

Visualizing Code and Coverage Changes for Code Review

Sebastiaan Oosterwaal
Arie van Deursen

Delft University of Technology
The Netherlands
sebastiaan.oosterwaal@gmail.com
Arie.vanDeursen@tudelft.nl

Roberta Coelho

Federal University of
Rio Grande do Norte
Brazil
souzacoelho@gmail.com

Anand Ashok Sawant

Alberto Bacchelli

Delft University of Technology
The Netherlands
A.A.Sawant@tudelft.nl
A.Bacchelli@tudelft.nl

ABSTRACT

One of the tasks of the reviewer is to verify that code modifications are well tested. Unfortunately, with current tool support, it is hard to understand precisely how changes to the code correspond to changes to the tests. In particular, it is hard to see if modified test code actually covers the modified code. To overcome this problem, we developed OPERIAS, a tool that provides a combined visualization of fine-grained source code differences and coverage impact. OPERIAS works both as a stand-alone tool on specific project versions and as a service hooked to GitHub. In the latter case, it provides automated reports for each new pull request, which the reviewer can use to assess the code contribution. OPERIAS works for any Java project that works with maven and its standard Cobertura coverage plugin. We present how OPERIAS can be used to identify a number of test-related problems in existing real-world pull requests. OPERIAS is open source and available on GitHub with a demo video: <https://github.com/SERG-Delft/operias>

1. INTRODUCTION

Code review consists in the manual assessment of source code changes by developers other than the author and is mainly intended to identify defects and quality problems before the deployment in a live environment [9]. Several studies provided evidence that code review supports software quality and reliability crucially [8, 18].

Modern code reviews (MCR) [9], as currently used in several large software and open-source software (OSS) projects, are informal, asynchronous, and supported by tools. Popular examples of code review tools are CodeFlow by Microsoft [9], Gerrit by Google [5], and the GitHub pull-request (PR) mechanism [4]. Although great for supporting the logistics of code review (e.g., inviting reviewers, in-line commenting, and accept/reject decisions), code review tools currently offer no support to help reviewers evaluating the quality of a change, other than basic highlighted textual differencing.

Particularly, current code review tools offer no information on how a code change under review affects test coverage, even though this is one of the most important pieces of information for developers when assessing a code change [21]. In fact, not only software

must evolve over time to stay useful [13], but newly added features should be tested as soon as possible to ensure that they work properly [19] and any change to existing code requires a retest, since any change could potentially invalidate the test suite [14].

In this paper we present OPERIAS, the tool we devised to try to fill this gap. OPERIAS enriches code review tools with *fine-grained test coverage change information*. It comprises two parts: (1) The core part, which accepts two versions of a software project, computes the differences in both source code and statement coverage, and outputs a report in XML and HTML format; and (2) the code review extension part, which runs the core as a service and connects it to GitHub, so that a report is automatically generated for every opened PRs and reviewers can see fine-grained test coverage information while evaluating the code.

As a form of initial evaluation of OPERIAS, we use it to analyze PRs from three OSS projects. Results indicate that OPERIAS would provide reviewers with new information for 27% to 71% of the PRs and that it would be useful in a number of scenarios, such as when it makes visible that a code change affects the coverage of a class not directly modified in the PR.

2. OPERIAS IN A NUTSHELL

OPERIAS is a tool to collect, analyze, and visualize simultaneously code change and related test coverage information to support the code review process. In the following we detail how it is implemented and the visualization it offers.

2.1 Implementation details

OPERIAS is available as an OSS project and is implemented for the Java eco-system. It builds upon the standard Maven [1] setup for Java projects (in which tests are executed with the maven Surefire plugin) and takes advantage of the standard maven Cobertura plugin to obtain both *statement* (or line) and *condition* coverage information.

Given two versions of such a maven project, OPERIAS produces an XML and a HTML report that provides the combined visualization of the changes in the code as well as in the test coverage. The two versions can be in two separate directories or can be identified as two commits (or tags) in a git repository. To get the changes between the two folders, we use Myer's diff algorithm [15] and annotate them with test coverage information; details about the diffing mechanism are available on the accompanying thesis [16].

