

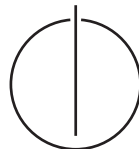
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Obtaining Coverage per Test Case

Florian Dreier





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Obtaining Coverage per Test Case

Testfallbasierte Erhebung von Coverage

Author: Florian Dreier
Supervisor: Prof. Dr. Dr. h.c. Manfred Broy
Advisor: Dr. Elmar Juergens (TUM) &
Dr. Andreas Göb (CQSE)
Submission Date: November 15, 2017



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, November 15, 2017

Florian Dreier

Acknowledgments

First of all, I would like to say thank you to Dr. Elmar Juergens who once again managed to propose a really interesting and challenging topic. I also want to thank him for his support on getting the co-operation with imbus and Tricentis started. In addition, I would like to say a sincere thank you to Dr. Andreas Göb for his constant support and dedication during the implementation and writing of the actual thesis. Furthermore, a big thank you to René Rohner (from imbus AG) and Roland Zuderstorfer (from Tricentis GmbH) for their time and support on getting started with the plugin development on their platforms.

Finally, a big thank you to Lavinia Eifler and Verena Dreier who carried out the most tedious work of correcting the language of this thesis and of proof-reading.

Abstract

To avoid bugs in software products a lot of effort is put into testing. Especially regression testing is important for software maintained over a longer period of time to make sure already existing functionality is not broken by newly introduced features and adjustments. However, regression testing often gets unreasonably expensive as test suite sizes are growing over time.

This thesis presents an algorithm for selecting and prioritizing test cases for Java applications based on a set of changes that have been applied to the system and per-test coverage collected on an older version of the system. The selection algorithm was integrated into the software Teamscale and plugins for various build systems have been implemented to prove that the technique can be easily integrated with existing test systems.

The technique was evaluated on twelve software systems of various sizes. I examined how stable coverage of individual tests is in general and how strongly changes have an effect on those coverage. I found that especially asynchronous actions invoked by a test case cause flickering coverage. To evaluate the effectiveness of the reduced test set with regard to revealing faults mutation analysis techniques have been applied to the study objects. I found that 99.2% of failures that can be detected by the whole test suite are also found by the reduced test suite. The results also show that ordering the tests makes more than 90% of the test fail within the first 10% of the total execution time of the selected tests.

Contents

1. Introduction	1
2. Terms and Definitions	3
3. Related Work	4
3.1. Research	4
3.1.1. Selective regression testing	4
3.1.2. Test case prioritization	5
3.2. Tools	6
3.3. Technical Fundamentals	7
3.3.1. Teamscale	7
3.3.2. JaCoCo	7
4. Approach	9
4.1. Overview	9
4.2. Coverage collection per test	10
4.2.1. Listen for test execution	10
4.2.2. JaCoCo	12
4.2.3. Report generation	12
4.2.4. Storage	13
4.3. Test suggestions	14
4.3.1. Regression test selection	14
4.3.2. Test ordering	17
4.4. Limitations	18
5. Evaluation	20
5.1. Research questions	20
5.2. Study Objects	21
5.3. Study Design	22
5.4. Procedure	23
5.5. Results	26
5.6. Discussion	31
5.7. Threats to Validity	32

Contents

6. Future Work	33
7. Conclusion	34
A. Study objects summary	35
List of Figures	36
List of Listings	37
List of Tables	38
Bibliography	39

1. Introduction

When maintaining an already functional software, priority number one is to not break existing functionality. Therefore, regression testing is an indispensable step in the software lifecycle. Regression test suites typically grow over the whole lifetime of a project up to an extent where running the complete test suites for every change gets expensive. Running only a subset of the regression tests, which can actually reveal newly introduced bugs, is hence reasonable (*selective regression testing*). Additionally ordering the tests in a way that bugs are found early during the test execution is desirable (*test case prioritization*).

Selecting and ordering the relevant test cases requires knowledge of all parts of the software that have been changed since the last regression test as well as which tests execute those changed portions of the code. For bigger systems the required knowledge is distributed amongst all developers of the system or has even been lost over time, which makes it a hard problem to select the tests manually.

Tool support is therefore critical for effective regression test selection. The goal of this thesis was to implement a tool, which assists in selecting the relevant tests for a given set of changes. The selection and ordering of tests has been integrated into Teamscale, a software quality analysis tool. This allows saving execution data from manual system and integration tests, automated unit tests and automated system tests in one central place.

The tool had to be compatible with Java and use JaCoCo for coverage collection. A proof of concept implementation of plugins for various build and test tools was also part of the thesis.

Selected tests are also prioritized so that tests that have a high probability of revealing faults are executed first. So if a more risky test selection should be done, the list of tests can be cut off at any point, which allows for a user specified budget.

The evaluation has shown that selecting only the relevant tests can massively decrease regression test time and makes tests fail faster by ordering them based on their additional coverage per execution time.

In the following chapter related tools and research on the topic are presented as well as the technical fundamentals my approach builds on.

Chapter 4 describes the approach with all its implementation details.

Chapter 5 presents a case study on 12 software systems investigating their basic

1. *Introduction*

coverage stability and the approach's ability to find faults.

2. Terms and Definitions

Retest-all Rerunning all regression tests without selecting a subset is called the *retest-all* approach. It retrieves the maximum amount of information that can be gained from the existing test suite.

Safe A selective regression testing technique is called *safe* if it guarantees that all information, which can be gained from retest-all, can also be gathered from the given subset of tests [10].

Test case prioritization *Test case prioritization* techniques schedule test cases for execution in an order that attempts to increase their effectiveness at meeting some performance goal. Various goals are possible; one involves rate of fault detection, a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during testing can provide faster feedback on the system under test and let software engineers begin correcting faults earlier than might otherwise be possible [11].

Error / Fault / Bug An *error* is a deviation from actual and expected value as a result of a human mistake. A *fault* is an incorrect step or computation performed by a machine, which is often the result of an error. A *bug* is a noticeable deviation from the expected behavior of the program, which is typically found by a tester.

Mutation testing Mutation testing is a technique to evaluate the quality of existing software tests. By running the existing tests against *mutants* - slightly modified versions of the original software - the tests' ability to detect the modified behavior can be measured. Those mutations that the test suite rejects are called *killed* mutants.

3. Related Work

3.1. Research

3.1.1. Selective regression testing

Research on the topic of selective regression testing can be categorized into four different strategies [2]:

- **DejaVu based** DejaVu (or DejaVOO) was first introduced by Rothermel and Harold [9] and was later adapted to Java [4]. Those techniques use fine-grained coverage information to infer which tests will execute code that has been modified. Most of them build a control flow graph for every function to select a minimal number tests which actually execute changed parts of the code. There are also variations of this technique that are specifically suited for database-based applications [14]. The fine-grained coverage makes the approaches computationally expensive, adds overhead to the test execution and makes it very language dependant.

My approach also belongs to this category, but tries to reuse the widely-used coverage profiler JaCoCo instead of implementing a new one. Furthermore, coverage is analyzed on method level to ensure compatibility with other languages and reduce computational overhead. Last but not least my approach integrates with Teamscale and various coverage collection environments. Teamscale hence provides a central coverage store and allows for wider collaboration instead of collecting coverage on a single machine. Teamscale's refactoring detection on method level also allows to further reduce the set of considered method changes [1].

- **Firewall based** Firewall based approaches as first introduced by Leung and White are focused on integration tests [6]. They build a dependency graph amongst the modules of a system. All modules, which have been modified or have a direct dependency to the modified modules according to some defined rules are considered inside the firewall. Afterwards all integration tests are selected for re-execution that execute code inside the firewall. This approach is safe, but also selects a lot of tests, which do not execute any changed code. In contrast to my

approach firewall based still relies on manually selecting the integration tests for the modules inside the firewall.

- **Dependency based** Dependency based systems like presented by Wu and Chen build a static function call graph to describe the dependencies of the individual software components [15]. All tests executing functions that have changed or are transitively affected by any changes are selected for re-execution. This technique is not dependant on any previously collected coverage, but are very specific for a certain language or even a specific framework when dependency injection is used in the system under test.
- **Specification based** For example Mao and Lu have proposed an XML based system to store changed methods and preconditions per component [7]. This technique was mainly intended to better understand what needs to be tested for third-party components. My approach in contrast does not consider changes made to third-party components.

