Or in other words, a test adequacy criterion defines what properties of a program must be exercised to constitute a thorough test, i.e., one whose successful execution implies no detected errors in a tested program [14, 15].

A formal definition of a test adequacy is the function C:

Formula 1: $C : P \times S \times T \rightarrow [0,1]$.

$C(p,s,t) = r$ means that the adequacy of testing program p by the test set t with respect to specification s is of the degree r according to criterion C [39]. In other word, if a program is tested by a given test set, a result between 0 and 1 is produced which denotes how well the program is tested according to a given criterion. In most cases this is presented in percentage.

During many years of research, different test adequacy criteria have been developed. The criteria used in this thesis are said to be program-based structural criteria and based on the control-flow of a program [38]. A program-based criterion is generally used during a unit test, and are used to test relatively small program pieces [35]. Structural testing, sometimes called white-box testing, stands for that the test cases are derived from the internal structure of the program, including branches and individual conditions. [11, 7] In the following sections, the most important structural criteria are described.

2.1.1 Statement Coverage

The very basic control-flow criterion is the statement coverage. It is the simplest and weakest form of testing [18]. It states that all statements in the program must be covered by at least one test case. Unfortunately, full statement coverage is not always possible due to pieces of code which cannot be reached, so called unreachable code, which cannot be found because its not decidable whether a piece of code is unreachable [34].

2.1.2 Branch (or Decision) Coverage

A stronger requirement of an adequate test is called branch coverage [18] or decision coverage. It states that every control transfer must be checked. In other words, for every decision in the program, every possible decision should exercised at least once [27].

Branch coverage is stronger than statement coverage, because if all branches are tested, all statements will be tested. This relationship is called the *subsumes* relation [38]. But in the end branch coverage still is rather weak, especially when decisions have more than one condition. In this case the number of combinations for the input variables increases exponentially, but only the outcome of the decision is taken in consideration.

2.1.3 Condition Coverage

A criterion which can be stronger sometimes is condition coverage [27]. For this criterion, a test case must exist so that each condition in every decision takes all possible outcomes at least once. We say it is sometimes stronger, because if we achieve 100% condition coverage, it does not necessarily means that the decision coverage is 100%. That is because condition coverage does not take the outcome of the decision into consideration. Consider the following example:

```
if (x > 1 && y == 0) {
        DoSomething();
} else {
        DoSomethingElse();
}
```

Now consider the test cases: $x = 1, y = 0$ and $x = 2, y = 1$. In both cases the decision will be false, but every condition was true and false in some case.

2.2 Test Coverage Tools

2.2.1 Cobertura

Cobertura [2] is an open source project based on jcoverage [3]. Using Cobertura, a developer can calculate the line and branch coverage of its code using test cases, made in jUnit[1] for example. Besides coverage, it also calculates the McGabe complexity of every method that is tested. Cobertura can be executed via Ant[2], the command line or Maven[3]. The tool creates HTML and XML reports based on the results.

Instrument

The first step Cobertura does, is instrumenting the compiled classes. Cobertura needs to instrument the classes to add new java byte code to the compiled classes. This is done by using ASM [1]. These extra lines of codes are used to collect metrics for the report. They communicates with code of Cobertura, which collects the information in the next phase.

Testing

When the instrumentation phase is done, the test suite can be executed on the newly created, edited classes of the project in subject. Because of this, Cobertura is independent of the way how the code is tested. This can be either by using jUnit, or for example GUI testing in which case a tester clicks on all buttons to see if all code is covered.

Report

Cobertura uses a datafile (default is Cobertura.ser) to generate the reports of the coverage. This .ser file is generated during the instrument phase and is modified during the execution of the code. All information about the classes and packages is stored in this datafile. The final step is to generate a report, which can be in either XML or HTML format. This way Cobertura can either be used as standalone test coverage tool, or use the generated XML to integrate the result into another application.

Statement Coverage In the report, the statement coverage (Cobertura calls this line coverage) for every package and class is shown. Also a detailed list of which actual line was covered and which not are provided.

Branch Coverage According to Cobertura, the report shows the branch coverage of the code under test. This is rather confusing, because in truth Cobertura shows the condition coverage instead of the decision coverage. The following example shows this. The number

[1]http://junit.org/
[2]http://ant.apache.org/
[3]http://maven.apache.org/

of branches a decision creates according to Cobertura is two times the number of conditions. This is incorrect, because decision coverage should only produces two branches for a decision, no matter how many conditions it consists of.

Two times the number of conditions equals the number of conditions that should be tested according to the condition coverage. Because of wrong naming of the test criterion, confusing results appear in the report. If a method does not have any decisions, there is one branch (or path) for that method, and thus if no statements were covered from that method, that branch was not covered and both branch and statement coverage will be 0%.

Cobertura says the branch coverage is 100% in this case, which is incorrect, unless Cobertura means condition coverage, then it is true that the condition coverage is 100%, since no decision were made in the method and thus no condition were present.

Also because of the short-circuit evaluation in Java, the condition coverage is not always calculated correctly. If we run the example test cases on the code of the previous section, we only get a condition coverage of 75%, because in the first test case, $y == 0$ will not be evaluated.

Nevertheless, Cobertura can serve a reliable tool for computing condition coverage, which is how we will use it in this thesis.

In this Thesis

Cobertura is used in this thesis to retrieve the statement and condition coverage for a project. To achieve this, the tool executes the following command on both versions:

```
   mvn clean cobertura:cobertura -Dcobertura.report.format=XML
-Dcobertura.aggregate=true
```

The argument cobertura.aggregate must be set to true if multi module projects are compared. This aggregate parameter will aggregate all the sub modules cobertura files into one big Cobertura file. After the execution of Cobertura is finished, a XML report can be found in the target directory of the project. This XML is parsed and is later used to retrieve information about any class.

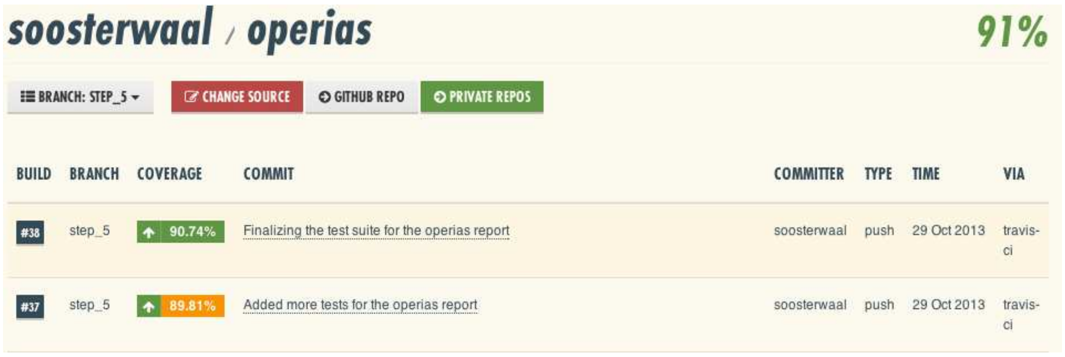
2.2.2 Surefire

Beside the Cobertura coverage information, the tool also collects information about which test case failed or generated an exception. This information is retrieved from the Surefire reports, which are automatically generated by Maven during the execution of the test suite.

A Surefire report is generated for every test class in the test folder of a Maven project. A report contains the information of every test case and the result. If the result is error or failure, it also contains the stack trace of the exception.

2.2.3 Coveralls.io

Coveralls.io [21] is a webbased application which analyzes the report created by Cobertura [2] for every commit or pull request made on Github. When a developer commits or creates a pull request, Cobertura executes the project and create a XML report with the coverage changes. This report is sent to Coveralls.io and compared to the previous report of a commit or branch. On the homepage, Coveralls shows an overview of comparisons of reports and the current coverage percentage, as seen in Figure 2.1. For every report, the change in coverage percentage is shown, on which branch the commit was done, the commiter and when the report was created.



BUILD	BRANCH	COVERAGE	COMMIT	COMMITTER	TYPE	TIME	VIA
#38	step_5	90.74%	Finalizing the test suite for the operias report	soosterwaal	push	29 Oct 2013	travis-ci
#37	step_5	89.81%	Added more tests for the operias report	soosterwaal	push	29 Oct 2013	travis-ci

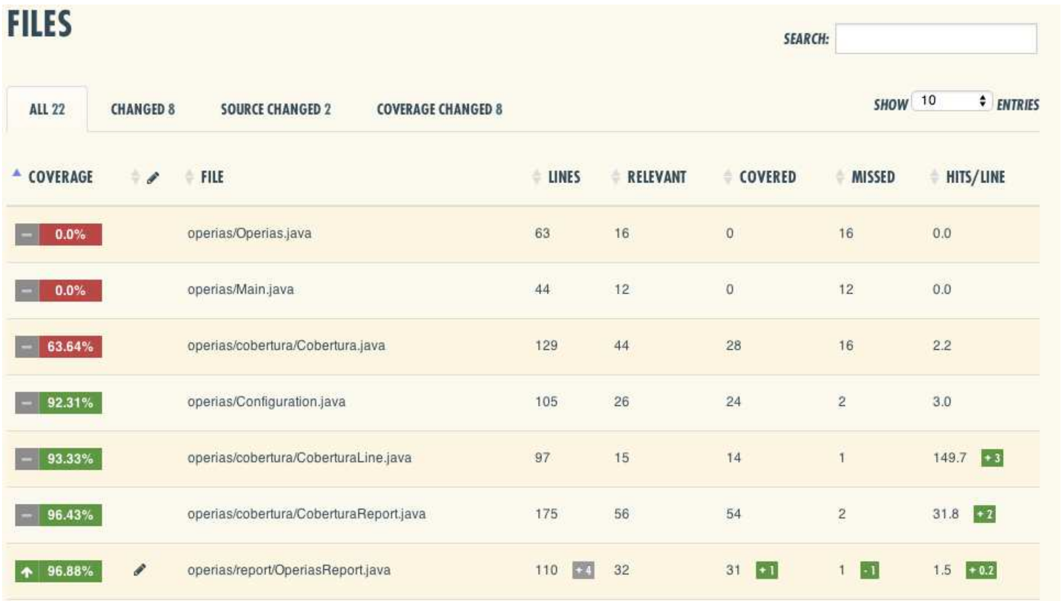
Figure 2.1: Overview of pushed reports

When opening a report, Coveralls shows an overview with the changed classes. It shows the difference in coverage percentage and a view with files with changes in the source and files of which the coverage has changed. This menu is illustrated in figure 2.2.

Something to notice here, is that some files in the list does not have any source changes or any coverage changes. So sometimes the number of changed files on, for example, GitHub differ from the information on the Coveralls website.

Another thing that can happen when a test case changes, is that the average hits per line changes a little bit. In this case, Coveralls notes this as changed coverage. This can make the report look messy and with bigger pull requests the report can become very hard to read. An example of this is seen in Figure 2.3. Here, these classes are in the tab *Coverage Changed*, but only the hits per line was changed. Hits per line is a usesless metric to show in a report like this.

Finally, when opening a changed file, the latest version of the source code and the latest state of the test coverage provided by Cobertura is shown, see Figure 2.4. A green line means that it is covered, a red line means it was not covered. The first thing to notice on such a page, is that it is impossible to see which lines were edited. If it says in the overview table, that the source was changed, we have to click on the button *previous version* on the top of the page to manually compare the source files and the changes. This is also the case for the coverage. It only shows the current state of the coverage on the source code, but it


FILES

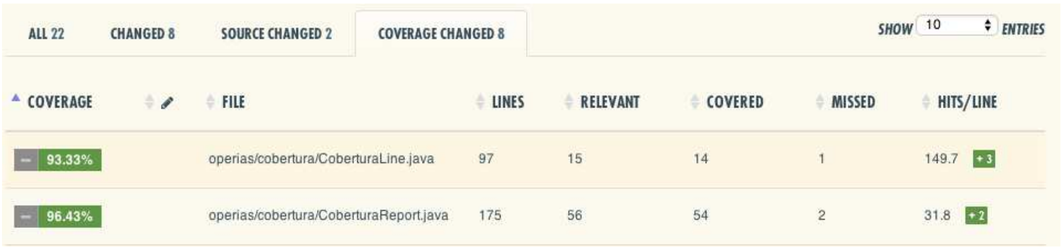
SEARCH:

ALL 22 | CHANGED 8 | SOURCE CHANGED 2 | COVERAGE CHANGED 8

SHOW 10 ENTRIES

COVERAGE		FILE	LINES	RELEVANT	COVERED	MISSED	HITS/LINE
0.0%		operias/Operias.java	63	16	0	16	0.0
0.0%		operias/Main.java	44	12	0	12	0.0
63.64%		operias/cobertura/Cobertura.java	129	44	28	16	2.2
92.31%		operias/Configuration.java	105	26	24	2	3.0
93.33%		operias/cobertura/CoberturaLine.java	97	15	14	1	149.7 +3
96.43%		operias/cobertura/CoberturaReport.java	175	56	54	2	31.8 +2
96.88%	✎	operias/report/OperiasReport.java	110 +4	32	31 +1	1 -1	1.5 +0.2

Figure 2.2: Overview of changed classes


ALL 22 | CHANGED 8 | SOURCE CHANGED 2 | COVERAGE CHANGED 8

SHOW 10 ENTRIES

COVERAGE		FILE	LINES	RELEVANT	COVERED	MISSED	HITS/LINE
93.33%		operias/cobertura/CoberturaLine.java	97	15	14	1	149.7 +3
96.43%		operias/cobertura/CoberturaReport.java	175	56	54	2	31.8 +2

Figure 2.3: Small changes in hits per line will mark the class as changed

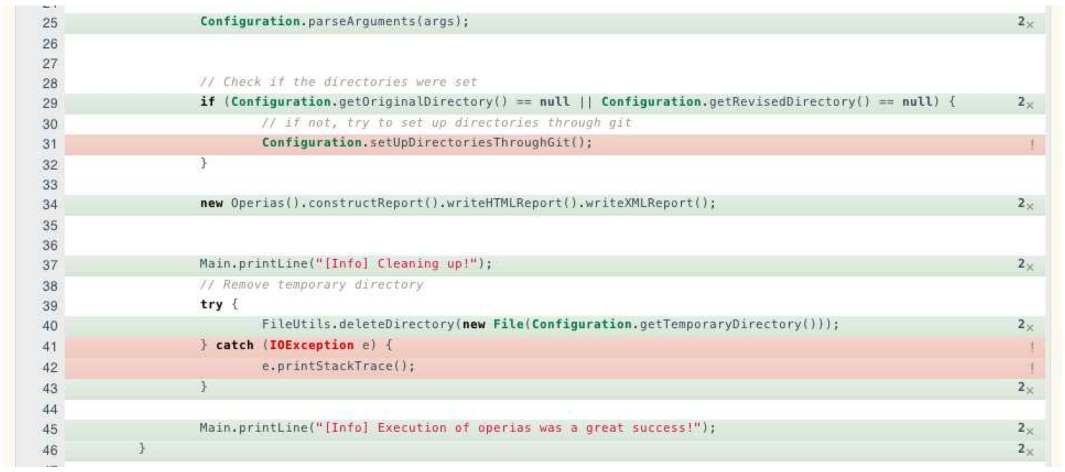
is impossible to actually see the difference. As said in the introduction, manually checking these changes is most of the time impossible due to the size of an average sized class.

2.2.4 SonarQube

SonarQube [32] is an open source product which can be used to support the manual inspection of a project. SonarQube supports multiple languages and can analyze code duplication, coverage, code complexity and more. In our work, only the coverage feature and the combination with the source changes of SonarQube is interesting to look into.

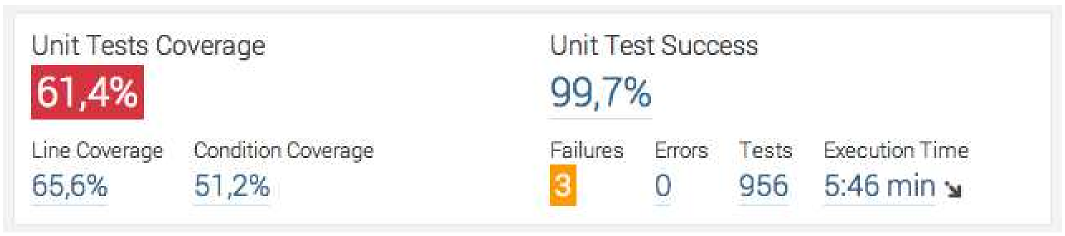
Figure 2.5 illustrates the test coverage part of a report created for an example project, Activity in this case. In this view we see a lot of information and statistics about test coverage. The current line and condition coverage, but also how many unit tests failed and how long it takes to run the test suite is shown.