OPERIAS can also operate as a service hooked to git or GitHub: With this service, whenever a PR is opened on GitHub, OPERIAS is run to visualize the changes in code and coverage introduced by this PR. The service notifies involved GitHub users, by automatically adding a comment to the appropriate PR and providing a link to the visualization (Figure 1).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FSE '16 Seattle, Washington USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

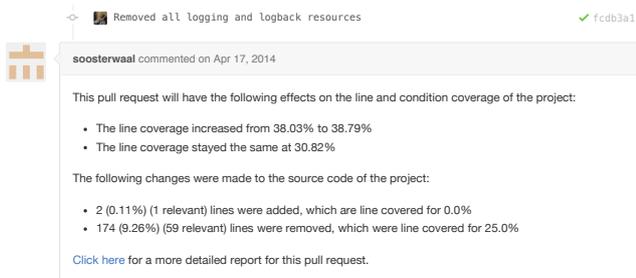


Figure 1: An OPERIAS generated comment on GitHub

2.2 Reporting changes and test coverage

OPERIAS generates a number of browsable reports to visualize code changes together with the corresponding test coverage information. We detail them from the least to the most fine-grained.

Project Overview. The ‘project overview’ is the first report (Figure 2), in which all packages are displayed. By clicking on a package, all changed classes within this package appear. For every class and package, two bars visualize the status of condition and statements coverage. These bars make use of four colors (also used in the ‘class view’ with the same semantic) with the following meaning: *light green* indicates parts that were covered in the original version and are still covered in the new version, *dark green* indicates an increase in coverage in the new version, *light red* indicates parts not covered in the original version and still not covered in the new version, and *dark red* indicates parts that were covered but are no longer covered in the new version. To increase readability, also coverage percentage points are visualized in the report.

The ‘project overview’ reports also provides an indication for deleted and newly created classes. 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.

Test View. The ‘test view’ report (Figure 3) contains information on source changes in test classes (coverage does not apply). To support reviewing tests, we show 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 (example images are available in the accompanying thesis [16]).

Class View. The finest-grained visualization is offered by the ‘class view’ report, which can be accessed by clicking on any class in the ‘project overview’. In the report, up to four code views are shown: *original file*, where 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*, which corresponds to the previous view, but showing the new version of the file; *source changes*, where only source changes between the versions (since red and green are used for conveying coverage information, we use the shaded background to mean 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); and *combined view*, the most characteristic view of OPERIAS, where it shows both source changes (similarly to the previous view) and coverage information for both versions (Figure 4) 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.

These four views are available for changed files. For added files, only the revised file view is shown including the coverage informa-

Table 1: Distribution of test coverage change across pull requests

Project	Test coverage across pull requests		
	Decreased	Stable	Increased
Bukkit	38%	29%	33%
JUnit	20%	73%	07%
Wire	25%	35%	40%

tion, 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.

3. REAL-WORLD USAGE SCENARIOS

We present real world usage scenarios to illustrate the benefits of the code change assessment support that OPERIAS offers by providing fine-grained test coverage information. We explore three OSS projects (JUnit [6], Wire [7], and Bukkit [2]) from different application domains and size, and hosted on the GitHub platform.

Overall applicability. As a first step, we get an indication of the general *applicability* of OPERIAS. We check the distribution of changes in test coverage across all the PRs of the selected project. We do so by running OPERIAS core on the entire code history and computing the effects of each single PR on the test coverage of the overall project. Table 1 summarizes the results, showing the proportion of PRs in which test coverage decreases, is stable, or increases. Results show that for JUnit (a well-tested system) only few PRs increase the coverage, while for Bukkit and Wire (with less coverage to start with) at least a third of PR increase it. More extensive metrics and underlying causes are discussed in the accompanying thesis [16]; here we note that OPERIAS would provide test coverage information, which is currently not available to reviewers, for a significant proportion of PRs (from a minimum of 27% PRs for JUnit, up to a maximum of 71% PRs for Bukkit).