3.1.2. Test case prioritization

Research on test case prioritization has been summarized by Elbaum and Rothermel in "Prioritizing Test Cases for Regression Testing" [11]. Basically test cases are ordered with respect to a certain performance goal with the aim to perform at least better than executing the tests in random order. The performance goals they had a look at deviated in the following dimensions:

- **Total vs. Additional coverage** The test cases are ordered based on the coverage they cause in the system under test. The additional coverage only looks at the coverage that can be gained in addition to the already selected tests.
- **Coverage granularity** The ordering was tested when considering coverage on statement and function level.
- **Fault-exposing potential (FEP)** Tests are (not) ranked based on their ability to detect faults. The FEP was calculated based on a mutation test executed beforehand.
- **Total fault index (FI)** Tests that have failed more often in the history of the system are prioritized.

They found that statement level coverage with the combination of additional coverage and including the FEP resulted in the best fault detection rates. In my approach I used line based coverage, which can be considered the same as statement level coverage in

combination with additional coverage weighted with the expected test execution time, to gain as much coverage in short time as possible.

An alternative approach without usage of static or dynamic analysis required was proposed by Saha et al.. They reduced the problem to a generic information retrieval problem. Tests are prioritized here based on the words used in the tests and the changed source code [12].

3.2. Tools

In the following I present three tools, which support doing selective regression testing and are already used in production systems.

Test impact analyzer The most prominent example is Microsoft's Test impact analysis (TIA), which is integrated into Visual Studio. It has been developed for several years in Microsoft Research [5]. It uses a DejaVu based approach with coverage on file level. The tool does not support test case ordering and only works for C# [3].

VectorCAST/QA Vector software has developed another tool called VectorCAST/QA, which supports their *Change-Based Testing* [13]. There is no further information available about what approach it uses and it works for C, C++ and Ada.

NCrunch Another tool for the C# world is NCrunch¹, which executes tests during the actual development and also intelligently selects tests for the currently modified code location.

¹<http://www.ncrunch.net/>

3.3. Technical Fundamentals

In the following the framework, which was used to implement this thesis' approach, is presented.

3.3.1. Teamscale

Teamscale is a software quality analysis tool that helps to monitor software quality goals and provides real-time feedback. It directly connects with the source code repository and analyses every commit. In addition, other external data such as code coverage can be uploaded and analyzed.

Teamscale features a so called *Test Gap analysis*. The analysis is able to detect code fragments that have been modified since the last version, but have not been tested yet. These modified code fragments are referred to as *Test Gaps*. When a method is considered as modified, refactorings have already been excluded.

My approach uses the data gathered by the Test Gap analysis. Especially information about which methods have been added, changed and deleted. I also used a modified version of the code coverage upload. The modified version is able to read and store coverage on a per-test level.

3.3.2. JaCoCo

JaCoCo is an open source code coverage library for the JVM. It allows to measure coverage on the fly and has been used for this thesis. To measure code coverage the *jacoco-agent* has to be specified as `javaagent` when running the system under test. This allows JaCoCo to hook into the class loading process. Whenever a new class is loaded, JaCoCo instruments it by inserting so called probes into the bytecode. A probe is a small portion of bytecode that has no side-effects on the system, but records that it has been executed into a boolean array `$jacocoData`, which has also been statically added to the class. To reduce runtime overhead the probes are only inserted at strategic places, which are determined by analyzing the control flow of the bytecode.

JaCoCo supports multiple ways to access the recorded coverage. The coverage can either be written to a file, accessed via TCP or via the Java Management Extension protocol (JMX). The output format is binary and basically is a list of tuples. Each tuple holds a class ID and the contents of `$jacocoData`. The class ID is the CRC64 checksum of the raw bytecode of this class. Not using the class name as identifier allows to keep the file size small and to use different versions of the same class side by side. The coverage at a specific point in time, which is written to the output stream, is called a *dump*.

Additionally, the JaCoCo agent allows to set a so called session ID, which can be used to identify a specific coverage dump. The session ID is written to the specified output stream as well. The functionality to set the session ID is exposed via the IAgent interface shown in Listing 3.1.

Listing 3.1: IAgent interface

```
public interface IAgent {  
  
    /** Returns version of JaCoCo. */  
        String getVersion();  
  
    /** Returns current a session identifier. */  
        String getSessionId();  
  
    /** Sets a session identifier. */  
        void setSessionId(String id);  
  
    /** Resets all coverage information. */  
        void reset();  
  
    /** Returns current execution data. */  
        byte[] getExecutionData(boolean reset);  
  
    /** Triggers a dump of the current execution data through the configured output. */  
        void dump(boolean reset) throws IOException;  
  
}
```

The agent can either be accessed via `RT.getAgent()` from within the same JVM as the JaCoCo agent is being executed or via JMX, which also exposes an IAgent object.

4. Approach

4.1. Overview

This approach implements a regression test selection and prioritization based on coverage that has been collected on a per-test-base for an older version of the system under test. The coverage collection was implemented based on JaCoCo for Java applications, but is designed to work across a wide range of languages and tools. The collected coverage gets stored in the tool Teamscale, which also is responsible for selecting and prioritizing the tests afterwards.

To make it easily useable in practice plugins for the following build and test tools have been implemented:

- Maven
- Gradle
- TestBench
- Tosca

The plugins either automatically collect coverage on per-test-basis or provide means to set it up. Furthermore, the plugins take care of generating the coverage report and uploading it to Teamscale. They also provide basic support for retrieving the selected regression tests from teamscale for a given set of changes.

The system was designed to collect coverage during the regularly scheduled nightly, weekly or monthly builds and use this information to be able to run only a relevant subset of tests after every single commit, in order to massively reduce feedback cycles.

4.2. Coverage collection per test

4.2.1. Listen for test execution

To record coverage per test case I needed to know when the execution of a single test case starts and ends. This obviously is highly dependent on the execution environment. For this thesis I considered tests written in *JUnit 4*¹, *JUnit 5*² and *TestNG*³ executed through *Maven*⁴ or *Gradle*⁵. Additionally, I implemented a proof of concept for *Tosca*⁶ and *TestBench*⁷, which are used in the industry for doing automated UI testing. In the following I will show how this was implemented for the considered environments.

JUnit 4 JUnit 4 natively provides a way to listen for test case execution through the `RunListener` interface. Maven's `surefire` plugin, which is used to run tests in Maven, supports setting this listener. In Gradle it's not possible to set the listener, so a custom JUnit-runner was used to detect when tests start and finish.

JUnit 5 JUnit 5 supports to listen for execution events as well by implementing `BeforeEachCallback` and `AfterEachCallback`. Classes implementing them can automatically be loaded via Java's `ServiceLoader` mechanism. But those only work for tests written for the JUnit-Jupiter platform, not for JUnit-Vintage.

TestNG TestNG supports specifying a global `ITestListener` - similar to JUnit 4 - but only Maven allows to set it. Since TestNG does not have the concept of a runner like JUnit 4, Gradle based TestNG projects would need to add `@Before` and `@After` annotated methods to all test classes to track test execution.

Tosca At the moment Tosca does not have a mechanism to listen for test executions. The only way to listen for test executions is to add custom test steps called *special execution tasks*. Those can be added to Tosca by adding a Dynamic Link Library (dll), which links against Tosca's C# TBox API. To be able to control the JaCoCo agent, I used the *Java Native Interface* library to bridge the calls from C# to Java as depicted in Figure 4.1.

