Test Accompanying Calculation of Test Gaps for Java Applications

Florian Dreier[1] Dr. Elmar Juergens[2] and Dr. Andreas Göb[3]

Abstract: To avoid bugs in software products a lot of effort is put into testing. Especially code that has been modified, is likely to induce new bugs and hence should be tested carefully. By analyzing coverage reports, test gap analysis assists in finding untested changes. For manual tests these reports of all tests are often collected in nightly jobs. The analysis then returns aggregated results for them. To improve efficiency for testers we developed a prototypic tool, which allows testers to get the fine-grained results of their own tests immediately after they tested a feature. The implementation is based on JaCoCo as coverage tool. The evaluation showed we can find relevant test gaps with high precision and recall.

Keywords: Test gap analysis, JaCoCo, Coverage, Real-time

1 Introduction

Successful software is typically developed once and maintained for a long time. To make sure any changes to the system do not break existing functionality it is crucial to carefully test those changes. To assist test managers in discovering changes that have not been tested so far, test gap analysis (TGA) has been introduced by CQSE GmbH. Their software *Teamscale*[4] analyzes source code to find code quality defects. Therefore Teamscale has knowledge about which parts of the code have been changed e.g. to implement a feature. Additionally, the analysis needs to detect what code gets tested. To do so, profilers run in the background, while the tests are executed. On shutdown of the system a report is generated. By combining the coverage reports with the knowledge about changes, untested changes (test gaps) can be found, before they produce bugs in production environments [Ed13].

Although testing is often automated, a lot of companies still rely on manual testing processes [JP16]. In those setups coverage reports of all testers are typically collected on a nightly basis and presented on an aggregated dashboard the next morning. But extracting actionable information from those dashboards is time-consuming and inefficient. To improve efficiency for testers and speed up the feedback cycle, we developed a prototypic tool, which allows testers to get the results of their own tests immediately after a feature has been tested. To do so we collect and analyze the coverage while the system under test is still running. The results are then presented on a per-feature basis.

[1] Technical University Munich, Department of Informatics, dreier@in.tum.de
[2] Technical University Munich, Department of Informatics, juergens@in.tum.de
[3] CQSE GmbH, goeb@cqse.eu
[4] https://www.cqse.eu/de/produkte/teamscale/landing/

2 Related Work

In the following the terms *feature*, *change request* and *issue* are used interchangeably and describe an entry in the issue tracker.

CRC Metric To get a better overview over which feature has low coverage and hence should be tested with priority, Jakob Rott proposed a metric called Change Request Coverage [Ro17]. The metric describes the percentage of the changed methods belonging to a change request that have been executed. Another insight was that there are some categories of methods that are typically not interesting for testers like getters, setters, `toString` methods etc., which can be excluded automatically.

We implemented the automatic exclusion to evaluate if we thereby miss any relevant methods. Based on that we also implemented the metric similar to CRC metric, but with a modified formula to consider fine-grained coverage data.

Test gap analysis Our approach is based on the implementation of test gap analysis in Teamscale [Ju11][JP16]. The analysis is responsible for mapping the changed portions of code to the coverage data provided by the profilers.

3 Approach

For our prototype, we used JaCoCo[5] to collect coverage data. Hence the JaCoCo agent was attached to the system under test and instructed to open a TCP interface for communication with the outside world.

We implemented a CoverageRecorder tool that connects to this interface, handle the report generation and upload the coverage to Teamscale.

In turn we extended Teamscale to show the collected coverage data on a per-feature basis.

3.1 JaCoCo agent

The JaCoCo agent was attached to the system under test as Java agent by appending the following options to the program's VM arguments:

```
-javaagent:jacocoagent.jar=output=tcpserver
```

The `tcpserver` option opens a TCP interface, which allows clients to retrieve coverage data without the need to stop the running application. The interface offers a command called `dump`, which returns everything covered until then and optionally discards the coverage afterwards (`reset`). Coverage data is recorded and transmitted in a non-human-readable format used internally by JaCoCo.