Usefulness. As a second step, we investigate the *usefulness* of OPERIAS from the point of view of the reviewer, in order to verify whether the newly provided information has the potential to help the reviewer in judging a code contribution. To do so, from the three projects we manually inspect several PRs in which test coverage is either increased or decreased. The complete analysis can be found in the accompanying thesis [16], here we limit ourselves to interesting PRs from JUnit.

PR/#767: In this PR, a new ‘plugin’ package is added. OPERIAS’ ‘project overview’ shows the reviewer that all the newly created classes are dark green and fully (100%) tested (figure omitted for space reasons, available in [16]). Furthermore, the PR changed another class and the reviewer can see a small dark red bar, indicating new code that is not tested. The reviewer is able to click on that class and, with the combined ‘Class View’ (Figure 5), see exactly which lines were added and where testing is lacking.

PR/#896: In this PR, the contributor makes a 1-line change to one class and adds 117 test lines for this class. While this sounds like a good PR, using OPERIAS the reviewer can see (Figure 6) that the change affects the statement coverage of a completely different class (‘EachTestNotifier’) reducing its coverage by 10%. Even though this class is not part of the original PR, OPERIAS shows it because its coverage is affected by the changes under review. Industrial reviewers reported that knowing which parts of the code are indirectly affected by a change is crucial to assess its quality [21]; using OPERIAS indirect changes in coverage are easy to detect.

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

Figure 2: The ‘project overview’ with package- and class-level information

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

Figure 3: The ‘test view’ with data on added/removed test lines

```

72 73 private void validateMember(FrameworkMember<?> member, List<Throwable> errors) {
74     validatePublicClass(member, errors);
75     validateStatic(member, errors);
76     validatePublic(member, errors);
77     validateTestRuleOrMethodRule(member, errors);
78 }
79
80 private void validatePublicClass(FrameworkMember<?> member, List<Throwable> errors) {
81     if (fStaticMembers && !member.isDeclaringClassPublic(member)) {
82         addError(errors, member, "must be declared in a public class.");
83     }
84 }
85
86 private void validateStatic(FrameworkMember<?> member, List<Throwable> errors) {
87     if (fStaticMembers && !member.isStatic()) {
88         addError(errors, member, "must be static.");
89     }
90     if (!fStaticMembers && member.isStatic()) {
91         addError(errors, member, "must not be static.");
92     }
93 }

```

Figure 4: The combined view in the ‘class view’ report

PR/#646: In this PR, 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; in fact, for a test case to be successfully executed within the JUnit project, it must be added to this class. Using OPERIAS, the reviewer can quickly see that the added test code affects neither line coverage nor condition coverage (Figure 7), thus indicating that the new tests are not executed and the absence of changes to the class ‘AllTests’ from the view.

Although anecdotal, these examples of PRs provide initial evidence on the potential of OPERIAS in supporting the code review process. As a future evaluation, we plan to design and conduct a controlled experiment to measure the causal effects of OPERIAS on the code review process, in particular with respect to the reviewing speed and number of changes suggested by reviewers. Moreover, an observational study can be conducted to see whether the usage of OPERIAS has a relation with a reduced number of further changes needed in code already accepted through PRs.

4. RELATED WORK

Previous research on the pull-based development model has highlighted the importance of tests in pull requests. First, pull requests are merged faster in a well-tested system [10]; then integrators, re-

```

65 72
73     private Plugin[] constructPlugins(Class<?> klass)
74         throws InitializationError {
75         Plug annotation = klass.getAnnotation(Plug.class);
76         if (annotation == null) {
77             return null;
78         }
79
80         Class<?> extends Plugin<?> pluginClasses = annotation.value();
81         if (pluginClasses == null || pluginClasses.length == 0) {
82             return null;
83         }
84
85         Plugin[] plugins = new Plugin[pluginClasses.length];
86         for (int i = 0; i < pluginClasses.length; ++i) {
87             Class<?> extends Plugin<?> pluginClass = pluginClasses[i];
88             Plugin plugin;
89             try {
90                 plugin = pluginClass.newInstance();
91             } catch (ReflectiveOperationException e) {
92                 throw new InitializationError(
93                     String.format(
94                         "Plugin class %s should have a public constructor with no parameters",
95                         pluginClass.getSimpleName());
96             )
97             plugins[i] = plugin;
98         }
99         return plugins;
100     }
101 }
66 102 //