¹<http://junit.org/junit4/>

²<http://junit.org/junit5/>

³<http://testng.org/>

⁴<https://maven.apache.org/>

⁵<https://gradle.org/>

⁶<https://www.tricentis.com/de/tricentis-tosca-testsuite/>

⁷<https://www.imbus.de/testbench/>

4. Approach

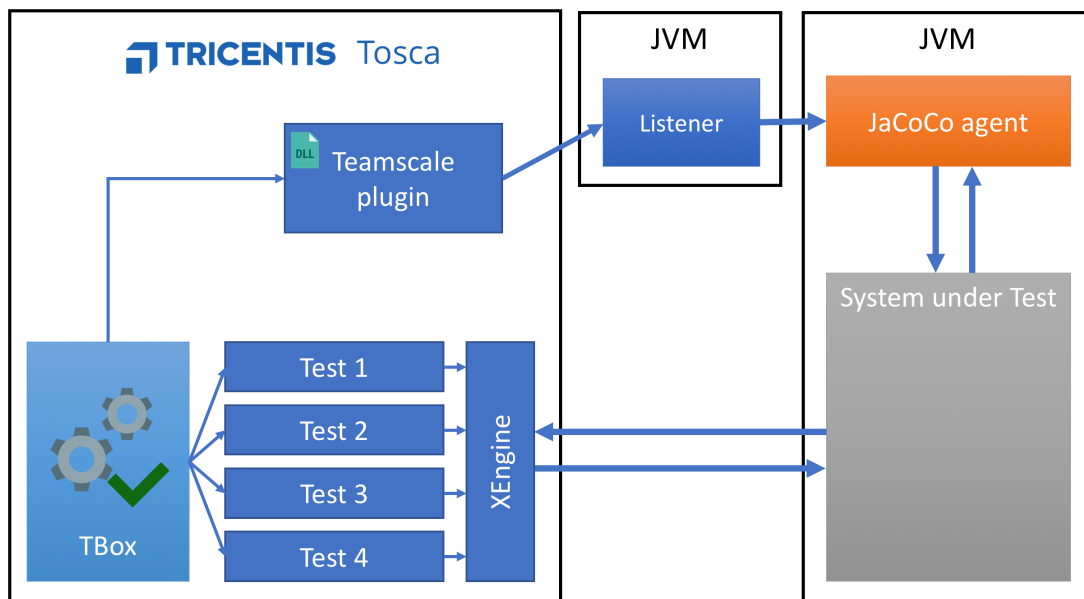


Figure 4.1.: Tosca integration overview

The implemented StartTest and FinishTest special execution tasks could then be inserted as first and last test step respectively as shown in Figure 4.2.

Insurance calculator test set					
Test passenger car insurance					PLANNED
StartTest					
testSetName	Insurance calculator test set	Input	String		
testName	Test passenger car insurance	Input	String		
Vehicle selection					
Fahrzeugauswahl	PKW/Kombi oder Wohnmobil b...	Input	String		
Zu den Fahrzeugdaten	X	Input	String		
Vehicle data					
Personal data					
Product choice					
Results					
FinishTest					
Test truck insurance					PLANNED
Test motorcycle insurance					PLANNED

Figure 4.2.: Tosca integration

TestBench After specifying and planning tests in the TestBench, a "test screenplay" can be exported via the imbus Test Execution Plugin (iTEP). A so called *iTEP-Wrapper*, which implements all test steps available for the domain, needs to be used to execute the actual test steps. To listen for start and end of tests, I have written a generic wrapper for the iTEP library, which is normally used to load the "test screenplay" into the *iTEP-Wrapper*. For tests that are executed manually a wrapper around the iTORX⁸ library would also allow to listen for test executions, but was not implemented in this thesis.

4.2.2. JaCoCo

To save the coverage on a per-test-basis into JaCoCo's binary format, I used JaCoCo's session IDs feature. As described in subsection 3.3.2 JaCoCo allows to set a session ID via its IAgent interface. When coverage gets dumped the session ID is written to the file and used as identifier for the portion of coverage together with the timestamps at which the session ID has been set and the timestamp at which the dump command was executed. I used the session ID to encode the test set name and the test name as shown in the following example: `okhttp3.AddressTest#!#addressToString`.

JaCoCo supports two ways of manipulating the session ID. For Maven and Gradle, where the listener code is executed in the same JVM as the JaCoCo agent, the agent can be directly accessed. In Tosca and TestBench JMX was used to remotely connect to the JaCoCo agent.

4.2.3. Report generation

To generate a report from the binary coverage data, JaCoCo reads the probes from the coverage files and reanalyses the class files to map those probes back to covered lines of code. My first prototypes have shown that reanalysing all classes for every test case creates an enormous performance overhead, so I implemented a probe cache to avoid reloading all class files for every test case. This led to a speed up of 40x for the report generation.

⁸imbus Test assistant for Online and Remote eXecution

To store the report I extended JaCoCo's regular XML report format with a new top-level node `<session class="..." name="..." duration="...">`.