3.2 CoverageRecorder

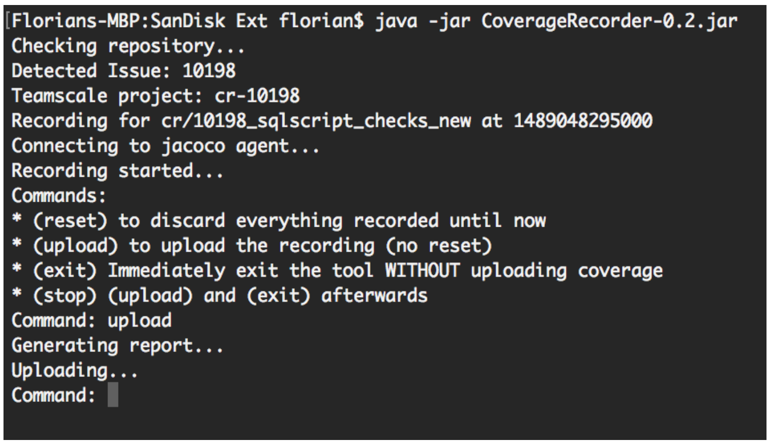
We implemented the CoverageRecorder as command line tool in Java. It must be invoked manually after the system under test has been started and connects to the TCP interface of the JaCoCo agent. Once started it connects with the JaCoCo agent and provides the tester with commands to `reset` and `upload` coverage to Teamscale. This is shown in figure 1.

The `reset` command makes it possible to explicitly discard code coverage that is executed during the startup of the system and hence allows to view coverage of a single test. `upload` in turn fetches the coverage data without resetting it, generates a report and sends it to Teamscale to analyze it. This sequence of events is also illustrated in figure 2.

The tool must be configured via a configuration file that contains the following information:

- The path to the working copy of the system under test. The given directory is scanned for class files, which are needed for the report generation. Furthermore, it expects a certain feature branch to be checked out. The issue ID is automatically extracted from the branch name with a regular expression. This means there is no additional configuration effort needed after the initial setup.

[5] http://www.eclemma.org/jacoco/



```
[Florians-MBP:SanDisk Ext florian$ java -jar CoverageRecorder-0.2.jar
Checking repository...
Detected Issue: 10198
Teamscale project: cr-10198
Recording for cr/10198_sqlscript_checks_new at 1489048295000
Connecting to jacoco agent...
Recording started...
Commands:
* (reset) to discard everything recorded until now
* (upload) to upload the recording (no reset)
* (exit) Immediately exit the tool WITHOUT uploading coverage
* (stop) (upload) and (exit) afterwards
Command: upload
Generating report...
Uploading...
Command: 
```