```

Figure 5: Effect of PR/#767 on the coverage of a changed class.

sponsible for merging, indicate that adequate testing is a key quality factor taken into account when deciding whether or not to accept a change. [11]. Pham et al discuss the testing culture on GitHub projects, and observe that projects indeed insist on tests in PRs [17].

Although many tools exist to either show differences between two versions of a piece of code or compute test code coverage (e.g., [3]), only a few combine both pieces of information in one view. The most popular are: Coveralls.io [12] and SonarQube [20].

Coveralls.io [12] is a web application that analyzes the report created by Cobertura [3] by comparing the test coverage metrics to a previous report. It shows an overview with detailed coverage information also showing whether test coverage increased or decreased, at the file level. Test coverage information is not integrated in the review process and Coveralls.io does not provide fine-grained information about which lines are affected in terms of test coverage.

SonarQube [20] is an extensive tool to evaluate the quality of a codebase and its changes; it visualizes information on code duplication, coverage, code complexity and more. Particularly, it shows current coverage information of a class and one can filter on selected changes or timeframes, showing lines to cover, branches to cover, uncovered lines and uncovered branches. Nevertheless, SonarQube does not provide any comparison view of test coverage between changes, but only reports on review specific statuses.

5. CONCLUSIONS

We created OPERIAS as a code review support tool that lets reviewers visualize fine-grained test coverage information while evaluating a code contribution on GitHub. Through real-world exam-

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.junit.internal.runners.model	-7.41%	0 (0.0%)	0.0%	0 (0.0%)	
EachTestNotifier	-9.52%	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: Effect of PR/#896 on the coverage of classes, including the *not* changed ‘EachTestNotifier’ class.

Name	Line coverage	# Relevant lines	Condition coverage	# Conditions	Source Changes
org.junit	-5.3%	+16 (6.56%)	-7.36%	+5 (10.64%)	
Assert	-7.41%	+16 (8.99%)	-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 7: Effect of PR/#646 on the coverage; since ‘AllTests’ does not include the new tests, there is no positive change in coverage.

ples we gave initial evidence of its usefulness in a number of reviewing scenarios and, potentially, for a large proportion of changes to review. The tool and a video of OPERIAS is available at <https://github.com/SERG-Delft/operias>

6. REFERENCES

- [1] Apache Maven. <https://maven.apache.org/>.
- [2] Bukkit. <http://bukkit.org/>.
- [3] Cobertura. <http://cobertura.github.io/cobertura/>.
- [4] Collaborative code review. <https://github.com/features>.
- [5] Gerrit code review. https://www.gerritcodereview.com.
- [6] Junit. <http://junit.org/>.
- [7] Wire. <https://github.com/square/wire>.
- [8] A. Ackerman, L. Buchwald, and F. Lewski. Software inspections: an effective verification process. *Software, IEEE*, 6(3):31–36, 1989.
- [9] 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.
- [10] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355, 2014.
- [11] G. Gousios, A. Zaidman, M. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: the integrator’s perspective. In *Proceedings International Conference on Software Engineering (ICSE)*, 2015.
- [12] L. H. Industries. Coveralls. <https://coveralls.io/>.
- [13] M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1980.
- [14] 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.
- [15] E. Myers. An (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.
- [16] S. Oosterwaal. Combining source code and test coverage changes in pull requests. Master’s thesis, Delft University of Technology, 2015. <http://repository.tudelft.nl/>.
- [17] 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.
- [18] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German. Open source peer review – lessons and recommendations for closed source. *To appear in IEEE Software*, 2012.
- [19] P. Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.
- [20] S. SA. Sonarqube. <http://www.sonarqube.org/>.
- [21] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of FSE 2012 (20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering)*, FSE ’12, pages 51:1–51:11. ACM, 2012.