Listing 4.1: Coverage report extract

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE report PUBLIC "-//JACOPO//DTD_Report_1.0//EN" "report.dtd">
<report>
  <session class="okhttp3.AddressTest" name="addressToString" duration="21">
    <package name="okhttp3/internal">
      <sourcefile name="Util.java">
        <line nr="372" mi="0" ci="5" mb="0" cb="0"/>
        <line nr="373" mi="2" ci="3" mb="1" cb="1"/>
        <line nr="376" mi="0" ci="3" mb="1" cb="1"/>
        <line nr="377" mi="2" ci="0" mb="0" cb="0"/>
        <line nr="380" mi="0" ci="2" mb="0" cb="0"/>
        ...
      </sourcefile>
      ...
    </package>
    ...
  </session>
  ...
</report>
```

To reduce the file sizes of several GB per report in bigger projects I used gzip compression. The resulting reports had on average 2-10% of the original size.

4.2.4. Storage

The upload service was extended to be able to read the modified JaCoCo report format and save the coverage accordingly. Teamscale previously used a class `LineCoverageInfo` to store the following sets per file: `fullyCoveredLines`, `partiallyCoveredLines` and `notCoveredLines`. I extended the data structure to additionally hold a `fullyCoveredLinesSessionMap` and a `partiallyCoveredLinesSessionMap`, which map the covered line numbers to a set of session IDs that executed the respective lines.

4.3. Test suggestions

The process of generating test suggestions is divided in two parts. First of all tests, which are likely to execute changed parts of the software are selected. Second, those tests get ordered in a way that tests, which are more likely to reveal bugs in the system, are executed first.

4.3.1. Regression test selection

Regression testing aims to find bugs introduced by changes to the system during the maintenance phase. Typically a large amount of the system stays the same. Therefore rerunning all existing tests (retest-all) is effective to find newly introduced bugs, but is inefficient. To make regression testing more efficient the goal is to run only tests, which execute changed parts of the code and therefore could reveal bugs.

To select a set of tests, I used a DejaVu based approach, which uses coverage information from previous test executions to predict which tests will execute the latest changes. To illustrate coverage and change information I will use the representation in Figure 4.3. The example depicts a version 1 (*baseline*) at which coverage has been recorded and a version 2, which is the current version under test. $t1$ represents a test, which has executed the method $m1$ at version 1. The method $m1$ on the right depicts the same method after some changes have been applied to it, which in turn is symbolized by the pencil symbol.

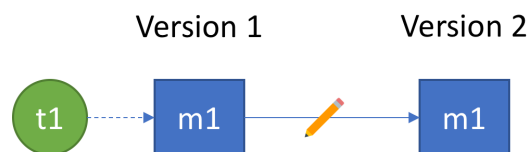


Figure 4.3.: Minimal example

In Figure 4.4 an example system is depicted. The system has had three tests $t1$, $t2$ and $t3$ at version 1. $t1$ executed method $m1$, $t2$ executed $m2$ and $m3$ and $t3$ executed $m3$. For version 2 $m3$ has been inlined at its call site $m2$ and $t3$ has been deleted accordingly. Additionally, $m4$ and $t4$ have been added.

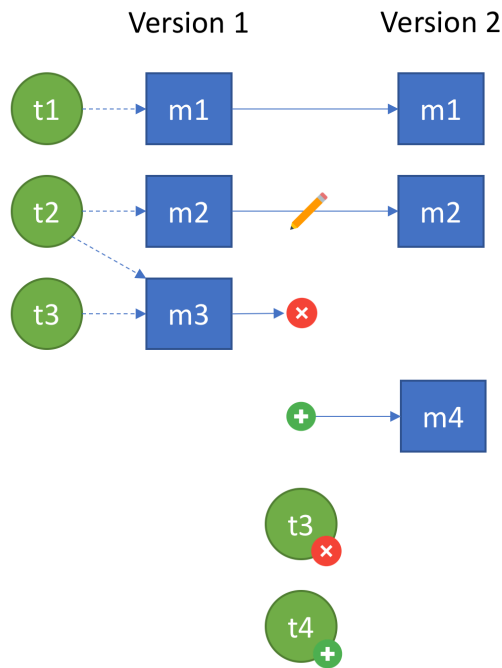


Figure 4.4.: Example system

My approach now selects all tests, which have executed methods that have been modified or deleted from version 1 to 2. In the example $m2$ and $m3$ have been modified and deleted respectively. $t2$ and $t3$ have executed those and therefore potentially execute the code changes. Since $t3$ has been deleted in the meanwhile it must be removed from the test suggestions. But $t4$ has been added and should hence get executed. Modified tests would be suggested as well. $m4$ could either be executed by the already suggested $t4$ or it is called from some other test or method. If the call to $m4$ has been added to some test, the test is modified and suggested for re-execution. If the $m4$ is called from another already existing method, tests executing this method are already suggested due to this modification.

To summarize the above: The implemented selection strategy selects all tests that did execute methods at the baseline, which have been modified or deleted in the meanwhile. Additionally, all tests that have been added or modified are suggested for re-execution. Test renamings and deletions also need to be considered.

4. Approach

The full algorithm is shown in Listing 4.2

Listing 4.2: Regression test selection

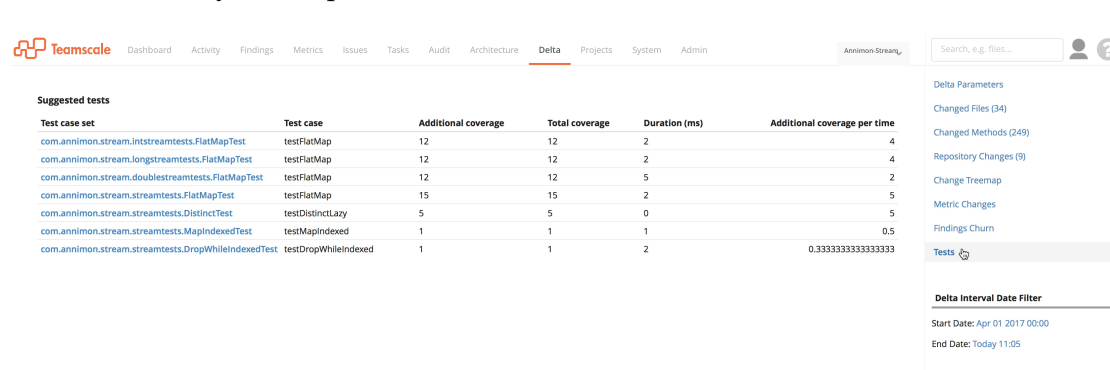
```
methodsToTest = modifiedOrDeletedMethodsBetween(baseline, end)
suggestedTests = getTestsCoveringMethods(methodsToTest)
suggestedTests += addedOrModifiedTestsBetween(baseline, end)
renameIfNameChanged(suggestedTests)
removeDeletedTestMethods(suggestedTests)
```

The suggestions are generated inside a Teamscale service. The service takes as parameters a baseline and an end commit descriptor, which is a tuple of branch name and commit timestamp. The service calculates test suggestions to test all changes that happened between baseline (exclusive) and end (inclusive).

To detect test cases in code I used a heuristic. If the method is annotated with `@Test` or the method name starts with `test` and the class name contains `Test`, the method is considered a test.

The method tracking was already implemented in Teamscale and uses a combination of name based and content based approach to track methods across versions. Teamscale's refactoring detection also filters out modifications, which are only caused by refactorings [1].

The results of this service are returned as a JSON list after the selected tests have been ordered. The results can either be inspected in Teamscale's web interface (Figure 4.5) or could be used by the respective test-tools to execute the tests.



Test case set	Test case	Additional coverage	Total coverage	Duration (ms)	Additional coverage per time
com.annimon.stream.intstreamtests.FlatMapTest	testFlatMap	12	12	2	4
com.annimon.stream.longstreamtests.FlatMapTest	testFlatMap	12	12	2	4
com.annimon.stream.doublestreamtests.FlatMapTest	testFlatMap	12	12	5	2
com.annimon.stream.streamtests.FlatMapTest	testFlatMap	15	15	2	5
com.annimon.stream.streamtests.DistinctTest	testDistinctLazy	5	5	0	5
com.annimon.stream.streamtests.MapIndexedTest	testMapIndexed	1	1	1	0.5
com.annimon.stream.streamtests.DropWhileIndexedTest	testDropWhileIndexed	1	1	2	0.3333333333333333

Figure 4.5.: Test suggestions shown in Teamscale

4.3.2. Test ordering

The already selected subset of test cases should now be ordered in a way that all changes are covered as fast as possible. This problem can be reduced to the *Set Cover Problem*, which is proven to be \mathcal{NP} -complete. Therefore, I used a greedy algorithm to retrieve a fast and still reasonably good ordering. My approach orders tests based on their additional coverage. This is the number of lines, which are part of modified methods that have not been executed by the already ordered tests. To additionally move fast tests, which add most additional coverage, to the front, the greedy choice property is $\frac{\text{additionalCoverage}}{\text{duration}+1}$. The +1 in the denominator is there to avoid undefined values for tests with 0ms duration.

The code for the ordering function is shown in Listing 4.3 and Listing 4.4. The `orderTests` function takes the list of already selected tests T' . At first `tests` is divided into two lists: `newTests` with all tests that have not been executed yet and `remainingTests` with tests where the coverage at baseline is known.

In the following tests are greedily selected until all lines in the changed methods would have been covered once in the baseline version (in the following referred to as one round). Then the already covered lines are reset and the selection process is restarted with the remaining tests until no tests are left. After round 1 also `newTests` are added to the list in undefined order, because no further information about them is known in advance.

Listing 4.3: Ordering algorithm (part 1)

```
orderTests(tests): TestCaseList {
    newTests = tests.filter { it.getTotalCoveredLines() == 0 }
    remainingTests = tests.filter { it.getTotalCoveredLines() != 0 }

    orderedTests = []
    orderedTests += selectAndOrderTestsForNextRound(remainingTests)
    orderedTests += newTests

    while (!remainingTests.isEmpty()) {
        orderedTests += selectAndOrderTestsForNextRound(remainingTests)
    }

    return orderedTests
}
```

The `selectAndOrderTestsForNextRound` function first resets the already covered lines for all remaining tests. Then searches for the test, which yields the most additional

coverage in the shortest time. As long as there are tests that cover yet unvisited lines, they are added to the ordered list. The additional coverage for the remaining tests is updated afterwards.

Listing 4.4: Ordering algorithm (part 2)