If we click on this test coverage percentage, we can select a package and eventually a



```
        Configuration.parseArguments(args);                                                                2x


        // Check if the directories were set
        if (Configuration.getOriginalDirectory() == null || Configuration.getRevisedDirectory() == null) {  2x
            // if not, try to set up directories through git
            Configuration.setUpDirectoriesThroughGit();                                                    !
        }

        new Operias().constructReport().writeHTMLReport().writeXMLReport();                                2x


        Main.printLine("[Info] Cleaning up!");                                                             2x
        // Remove temporary directory
        try {
            FileUtils.deleteDirectory(new File(Configuration.getTemporaryDirectory()));                    2x
        } catch (IOException e) {                                                                          !
            e.printStackTrace();                                                                           !
        }                                                                                                  2x

        Main.printLine("[Info] Execution of operias was a great success!");                                2x
    }                                                                                                      2x
```

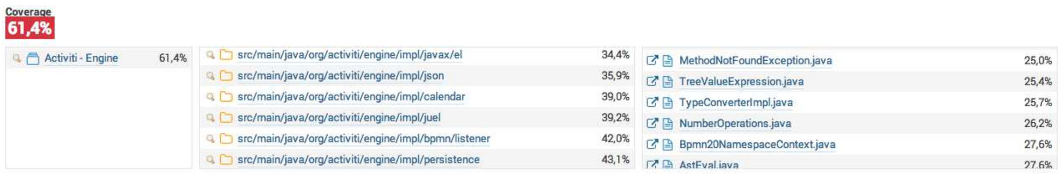
Figure 2.4: Class view of Coveralls only shows the new versions and no visible changes



Unit Tests Coverage		Unit Test Success			
61,4%		99,7%			
Line Coverage	Condition Coverage	Failures	Errors	Tests	Execution Time
65,6%	51,2%	3	0	956	5:46 min

Figure 2.5: Test coverage overview of the Activity project in SonarQube

class, as shown in figure 2.6. This view is not very clear, and again, it only shows the current coverage state of the file.



Coverage
61,4%

Activiti - Engine	61,4%

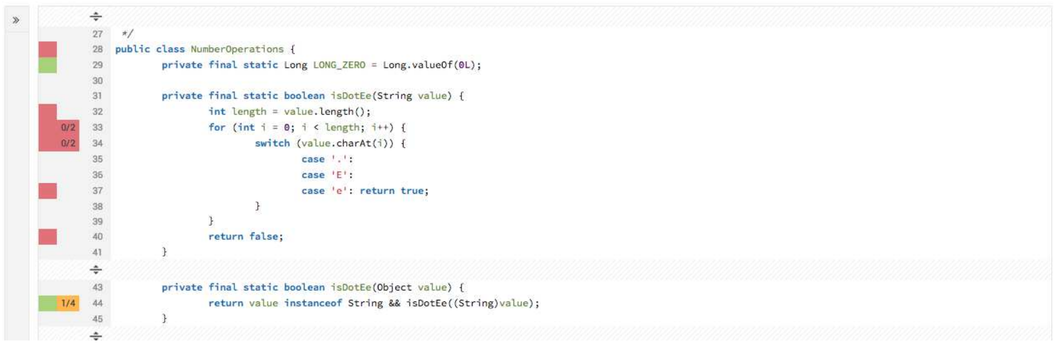
src/main/java/org/activiti/engine/impl/javax/el	34,4%
src/main/java/org/activiti/engine/impl/json	35,9%
src/main/java/org/activiti/engine/impl/calendar	39,0%
src/main/java/org/activiti/engine/impl/juel	39,2%
src/main/java/org/activiti/engine/impl/bpmn/listener	42,0%
src/main/java/org/activiti/engine/impl/persistence	43,1%

MethodNotFoundException.java	25,0%
TreeValueExpression.java	25,4%
TypeConverterImpl.java	25,7%
NumberOperations.java	26,2%
Bpmn20NamespaceContext.java	27,6%
AstEval.java	27,6%

Figure 2.6: Package and class selection

After selecting a file the current coverage information of the class is shown. In Figure 2.7, an example is shown that besides green and red, SonarQube uses yellow to show that a branch was not fully covered.

There are several ways to filter the changes. It is possible to show the lines to cover, branches to cover, uncovered lines and uncovered branches. YIt is also possible to also select a timeframe, but by default it shows the current state of the file. A time frame of 90 days, 30 days or since the last analysis can be selected. For those last three, it can be seen which untested parts have been added since that date. Downside is that it is impossible to show the difference between two versions, but just shows the difference between two dates.



```
 */
public class NumberOperations {
        private final static Long LONG_ZERO = Long.valueOf(0L);

        private final static boolean isDotEe(String value) {
                int length = value.length();
                for (int i = 0; i < length; i++) {
                        switch (value.charAt(i)) {
                                case '.':
                                case 'E':
                                case 'e': return true;
                        }
                }
                return false;
        }

        private final static boolean isDotEe(Object value) {
                return value instanceof String && isDotEe((String)value);
        }
```

Figure 2.7: Coverage information of the current version

Chapter 3

Requirements

This chapter describes the goal of the thesis and the non-functional and functional requirements for creating a tool which combines the information of code changes and test coverage changes.

3.1 Operias

The work in this thesis is the tool named Operias. Operias is a review tool for developers to increase the awareness of the change in the code and change in test coverage at the same time. It can be used to track the co-evolution of code changes and test code changes by combining information into a single report. The tool generates a HTML and XML report based on two versions of a project and its test suite. A report contains information about the changes in the code and the changes in the test coverage of the code.

With the increasing popularity of GitHub, open-source projects should have a way to validate the pull requests made for their projects. Operias is the ideal tool to be used in version control systems. The goal of this thesis is to provide information on how such a tool should work and what a generated report from the tool would look like. Therefore, the tool will be implemented and evaluated using real open-source projects as examples.

3.2 Non-functional requirements

There are a few non-functional requirements for a desired execution of the tool. First of all, the tool must be easy to use. This means that no setup or additional dependencies should be required to run it.

The tool must be configurable to run for different projects and folders, which means that it should be possible to provide a configuration. These options should cover the setup for the two versions of a project which should be compared, but also where the generated reports should be stored.

To compare two versions, both versions should be available for the tool. These versions should compile without error, and there must be a uniform way of executing a test suite.

3.3 Functional requirements

In this section, the functional requirements for the tool is described and grouped into four different parts:

- Collecting code coverage information for both projects
- Compare source code for changes
- Combine the information and calculate the final changes
- Construction of a report of the changes, both in HTML and XML

3.3.1 Collecting code coverage information

To get the code coverage information of a project, an external tool can be used to collect the information during the execution of the test suite. A requirement for the tool is that it can be used on a project without any changes or additions to the project. So a code coverage tool should be selected which can run independent from the project and its test suite.

The code coverage tool should produce a report in a computer readable format. The result should consists of at least information about statement coverage and should include which lines were covered and which lines were not. This line by line information is needed to visualize the code coverage difference.

3.3.2 Compare source code

Besides code coverage, we also need the code changes between the two version of the software. There are already several tools which can be used to produce a highlighted difference between source files. Some of them can also be used to create a XML report. A downside is that most of these tools only compare one file to another and cannot be used to create a diff report for a whole directory.

Another way to get the differences is by using an algorithm, for example the O(ND) Difference Algorithm [26] of Myers or the algorithm of Hunt et al [20]. These algorithms provide an effecient implementation for the longest common subsequence (LCS) problem [19].

The only requirements for the code changes is that for every file in the project, the inserted, deleted and changed lines are given.

3.3.3 Combine the information

The most important step is to combine the information from the first two phases. To combine the information, a file is marked as changed if either the coverage has changed or if there are changes in the source code. A change in coverage means that lines were changed from covered to uncovered or vice versa or if coverable lines are added or removed. A coverable line is a line which can be executed, i.e., a line which contains at least one statement.

The result of this phase is a set of files which are changed, deleted or newly added. The same steps are taken for test files, for which we only consider code changes.

3.3.4 Construction of a report

The final part of the execution is to create report of the changes in the two versions. The tool should be able to construct a report in HTML and XML format. The HTML format is usable for code reviews by other developers, and the XML report can be used by another program to parse the results and check for anomalies.

HTML Site

When opening the constructed site, a clear overview should be given of which files were changed, added and deleted. The classes should be grouped together by package for easy lookup. For every package and class, both a visual and textual representation of the change should be provided. This way, one can easily see the impact of the changes made in the new version of the project.

But just seeing the change in coverage does not tell the full story about the impact of the changes. If also the percentage of difference in source changes is shown, that number can be related to the code coverage change. For example, if we have an code size increased by 50%, and the code coverage decrease by 1%, we can most likely conclude that a lot of the new code is covered, under condition that the original coverage percentage was high. So a decrease in coverage percentage does not mean, by definition, that the newly added code is badly tested.

Besides the changes in the classes, the source changes in the test cases are also important to understand the coverage change of the new code. If small source changes in a class are made, but there is a big decrease in code coverage, then probably the test case is removed, altered or in some cases not included in the test suite. Providing an overview of the changed test classes may help the user to identify these problems.

XML

The XML report should contain the most important metrics of the changes made. For example, the amount of added coverable lines and removed coverable lines, but also the rate of which these lines were covered. For both test classes and normal classes, the number of source line changes should be given, together with the percentage change.

This information should be given per class and in a short summary. This way, another application can easily construct their own summary or conclusions based on the execution of the tool.

3.4 Deployment

To ensure the succes of a tool with the goal to help developers to review changes, the tool should have the ability to run as a service. This way, reports can be generated automatically and it will be more likely that users will use the tool.

Chapter 4

Operias

This chapter provides a short overview on how the tool must be used by developers. After that, the most important implementation details of Operias are described.

4.1 Tool Overview

The constructed tool Operias has been designed to be easy to use for a developer. The only thing that a developer needs to generate the reports, is a folder with the original version of the project and a folder containing the revised version of the project. To achieve this, the developer should keep track of different versions of the project.

To execute Operias, a project must be written in Java and must be compiled and tested using Maven. This allows a uniform way of executing the test suites. Also the folder structure is clear for Operias to analyze and to make a distinction between normal code and test code.

With the two versions of the project in separate folders, the developer can execute the tool using these location of the folders as arguments. This process is illustrated in Figure4.1. After successful execution of Operias, a HTML site and a XML report are generated which can be used to review the changes in the code.

4.2 Implementation details

In the upcoming sections, the architecture of Operias will be described in more detail. Its packages are defined in the class diagram of Operias, illustrated in Figure 4.2.

4.2.1 Package: operias

The core package of Operias provides code for setting up the directories which are compared. In Table 4.1, the initial arguments for Operias can be found. The arguments are parsed and saved in the *Configuration* class of Operias. Of these parameters, both the original and the revised directories are required parameters. The temporary and destination

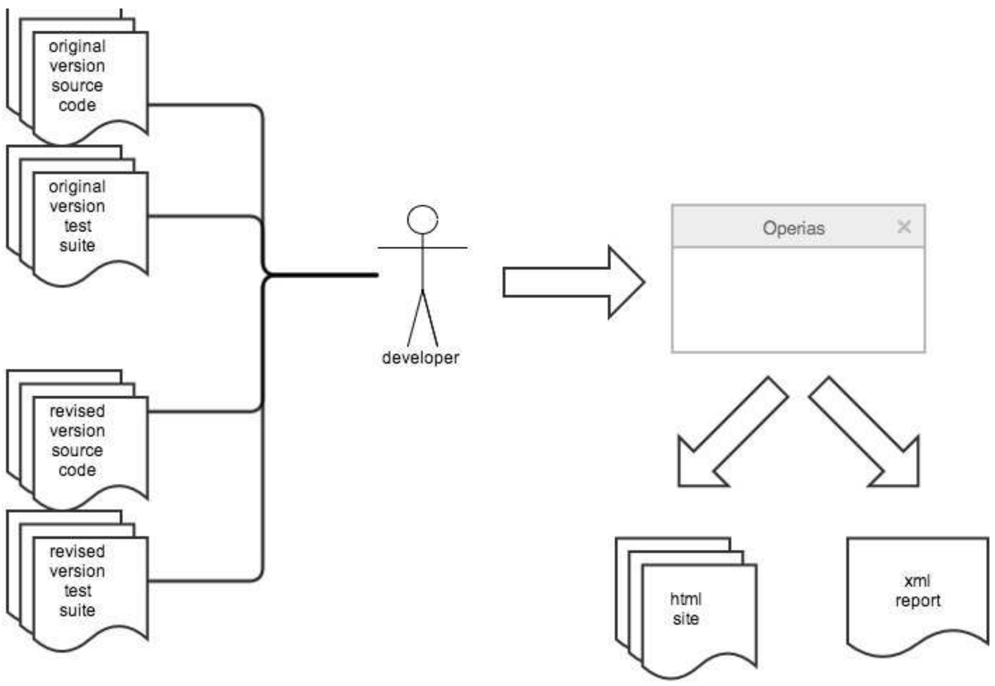

original version source code
original version test suite
revised version source code
revised version test suite
developer
Operias
html site
xml report

Figure 4.1: Visualization of the process a developer needs to follow to create a report with Operias

directories are optional parameters to prevent any privilege issues on the system on which the tool runs.

By enabling the verbose option, Operias shows the current state of the execution or any of the following warnings: deleted or added inner classes, missing information or invalid folder structures. If Operias fails to execute, an error message is returned if the verbose parameter is provided.

After parsing the arguments the class *Operias* collects the information needed and constructs the defined reports.

4.2.2 Package: operias.coverage

The package *operias.coverage* is used to create the coverage report. To retrieve the coverage information, Operias can use a third-party library or application. Operias uses the open-source tool Cobertura [2]. Cobertura can easily be executed on a Maven project. The advantage of using Cobertura is that it is not necessary to add Cobertura to the pom of a project for it to work.

Argument name	Description
original-directory	A local directory containing the root directory of the original version of the project to which a revised version is compared to
revised-directory	A local directory containing the root directory of the revised version of the project which is compared to the original project version
temp-directory	A temporary directory which Operias uses to store intermediate results
destination-directory	A local directory in which Operias stores the generated site
verbose	If this parameter is provided, Operias prints info, warning and error messages during the execution

Table 4.1: Operias command-line parameters

4.2.3 Package: operias.diff

The package *operias.diff* is used to create a report, *DiffReport*, which contains the code changes. To get the changes between the two folders Myer's diff algorithm [26] is used. The result of this algorithm applied to two files, is a set of deltas. Three different deltas are produced, an InsertDelta, a DeleteDelta and a ChangeDelta. These deltas consist of the following information:

- Number of lines in the original and revised piece of code, where the InsertDelta has zero original lines and the DeleteDeltra has zero revised lines
- The start line number in both the original and revised source, again zero if we have an InsertDelta or DeleteDelta for the original and revised respectively
- The actual lines which were inserted, deleted or changed

If the result of the comparison consist of at least one delta, we can say that the source of that file was changed. There are four different states (*SourceDiffState*) used: same, changed, new, deleted. In Operias, the information about directories and its files are stored in the *DiffDirectory* and *DiffFile* classes for easy access in a later phase.

4.2.4 Package: operias.report

After collecting both the information on test coverage changes and code changes, it is time to merge this information and calculate which file has either a change in test coverage or in the code, or both. This process is done the class *OperiasReport* in the package *operias.report*.

To determine whether a file must be marked as changed, we can first check whether a class still exists in the new version or was added in the new version. In these cases, no comparison can be made. The process to determine this, is described in Algorithm 1 and is executed in *OperiasReport*.

Algorithm 1 Check which files can be compared and which are added or removed

```
for all $originalClass$ in $originalCoberturaReport$ do
  $revisedClass \leftarrow revisedCoberturaReport.getClass(originalClass.getName())$
  if $revisedClass$ exists then
    $sourceDiff \leftarrow getSourceDiff(originalClass.getFileName())$
    $addChangedClass(compareClass(originalClass, revisedClass, sourceDiff))$
  else
    $addChangedClassAsDeleted(originalClass)$ {Class was deleted}
  end if
end for
{Also check for added files}
for all $revisedClass$ in $revisedCoberturaReport$ do
  $originalClass \leftarrow originalCoberturaReport.getClass(revisedClass.getName())$
  if $originalClass$ not exists then
    $addChangedClassAsNew(revisedClass)$ {Class was newly added}
  end if
end for
```

The result of this algorithm is a list of changed classes, which will be processed to be reported. There are three possible ways to add a class to this list:

- **Deleted** To determine if a class was deleted, the algorithm checks for every class in the original version whether or not coverage information exists for this class in the revised version. If not, the class is marked as deleted, and the code changes will be set to one DeleteDelta, which consists of the complete file.
- **New** To determine if a class was newly added, the algorithm checks the classes in the same way as for deleted classes, but now the other way around. If the class only exists in the revised coverage report, it marks the class as new and the code changes will be set to one InsertDelta.
- **Changed** If a class is found in both the revised and original coverage report, the classes can be compared. After retrieving the corresponding code changes from the code change report, the classes are compared in the subroutine *compareClass*

In the subroutine *compareClass*, a class is compared based on the two coverage reports and the code change report. This process is described in Algorithm 2 and is executed in the class *OperiasFile*. The result of this algorithm is a list of zero or more instances of *OperiasChange*. There are five possible changes which can be identified:

- *CoverageIncreaseChange* This type of change is created for every line with an increase in coverage, but no changes in the actual code

- *CoverageDecreasedChange* This type of change is created for every line with an decrease in coverage, but no changes in the actual code

- *DeleteSourceChange* The DeleteDelta from the code change report is converted to this type of change, and any coverage information for the deleted piece of code is added

- *InsertSourceChange* The InsertDelta is converted to this type, and again any coverage information is also added

- *ChangeSourceChange* The final type of change is also a converted ChangeDelta. Now coverage information from both the original and revised coverage reports are added

Every type of change contains the original and revised line numbers to indicate where the change is made, and information about the coverage state of the line or lines which are involved.