A walk through Operias

In this walk through we introduce John, an open-source software (OSS) developer interested in contributing to SimpleMath, an OSS project hosted on GitHub that. SimpleMath is a utility that provides functionality to multiply and divide two integers; Figure 1 shows the two classes in this project.

```
1 package multiplication;
2
3 public class Multiplier {
4
5     public int multiply(int a , int b) {
6         return a * b;
7     }
8
9 }
10
```

(a) Multiplier class in SimpleMath

```
1 package division;
2
3 public class Divider {
4
5     public double divide(int a, int b) {
6         return (double) a / b;
7     }
8
9 }
10
```

(b) Divider class in SimpleMath

Figure 1: Classes in SimpleMath

```
1 package multiplication;
2
3 import static org.junit.Assert.assertEquals;
4
5
6 public class MultiplierTest {
7
8
9     @Test
10    public void testMultiply() {
11        Multiplier multi = new Multiplier();
12        assertEquals(multi.multiply(3, 2), 6);
13    }
14
15    @Test
16    public void testMultiplyByZero() {
17        Multiplier multi = new Multiplier();
18        assertEquals(multi.multiply(3, 0), 0);
19    }
20
21 }
```

(a) MultiplierTest class

```
1 package division;
2
3 import static org.junit.Assert.assertEquals;
4
5
6 public class DividerTest {
7
8
9     @Test
10    public void testDivide() {
11        Divider divider = new Divider();
12        assertEquals(divider.divide(4, 2), 2.0, 0.0);
13    }
14
15    @Test
16    public void testDivideZeroNumerator() {
17        Divider divider = new Divider();
18        assertEquals(divider.divide(0, 2), 0.0, 0.0);
19    }
20
21 }
```

(b) DividerTest class

Figure 2: Unit tests in SimpleMath

The developers of this project require every line of code to be tested. This results in the test classes that can be seen in Figure 2.

John uses the SimpleMath module in his application to perform simple division. He notices that sometimes his application crashes when the divisor is 0. The stack trace shows that this error originates from the Divider class. He looks at the implementation of the class and discovers an obvious flaw in the divide method, wherein there is no check for division by 0. To fix this, he adds a check (Figure 3) and issues a Pull Request (PR) on the repository.

```

1 package division;
2
3 public class Divider {
4
5     public double divide(int a, int b) {
6         if (b != 0) {
7             return (double) a / b;
8         } else {
9             return Double.MAX_VALUE;
10        }
11    }
12
13 }

```

Figure 3: Fixed Divider class

Once the PR is issued, Operias checks out the base and the revised version of the code and analyzes both. Afterwards, it posts a comment (Figure 4) on the PR summarizing the changes with respect to code coverage and a link to a detailed report with a line-by-line overview.

operias commented 2 minutes ago

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

- The line coverage decreased from 100.0% to 83.33%
- The condition coverage decreased from 100.0% to 50.0%

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

- 6 (66.67%) (3 relevant) lines were added, which are line covered for 33.0%
- 2 (22.22%) (1 relevant) lines were removed, which were line covered for 100.0%

[Click here](#) for a more detailed report for this pull request.

Figure 4: A comment to a PR automatically generated by Operias

Alex is a core developer of SimpleMath who reviews the PR created by John. First, he sees the comment posted by Operias that the PR has reduced the statement coverage and the conditional coverage of the project. He sees that only 33.3% of the added lines are tested. He then inspects the detailed report (Figure 5) to get a clear idea of exactly which changes negatively impacted the coverage.

In the detailed report Alex sees that a change to the Divider class (A) has reduced the line coverage by 25% (B) and the branch coverage by 50% (C) due to the introduction of one new branch statement (D). By clicking on the class name he sees a detailed view of the change in line-by-line coverage statistics and the changed code of the class (Figure 6).