```
selectAndOrderTestsForNextRound(remainingTests): TestCaseList {
    orderedTests = []
    remainingTests.forEach { it.resetAdditionalCoverage() }
    while (!remainingTests.isEmpty()) {
        testWithBiggestGain = remainingTests
            .max { it.getAdditionalCoverageDurationRatio() }
        if (testWithBiggestGain.getAdditionalCoverage() == 0) {
            break
        }
        orderedTests += testWithBiggestGain
        remainingTests -= testWithBiggestGain

        remainingTests.forEach { it.substractCoveredLines(testWithBiggestGain) }
    }
    return orderedTests
}
```

4.4. Limitations

The approach has some limitations listed below:

- **Assumes stable coverage** The approach assumes that every test case has a stable set of covered methods, which is not always the case (see chapter 5 RQ1).
- **No parallel test execution** Currently tests cannot be executed in parallel in one JVM when recording coverage, because JaCoCo cannot distinguish which coverage is caused by which test case.
- **Parameterized tests have to be named** If parameterized tests are not uniquely named or not named at all, it is not possible to name and execute them separately afterwards.
- **Test detection heuristic** The currently implemented test detection is only a heuristic and does not always work in complex setups e.g. in the Apache Commons Math library, which extensively uses subclassing also in test code. The solution would

be to upload a list of currently available tests and when they haven been changed before getting the suggested tests. The available tests can be extracted from the bytecode of the system instead of source code for JUnit tests and from the plugin APIs for TestBench and Tosca. The names of all available tests are needed for two reasons: First, to suggest all tests, which have been added since the last coverage collection. And second, to re-execute tests that have been changed.

- **Changes outside of methods** Changes to field initializations and constants are currently not considered. Hence errors introduced here are not detected. A solution would be to mark all constructors as modified if changes are detected.
- **Changes to resources** Changes to configuration files, third-party code or other external factors, like environment variables or similar, are not considered in this approach, but should stay constant anyway in a testing environment to produce reproduceable results.
- **Indirect invocations** Does not work if code is indirectly invoked e.g. via reflection. Imagine a test t , which searches for all classes annotated with `@MyDomainObject` and executes a certain method to ensure consistency amongst the implementations. Adding a new `@MyDomainObject`-annotated class x would not lead to any code changes in t . Therefore t would not be suggested for re-execution even though t could potentially find bugs in x .

5. Evaluation

5.1. Research questions

RQ1: How stable is the set of methods that are executed by a test? The chosen approach basically relies on the assumption that executing the same tests on the same codebase multiple times also causes the same methods to be executed for every run. This research question aims to investigate whether this is always the case or if our intuition is wrong here and in what causes lead to unstable coverage.

RQ2: How reliably can we infer tests from a given code fragment? Furthermore, the approach uses coverage data that has been recorded for a previous version of the code. I therefore expect the set of tests, which execute a given method, to stay the same over time. For RQ2 I wanted to investigate the impact of code changes on this mapping.

RQ3: How many bugs do we find by executing only the suggested tests? Only covering changed code regions is often not enough to also detect faulty behavior[8]. Therefore, investigation is required to find out how many bugs can be found by the suggested tests, compared to the retest-all approach.

RQ4: Does ordering the test cases make them fail faster? The approach aims to not only reduce the number of executed test cases, but also should make them fail faster by ordering them. This research question examines whether the implemented approach makes the tests significantly fail faster.

RQ5: How rapidly does the list of suggested tests grow? A common problem of regression test selection is that the number of suggested tests grows quickly when numerous changes are involved, which in the worst case results in suggesting all available tests. This would make suggesting tests in the first place superfluous. RQ5 aims to find out for change size using this technique still produces reasonably smaller test suites.

5.2. Study Objects

The study was done on a selection of, for the most part, well known open source projects. The projects were selected on GitHub based on their popularity and repository size to include both popular and huge projects as well as medium and small sized projects. Besides 11 open source projects, Teamscale was chosen as a closed source project, with high number of commits. The study objects are listed with their most important metrics in Table 5.1 followed by a short description of the projects. A more detailed table with the study objects can be found in Appendix A.

Table 5.1.: Study objects

Study object	build	framework	kLoC			commits
			total	source	test	
Apache Commons Collections	maven	JUnit 4	62	31	31	3235
Apache Commons Lang	maven	JUnit 4	75	27	48	5486
Apache Commons Math	maven	JUnit 4	178	87	91	7156
Histone Template Engine 2	maven	JUnit 4 & 5	14	12	2	1133
JabRef	gradle	JUnit 4	122	94	27	10645
Joda-Time	maven	JUnit 3	83	28	55	2105
Lightweight-Stream-API	gradle	JUnit 4	23	8	15	529
LittleProxy	maven	JUnit 4	9	4	5	1037
OkHttp	maven	JUnit 4	52	26	26	3548
RxJava	gradle	TestNG	242	84	158	6000
Symja Commons Math Parser	maven	JUnit 4	7	6	2	44
Teamscale	gradle	JUnit 4	336	270	67	82164

Apache Commons Collections Utility classes for working with any type of collections in Java.

Apache Commons Lang A package of Java utility classes for the classes that are in java.lang's hierarchy, or are considered to be so standard as to justify existence in java.lang.

Apache Commons Math Provides mathematics and statistics components that are addressing the most common practical problems and which are not directly supported by Java.

Histone Template Engine 2 Histone allows to generate text based content e.g. HTML based on a powerful template system.

JabRef JabRef is a graphical Java application for editing and organizing BibTeX and Biblatex databases.

Joda-Time Joda-Time is the widely used replacement for the Java date and time classes prior to Java SE 8.

Lightweight-Stream-API Is a library that back-ports the Java 8 stream API to Java 7.

LittleProxy LittleProxy is a high performance HTTP proxy, which can be used to intercept network connections.

OkHttp Performant HTTP client for Android and Java applications with support for HTTP 2, caching and transparent gzip-compression.

RxJava A library allowing to write Java applications in reactive programming style.

Symja Commons Math Parser A implementation of a math expression parser.

Teamscale A software quality analysis tool, which provides analysis results in realtime for analyses like architecture conformance, code clone detection, coding guideline conformance, test gap analysis and a lot more for over 20 programming languages.

5.3. Study Design