To compare the class, the algorithm compares the files line by line. First it checks if a code change exists on compared lines in the code.

If there is a delta, it converts this to the corresponding new *OperiasChange* and use the coverage reports to retrieve coverage information about the lines in the code change delta. After that, it jumps to the next comparison by adding the size of the delta to the line numbers.

If no delta was found for the given line numbers, the algorithm retrieves information about the lines from the coverage reports and checks if there is a difference in coverage between these lines, assuming we have information about both of the lines. If there is a change, the corresponding *OperiasChange* is created and added to the list of changes

Finally, if there was at least one *OperiasChange* instance constructed for the file and added to the list, the *OperiasFile* instance containing these changes is stored in the *OperiasReport* instance.

4.2.5 Package: operias.report.xml

In the *operias.output.xml* package, the XML report for is generated. This report is based on the list of changed classes in *OperiasReport*. This report can be used by other applications to read the results of the comparison. The XML includes several important elements. For every class in the project that has changed, the number of lines added and removed are given and for these lines also the coverage percentage. This information is easily retrieved by looping through all the deltas and counting the coverable lines and calculating what percentage of these are covered. An example can be seen in listing 4.1, where the line coverage increased from 42% to 50%. In the example are five lines removed, which are covered for 20%, but also 10 lines are added, of which 90% are covered. So in this case, most of the removed lines were uncovered, but almost every line that is added is covered.

Algorithm 2 Compare and merge coverage and source difference information

```
**CompareLines**(originalLineNumber, revisedLineNumber):
**if** end of file **then**
    **return**
**else**
    $delta \leftarrow sourceDiff.getChange(originalLineNumber, revisedLineNumber)$
    **if** $delta$ **exists then**
        **if** $delta$ **instanceof** $InsertDelta$ **then**
            add revised coverage information to $delta$
            addToChanges($delta$)
        **else if** $delta$ **instanceof** $DeletetDelta$ **then**
            add original coverage information to $delta$
            addToChanges($delta$)
        **else if** $delta$ **instanceof** $ChangeDelta$ **then**
            add original and revised coverage information to $delta$
            addToChanges($delta$)
        **end if**
        **CompareLines**($originalLineNumber + delta.originalLineCount(), revisedLineNumber + revisedLineCount()$)
    **else** {No change was found, check for coverage change}
        $originalLine \leftarrow originalClass.getLine(originalLineNumber)$
        $revisedLine \leftarrow revisedClass.getLine(revisedlLineNumber)$
        **if** $originalLine.isCovered()$ **and** $!revisedLine.isCovered()$ **then**
            addToChanges(IncreasedCoverageChange)
        **else if** $!originalLine.isCovered()$ **and** $revisedLine.isCovered()$ **then**
            addToChanges(DecreasedCoverageChange)
        **end if**
        **CompareLines**($originalLineNumber + 1, revisedLineNumber + 1$)
    **end if**
**end if**
```

XML 4.1: Example XML part

```
...
<classFile classname="example.test"
        conditionCoverageOriginal="0.0"
        conditionCoverageRevised="0.0"
        filename="/src/main/java/example.test.java"
        lineCoverageOriginal="0.42"
        lineCoverageRevised="0.50"
        sourceState="CHANGED">
        <coverageChanges>
```

```
                <relevantLinesRemoved>
                        <lineCount>5</lineCount>
                        <lineRate>0.2</lineRate>
                </relevantLinesRemoved>
                <relevantLinesAdded>
                        <lineCount>10</lineCount>
                        <lineRate>0.9</lineRate>
                </relevantLinesAdded>
        </coverageChanges>
        ...
</classFile>
...
```

Besides a short overview per class, the XML report also contains a summary for the complete changes. This makes it easy to automatically evaluate the differences between the version. The summary contains the total change in condition and line coverage, and again the number of lines removed and added, and to which extent these lines are covered.

XML 4.2: Example summary

```
...
<coverageChanges originalConditionRate="0.0"
        originalLineRate="0.42"
        revisedConditionRate="0.0"
        revisedLineRate="0.50">
        <totalRelevantLinesRemoved>
                <lineCount>10</lineCount>
                <lineRate>0.2</lineRate>
        </totalRelevantLinesRemoved>
        <totalRelevantLinesAdded>
                <lineCount>18</lineCount>
                <lineRate>0.42</lineRate>
        </totalRelevantLinesAdded>
</coverageChanges>
<sourceChanges>
        <addedLineCount>21 (1.11\%)</addedLineCount>
        <removedLineCount>49 (2.6\%)</removedLineCount>
</sourceChanges>
...
```

4.2.6 Package: operias.report.HTML

The feature of Operias that is most visible to developers, is the HTML site Operias generates. This report is generated in the *operias.output.HTML* package in the class *HTMLReport*. On this generated site, developers and reviewers can check all the changes visually. There are three main views to find in the site. The class overview, in which a clear overview of the changed packages and its classes can be found. Below that, a short overview of the test suite is given. The last view is shown when clicked on a class. A new page opens with a detailed view of the source code. In the section below these views are described in more detail.

Class: HTMLOverview

The class overview is a first view in which all packages are displayed, as shown in Figure 4.3. By clicking on a package, all changed classes within this package appear. For every class and package, we can see what the change is in condition and statements coverage by looking at the bars. These bars consists of four different colors which are also used in the class view. The colors have the following meanings:

- **Light green** is used for the part which was covered in the original version, and is still covered in the revised version.

- **Dark green** indicates the increase in coverage for that class of package. The final coverage percentage in the revised version is the light green and dark green combined.

- **Light red** means that this part was not covered in the original version, and also not in the revised version.

- **Dark red** indicates the percentage points decrease in coverage in the revised version. So if a dark red part is seen, the original coverage is the light green and dark red parts combined, and the revised coverage only consists of the light green part.

So, if no dark red or dark green part is present in the bar, the coverage stayed the same. This can also be checked by the number next to the bars, which shows the change in percentage points. In the last column, the change in source is showed for the class or package.

Besides the colors used in the bars, an indication for deleted and newly created classes is also used. A shaded row means that package or class was deleted. In that case, the coverage bars indicate the coverage of the original file by only using the light colors. If a class is new, the bars consists of only dark red and dark green parts, which indicate the revised coverage percentage of the class. Besides the bars, the text *new* or *deleted* is written next to the bars as an extra indication of the state of the class.

Class: HTMLTestView

Below the class overview, the test overview can be found. An example is shown in Figure 4.4. This overview is much shorter, containing only the information on source changes in test classes, since no coverage information on test suites is available in Cobertura.

Another thing that can be shown besides the source modifications, is the outcome of the execution of the test cases. For both the original and revised versions, a list of failed or errored test cases are shown. When clicking on a test case in the list, it shows whether it failed or errored and see the complete stacktrace generated by the test suite. Information on the test cases is retrieved by parsing the Surefire reports generated by Maven. An example can be seen in Figure 4.5.

Class: HTMLClassView

The final and most important view is the class view. The class view can be accessed by clicking on one of the classes in the list in the class overview. In the view, a maximum of four code views are shown:

- **Original file** In this view, see Figure 4.6, the original file is shown with the coverage information for that version of the code. As expected, green means covered, red means not covered.
- **Revised file** Similar to the original file view, except for showing the revised version of the file. The revised version of the example code is seen in Figure 4.7.
- **Source changes** This view only shows the source changes between the versions. The shaded background again means that the line was deleted and a box around a line or a group of lines means that these lines were inserted in the new version. An example of this view can be seen in Figure 4.8.
- **Combined view** This is the most useful part of Operias. It is similar to the source changes view, but it also shows the coverage information for both version. The coverage information in the deleted and inserted parts is easily to understand, since a comparison is not possible here. For the other parts, the comparison is done using the four colors that are used to indicate a change in coverage in the same way as described above, but now for specific lines of code. The final combined view for the example code is seen in Figure 4.9.

These four views are available for changed files. For added files, only the revised file view is shown including the coverage information, for deleted files, only the original file view is shown. When opening a changed test file, only the source diff view is viewable since there is no information about coverage available.

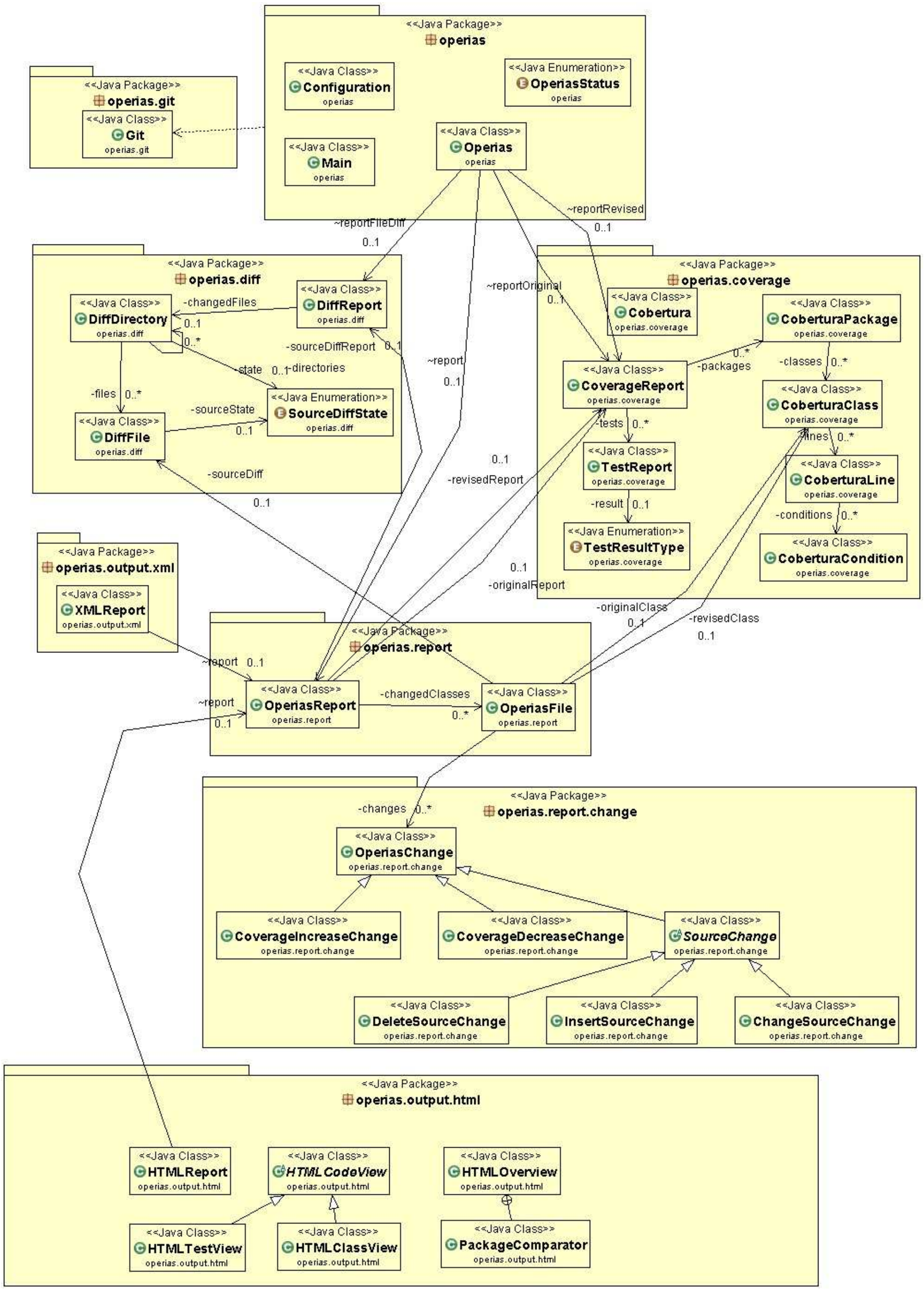
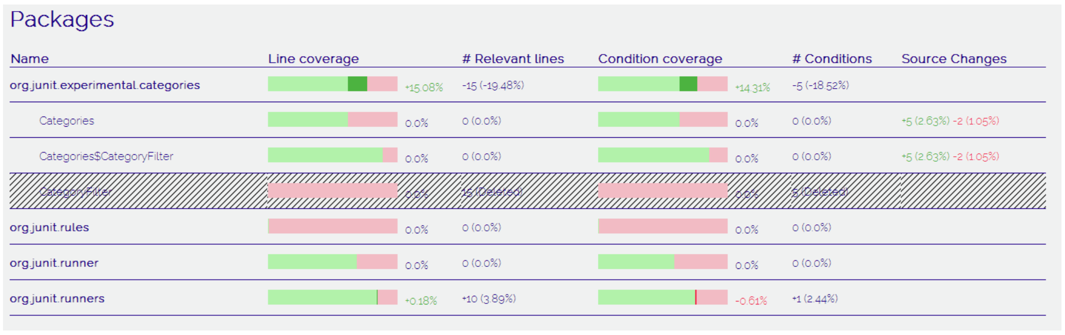


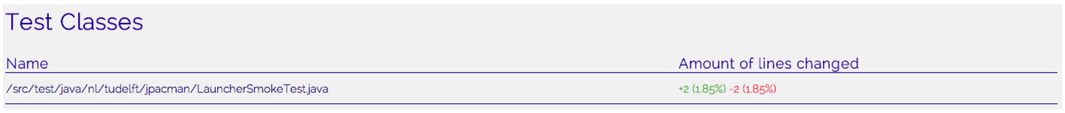
Figure 4.2: Class diagram of Operias



Packages

Name	Line coverage		# Relevant lines	Condition coverage		# Conditions	Source Changes
org.junit.experimental.categories		+15.08%	-15 (-19.48%)		+14.31%	-5 (-18.52%)	
Categories		0.0%	0 (0.0%)		0.0%	0 (0.0%)	+5 (2.63%) -2 (1.05%)
Categories$CategoryFilter		0.0%	0 (0.0%)		0.0%	0 (0.0%)	+5 (2.63%) -2 (1.05%)
CategoryFilter		0.0%	15 (Deleted)		0.0%	5 (Deleted)	
org.junit.rules		0.0%	0 (0.0%)		0.0%	0 (0.0%)	
org.junit.runner		0.0%	0 (0.0%)		0.0%	0 (0.0%)	
org.junit.runners		+0.18%	+10 (3.89%)		-0.61%	+1 (2.44%)	

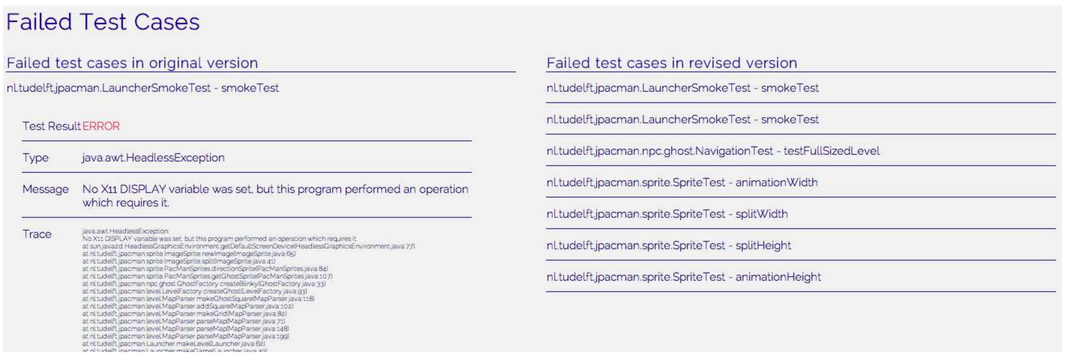
Figure 4.3: Class overview containing packages and classes which are changed



Test Classes

Name	Amount of lines changed
/src/test/java/nl/tudelft/jpacman/LauncherSmokeTest.java	+2 (1.85%) -2 (1.85%)

Figure 4.4: A list of test files which were changed and showing how many lines were added and removed



Failed Test Cases

Failed test cases in original version

nl.tudelft.jpacman.LauncherSmokeTest - smokeTest

Test Result	ERROR
Type	java.awt.HeadlessException
Message	No X11 DISPLAY variable was set, but this program performed an operation which requires it.