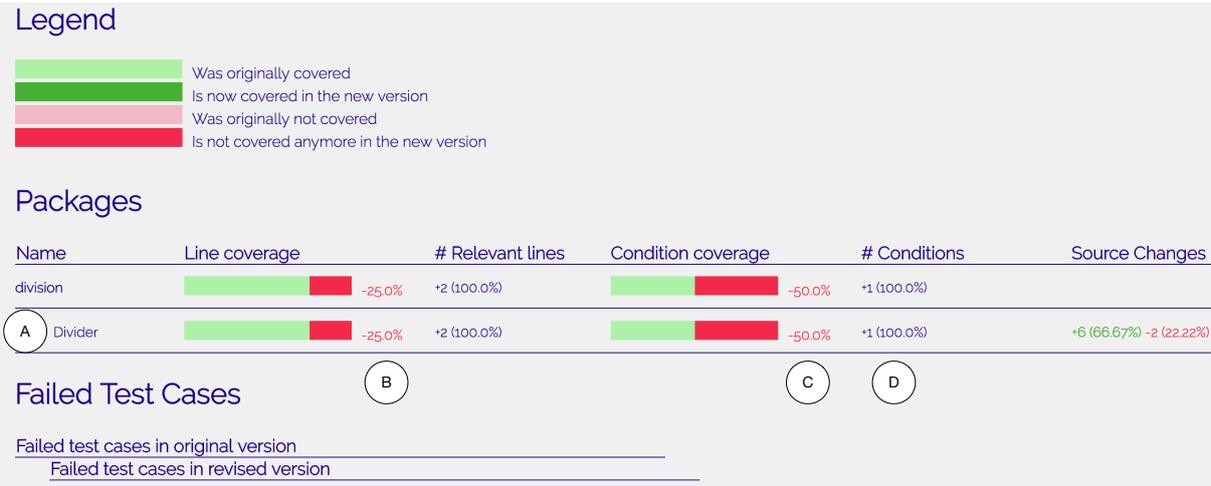


Figure 5: Detailed report on changed test coverage in a submitted PR, generated by Operias