RQ1: How stable is the set of methods that are executed by a test? For the first research question I first recorded coverage for all tests at a specific baseline version for each study object. Then I ran the coverage recording again for one more time on the same baseline to find out whether the set of methods covered by a tests changed amongst those executions.

RQ2: How reliably can we infer tests from a given code fragment? To determine how code changes have an impact on the code to tests mapping I ran all the tests at a baseline version and at two distinct versions after the baseline. Since nightly or at least weekly builds, which execute all tests are common practice, I ran the tests again after one day and one week of development activity. This way if integrated into the development process, developers could always run the suggested tests for their current changes based on the coverage of the last nightly or weekly build.

RQ3: How many bugs do we find by executing only the suggested tests? To answer the question I have generated mutations of the original code to simulate a newly introduced bug. For each mutation the suggested tests were calculated and executed on the mutated version of the software. If none of the suggested tests did fail, all remaining tests were executed to see if any of them were able to reveal the bug.

RQ4: Does ordering the test cases make them fail faster? Based on the setup of RQ3 for those suggestions that did find the fault, I measured the time it took until the first test failed.

RQ5: How rapidly does the list of suggested tests grow? For each study object the list of suggested tests was calculated for all commits after the baseline, based on the coverage for the baseline.

5.4. Procedure

RQ1: How stable is the set of methods that are executed by a test? To record the coverage I configured the maven and gradle plugins for each of the study objects. Some projects contained tests that did not pass on my laptop for various reasons. They were ignored by adding @Ignore annotations. After running the tests once and uploading the coverage to Teamscale all temporary build data was deleted by running the clean task for the respective build system before rerunning the tests.

RQ2: How reliably can we infer tests from a given code fragment? For every method I compared the tests that executed this method at the baseline with the tests that executed the method one day and one week later. The commit that was considered as one day later was chosen to contain at least one day of development activity and at least 2 commits. For projects with little development activity those two commits were also considered as one day, if they were scattered across multiple days. Assuming that the tests that executed the method at baseline plus the tests that have been added or changed are all tests that cover the methods at later timestamps, I calculated the methods, which broke this assumption.

RQ3: How many bugs do we find by executing only the suggested tests? To generate mutations of the original source code I used the *PIT mutation testing tool* (Pitest)¹. Pitest can be integrated into build systems like Maven and Gradle and provides an

¹<http://pitest.org/>

end-to-end solution for generating mutants, running tests on the mutations and generating a report afterwards. Through its plugin mechanism I used it to randomly select a number of mutations. When executing the tests Teamscale was queried to generate test suggestions that were afterwards executed in the provided order to test if they kill the mutant. The information about the mutant was passed to Teamscale as a tuple of file name and modified line number. The method containing that line was then treated as modified.

Each project was ran against 100-1000 mutations depending on the size of the test suite to finish the analysis in an reasonable time of under 5 hours.

I applied the following types of mutations, because they are similar to bugs that are likely to happen in a real development scenarios as well:

- **Conditionals Boundary Mutator** Replaces occurrences of e.g. `<` to `<=`.
- **Increments Mutator** Exchanges `++` with `--` operator and vice versa.
- **Void Method Call Mutator** Removes a call to a method with void return type.
- **Return Vals Mutator** Depending on the methods return type, returns a slightly different result. For long e.g. `+1` is added to the original return value.
- **Math Mutator** Arithmetic operations are replaced with another operation e.g. `+` with `-`
- **Negate Conditionals Mutator** Replaces comparison operators with its negated version e.g. `==` gets replaced with `!=`.

There are multiple reasons for which a mutant can be considered as killed, all of which were considered as fault revealing in RQ3:

- **Killed** The test was executed and failed.
- **Memory error** The test was run and caused a memory error e.g. stack overflow caused by endless recursion.
- **Timed out** The mutation caused an endless loop and terminated due to a predefined timeout.

To make the results better reflect real-life situations, a set of changes was considered in addition to the mutation itself. The coverage used to generate the test suggestions was recorded at a baseline and the mutation was generated and the tests executed for the code at the version end. The end version was selected to be one or two version numbers later, because some follow-up bugfix versions only contained up to two commits. The versions used for each study object are shown in Table 5.2.

Table 5.2.: Considered changes

Study object	baseline	end	commits	days
Apache Commons Collections	4.1	master ²	37	535
Apache Commons Lang	3.6-RC2	3.6-RC4	47	75
Apache Commons Math	3.6.0	3.6.1	14	75
Histone Template Engine 2	1.7.1	1.9.1	39	191
JabRef	4.0-beta1	>4.0-beta1	51	4
Joda-Time	2.9.7	2.9.8	12	93
Lightweight-Stream-API	1.1.6	1.1.8	49	115
LittleProxy	1.1.1	1.1.2	24	69
OkHttp	3.8.0	3.8.1	43	71
RxJava	2.1.1	2.1.2	59	87
Symja Commons Math Parser	1.0.0	master	9	260
Teamscale	3.1.1	>3.1.1	9 ³	7

RQ4: Does ordering the test cases make them fail faster? The procedure for RQ3 is identical with RQ4. Except that for the timings in RQ4 only **Killed** mutants were considered, because **Timed out** and **Memory error** are dependant on the chosen timeout factor and currently available heap size and therefore don't produce reliable and reproducible results.

RQ5: How rapidly does the list of suggested tests grow? For projects like Teamscale with a very high commit density only every 10th commit was analyzed, so that the analysis completed within 3 hours.

²At Jul 26 2017. No other version in between

³Only merges of feature branches with a lot of changes

5.5. Results

RQ1: How stable is the set of methods that are executed by a test? Table 5.3 shows for each study object the number of total tests and how many of them did have differing coverage in the second run. While half of them did have exactly the same coverage, others had a large amount of tests with differing coverage. The causes have been explored manually. For RxJava and Teamscale only a random sample has been considered. The root cause of most inspected differences in coverage was asynchronism, which in turn was often a result of a scheduler or thread pool being organized across test case boundaries. Calls to logging frameworks, which asynchronously write the log messages either to disk or to the user interface, also involved asynchronous calls. These observations were made across all study objects. The Apache Commons Math coverage differences were completely caused by calls methods based on random numbers.