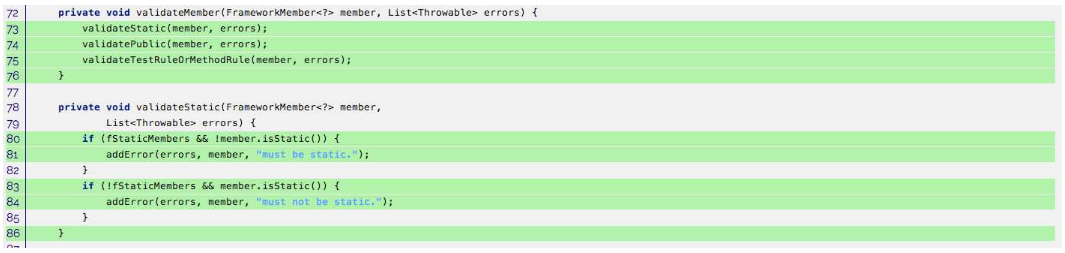
Trace

```
java.awt.HeadlessException:
No X11 DISPLAY variable was set, but this program performed an operation which requires it.
at sun.java2d.HeadlessGraphicsEnvironment.getDefaultScreenDevice(HeadlessGraphicsEnvironment.java:77)
at nl.tudelft.jpacman.sprite.ImageSprite.newImage(ImageSprite.java:65)
at nl.tudelft.jpacman.sprite.ImageSprite.split(ImageSprite.java:41)
at nl.tudelft.jpacman.sprite.PacManSprites.directionSprite(PacManSprites.java:84)
at nl.tudelft.jpacman.sprite.PacManSprites.getGhostSprite(PacManSprites.java:107)
at nl.tudelft.jpacman.npc.ghost.GhostFactory.createBlinky(GhostFactory.java:33)
at nl.tudelft.jpacman.level.LevelFactory.createGhost(LevelFactory.java:93)
at nl.tudelft.jpacman.level.MapParser.makeGhostSquare(MapParser.java:118)
at nl.tudelft.jpacman.level.MapParser.addSquare(MapParser.java:102)
at nl.tudelft.jpacman.level.MapParser.makeGrid(MapParser.java:82)
at nl.tudelft.jpacman.level.MapParser.parseMap(MapParser.java:71)
at nl.tudelft.jpacman.level.MapParser.parseMap(MapParser.java:148)
at nl.tudelft.jpacman.level.MapParser.parseMap(MapParser.java:199)
at nl.tudelft.jpacman.Launcher.makeLevel(Launcher.java:62)
at nl.tudelft.jpacman.Launcher.makeGame(Launcher.java:40)
```

Failed test cases in revised version

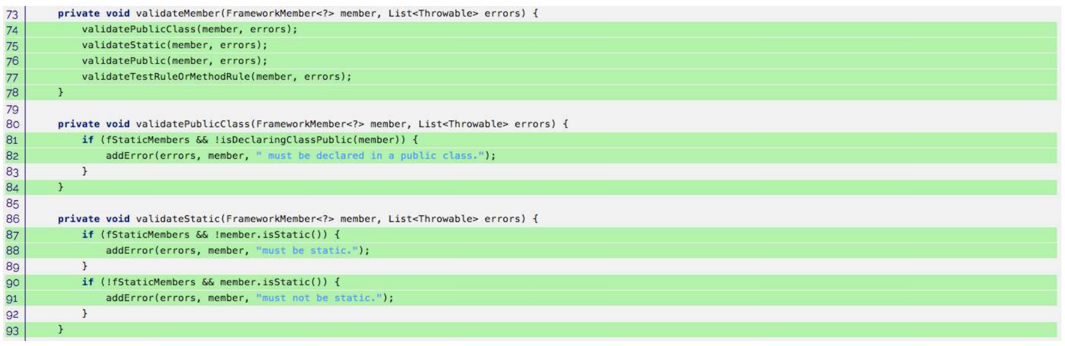
- nl.tudelft.jpacman.LauncherSmokeTest - smokeTest
- nl.tudelft.jpacman.LauncherSmokeTest - smokeTest
- nl.tudelft.jpacman.npc.ghost.NavigationTest - testFullSizedLevel
- nl.tudelft.jpacman.sprite.SpriteTest - animationWidth
- nl.tudelft.jpacman.sprite.SpriteTest - splitWidth
- nl.tudelft.jpacman.sprite.SpriteTest - splitHeight
- nl.tudelft.jpacman.sprite.SpriteTest - animationHeight

Figure 4.5: A list of failed or errored test cases, the test cases are clickable. After clicking, a short overview of the exception is shown, as shown on the left



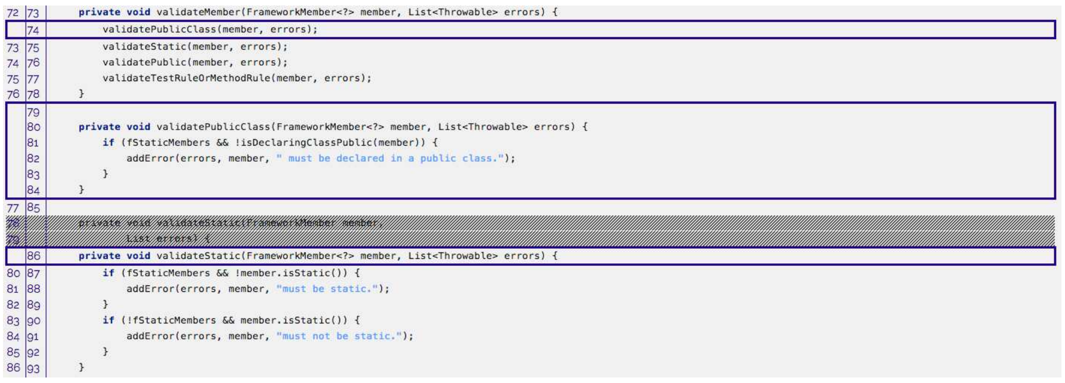
```java
    private void validateMember(FrameworkMember<?> member, List<Throwable> errors) {
        validateStatic(member, errors);
        validatePublic(member, errors);
        validateTestRuleOrMethodRule(member, errors);
    }

    private void validateStatic(FrameworkMember<?> member,
            List<Throwable> errors) {
        if (fStaticMembers && !member.isStatic()) {
            addError(errors, member, "must be static.");
        }
        if (!fStaticMembers && member.isStatic()) {
            addError(errors, member, "must not be static.");
        }
    }
```

Figure 4.6: Example of the original part of a piece of code and the coverage for that part, in this case, 100% covered



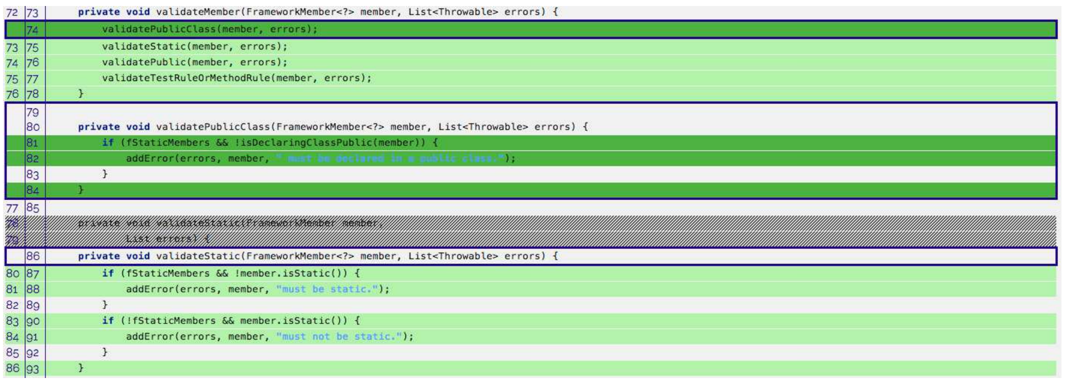
```
    private void validateMember(FrameworkMember<?> member, List<Throwable> errors) {
        validatePublicClass(member, errors);
        validateStatic(member, errors);
        validatePublic(member, errors);
        validateTestRuleOrMethodRule(member, errors);
    }

    private void validatePublicClass(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !isDeclaringClassPublic(member)) {
            addError(errors, member, " must be declared in a public class.");
        }
    }

    private void validateStatic(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !member.isStatic()) {
            addError(errors, member, "must be static.");
        }
        if (!fStaticMembers && member.isStatic()) {
            addError(errors, member, "must not be static.");
        }
    }
```

Figure 4.7: Example of the revised part of a piece of code and the coverage for that part, in this case, 100% covered



```
    private void validateMember(FrameworkMember<?> member, List<Throwable> errors) {
        validatePublicClass(member, errors);
        validateStatic(member, errors);
        validatePublic(member, errors);
        validateTestRuleOrMethodRule(member, errors);
    }

    private void validatePublicClass(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !isDeclaringClassPublic(member)) {
            addError(errors, member, " must be declared in a public class.");
        }
    }

    ~~private void validateStatic(FrameworkMember member,~~
            ~~List errors) {~~
    private void validateStatic(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !member.isStatic()) {
            addError(errors, member, "must be static.");
        }
        if (!fStaticMembers && member.isStatic()) {
            addError(errors, member, "must not be static.");
        }
    }
```

Figure 4.8: Example of the source change view, showing the difference in source between the code in Figures 4.6 and 4.7



```
    private void validateMember(FrameworkMember<?> member, List<Throwable> errors) {
        validatePublicClass(member, errors);
        validateStatic(member, errors);
        validatePublic(member, errors);
        validateTestRuleOrMethodRule(member, errors);
    }

    private void validatePublicClass(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !isDeclaringClassPublic(member)) {
            addError(errors, member, " must be declared in a public class.");
        }
    }

    ~~private void validateStatic(FrameworkMember member,~~
            ~~List errors) {~~
    private void validateStatic(FrameworkMember<?> member, List<Throwable> errors) {
        if (fStaticMembers && !member.isStatic()) {
            addError(errors, member, "must be static.");
        }
        if (!fStaticMembers && member.isStatic()) {
            addError(errors, member, "must not be static.");
        }
    }
```

Figure 4.9: Example of the combined view, showing the difference in source between the code in Figures 4.6 and 4.7 and also adds the coverage information

Chapter 5

Git extension

In this chapter, the extension of Operias for use in combination with git and GitHub is described. First an overview of the workflow of a pull request on GitHub is described. After that, more details are provided on how Operias is extended to be used with a Git repository. At last, the implementation details of the complete webhook extension are described. This webhook is used to automatically connect Operias to GitHub.

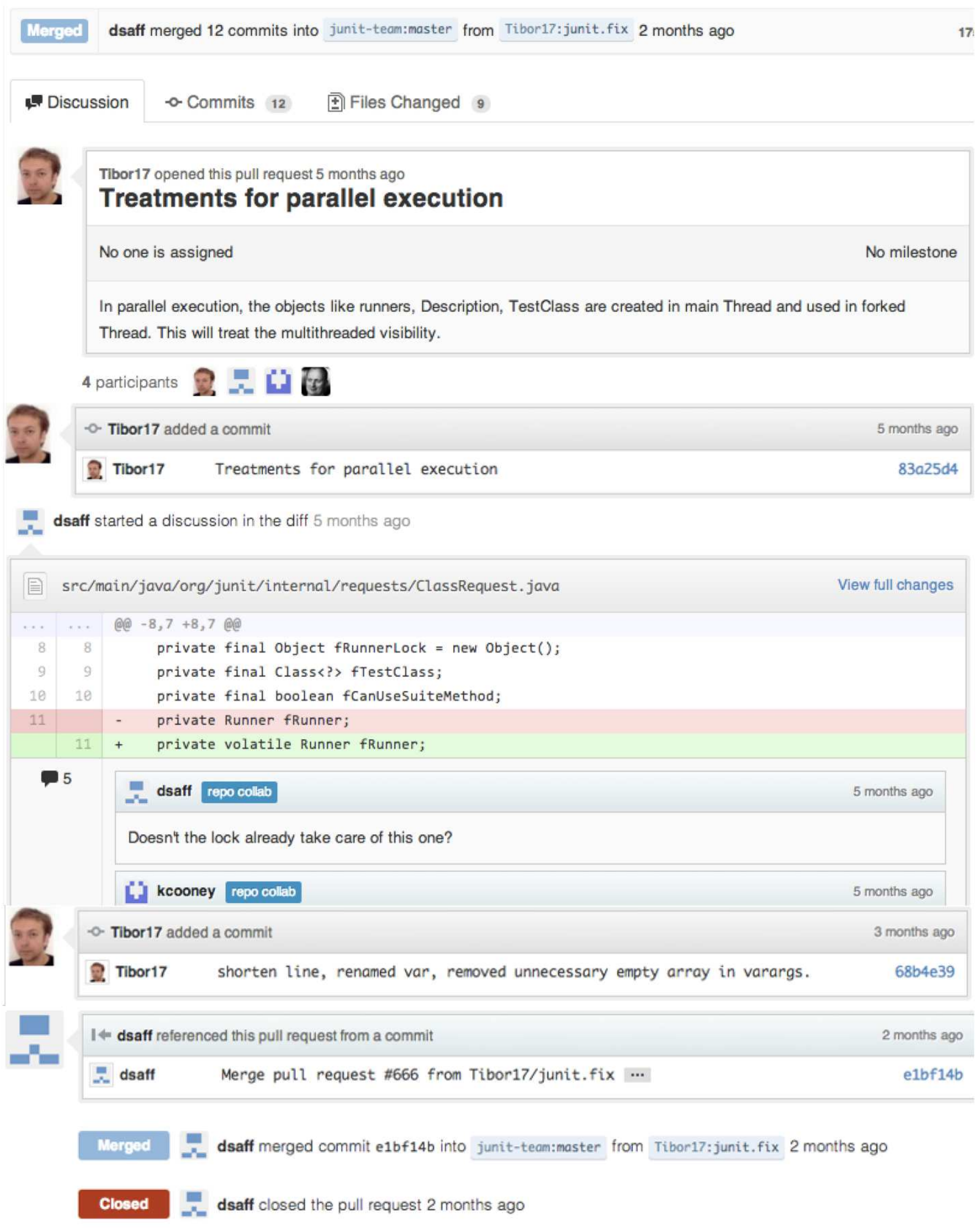
5.1 GitHub workflow

From this point, the extension to a version control system seems natural, since small code reviews by the developer or other developers in a team can be done before accepting a change into the repository. We chose Git as the version control system to which Operias will be extended. Git has growing support, GitHub for example, and offers the pull request mechanism. As of august 2013, GitHub reports more than 7 million repositories and 4 million users and still growing [16].

A pull request is a request to pull changes into a certain branch in the repository. Every developer in an open-source project can create a pull request, but only a selected group of developers can accept the changes in a pull request after reviewing the changes. An example of a pull request on GitHub is shown in Figure 5.1.

GitHub offers highlighted source differences to let the developers check the changes in a pull request, but does not offer anything to check for coverage changes. On average, every review on a pull request receives 2,89 discussion and code review comments and about 75% of the opened pull request are merged using the GitHub web interface [16]. Therefore, an extension to the Operias tool to offer such coverage change addition can be very useful and widely used.

In Figure 5.2, a sequence diagram is shown about how a pull request progresses. A contributor who wants to make an addition or change in the code, must fork the repository of the original, since not everyone has the same access rights to a repository. After committing the changes into the forked version, the contributor can decide to create a pull request for the original repository. After this, the core team (repository owners or admins) can inspect and comment on the changes and can decide to merge after accepting the changes. After merging, the commits are pulled into the repository and the pull request can be closed.



Merged dsaff merged 12 commits into junit-team:master from Tibor17:junit.fix 2 months ago

Tibor17 opened this pull request 5 months ago

Treatments for parallel execution

No one is assigned — No milestone

In parallel execution, the objects like runners, Description, TestClass are created in main Thread and used in forked Thread. This will treat the multithreaded visibility.