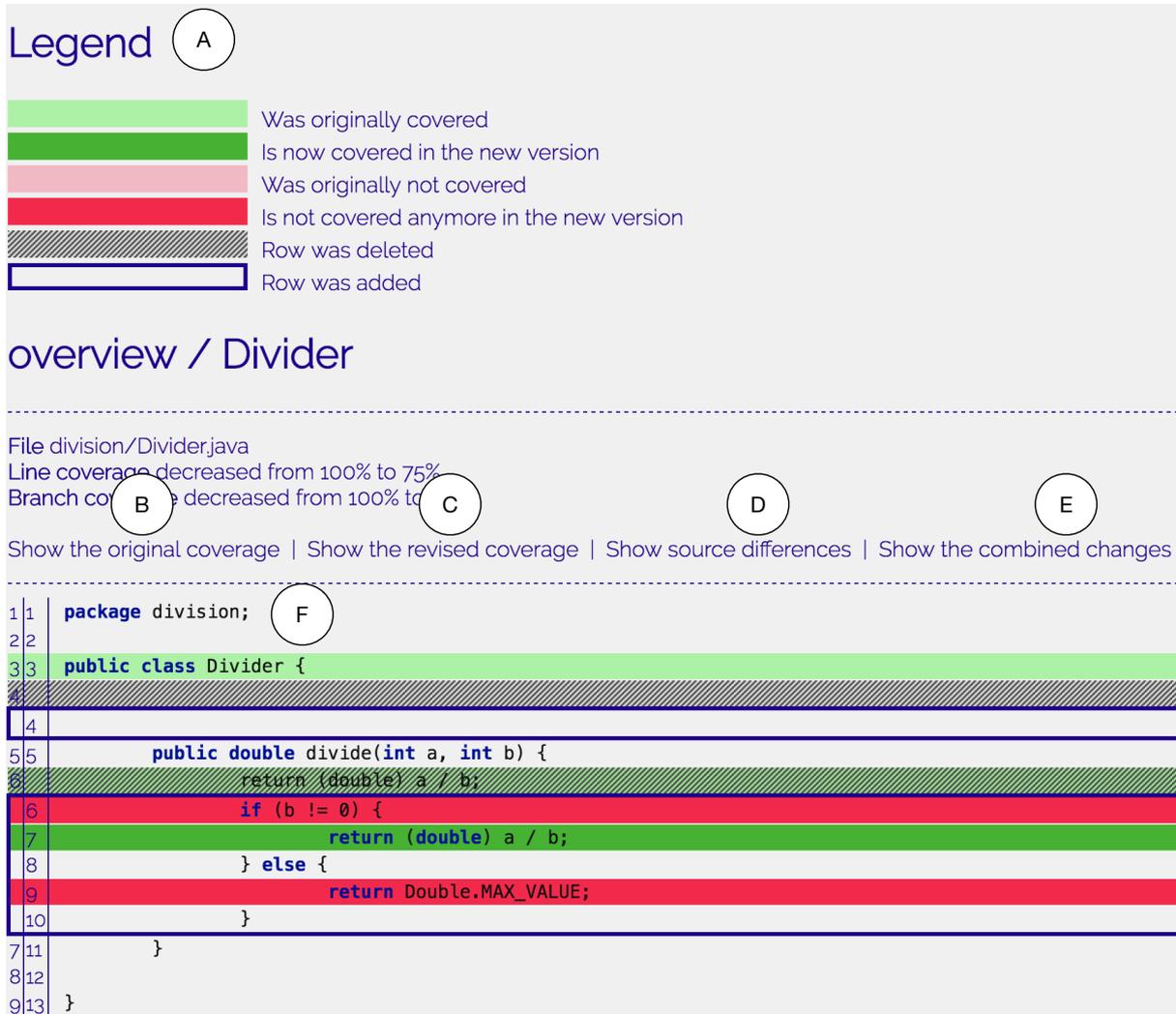


Figure 6: Line-by-line detail for Divider class

In the detailed overview for the class, Alex sees the delta in terms of coverage on a line-by-line basis. The legend (A) helps him identify the the various color codings in the detailed report.

For instance red indicates the lines that have not been covered in the new version of the code. To make his task easier, there are four options that allow him to see the original coverage (B), the revised coverage (C), the source difference (D) and a combined view of all 3 (E, F).

Alex clicks on the original coverage tab to see what the file looked like before any modifications (Figure 7). Here he notices that the file was covered in its entirety. He then proceeds to click on the revised coverage tab and sees that the if statement is not fully covered and needs to be tested (Figure 8).

```
File division/Divider.java
Line coverage decreased from 100% to 75%
Branch coverage decreased from 100% to 50%

Show the original coverage | Show the revised coverage | Show source differences | Show the combined changes
```

```
1 package division;
2
3 public class Divider {
4
5     public double divide(int a, int b) {
6         return (double) a / b;
7     }
8
9 }
```

Figure 7: Original coverage report

This information leads Alex to reject the pull request as it has a negative impact on code coverage and thus does not fully comply with the coding practices of the SimpleMath project. He asks John to re-issue a new PR wherein the added code has been tested sufficiently.

```
File division/Divider.java
Line coverage decreased from 100% to 75%
Branch coverage decreased from 100% to 50%

Show the original coverage | Show the revised coverage | Show source differences | Show the combined changes
```

```
1 package division;
2
3 public class Divider {
4
5     public double divide(int a, int b) {
6         if (b != 0) {
7             return (double) a / b;
8         } else {
9             return Double.MAX_VALUE;
10        }
11    }
12
13 }
```