Table 5.3.: Tests changing their coverage

Study object	Total	Changed
Apache Commons Collections	5628	2
Apache Commons Lang	3975	0
Apache Commons Math	5964	4
Histone Template Engine 2	27	0
JabRef	2510	10
Joda-Time	4201	0
Lightweight-Stream-API	1002	0
LittleProxy	183	3
OkHttp	2084	12
RxJava	8735	926
Symja Commons Math Parser	79	0
Teamscale	3670	431

In JabRef 141 additional tests did change their names amongst the compared test runs. The differing names were caused by named parameterized tests. Normally those names are used to uniquely identify parameterized test instances. But in those tests the names contained object references, which differed in consecutive runs. For example `getKeyReturnsNotEmpty[8: ...MinifyNameListFormatter@73b91e06]`

RQ2: How reliably can we infer tests from a given code fragment? Table 5.4 shows the results of RQ2. The table lists the total number of methods the project had at the baseline, how many of them had a changed test set covering them after 1 day and 1

week as well as how many tests per method were not part of the baseline coverage on average.

Table 5.4.: Methods with changed test sets

Study object	Methods	1 day		1 week	
		Methods	Tests	Methods	Tests
Apache Commons Collections	4699	3	1.0	5	1.0
Apache Commons Lang	4653	0	0	0	0
Apache Commons Math	9954	25	1.4	25	1.4
Histone Template Engine 2	290	0	0	0	0
JabRef	3732	151 ⁴	4.76	204 ⁵	3.89
Joda-Time	7262	0	0	3	1
Lightweight-Stream-API	801	15	1.4	15	1.4
LittleProxy	511	8	1.0	8	1.0
OkHttp	3522	44	1.29	36	3.03
RxJava	12700	272	27.10	167	11.11
Symja Commons Math Parser	285	0	0	0	0
Teamscale	16959	815	2.51	926	2.86

RQ3: How many bugs do we find by executing only the suggested tests? The results in Table 5.5 show that 99.2% of the mutations, which the whole test suite was able to kill, were also killed by the suggested tests. The causes for missing certain tests varied. The 10 mutations in Apache Commons Lang were missed, because the Pitest applied mutations to methods of an enum, which Teamscale did not consider as methods and therefore did not track this modification. The missed tests in RxJava were caused by flickering base coverage and in Teamscale some mutations were applied to generated code, which was excluded in the Teamscale analysis and was therefore not considered. I was not able to find the root cause for Joda-Time and Lightweight-Stream-API though.

RQ4: Does ordering the test cases make them fail faster? The failure times have been sorted and accumulated and are shown in blue in Figure 5.1. Compared to the red line that shows the failure reveal times when running the suggested tests in random order assuming that the failure revealing tests are equally distributed. The black line in contrast shows the average failure reveal times when running all tests, not only the suggested ones.

⁴26 when ignoring tests, which include a string representation of the object reference

⁵73 when ignoring tests, which include a string representation of the object reference

5. Evaluation

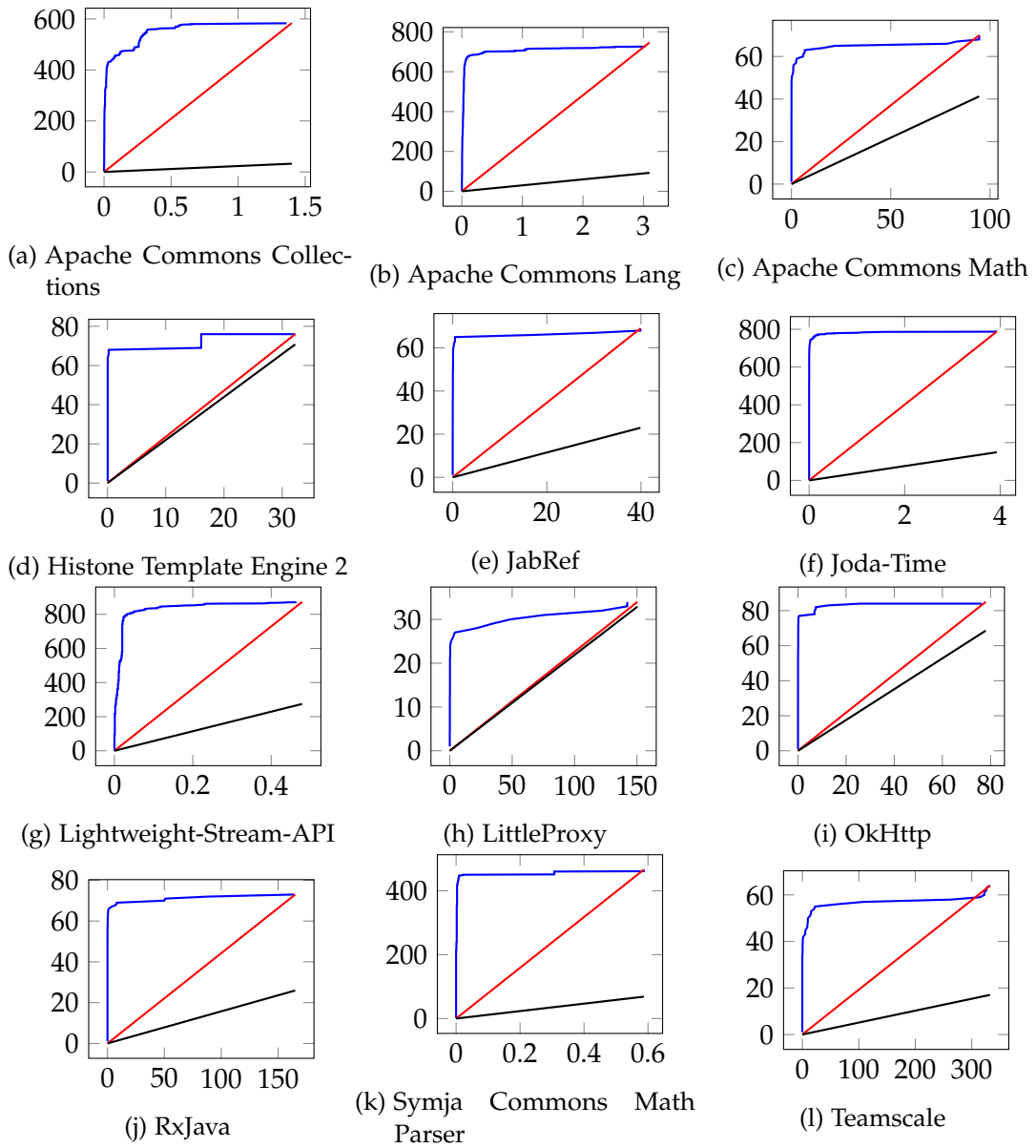


Figure 5.1.: Test failure times (x=Time in seconds, y=Failed tests)

Table 5.5.: Mutations missed by suggested methods

Study object	Mutations	Killed	Missed by suggestions
Apache Commons Collections	1000	584	0
Apache Commons Lang	996	747	10
Apache Commons Math	100	85	0
Histone Template Engine 2	839	85	0
JabRef	196	92	0
Joda-Time	986	792	3
Lightweight-Stream-API	1000	953	2
LittleProxy	96	50	0
OkHttp	199	85	0
RxJava	100	87	8
Symja Commons Math Parser	995	478	0
Teamscale	154	64	6

RQ5: How rapidly does the list of suggested tests grow? The number of suggested tests for each project is shown in Figure 5.2 for a timespan of 100 days. The x-axis shows the percentage of files that have been modified during this time. The y-axis depicts the number of tests, whereby the maximum visible y position equals the total number of tests available in the project. The blue line shows the number of all suggested tests and the red line shows the number of tests selected during round 1 of the algorithm (see section 5.6). Computing the lists took between 10 minutes and 2 hours depending on the project size and number of commits.

5. Evaluation

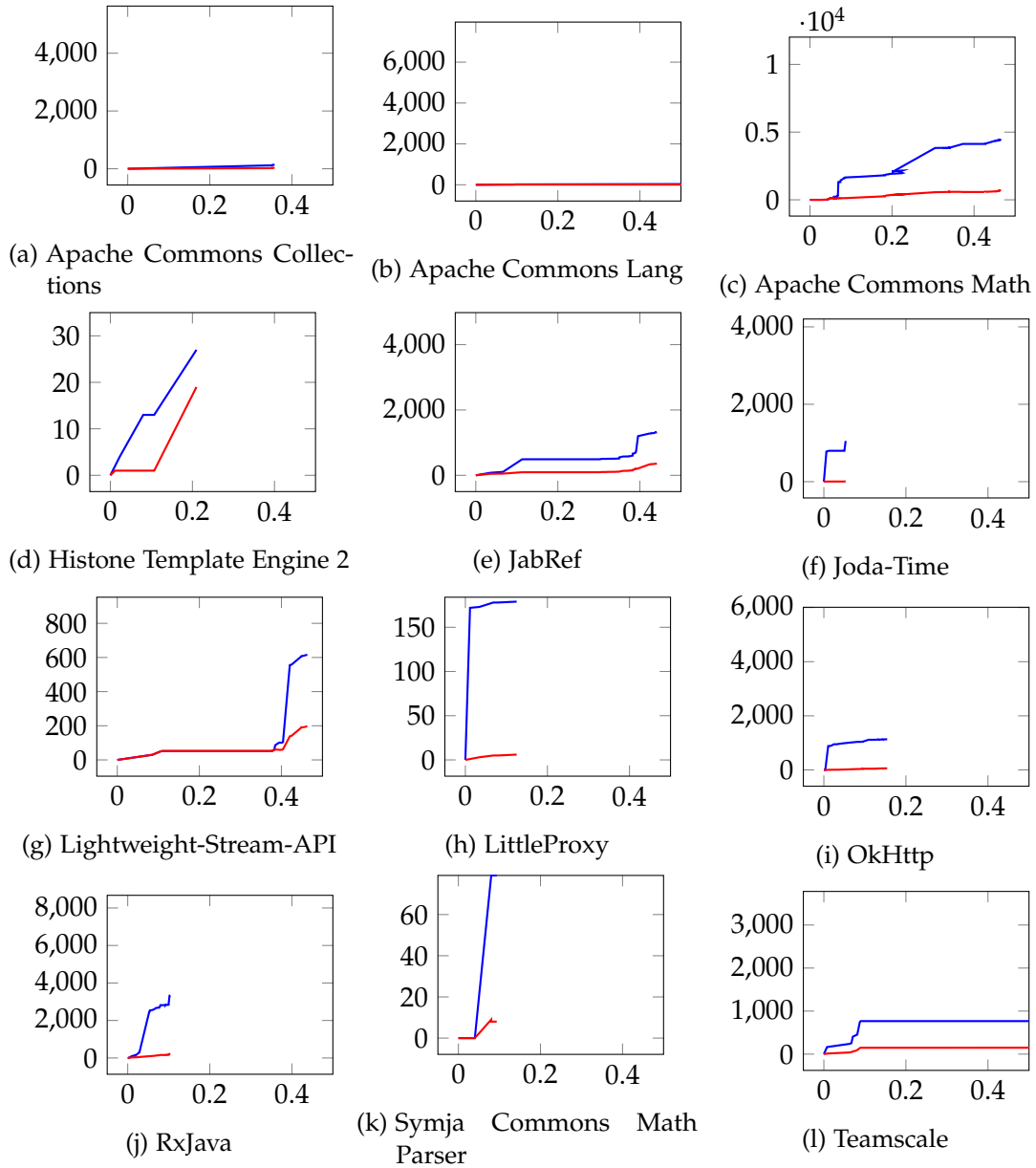


Figure 5.2.: Number of suggested tests (x=Percentage of files changed, y=Tests)

5.6. Discussion

RQ1: How stable is the set of methods that are executed by a test? The results of RQ1 show that a reliable mapping from code to tests is not always possible for code, which is called asynchronously. This also implies that a reduced test set generated based on a single coverage report can never be guaranteed to select all failure revealing tests. But the data gathered could be used to detect potentially flickering tests and could also be a means to find the root cause.

RQ2: How reliably can we infer tests from a given code fragment? As expected the study objects, which did have differing coverage when testing the same code version, also had errors in predicting the tests executing a method after one day and one week of development activity. On average only 2% of all methods were affected by incorrect test predictions. The total number of missed test predictions amongst those was relatively small with 3 tests on average per affected method. But the difference between using one day and one week old coverage as basis to predict method executions was insignificant. This means running all tests only once per week and executing only a subset of tests for every commit in between — based on the last weekly coverage — would be a viable option.

RQ3: How many bugs do we find by executing only the suggested tests? As shown 99.2% of the mutations, that the whole test suite was able to reveal, were also suggested by the presented approach. Inspecting the causes for missed tests indicates that most of them could also be avoided by fixing the underlying issues in Teamscale. For those missed tests that are caused by flickering base coverage, an addition to the approach, which also suggests tests that sometimes execute the changed methods could overcome the problems there.

RQ4: Does ordering the test cases make them fail faster? The results indicate that ordering the tests makes the tests fail faster. On average more than 90% of the tests have already failed after 10% of the execution time. 80% of those tests that killed mutations were suggested by the algorithm during round 1. So for risk-based testing, where even executing the reduced test suite is not feasible, the ordering provides a good means to adjust the test suite size to the available budget.

RQ5: How rapidly does the list of suggested tests grow? The results indicate that the approach tends to quickly grow for small projects like Histone Template Engine 2 (Figure 5.2d), LittleProxy (Figure 5.2h) and Symja Commons Math Parser (Figure 5.2k).

Some rare commits (see Figure 5.2g, Figure 5.2h, Figure 5.2k) result in a high increase of the total tests suggested. If core components are altered, because all tests are suggested that execute this core component. When only considering tests of e.g. round 1, the effect of those commits can be avoided, because by definition round 1 only produces as much tests as necessary to cover every modified line once. In practice this means the same as already described in the previous paragraph. The test execution can simply be stopped if the given time budget has been exceeded.