Fig. 1: CoverageRecoder interface

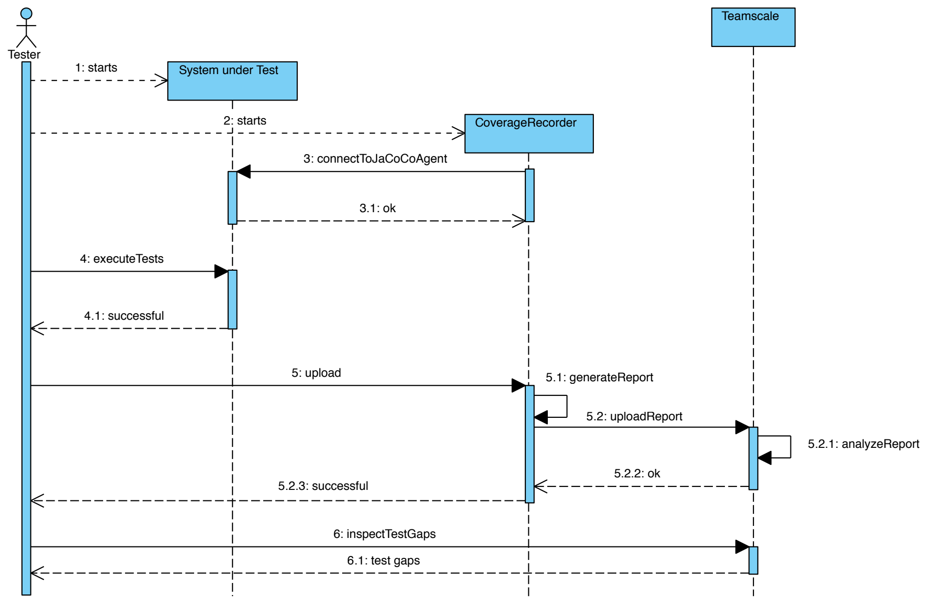


Fig. 2: Sequence diagram

- The tester's ID, which enables the system to differentiate between distinct testers. This allows multiple distinct testers to upload coverage for the same feature independently.
- Authentication information for Teamscale including the teamscale server URL and an access token for a user entitled to upload coverage.

3.3 Teamscale UI enhancement

A common way to visualize coverage data and test gaps of a software system is a *treemap*. An example is shown in figure 3. The treemap shows a whole software system. Each of the small rectangles represents a method. The size of the method corresponds with the size of the rectangle and methods within the same class are rendered close to each other in a bigger logical rectangle. Methods drawn in orange color have been changed, red methods have been added since a defined baseline. These are overlaid by green for methods that have been executed during testing. This is a very convenient solution to get an overview of what has been changed since e.g. the last release.

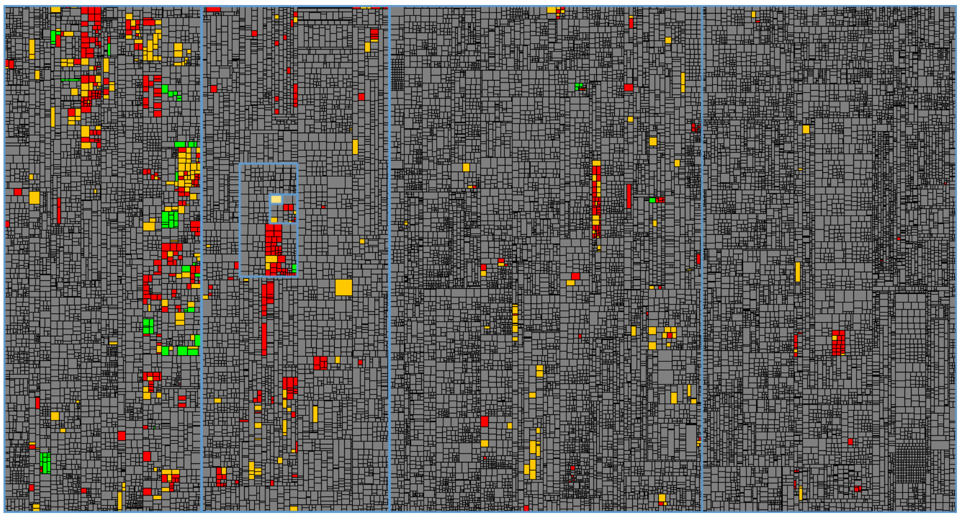
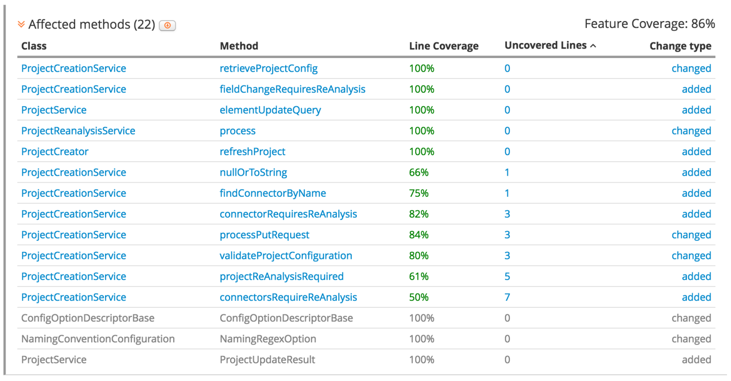