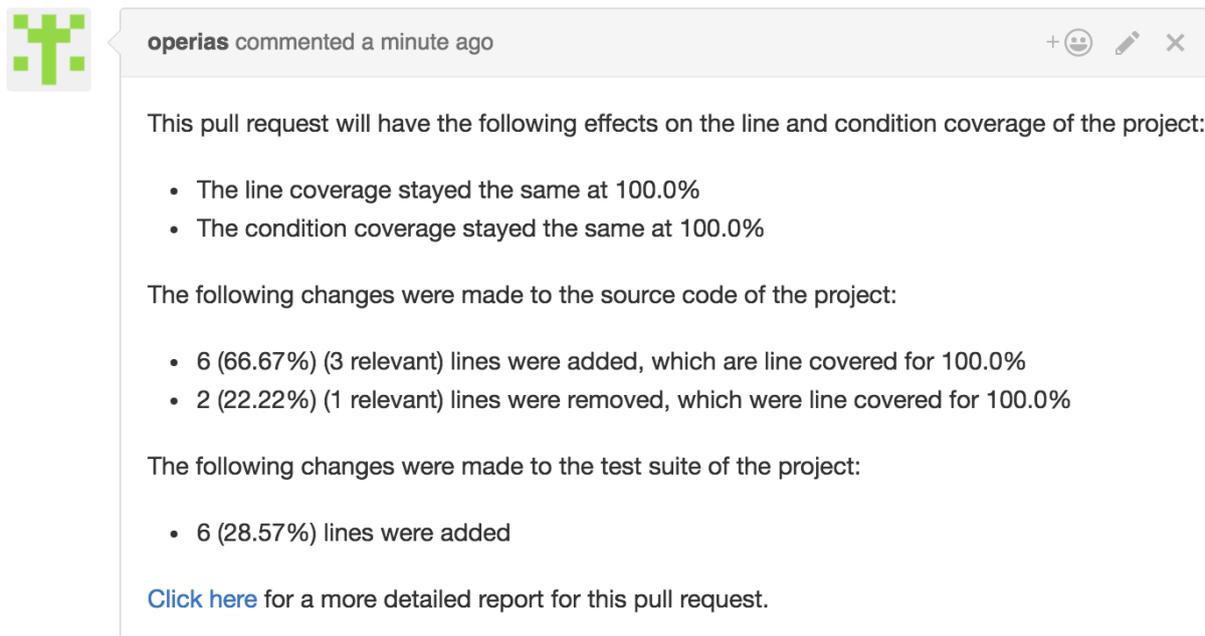
Figure 8: Revised coverage report

John understands that he has to add a test that executes the else branch of the if statement as well and to that end he adds an additional test to the DividerTest class as seen in figure 9. On adding the new test, John issues a new PR on the SimpleMath library. This time Operias posts the comment as seen in figure 10. Alex reviews this new PR and sees that the coverage statistics are all at 100% again, and that this is due to the addition of a new test case that tests

the previously added code. This time he happily accepts the PR as it meets his expectation and thanks John for his contribution.

```
1 package division;
2
3+ import static org.junit.Assert.assertEquals;
4
5
6
7 public class DividerTest {
8
9-     @Test
10     public void testDivide() {
11         Divider divider = new Divider();
12         assertEquals(divider.divide(4, 2), 2.0, 0.0);
13     }
14
15-     @Test
16     public void testDivideZeroNumerator() {
17         Divider divider = new Divider();
18         assertEquals(divider.divide(0, 2), 0.0, 0.0);
19     }
20
21-     @Test
22     public void testDivideZeroDivisor() {
23         Divider divider = new Divider();
24         assertEquals(divider.divide(2, 0), Double.MAX_VALUE, 0.0);
25     }
26
27 }
```

Figure 9: Updated DividerTest class



The screenshot shows a comment from a user named 'operias' on a pull request. The comment is enclosed in a light gray box with a green 'T' icon on the left. The text of the comment provides a detailed analysis of the pull request's effects on the project's line and condition coverage. It lists the following points:

- This pull request will have the following effects on the line and condition coverage of the project:
 - The line coverage stayed the same at 100.0%
 - The condition coverage stayed the same at 100.0%
- The following changes were made to the source code of the project:
 - 6 (66.67%) (3 relevant) lines were added, which are line covered for 100.0%
 - 2 (22.22%) (1 relevant) lines were removed, which were line covered for 100.0%
- The following changes were made to the test suite of the project:
 - 6 (28.57%) lines were added

At the bottom of the comment, there is a blue link that says "Click here for a more detailed report for this pull request."

Figure 10: Operias comment on new PR