4 participants

Tibor17 added a commit — 5 months ago

Tibor17 Treatments for parallel execution 83a25d4

dsaff started a discussion in the diff 5 months ago

src/main/java/org/junit/internal/requests/ClassRequest.java — View full changes

```
@@ -8,7 +8,7 @@
     private final Object fRunnerLock = new Object();
     private final Class<?> fTestClass;
     private final boolean fCanUseSuiteMethod;
-    private Runner fRunner;
+    private volatile Runner fRunner;
```

5

dsaff repo collab — 5 months ago

Doesn't the lock already take care of this one?

kcooney repo collab — 5 months ago

Tibor17 added a commit — 3 months ago

Tibor17 shorten line, renamed var, removed unnecessary empty array in varargs. 68b4e39

dsaff referenced this pull request from a commit — 2 months ago

dsaff Merge pull request #666 from Tibor17/junit.fix … e1bf14b

Merged dsaff merged commit e1bf14b into junit-team:master from Tibor17:junit.fix 2 months ago

Closed dsaff closed the pull request 2 months ago

Figure 5.1: Example of a discussion on a pull request on GitHub [16]

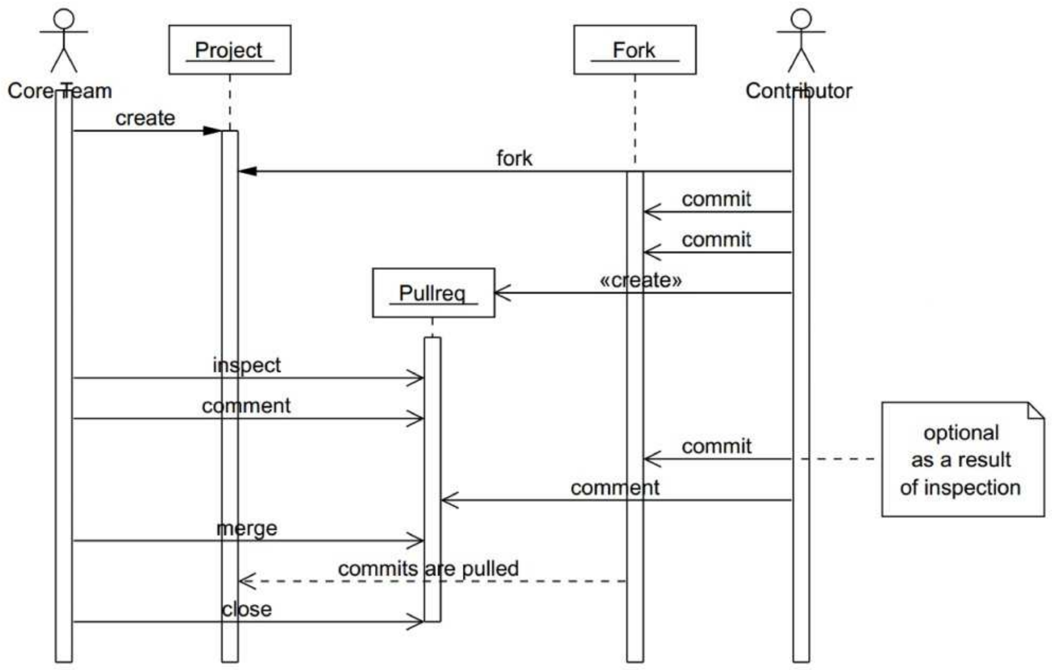


Figure 5.2: Sequence diagram of a pull request [16]

5.2 Extending Operias with Git support

To achieve the proposed extension, Operias must be extended to support Git and generate the result based on a certain commit or pull request. Because a pull request is basically the result of merging two commits, Operias should be able to receive two commit IDs and clone the repositories and check the correct branch and commit out.

Several new command line arguments will be introduced to support this extension. An overview of these new arguments is given in table 5.1.

Either a repository url for both versions should be provided or a different url for each of them. This allows comparing versions between different repositories, for example a forked project. For both the original and revised version, it is possible to give a branch name and a commit ID. Both are optional, by default Operias will use the latest commit in the master branch.

These arguments work in combination with the arguments described in table 4.1. In other words, it is possible to compare a local working version to a certain version which is in a git repository, and also the other way around. This feature is especially useful for checking code while working on it, without having to commit it or clone a previous version.

The process of cloning the repositories and checking out the correct branches and commits is located in the package *operias.git*. This package is also illustrated in the class diagram in Figure 4.3.

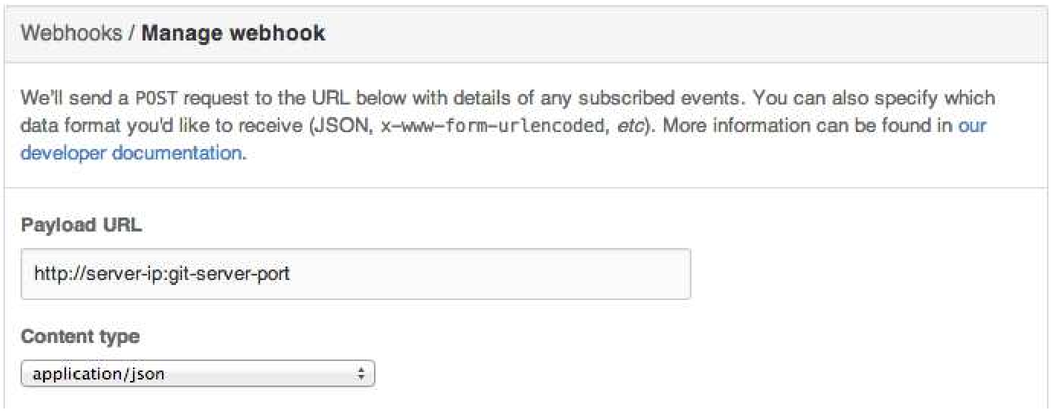
Argument name	Description
repository-url	The clone-url of the repository for both the original and the revised source code
original-repository-url	The clone-url of the repository for the original source code
revised-repository-url	The clone-url of the repository for the revised source code
original-branch-name	The branch name of the original
revised-branch-name	The branch name of the revised version
original-commit-id	The commit ID of the version which should be used as the original source code
revised-commit-id	The commit ID of the version which should be used as the revised source code

Table 5.1: New Operias command-line parameters to support the Git extension

5.3 Extending GitHub workflow with Operias

In this section the process to extend the pull request workflow with Operias is described. We want to achieve a process which automatically executes Operias after a pull request is created. For this, we constructed a webhook for GitHub which retrieves information about a pull request and executes Operias based on the received data.

A webhook is GitHub's way to let a repository communicate with an external web server whenever there is an event triggered for that repository.[1] A webhook is easily set up for a repository. The only thing that must be provided is the server IP and port. An example of this is illustrated in Figure 5.3



Webhooks / **Manage webhook**

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, *etc*). More information can be found in our developer documentation.

Payload URL

http://server-ip:git-server-port

Content type

application/json

Figure 5.3: Setting up the GitHub webhook

[1]More information on webhooks: https://developer.github.com/webhooks

In the upcoming section, the webhook extension is described. First an overview of the extension is provided and after the implementation details are discussed.

5.3.1 Extension overview

The workflow of the execution of Operias changes because of this extension. The sequence diagram in Figure 5.4 can be extended to include Operias, as illustrated in Figure 5.3. Operias is executed right after the pull request is created and posts a comment on the pull request as soon as Operias is done creating the report. The report is then available for inspection to the core team and can be used to decide whether or not to merge the commits.

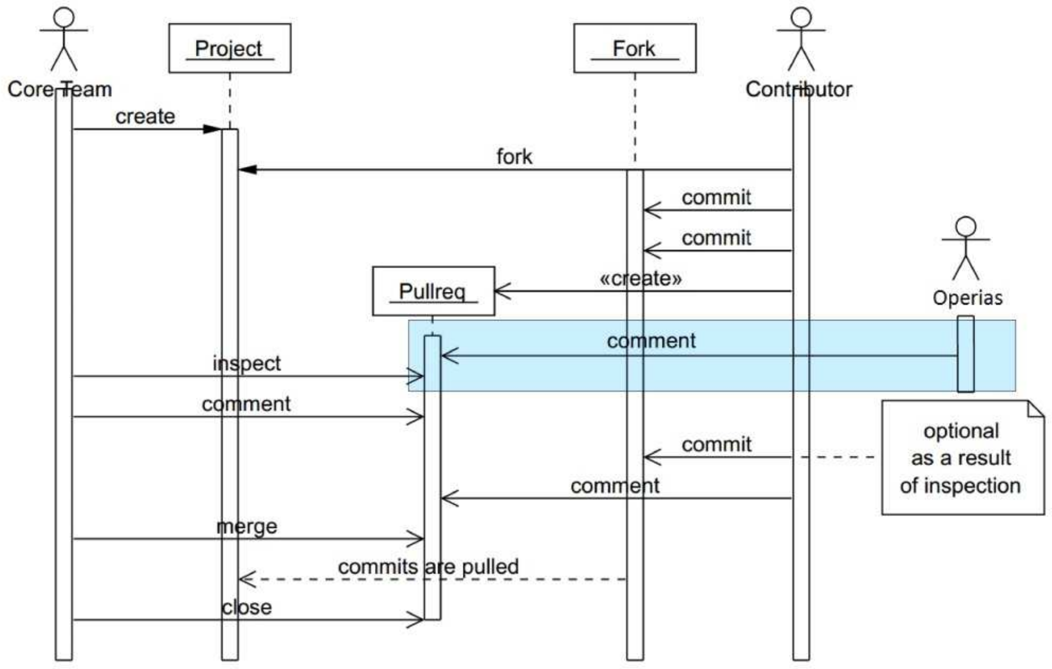


Figure 5.4: Sequence diagram of a pull request [16], extended with the comment Operias will add after creating a pull request

After setting up the webhook as illustrated in Figure 5.3, GitHub automatically send all information about pull requests to the given address. From this data, it is possible to determine which commits must be compared to each other. It also provides the repository clone urls and the branch names. In Figure 5.5, the process from the creating of a pull request to executing Operias to commenting on the pull request is illustrated.

GitHub will send its data to Operias' webhook. This webhook will parse the data and present the arguments in the correct form as described in Table 5.1 to Operias. After that generates the HTML site and XML report. The webhook of Operias will parse the XML and construct a short message which represents the changes in the pull request. This message is

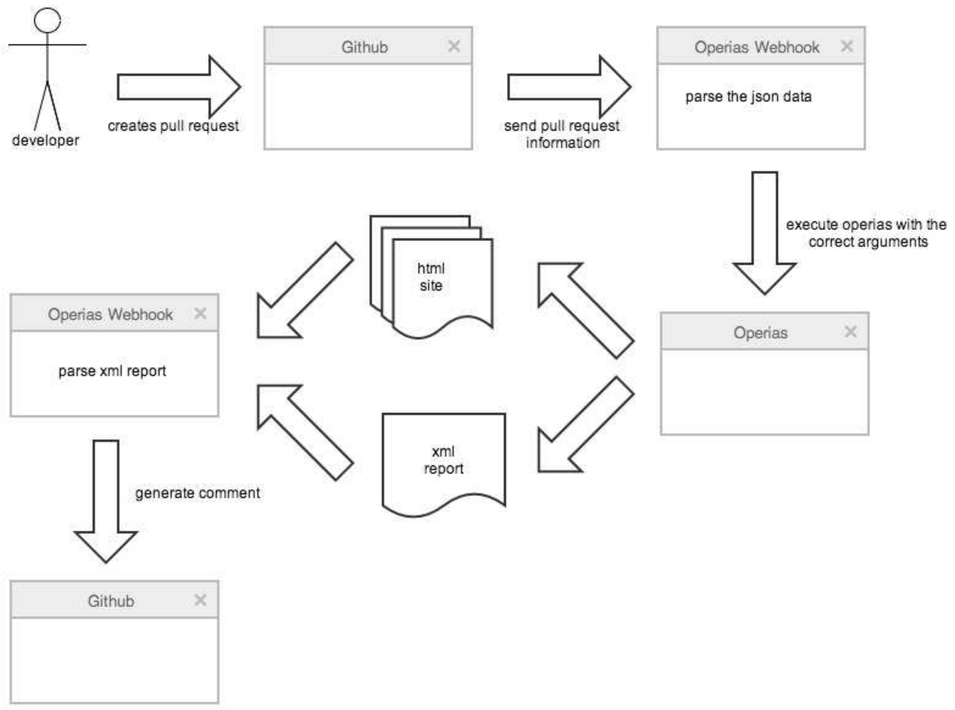


Figure 5.5: Visualization of the process from creating a pull request to posting a comment with the changes generated by Operias

then sent to GitHub and posted as a comment. In this comment, there is also a link to the HTML site.

5.3.2 Implementation Details

In this section, the implementation details of the webhook are presented. The webhook implementation uses Jetty as library for creating a webservlet. This webservlet will retrieve the data and processes it. The class diagram for this webhook is fairly simple and is illustrated in Figure 5.6. In the upcoming sections, the implementation of the individual classes is described.

Class: Configuration

As for Operias, the webhook extension consists of a *Configuration* class, where the command line arguments are parsed and stored. The arguments for the extension are shown in

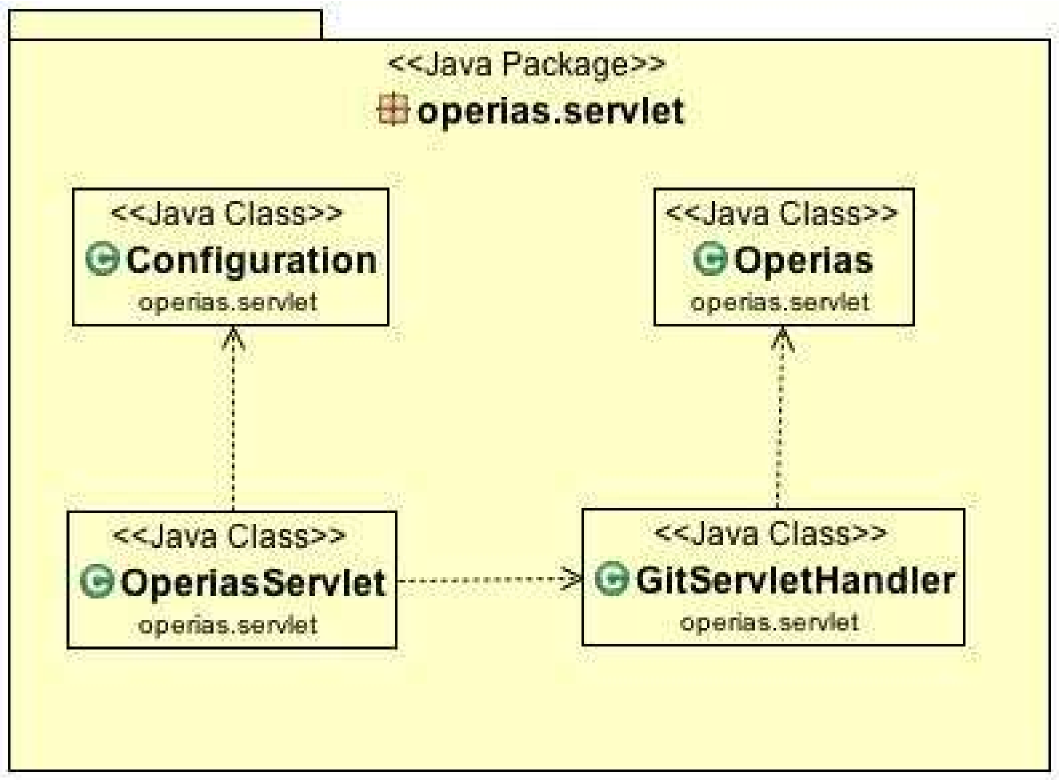


Figure 5.6: Class diagram of the webhook extension for Operias

Table 5.2. The server IP address and the git server port must be the same as filled in on GitHub. The webhook needs to know it own server IP address, because it adds a link to the generated HTML report. This HTML report can be accessed on this IP address and the HTML server port. Finally, a username and password for a GitHub account should also be provided. This account is used to comment on a pull request.

Class: OperiasServlet

This is the main class of the webhook, where the configuration is created and where Jetty is initialized. Both a handler for the HTML site requests and for accepting the requests from GitHub are constructed here.

Class: GitServletHandler

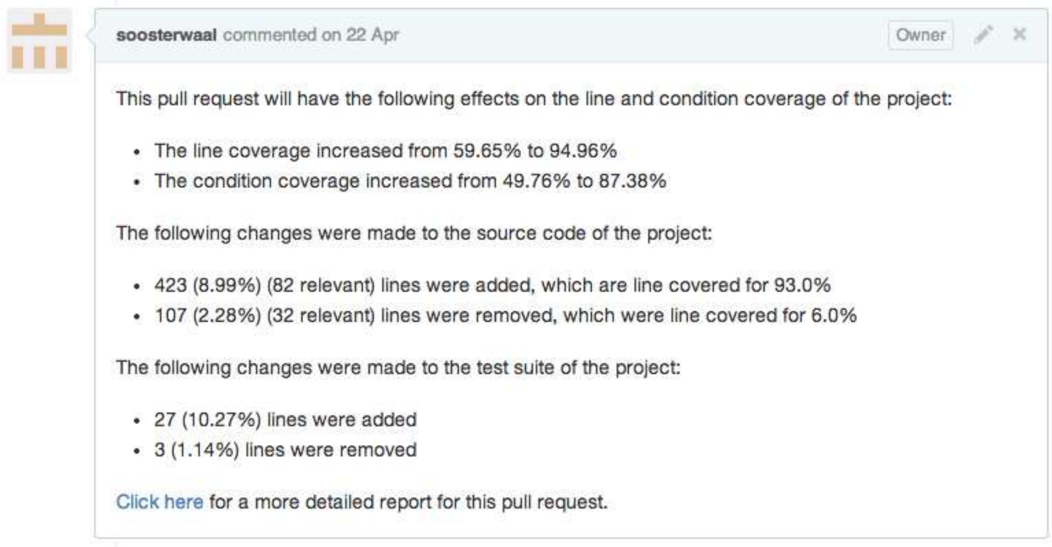
This is handler for Jetty which accepts the JSON data from GitHub. In this class, the JSON is parsed and the correct parameters for Operias are determined. If all the arguments are present in the data, Operias can be executed using the class *Operias* in this webhook.