Fig. 3: Treemap

While this is a very convenient solution to get an overview, it is not suitable to show fine-grained coverage data. Since there already is an *issue* perspective in Teamscale, which shows details about modified files and commits made for a specific issue together with information from the issue itself, it was a straight forward place to integrate information about methods that have been changed and their corresponding coverage on a per-line basis. What we came up with is shown in figure 4.

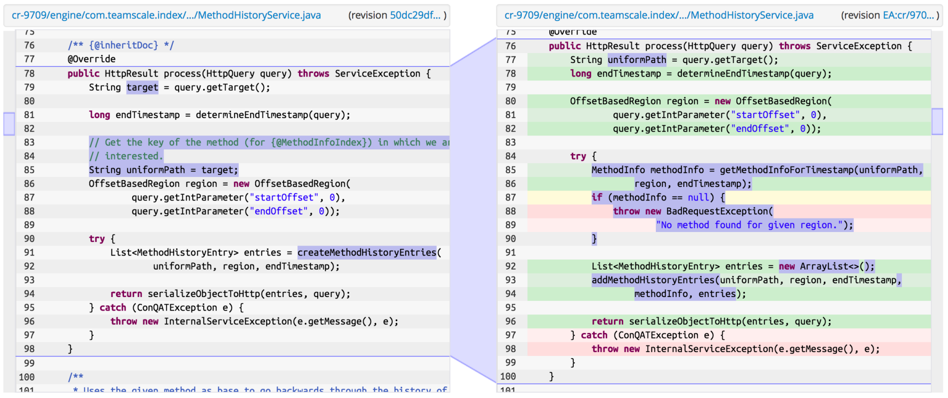
Besides the class and method name, the current line coverage, the absolute number of lines that are still uncovered and the detected change type are shown. All columns are sortable and rows are linked to diff views that show the method before work on the feature has been started in comparison to the most recent code (see figure 5). Methods classified as non-relevant are displayed in gray. The feature coverage is shown at the top of the table.



Affected methods (22) — Feature Coverage: 86%

Class	Method	Line Coverage	Uncovered Lines	Change type
ProjectCreationService	retrieveProjectConfig	100%	0	changed
ProjectCreationService	fieldChangeRequiresReAnalysis	100%	0	added
ProjectService	elementUpdateQuery	100%	0	added
ProjectReanalysisService	process	100%	0	changed
ProjectCreator	refreshProject	100%	0	added
ProjectCreationService	nullOrToString	66%	1	added
ProjectCreationService	findConnectorByName	75%	1	added
ProjectCreationService	connectorRequiresReAnalysis	82%	3	added
ProjectCreationService	processPutRequest	84%	3	changed
ProjectCreationService	validateProjectConfiguration	80%	3	changed
ProjectCreationService	projectReAnalysisRequired	61%	5	added
ProjectCreationService	connectorsRequireReAnalysis	50%	7	added
ConfigOptionDescriptorBase	ConfigOptionDescriptorBase	100%	0	changed
NamingConventionConfiguration	NamingRegexOption	100%	0	changed
ProjectService	ProjectUpdateResult	100%	0	added

Fig. 4: User interface



cr-9709/engine/com.teamscale.index/.../MethodHistoryService.java (revision 50dc29df...)

```java
    /** {@inheritDoc} */
    @Override
    public HttpResult process(HttpQuery query) throws ServiceException {
        String target = query.getTarget();

        long endTimestamp = determineEndTimestamp(query);

        // Get the key of the method (for {@MethodInfoIndex}) in which we ar
        // interested.
        String uniformPath = target;
        OffsetBasedRegion region = new OffsetBasedRegion(
                query.getIntParameter("startOffset", 0),
                query.getIntParameter("endOffset", 0));

        try {
            List<MethodHistoryEntry> entries = createMethodHistoryEntries(
                    uniformPath, region, endTimestamp);

            return serializeObjectToHttp(entries, query);
        } catch (ConQATException e) {
            throw new InternalServiceException(e.getMessage(), e);
        }
    }

    /**
     * Uses the given method as base to go backwards through the history of
```

cr-9709/engine/com.teamscale.index/.../MethodHistoryService.java (revision EA:cr/970...)

```java
    @Override
    public HttpResult process(HttpQuery query) throws ServiceException {
        String uniformPath = query.getTarget();
        long endTimestamp = determineEndTimestamp(query);

        OffsetBasedRegion region = new OffsetBasedRegion(
                query.getIntParameter("startOffset", 0),
                query.getIntParameter("endOffset", 0));

        try {
            MethodInfo methodInfo = getMethodInfoForTimestamp(uniformPath,
                    region, endTimestamp);
            if (methodInfo == null) {
                throw new BadRequestException(
                        "No method found for given region.");
            }

            List<MethodHistoryEntry> entries = new ArrayList<>();
            addMethodHistoryEntries(uniformPath, region, endTimestamp,
                    methodInfo, entries);

            return serializeObjectToHttp(entries, query);
        } catch (ConQATException e) {
            throw new InternalServiceException(e.getMessage(), e);
        }
    }
```

Fig. 5: Diff view

Feature coverage The calculation of the feature coverage is illustrated based on the example methods listed in table 1. M_r is the set of relevant methods.

As proposed by Jakob Rott in his bachelor's thesis, the CRC is a means to express how much progress has already been made in testing a change request. The CRC is calculated as the number of methods that have been executed divided by the total number of methods that have been changed by the change request.

$$CRC = \frac{\sum_{m \in M_r} \left\lceil \frac{coveredLines(m)}{totalLines(m)} \right\rceil}{|M_r|} = \frac{4}{4} = 100\%$$

A CRC value of 100% for this sample set would indicate for the tester that the feature has been tested completely. Even though there is still a decent amount of code untested. To better reflect the actual progress of testing the change request we adjusted the formula and named it feature coverage. The feature coverage in contrast to CRC is calculated on a

Method	LOC	Covered lines	Uncovered lines	Line coverage
processRequest	49	20	29	$\frac{20}{49} = 41\%$
appendExtraInfo	5	5	0	$\frac{5}{5} = 100\%$
findByName	4	4	0	$\frac{4}{4} = 100\%$
nullOrToString	3	2	1	$\frac{2}{3} = 66\%$

Tab. 1: Example methods with line coverage

per-line basis as the line coverage of all relevant methods weighted by their total number of lines.

$$Feature\ coverage = \frac{\sum_{m \in M_r} coveredLines(m)}{\sum_{m \in M_r} totalLines(m)} = \frac{31}{61} = 51\%$$

Here the tester can see that 49% of the lines are still uncovered and therefore more testing is required.

Exclusion of non-relevant methods As Jakob Rott discovered in his bachelor thesis, there are three kinds of methods that are typically of no interest for testers [Ro17]. These are getters, methods that are ”too trivial to test”, `toString` and `hashCode` methods. To implement this filter, we generalized those to the following rules:

- Method only contains one **single return statement** and consists of less than 10 tokens
- Method only contains a **sequence of simple assignments** and super constructor calls. Simple assignments are defined as assignment statements with less than 10 tokens out of {`THIS`, `DOT`, `EQ`, `NULL`, `LITERAL`, `IDENTIFIER`, `SEMICOLON`}.

Methods matching one of these rules are displayed at the bottom of the table in grey color, and are considered as non-relevant and not included in the feature coverage calculation.