Argument name	Description
server-ip	This is the IP of the server on which the web-hook run
git-server-port	The port used for received the data from GitHub
html-server-port	The port on which the generated HTML sites are hosted
temporary-directory	A local directory which Operias can use to store intermediate results during execution
results-directory	A local directory which is used to store the generated reports in
username	The username of the GitHub account, which is used to post the comments
password	The password of the GitHub account

Table 5.2: Command line parameters of the Operias Servlet

Class Operias

In this class, the Operias tool is executed using the arguments from the *GitServletHandler*. It executes Operias and after execution, the XML report is parsed. Using the information of the XML report, a short message is constructed and posted back to GitHub with the specified account. An example of such a comment on GitHub is shown in Figure 5.7.



This pull request will have the following effects on the line and condition coverage of the project:

- The line coverage increased from 59.65% to 94.96%
- The condition coverage increased from 49.76% to 87.38%

The following changes were made to the source code of the project:

- 423 (8.99%) (82 relevant) lines were added, which are line covered for 93.0%
- 107 (2.28%) (32 relevant) lines were removed, which were line covered for 6.0%

The following changes were made to the test suite of the project:

- 27 (10.27%) lines were added
- 3 (1.14%) lines were removed

Click here for a more detailed report for this pull request.

Figure 5.7: An automatically generated comment on GitHub by Operias

Chapter 6

Evaluation

After specifying and developing the tool, it is also important to know whether the tool is actually useful for developers and reviewers. To determine this, we look into pull requests on GitHub and try to answer the main question whether or not the tool is useful. First, some interesting pull requests are selected and the visual aspect of Operias is checked to see whether the use of the tool can give any useful information. Next, metrics on pull requests are collected using Operias to determine in how many cases the tool could be useful. For example, in how many pull requests do we see newly added code which is not covered.

6.1 Selected Projects

For this evaluation, three projects are selected. The project selection is constrained by the capabilities of Operias. Operias supports, for now, only projects written in Java with a test suite executed by Maven. The selected projects, all available on Github, are JUnit from JUnit-team, Bukkit from Bukkit and Wire from Square. For these projects, every pull request are sent to the webserver of Operias for analysis. Not every pull request could be analyzed, since not every pull request was accepted and certain branches or forked repositories do not exists anymore. Because of deleted branches, it is not possible to determine which versions should be compared.

JUnit

JUnit is a simple framework to write repeatable tests, which can be used in combination with Maven. The JUnit[1] project is selected for the evaluation, because the expectations are that this project is very well tested, since it is a framework for testing. For JUnit, more than 250 pull request are analyzed, coming from more than 80 contributors. Therefor, we expect that this project is very well tested.

[1]https://github.com/junit-team/junit

Bukkit

Bukkit[2] is an up-and-coming Minecraft Server mod that will completely change how Minecraft server run and are modified. This makes managing and creating servers easier and provides more flexibility. Bukkit is a project which is less tested than the JUnit, but has more contributors. Over 400 pull requests are analyzed from this project, from more than a 100 different contributors. The Bukkit project is selected, because almost every change is added using the pull request mechanism and because this project is substantially less tested than the other two.

Wire

Wire[3] is a library for lightweight protocol buffers for mobile Java. Code generated by Wire has fewer methods than standard protocol buffer code, which helps applications avoid the size limits on methods in Android applications. Wire is a smaller project with less than 100 pull request analyzed and only 12 different contributors. This project is better tested than Bukkit, but is expected to be less tested than JUnit.

6.2 Selected Pull Requests

Our first approach consists of collecting pull requests which show interesting cases. The goal for this evaluation is to analyze the usefulness of Operias from the point of view of the code reviewer. To determine the usefulness, pull requests of the projects are manually inspected and interesting cases are selected. We looked for pull requests which show a good example of testing and a few which show less tested parts and explain how Operias is useful in the review process of these pull requests.

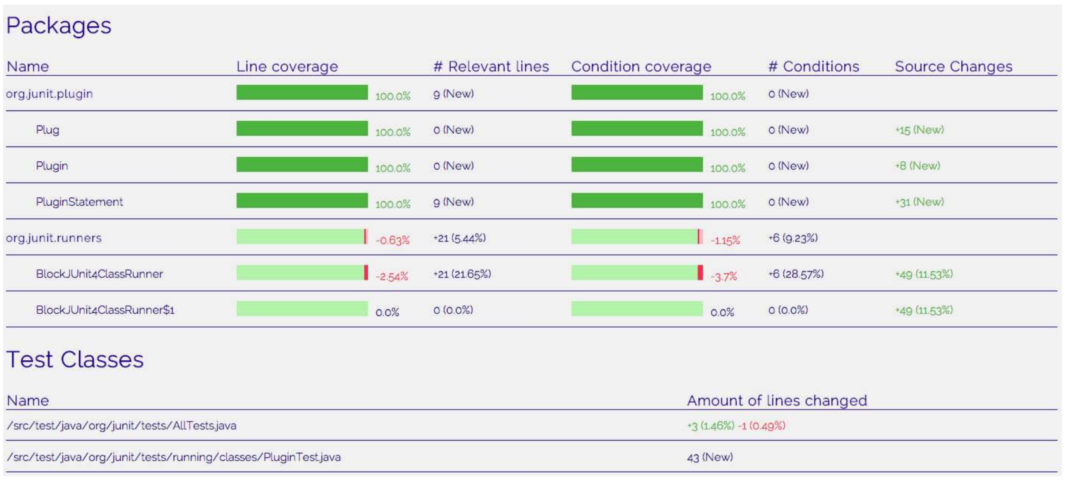
Pull request #767 from JUnit

The first example of a pull request which is correctly tested is illustrated in Figure 6.1. The newly added package *Plugin* is fully covered by the added test *PluginTest*. We can also see that the test is correctly added, because the test class *AllTests* is also edited. In JUnit, all tests must be added to the class *AllTests* in order to be executed. For the class *BlockJUnit4ClassRunner*, we can see that it is initially covered for 100%, but after adding 21 coverable lines and 6 conditions, both statements and condition coverage dropped a little bit. This means that the added code in this class is not completely covered.

If we open the class and check the combined view, we see that a new code block is added with a few lines that are not covered. This piece of code is shown in Figure 6.2.

For this pull request, one might say that is was well covered, but using the Operias tool, the reviewer and developer are able to easily view the uncovered lines which are added.

[2]https://github.com/Bukkit/Bukkit
[3]https://github.com/square/wire



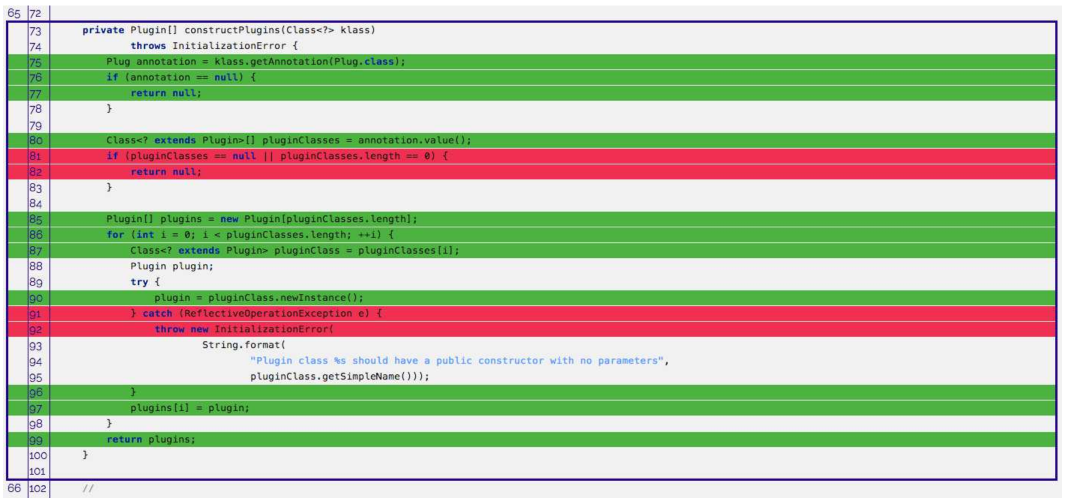
Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.junit.plugin	100.0%	9 (New)	100.0%	0 (New)	
Plug	100.0%	0 (New)	100.0%	0 (New)	+15 (New)
Plugin	100.0%	0 (New)	100.0%	0 (New)	+8 (New)
PluginStatement	100.0%	9 (New)	100.0%	0 (New)	+31 (New)
org.junit.runners	-0.63%	+21 (5.44%)	-1.15%	+6 (9.23%)	
BlockJUnit4ClassRunner	-2.54%	+21 (21.65%)	-3.7%	+6 (28.57%)	+49 (11.53%)
BlockJUnit4ClassRunner$1	0.0%	0 (0.0%)	0.0%	0 (0.0%)	+49 (11.53%)

Test Classes

Name	Amount of lines changed
/src/test/java/org/junit/tests/AllTests.java	+3 (1.46%) -1 (0.49%)
/src/test/java/org/junit/tests/running/classes/PluginTest.java	43 (New)

Figure 6.1: Generated site by Operias for pull request #767 from the JUnit project, a clear example of a well tested pull request

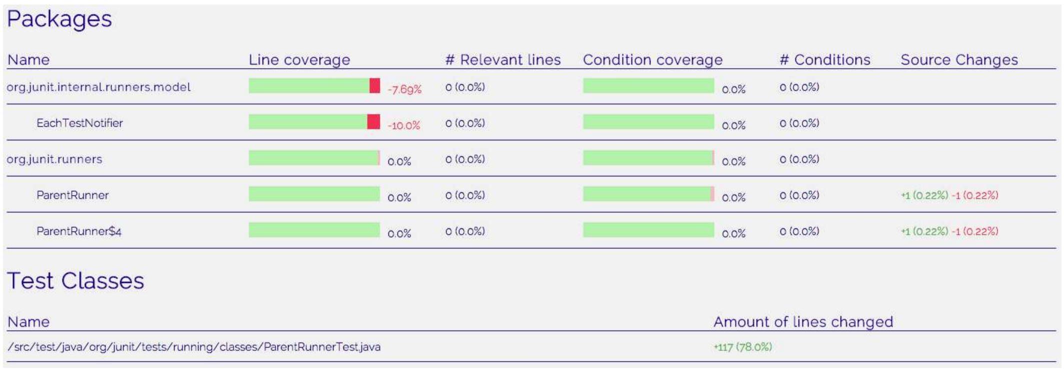


```

    private Plugin[] constructPlugins(Class<?> klass)
            throws InitializationError {
        Plug annotation = klass.getAnnotation(Plug.class);
        if (annotation == null) {
            return null;
        }

        Class<? extends Plugin>[] pluginClasses = annotation.value();
        if (pluginClasses == null || pluginClasses.length == 0) {
            return null;
        }

        Plugin[] plugins = new Plugin[pluginClasses.length];
        for (int i = 0; i < pluginClasses.length; ++i) {
            Class<? extends Plugin> pluginClass = pluginClasses[i];
            Plugin plugin;
            try {
                plugin = pluginClass.newInstance();
            } catch (ReflectiveOperationException e) {
                throw new InitializationError(
                        String.format(
                                "Plugin class %s should have a public constructor with no parameters",
                                pluginClass.getSimpleName()));
            }
            plugins[i] = plugin;
        }
        return plugins;
    }

    //
```

Figure 6.2: Added code in *BlockJUnit4ClassRunner* in pull request #767 which is covered for almost 75%

Pull request #896 from JUnit

In pull request #896 from JUnit, a small change in a class is made, and a large number of test lines is added for this class. That's sounds like a good pull request, but it affected the statement coverage of a completely different class in another package. Using Operias, these indirect changes in coverage can easily be detected, of which this is a clear example, illustrated in Figure 6.3.



Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.junit.internal.runners.model	-7.69%	0 (0.0%)	0.0%	0 (0.0%)	
EachTestNotifier	-10.0%	0 (0.0%)	0.0%	0 (0.0%)	
org.junit.runners	0.0%	0 (0.0%)	0.0%	0 (0.0%)	
ParentRunner	0.0%	0 (0.0%)	0.0%	0 (0.0%)	+1 (0.22%) -1 (0.22%)
ParentRunner$4	0.0%	0 (0.0%)	0.0%	0 (0.0%)	+1 (0.22%) -1 (0.22%)

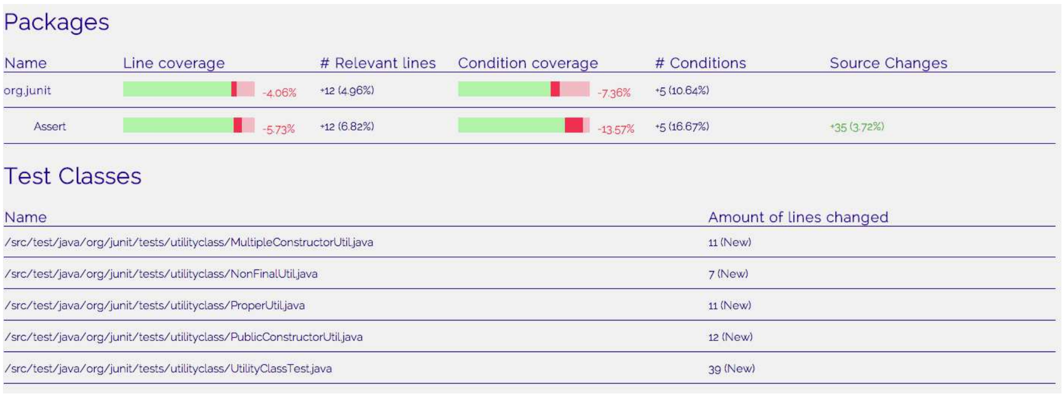
Test Classes

Name	Amount of lines changed
/src/test/java/org/junit/tests/running/classes/ParentRunnerTest.java	+117 (78.0%)

Figure 6.3: A change in the code can indirectly affect the coverage in a completely different class or package

Pull request #646 from JUnit

For a test case to be successfully executed within the JUnit project, it must be added to the class *AllTests*. In pull request #646 from JUnit, illustrated in Figure 6.4, there are five new test cases added to the project, next to a few changes in the code. Even if the test cases would properly test new or existing code, they are not executed because they are not added to the *AllTests* class. Again, Operias is useful here to easily see that the added test code does not affect either line coverage or condition coverage positively.



Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.junit	-4.06%	+12 (4.96%)	-7.36%	+5 (10.64%)	
Assert	-5.73%	+12 (6.82%)	-13.57%	+5 (16.67%)	+35 (3.72%)

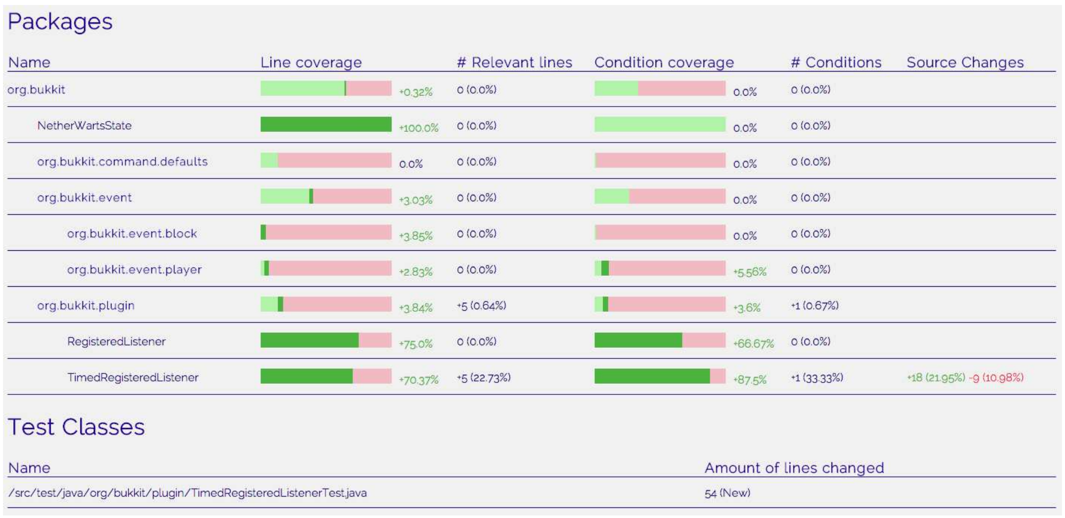
Test Classes

Name	Amount of lines changed
/src/test/java/org/junit/tests/utilityclass/MultipleConstructorUtil.java	11 (New)
/src/test/java/org/junit/tests/utilityclass/NonFinalUtil.java	7 (New)
/src/test/java/org/junit/tests/utilityclass/ProperUtil.java	11 (New)
/src/test/java/org/junit/tests/utilityclass/PublicConstructorUtil.java	12 (New)
/src/test/java/org/junit/tests/utilityclass/UtilityClassTest.java	39 (New)

Figure 6.4: Newly added test files should be added to *AllTests* before executing

Pull request #630 from Bukkit

Adding a test case for a class where changes were made is a good thing. A great example for this is shown in Figure 6.5. Here the changed class *TimedRegisteredListener* was covered for more than 70% by the added test case. Using Operias, a reviewer can also easily see what effect this has on the complete project. The coverage of several other packages and classes in this example also increased. This increase in coverage is a positive change, but also indicates that the class coupling in the project is high.



Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.bukkit	+0.32%	0 (0.0%)	0.0%	0 (0.0%)	
NetherWartsState	+100.0%	0 (0.0%)	0.0%	0 (0.0%)	
org.bukkit.command.defaults	0.0%	0 (0.0%)	0.0%	0 (0.0%)	
org.bukkit.event	+3.03%	0 (0.0%)	0.0%	0 (0.0%)	
org.bukkit.event.block	+3.85%	0 (0.0%)	0.0%	0 (0.0%)	
org.bukkit.event.player	+2.83%	0 (0.0%)	+5.56%	0 (0.0%)	
org.bukkit.plugin	+3.84%	+5 (0.64%)	+3.6%	+1 (0.67%)	
RegisteredListener	+75.0%	0 (0.0%)	+66.67%	0 (0.0%)	
TimedRegisteredListener	+70.37%	+5 (22.73%)	+87.5%	+1 (33.33%)	+18 (21.95%) -9 (10.98%)

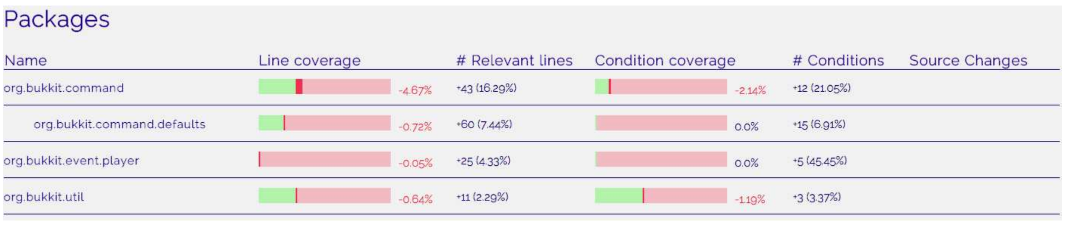
Test Classes

Name	Amount of lines changed
/src/test/java/org/bukkit/plugin/TimedRegisteredListenerTest.java	54 (New)

Figure 6.5: Changing one class and adding a test case for this does not only affect that code, but the complete project

Pull request #670 from Bukkit

Besides well-tested pull requests, there are also several pull requests in which there are several changes, but hardly or no test cases are added. A good example of this is pull request #670 from Bukkit, illustrated in Figure 6.6. Here, around 150 lines were added to the code, whch contain 30 conditions are added. This indicates that the complexity of the added code is high. Even so, no test cases are added which resulted in a drop of coverage.

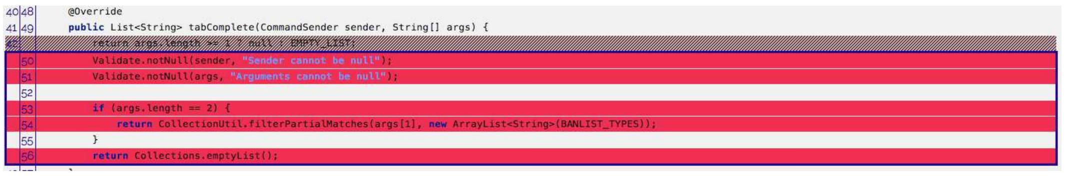


Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.bukkit.command	-4.67%	+43 (16.29%)	-2.14%	+12 (21.05%)	
org.bukkit.command.defaults	-0.72%	+60 (7.44%)	0.0%	+15 (6.91%)	
org.bukkit.event.player	-0.05%	+25 (4.33%)	0.0%	+5 (45.45%)	
org.bukkit.util	-0.64%	+11 (2.29%)	-1.19%	+3 (3.37%)	

Figure 6.6: Coverage goes down when no test cases are added for new or changed code

Operias is useful here to quickly identify that no test cases were changed or added. By clicking on a class and opening the combined view, one can see which parts are added but not covered. An example from this pull request is shown in Figure 6.7. Here the changes which are not covered are highlighted with dark red lines. So, in this case, a reviewer can see that the code is not covered, and can still use Operias' generated site to review the changes in the code manually.

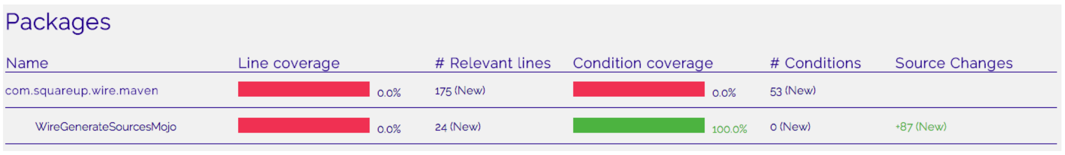


```
    @Override
    public List<String> tabComplete(CommandSender sender, String[] args) {
        return args.length >= 1 ? null : EMPTY_LIST;
        Validate.notNull(sender, "Sender cannot be null");
        Validate.notNull(args, "Arguments cannot be null");

        if (args.length == 2) {
            return CollectionUtil.filterPartialMatches(args[1], new ArrayList<String>(BANLIST_TYPES));
        }
        return Collections.emptyList();
```

Figure 6.7: Not covered changes can be identified easily with Operias

Pull request #92 from Wire

In pull request #92, illustrated in Figure 6.8, one file was added to the project, *WireGenerateSourcesMojo*. As seen in the report, no test cases were added in this pull request and so this class is also not tested. Operias can be used here to quickly determine that the class is indeed not covered. The condition coverage is 100% here which can be confusing, because the class consists of zero conditions so zero of zero conditions are covered, which Cobertura indicates as 100%.



Packages

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
com.squareup.wire.maven	🟥 0.0%	175 (New)	🟥 0.0%	53 (New)	
WireGenerateSourcesMojo	🟥 0.0%	24 (New)	🟩 100.0%	0 (New)	+87 (New)

Figure 6.8: Newly added class, which was not tested

6.3 Pull Request Metrics

Another way to determine the usefulness of Operias, is to study pull requests and retrieve information about the changes in source code and test coverage. We look for metrics which can give us an indication about how often the tool could be useful. For example, if we can see that newly written code is not or poorly covered, we can say that Operias could be of use to detect these faults.

6.3.1 Experimental setup

To evaluate Operias we analyzed the pull requests of a number of projects. We try to answer the question whether or not the tool can be a useful addition to the review process of a pull request. By analyzing pull requests of the three projects, we can identify how well these pull request are tested.

For the evaluation of the projects, the Goal Question Metric (GQM) approach is used. This method consists of a goal for the study, multiple questions to be answered and metrics related to the questions. The goal of this study is to observe the impact of pull requests on test coverage from the point of view of the code reviewer. To achieve the goal, we set up a few questions and derive metrics to answer the questions.

Question 1. *What is the impact of a pull request on the statement coverage for the complete project?*

To start with, the impact of a pull request on the statement coverage of the complete project is analyzed. If the impact of a project is negatively, Operias could be useful to help the developers identify these pull requests. For this question, one metric is collected:

M_1: **percentage change in coverage statement for the complete project.** The XML report Operias produces provides the statement coverage for the original, base version and the revised version. We expect that more than half of the pull request will have a positive impact on the statement coverage, but a significant number of the pull requests will impact the statement coverage negatively. We believe that this is the case, since the selected projects all contain a test suite and during the development of the initial version of a project, the coverage start from 0% and slowly working its way up to a higher percentage.

Question 2. *What is the impact of a pull request on the statement coverage for the changed classes in the pull request?*

This question is a more in depth version of question 1. Here the impact of a pull request is checked on class level. Again, here Operias could be useful be identify the classes which are negatively impacted by a pull request, or identify that classes are not impacted as much as the reviewer would want. For this question, one metric is collected:

M_2: **percentage change in statement coverage for every changed class.** The XML report also provides the statement coverage for the original version of a class and a revised version of a class if a class was marked as changed. Again, as for M_1, we expect that most classes are positively impacted by a pull request, but large part of classes will also be negatively affected for the same reason that a class starts with 0% coverage.

Question 3. *What is the statement coverage for the newly added classes in the pull request?*

Newly added classes should be properly tested. Since these classes are often not already affected by existing tests, a reviewer would like to see that these classes have very high coverage percentages. Operias can be used to identify poorly tested new classes. Again, only one metric is collected:

M_3: **statement coverage percentage for every new class.** Again, the XML report Operias generates provides the statement coverage for every new class in the pull request.

Question 4. *What is the impact of a pull request on the statement coverage of classes which are not changed, added or deleted in the pull request?*

Besides the coverage of the changed classes and added classes, the coverage of other classes can also be affacted. Classes which are tested indirectly can be affected by a change in another class. An indirectly tested method is a method which is used by another method which is tested, but is not tested itself separately. Also, if a test case fails or is edit, the coverage of a class can change without a change in the source of that class. To answer this question, the following metric is collected:

M_4: **change in statement coverage for unchanged classes.** The XML report of Operias consist of files with four different states: new, changed, deleted and same. For this metric, the change in coverage for the classes with the state *same* is collected. We expect that most of the classes are affected positively, but that there also are classes which show a big drop in coverage, because they are tested indirectly.

6.3.2 Analysis and Interpretation

In this section, the results per question for each project are shown and briefly discussed.

Q1: What is the impact of a pull request on the statement coverage for the complete project?

To answer the first question, we analyzed metric M_1 and plotted the results for each project. In Figure 6.9, the results are illustrated for the three projects. We can see for all projects, that a large portion of the pull requests does not affect the test coverage. If we ignore these pull requests, we can see that for JUnit and Wire, more pull request affects the statement coverage positively than negatively. For the Bukkit project, we see that only a small portion of the pull request affects the statement coverage positively. Considering the large number of pull requests that affect the test coverage negatively for all three projects (20% in JUnit, 38% in Bukkit and 25% in Wire) we can say that Operias would be a useful tool to use in the review process of a pull request.

Reasons behind pull requests that do not affect the test coverage. In a large portion of all projects, the coverage is not affected by the pull request, 43% in JUnit, 42% in Bukkit and 35% in Wire. A lot of these pull requests that did not affect the coverage do not have any change in the code, but changes in resources for example. This is the case for 73% of the pull requests for JUnit, 29% for Bukkit and 35% for Wire.

Reasons behind a large change in coverage. Almost every pull request that we analyzed only had a small change in the test coverage of the project. This is because we analyzed the change the complete project. In most cases the amount of code changed in a pull request is only a fraction of the whole project, so the test coverage can only change a little bit.

For JUnit we see that two pull requests had a change of -10 to -15% and for Wire we see a pull request with a change of -30 to -25% and one with +20 to +25% in statement coverage. These large numbers of changes can be explained by either a pull request containing failed test cases or they were created in an early stage of the project where the project size was much smaller.

Q2: What is the impact of a pull request on the statement coverage for the changed classes in the pull requests?

In Figure 6.10, the metric M_2 for the three projects are shown. Using this metric we can answer question 2. For all the projects, we can see that the coverage of almost every changed class is not affected by the pull request. If the class is changed, we can see that for most cases, the coverage only changed a little bit, except for the Bukkit project, where a lot of classes have a larger increase. For both JUnit and Wire we can see some classes with a very large decrease. Overall, we can see the same trend as with metric M_1. If we only consider the classes that do have a change in coverage, then a significant part of those have a decrease in coverage. So again, Operias can support reviewers and developers in order to increase the test coverage.

Reasons behind a large part of classes without a change in coverage.

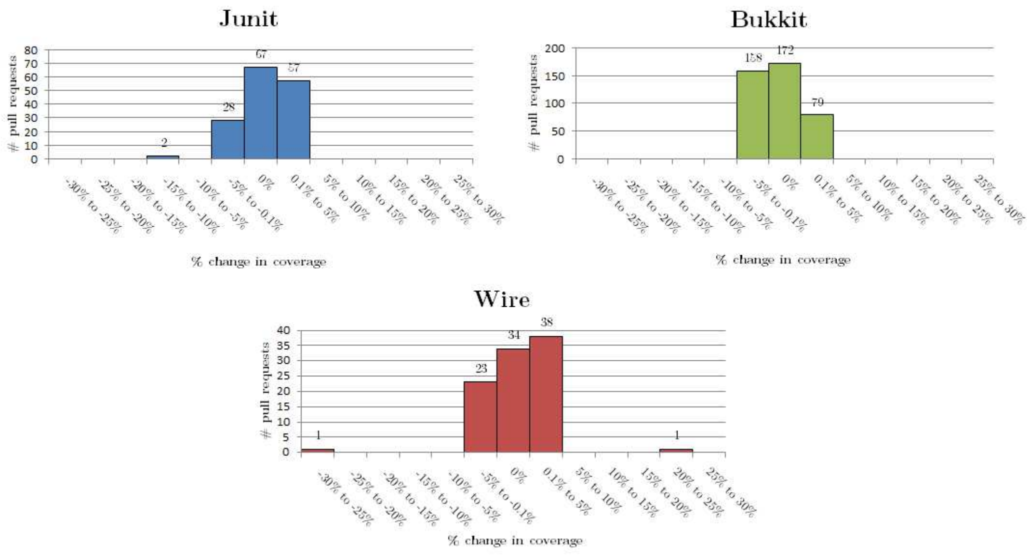


Figure 6.9: Change in overall test coverage because of a pull request

There are four main reasons how a class can have source changes, but without a change in test coverage:

1. The class is either covered for 0% or 100%, and the added or removed lines did not change the coverage.
2. The class is covered for more than 0% and less than 100% and there is a change in one or more lines which do not have effect on the execution of the test suite.
3. Due to the current limitations of Operias, a file is marked as changed if the changes in the file only consist of comments or whitespace. In this case, it is likely that the coverage did not change.
4. Another limitation of Operias is that if the parent class has changes, an inner class is also marked as changed. This is because the inner class has the same source difference as the class it resides in. It is possible that the coverage for this inner class was not affected, and thus resulting in unncessary marking of the class.

To determine how many of the classes are marked as changed unnecessary due to the tools limitations, we did a manual inspection of a fraction of the changed classes to create an estimation of the distribution for the four cases described above. This inspection was done by checking the source difference for 40 classes for each project. The result can be seen in Table 6.1.

Classes marked because of case 3 and 4 can be seen as marked unneccesary, since this can be prevented in the future. We can see that many classes are marked unnecessary (41%

	JUnit	Bukkit	Wire
Case 1	26%	31%	22%
Case 2	33%	14%	11%
Case 3	26%	41%	30%
Case 4	15%	14%	37%

Table 6.1: Distribution of changed classes with no coverage change

in Junit, 55% in Bukkit and 67% in Wire). For JUnit and Wire, we observed that a lot of classes are already covered for 100% and for Bukkit a lot of classes were still covered for 0%. Also, in JUnit, for approximately 33% of the changed classes, only a small change in the class is made which did not affect the test coverage. An example of such a change is the renaming of a variable or method. This number is significantly smaller in the other projects (14% for Bukkit and 11% for Wire).

Reasons behind a large decrease in coverage. In the graph for JUnit, we can see that three classes have a decrease of -95% to -100%. The coverage of these three classes completely dropped to 0%. If we look closer to the report Operias generated, we found that these classes are tested indirectly. By indirect tested, we mean that these classes do not have a test suite but are used by other classes which are tested. So when the class with the test suite stopped using one of these three classes, the coverage dropped to 0%. Coverage changes like this are normally hard to find, but with a report of Operias, reviewers and developers can quickly see these indirect test changes.