3.4 Current limitations

Since only JaCoCo has been considered as coverage framework, the approach is currently limited JVM languages even though we only had a look at Java. Nevertheless, the approach can be applied to other languages, but a shutdown of the system under test may be necessary after each test to retrieve the coverage data.

Our approach currently relies on a strict naming scheme for the commit messages. All commits belonging to a feature must contain the ticket ID from the issue tracker. The server side implementation of the enhanced Teamscale UI also expects the project to follow a feature-branch-based development strategy to be able to show a diff of the isolated changes for one feature. Furthermore, since merge commits do typically not follow the naming scheme and contain a change set that cannot easily be assigned to a specific ticket ID, they are excluded from the analysis.

4 Evaluation

4.1 Research questions

RQ1: Do we find the relevant test gaps? We want to find out if the test gaps we find are relevant, because finding relevant test gaps means potentially spotting bugs before they get released. We defined that a method is relevant if the tester did classify the missed lines as test-worthy and therefore performed another test to get it covered.

RQ2: Do we find bugs? Since the main reason for testing is to find bugs the aim was to analyze if this is possible with the described technique.

RQ3: How useful is the implemented approach? Since test gaps are currently always visualized as treemaps in Teamscale, we wanted to know if the implemented table format is a better way to show test gaps for a feature in a finer grained fashion.

4.2 Study objects

The evaluation was done at CQSE GmbH on features that were developed for an upcoming release of Teamscale itself. It was the easiest way to get in contact with developers and test the prototype in a real software engineering context to get qualitative feedback. Due to the lack of a separate testing role, we integrated the study into the usual peer review process, so that the reviewer ran a manual test of the feature before doing the actual code review. Teamscale is designed as a client-server application: the front-end is written in JavaScript and consists of about 90k lines of code, the backend is implemented in Java and comprises about 650k lines of code. As the coverage analysis was only implemented for Java, features were selected for the study that mainly contained Java code changes. The selected features contained an average of 24 changed methods with absolute numbers ranging from 7 to 43.

4.3 Study design

RQ1 To answer RQ1 the reviewers were asked for each test gap that was found during the manual test if they consider the missed lines as test-worthy. A method was considered a test gap if it did yield less than 100% coverage. Automatically classified non-relevant methods have also been inspected by the reviewer.

RQ2 For RQ2 we noted the bugs found by the reviewers during the tests. Questions that were asked:

- *"Did you find any bug during the review?"*
- *"If not, do you think you could find bugs using this setup?"*

RQ3 RQ3 was qualitatively evaluated via interview questions.

- *"Did you get an overview over the feature's functionality and scope with the tested user interface?"*
- *"How much time did it take you to use the tool in addition to testing the feature?"*

We only needed qualitative feedback whether it's worth to further invest in this technique. Hence we stopped the evaluation after four interviews, since it already became clear that all participants agreed in the most central questions.

4.4 Results

RQ1: Do we find the relevant test gaps? During the tests, we looked at a total number of 95 methods. After a first manual test 38 of those methods had less than 100%-line-coverage whereat 8 of them were classified as not relevant due to the rules listed in the previous chapter. As test gaps we considered the remaining 30 non-trivial methods with less than 100% coverage. 23 of those 30 test gaps have been classified by the reviewers as test-worthy. This yields a precision of $\frac{23}{30} = 76.6\%$. The 7 false-positives either fall in the category "Too trivial to test" or "Too difficult to test" and included edge cases, tests for null, error handling and error propagation etc. None of the 8 automatically filtered methods were considered as relevant by the reviewers resulting in a recall of 100%.

		Suggested		
		Relevant	Not relevant	Σ
Actual	Relevant	23	0	23
	Not relevant	7	8	15
	Σ	30	8	38

Tab. 2: test gap confusion matrix

4 out of the 7 non-relevant methods could have been excluded automatically by introducing an additional rule, which ignores uncovered lines if they start with the keyword `throw`.