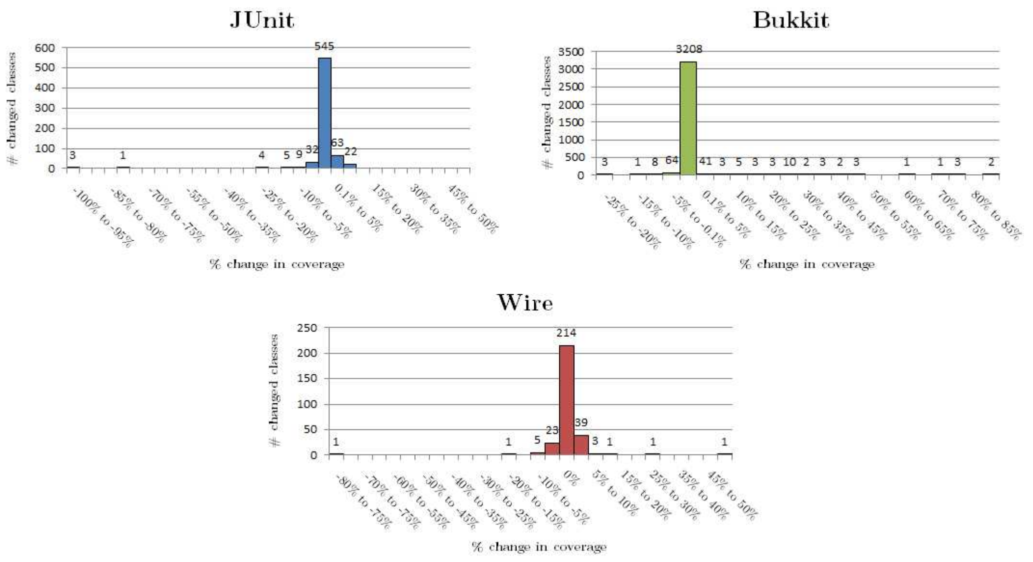


Figure 6.10: Change in test coverage for every changed class made by a pull request

Q3: What is the statement coverage for the newly added classes in the pull request?

Question 3 can be answered by plotting M_3. In Figure 6.11, these plots are illustrated. We can see that all three projects have a different behavior concerning how the added classes are covered. JUnit is an example of a well tested project, where 73% of the newly added classes are fully covered and only 5% of the classes lack a test suite. Bukkit is the other way around, 72% of it's newly added classes are not covered at all. The behavior of Wire is somewhere between JUnit and Bukkit, where almost 20% of the added classes does not come with a test suite.

Reasons behind untested added classes. The different behavior of the projects can be explained by the different co-evolution process [36, 37], where JUnit and Wire can follow a synchronously approach and Bukkit a phased testing approach. Using a phased testing approach, a class can first be added to the project with a pull request without a test case, and afterwards the test cases can be added separately in order to test the added classes. Because of this, Operias shows newly added classes as poorly tested at first. If this is the case for Bukkit, we should see classes with an increase in coverage for Bukkit when answering Q4.

As with the previous questions, we can also say that Operias is useful for reviewing the added classes. Especially when using a synchronous co-evolution, badly tested classes are easily identified.

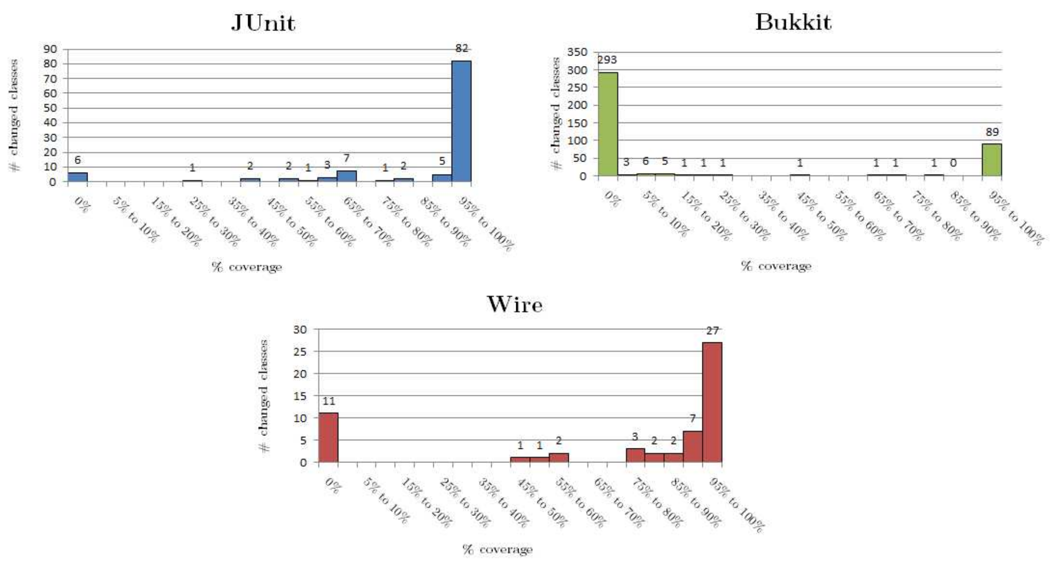


Figure 6.11: Test coverage for every added class made by a pull request

Q4: What is the impact of a pull request on the statement coverage of classes which are not changed, added or deleted in the pull request?

In Figure 6.12, the change in test coverage for classes which are not changed in the pull request are shown. We observed that for Wire, only seven classes had a change in coverage because of a pull request. This could indicate that the class coupling for this project is very low, or that testing only takes place very low in the dependency hierarchy.

For JUnit, we see something opposite as expected. Where newly added classes are very good covered, almost all classes that have a change in coverage but no change in source, decreases in statement coverage. Especially for these classes Operias is very useful, because these classes are not marked as changed in GitHub.

Finally, for Bukkit, we see that indeed a large portion of the classes have a high increase in coverage without changes in the code. This could indicate the phased testing approach as described in Q3. On the other hand, almost an equal number of classes have a decrease in coverage of which most have a decrease of 100%.

Operias is useful to identify every class in these graphs that have a decrease in coverage. A developer can get a good insight on what the impact of his pull request is on the coverage of the rest of the project.

Reasons for a decrease in coverage. There are several reasons for the decrease in coverage. We think that the main reason for a decrease is the fact that a lot of classes are indirectly tested. A change in class A can affect the coverage in class B, for example when a dependency is removed. This indicate high class coupling, which is probably the case in the JUnit project. It also indicates that class B is not tested properly, it should have its own test suite instead of being covered indirectly.

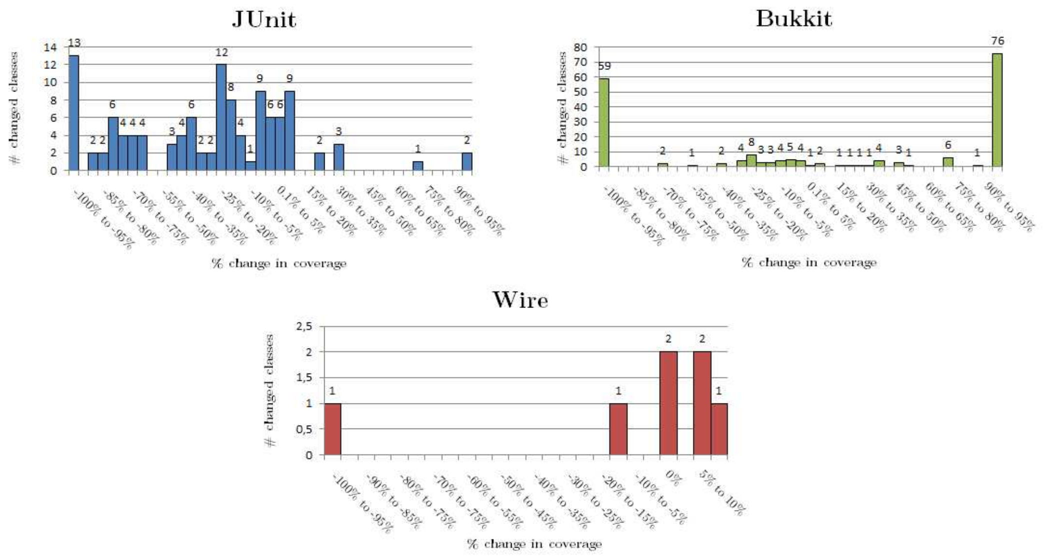


Figure 6.12: Change in test coverage for classes which were not changed in the pull request

Chapter 7

Discussion

Operias has been designed as a great tool for developers and reviewers to understand what the impact of changes are for a project. It helps the users to identify how the changes affect the rest of the project. But as with everything, this tool has its limitiations. This chapter discusses these limitations of Operias and also presents future work that has to be done to further enhance this tool for developers to track the co-evolution of their projects.

7.1 Limitations

The tool Operias has its limitations. For now, only projects in Java which use Maven to build and execute its test suite are supported. The project which must be analyzed should contain the default folder structure of Maven. The test coverage tool which Operias uses, Cobertura, can also be used with Ant, but Operias does not support this yet.

Another limitation of the tool is that it cannot process inner classes properly. An inner class is a class which is defined within another class. Cobertura does give a separate report for the parent class and its inner classes, but Operias matches each Cobertura report to the same source changes, it is unable to detect which part of the file is the innerclass. The result of this limitation is that if there is a change in the parent class or any inner class, all of them are marked as changed, even for the classes which do not have a source change or coverage change.

Comments and whitespace changes are also common in a lot of projects. If a file only has changes in the comments or whitespace, Operias show it as changed. This file is not necessary when reviewing the code changes with respect to test coverage changes.

Besides changes in classes and the test suite, Operias also shows which tests succeeded and which failed. Operias uses the surefire reports automatically generated by Maven to determine which tests failed. A limitation of the surefire plugin is that it does not support aggregation. If a project consists of multiple modules, Operias can ́t collect the test suite results yet, because the results are not aggregated to the top level directory.

7.2 Future Work

In the future, Operias can be extended in several ways. Besides statement coverage, Cobertura also provides condition coverage in their reports. This is also very valuable information and is also shown in the Operias reports. However, when viewing the changes in a specific class, only statement coverage changes and code changes are shown. An more than welcome extension would be to show the changes in condition coverage in this view.

With the current version of Operias, it is possible to compare two folders on a local machine and two versions on GitHub. It is alo possible to compare a version on a local machine with a version on GitHub. This feature is the ideal setup to create an IDE plugin. Using such plugin, a developer can review their own code before creating a pull request.

To allow further integration with Github, the tool Operias should be available to be used in, for example, Travis CI. Travis CI can be used to automatically run a test suite or other tasks when a developer commits to a repository on Github, or when a pull request is created. If Operias can be integrated with Travis, projects can use their own set up as environment for executing their tests and no extra setup would be needed for the servlet presented in the Git chapter of this thesis.

This thesis gives a schematic on how to create a tool like Operias, and a C# version of the tool would be appreciated by lots of developers.

Chapter 8

Related Work

A tool such as Operias has to our knowledge not yet been made, so projects directly related to this thesis are hard to find or non existing. However, the tool covers several important subjects in software engineering, in the upcoming sections some of the related work in these subjects is discussed.

8.1 Code Review

Belachandran [9] shows that integrating static analysis tool with code review process can improve the quality of the code. Such a tool, like Operias, reduces the human effort and helps to detect common defect patterns.

Gousios et al [17] investigated how developers are using the pull request mechanism in Github. In most of the cases (79%), developers and reviewers use the GitHub facilities, such as the merge button, to accept and merge the pull request. More than 70% of the integrators indicated that they always review the code in a pull request and more than 60% think that it is obligatory to review the code in a pull request.

Marinescu et al [24] constructed the tool KATCH which can be used as an automated tool for testing software patches. It is capable of finding bugs in patches using symbolic execution.

Tao et al [33] did a large-scale quantitative and qualitative study at Microsoft. The study investigates the need for understanding the changes and the role of understanding code changes during the software-development process.

Stakeholders interact differently with one another during the review process, Rigby et al [30] examined reviews of open-source project and provide insight into the community-wide techniques that developers use to effectively manage large quantities of reviews.

For a good integration of Operias with GitHub, it is important to know what testing strategies are important. Pham et al [28] suggests guidelines for promoting a sustainable testing culture in software development projects.

Github offers pull-based developments using pull requests, Gousios et al [16] found that the pull request model offers fast turnaround, increased opportunities for community engagement and decreased time to incorporate contributions.

8.2 Co-evolution

The tool ChronoTwigger [13] helps to understand the co-evolution of source and test code. It is a visual analytics tool which can be run on the history of a project and helps to give insight about its evolution.

Zaidman et al [37] showed different ways how co-evolution within an open-source project occurs. It describes, for example, the difference between asynchronous and synchronous testing and how this affects the co-evolution.

As follow up on KATCH, Marinescu et al [23] constructed the lightweight framework Covrig which can be used to analyze the evolution of code, test and coverage. Covrig executes the test suite of every version available in a version control system and plots the evolution in different ways. Operias can be used for the same purpose, but it also includes a combined view of source and test coverage per file.

Chapter 9

Conclusion

This chapter reflects on the contributions of the thesis, as well as summarizes the conclusions that can be made after the evaluation of Operias.

9.1 Contributions

This thesis has several main contributions:

- A high level description on how a review tool such as Operias can be constructed. This architecture can easily be implemented in any other program language or for different coverage tools.
- The actual implementation of the tool, which is available for free on GitHub. It is possible to use the tool on a local machine or as a service which can be added to a GitHub project.
- The evaluation of the tool, in which we show how the tool can be useful in three projects.

9.2 Conclusions

During the evaluation of the tool, several examples were given to show the visual advantages of using Operias as review tool. The HTML site provides a clear overview on what changed. It also helps reviewers identify changes in test coverage in other packages because of indirectly tested pieces of code.

The second part of the evaluation consisted of a *Goal Question Metrics* approach to determine in how many cases Operias could be useful. We saw that for all three projects the project's statement coverage was negatively affected by pull requests (20% in JUnit, 38% in Bukkit and 25% in Wire).

Pull requests which contain newly added classes show very different behavior in every project. Where JUnit's new classes come with a proper test suite in about 80% of the time, Bukkit's new classes lack a test suite in 72%. Wire is half way between the other two projects, where 20% of the classes are missing a test suite. Operias can be used here to quickly identify those newly added classes which were not covered properly.

If we look closer at files with a change in test coverage, but no change in source code, we see that there are several cases in which the coverage drops significantly. This can be an indication of indirect testing and for identifying these cases, we think Operias is very useful.

Based on the evaluation we can conclude that Operias is a useful tool for reviewers. It can help to identify special cases in which the coverage was affected negatively, and also shows cases in which the developer did a good job in writing test cases.

Bibliography

[1] Asm. `http://asm.ow2.org/`.

[2] Cobertura. `http://cobertura.github.io/cobertura/`.

[3] jcoverage. `http://java-source.net/open-source/code-coverage/jcoverage-gpl`.

[4] Ansi/ieee standard 1008-1987, ieee standard for software unit testing. *IEEE*, 1986.

[5] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. Software inspections: an effective verification process. *Software, IEEE*, 6(3):31–36, 1989.

[6] A.F. Ackerman, P.J. Fowler, and R.G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40. Elsevier North-Holland, Inc., 1984.

[7] P. Ammann and F. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[8] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[9] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 931–940. IEEE Press, 2013.

[10] B. Beizer. Software testing techniques. 1990. *New York, ISBN: 0-442-20672-0*.

[11] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[12] E.W. Dijkstra. *Notes on structured programming*. Technological University Eindhoven Netherlands, 1970.

[13] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J.E. Young, and P. Irani. Chronotwigger: A visual analytics tool for understanding source and test co-evolution.

[14] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *Software Engineering, IEEE Transactions on*, (2):156–173, 1975.

[15] J.B. Goodenough and S.L. Gerhart. Toward a theory of testing: Data selection criteria. *Current trends in programming methodology*, 2:44–79, 1977.

[16] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355, 2014.

[17] G. Gousios, A. Zaidman, M. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. Technical report, Proceedings International Conference on Software Engineering (ICSE), 2015., 2014.

[18] W.C. Hetzel and B. Hetzel. *The complete guide to software testing*. John Wiley & Sons, Inc., 1991.

[19] D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM (JACM)*, 24(4):664–675, 1977.

[20] J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[21] Lemur Heavy Industries. Coveralls. `https://coveralls.io/`.

[22] M.M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.

[23] P. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 93–104. ACM, 2014.

[24] P.D. Marinescu and C. Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[25] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software evolution*, pages 173–202. Springer, 2008.

[26] E.W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[27] G.J. Myers, C. Sandler, and T. Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[28] R. Pham, L. Singer, O. Liskin, K. Schneider, et al. Creating a shared understanding of testing culture on a social coding site. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 112–121. IEEE, 2013.

[29] P.C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D.M. German. Open source peer review lessons and recommendations for closed source. *To appear in IEEE Software*, 2012.

[30] P.C. Rigby and M. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 541–550. ACM, 2011.

[31] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

[32] SonarSource SA. Sonarqube. `http://www.sonarqube.org/`, 2008.

[33] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[34] E.J. Weyuker. The applicability of program schema results to programs. *International Journal of Computer & Information Sciences*, 8(5):387–403, 1979.

[35] E.J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[36] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.

[37] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[38] H. Zhu, P.A.V. Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.

[39] Hong Zhu, Patrick Hall, and John May. Inductive inference and software testing. *Software Testing, Verification and Reliability*, 2(2):69–81, 1992.