RQ2: Do we find bugs? The participants did not find any bug during the reviews. On the one hand methods, which were not fully covered and were therefore marked as test gaps, obviously received special attention. All reviewers were positive about better finding bugs using this technique, because they were forced to consider the code and think about why a certain set of lines has not been covered. On the other hand, code that was covered completely during the first run did not receive any additional attention and therefore the likelihood of finding bugs in there was rated as low.

RQ3: How useful is the implemented UI? Since the selected features contained an average of 24 changed methods, the tables were small enough to fit on the screen without scrolling. All reviewers agreed that the table was a good means to get an overview over the scope of reviewed feature as well as the amount of functionality they have missed during their first test.

All four participants agreed, that using the CoverageRecorder and the Teamscale UI did not take them noticeably more time, but one mentioned that he would not have tested that much without it and therefore it took him longer. But testing more thoroughly should also be considered as a positive outcome.

Other insights The strategies to find out about what functionality has been implemented, before starting to test those, varied:

- One participant used the enhanced Teamscale UI to navigate to the changed parts of the code in the web browser.
- One participant used the enhanced Teamscale UI to find changed or added methods and looked them up in the IDE afterwards to read the code.
- One participant tried to find the new functionality in the software based on the title of the reviewed feature by trial and error.
- One participant read the description in the issue tracker to find out what to test.

Besides that we made a few other interesting observations:

- In three of four reviews those manual tests turned out to be especially useful to find unreachable code. In those cases a static analysis would not have detected the code as unused.
- Three participants pointed out that it would also be useful for them to show the coverage results of automated tests in the issue tracker.
- Two participants suggested an additional treemap at the top of the table, which does not show the whole system, but only methods changed by the feature's implementation.
- All participants noticeably enjoyed exploring the reviewed feature, intuitively tried to get everything covered and stated that this is a ”Pretty cool feature”.

4.5 Discussion

As the results indicate it can be useful to apply manual testing on a per-feature basis during development to get earlier feedback about test gaps, which can potentially contain bugs. Additionally, the approach showed its usefulness in detecting unused code. In those cases, quick feedback can be especially important, because once the code is part of the system it is more expensive to remove it. This is also commonly known in software engineering as Lava-Flow antipattern[6].

5 Threats to Validity

This evaluation only had a very limited set of study participants and therefore the results might not be generalizable.

Another threat is that we only considered the language Java, the software Teamscale and their developers as study objects. Hence, we didn't have dedicated testers to do the tests, but other developers. So, under other circumstances the results may have looked different.

Furthermore we only implemented one UI that seemed to fit for the use case, but we did not do A/B testing with different approaches, that could have led to other results.

6 Conclusion and Future Work

Our prototype showed that there is a use case for real-time feedback for testing individual features. But there is still some work left to further develop the prototype, implement improvements that have been suggested by the study participants like combining the benefits of treemaps and the table by showing both below each other. And further studies will be necessary to validate the results in different technological or organizational environments.

Literatur

[Ed13] Eder, Sebastian; Hauptmann, Benedikt; Junker, Maximilian; Juergens, Elmar; Vaas, Rudolf; Prommer, Karl Heinz: Did we test our changes? Assessing alignment between tests and development in practice. 2013 8th International Workshop on Automation of Software Test, AST 2013 - Proceedings, S. 107–110, 2013.

[JP16] Juergens, Elmar; Pagano, Dennis: Did We Test the Right Thing? Experiences with Test Gap Analysis in Practice. Whitepaper, CQSE GmbH, 2016.

[Ju11] Juergens, Elmar; Hummel, Benjamin; Deissenboeck, Florian; Feilkas, Martin; Schlögel, Christian; Wübbeke, Andreas: Regression test selection of manual system tests in practice. Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, S. 309–312, 2011.

[6] http://antipatterns.com/lavaflow.htm

[Ro17] Rott, Jakob; Niedermayr, Rainer; Juergens, Elmar; Pagano, Dennis: Ticket Coverage: Putting Test Coverage into Context. In: Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM'17). 2017.