5.7. Threats to Validity

The study objects mainly consisted of open source libraries. Even though I used Teamscale as non-open source study object the results may not be generalizable. The presented approach is designed to work for both unit and integration tests, but most of the study objects mainly had unit tests, so a general applicability to integration tests is not given. As setting up the configuration for the study on an old version of the system was not trivial, I only considered one baseline for all the research questions. Other baselines may have yielded different results.

6. Future Work

Since the Test Impact Analyzer from Microsoft uses coverage on file level, it would be interesting to compare the number of suggested tests produced by coverage on file level with coverage on method level. Furthermore the study should be applied to even bigger industrial projects to measure its effectiveness there. Also the implemented plugins for Maven, Gradle, Tosca and TestBench should be enhanced to allow automatically executing the suggested tests.

The coverage collected in Teamscale could be used to build other analyses as well. For example an analysis to discover redundant tests, which execute the same functionality and could therefore be removed to speed up test executing. Another interesting follow-up analysis would be *failure clustering*. Since Teamscale can also process JUnit reports, which contain information about which tests did pass and fail, the coverage data per-test could be used to provide hints for the cause of the multiple failing test cases, by inspecting, which parts of the changed code are covered by multiple failing test cases.

As shown in RQ1 of the evaluation asynchronous execution of tasks is a problem for the reliability of the test suggestions. On the one hand JaCoCo could be extended to support tracking of which threads were started by which test case. This could make it possible to map coverage that was recorded after the test has finished to the correct test case. On the other hand an analysis could be built to support localizing the root cause of flickering tests.

7. Conclusion

The aim of this thesis was to implement a solution for getting coverage on a per-test level by using the already existing and widely used JaCoCo coverage profiler. Integrations with Gradle, Maven, Tosca and TestBench have shown that the approach can be used independently of the used testing environment. Based on this coverage data an approach to automate selective regression testing and prioritization was implemented in the software quality analysis software Teamscale.

The approach used a DejaVu based approach to select the tests for re-execution. The selection strategy used the change information available in Teamscale to select tests which execute methods that have been modified. Afterwards a greedy algorithm was used to sort the tests by their additional coverage on method level. In contrast to other approaches I used the additional coverage per execution time to execute fast tests first.

The evaluation on twelve study objects showed that coverage is not always stable due to asynchronous tasks, which ran longer than the actual test (RQ1). These unstable test to code mappings cause incomplete predictions when applied to a modified version of the code in turn. But the study has indicated that there is no huge difference in using a weekly full build and a nightly full build (RQ2). Using older coverage to infer, which tests will execute changes introduced to the code, can therefore not be considered completely safe, but has demonstrated to work well enough to catch 99.2% of the introduced bugs (RQ3). The results have also shown that ordering the tests makes more than 90% of the test fail within the first 10% of the total execution time of the selected tests (RQ4). This makes the suggestions even interesting for risk based testing, because the execution could simply be stopped if a certain time budget is exhausted. Last but not least for larger projects the study indicated that the number of suggested tests stays reasonably small (RQ5).

Based on the per-test coverage a lot of interesting analyses could be implemented in the future.

A. Study objects summary

Study object	URL	build	framework	kLoC	source	test	commits	methods	tests	baseline	end	commits	days
Apache Commons Collections	https://commons.apache.org/collections/	maven	JUnit 4	62	31	31	3235	4699	5628	4.1	master	37	535
Apache Commons Lang	https://commons.apache.org/lang/	maven	JUnit 4	75	27	48	5486	4653	3975	3.6-RC2	3.6-RC4	47	75
Apache Commons Math	https://commons.apache.org/math/	maven	JUnit 4	178	87	91	7156	9954	5964	3.6.0	3.6.1	14	75
Histone Template Engine 2	https://github.com/MegafonWebLab/histone-java2	maven	JUnit 4 & 5	14	12	2	1133	290	27	1.7.1	1.9.1	39	191
JabRef	http://www.jabref.org/	gradle	JUnit 4	122	94	27	10645	3732	2510	4.0-beta1	>4.0-beta1	51	4
Joda-Time	http://www.joda.org/joda-time/	maven	JUnit 3	83	28	55	2105	7262	4201	2.9.7	2.9.8	12	93
Lightweight-Stream-API	https://github.com/aNNiMON/Lightweight-Stream-API	gradle	JUnit 4	23	8	15	529	801	1002	1.1.6	1.1.8	49	115
LittleProxy	https://github.com/adamfisk/LittleProxy	maven	JUnit 4	9	4	5	1037	511	183	1.1.1	1.1.2	24	69
OkHttp	http://square.github.io/okhttp/	maven	JUnit 4	52	26	26	3548	3522	2084	3.8.0	3.8.1	43	71
RxJava	https://github.com/ReactiveX/RxJava	gradle	TestNG	242	84	158	6000	12700	8735	2.1.1	2.1.2	59	87
Symja Commons Math Parser	https://github.com/axkr/symja-parser	maven	JUnit 4	7	6	2	44	285	79	1.0.0	master	9	260
Teamscale	https://teamscale.io	gradle	JUnit 4	336	270	67	82164	16959	3670	3.1.1	>3.1.1	9	7

List of Figures

4.1. Tosca integration overview	11
4.2. Tosca integration	11
4.3. Minimal example	14
4.4. Example system	15
4.5. Test suggestions shown in Teamscale	16
5.1. Test failure times	28
5.2. Number of suggested tests	30

List of Listings

3.1. IAgent interface	8
4.1. Coverage report extract	13
4.2. Regression test selection	16
4.3. Ordering algorithm (part 1)	17
4.4. Ordering algorithm (part 2)	18

List of Tables

5.1. Study objects	21
5.2. Considered changes	25
5.3. Tests changing their coverage	26
5.4. Methods with changed test sets	27
5.5. Mutations missed by suggested methods	29

Bibliography

- [1] F. Dreier, E. Juergens, and A. Göb. "Detection of Refactorings." Bachelor's Thesis. Technical University of Munich, 2015.
- [2] E. Engström, P. Runeson, and M. Skoglund. "A systematic review on regression test selection techniques." In: *Information and Software Technology* 52.1 (2010), pp. 14–30. ISSN: 09505849. DOI: 10.1016/j.infsof.2009.07.001.
- [3] P. Hammant. *The Rise of Test Impact Analysis*. <https://martinfowler.com/articles/rise-test-impact-analysis.html>. 2017.
- [4] M. J. Harrold, J. a. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. "Regression test selection for Java software." In: *ACM SIGPLAN Notices* 36 (2001), pp. 312–326. ISSN: 03621340. DOI: 10.1145/504311.504305.
- [5] J. Hartmann. "30 Years of Regression Testing : Past, Present and Future." In: *PNSQC 2012 Proceedings* (2012), pp. 1–8.
- [6] H. K. N. Leung and L. White. "A study of integration testing and software regression at the integration level." In: *Proceedings Conference on Software Maintenance 1990* (1990), pp. 290–301. DOI: 10.1109/ICSM.1990.131377.
- [7] C. Mao and Y. Lu. "Regression testing for component-based software systems by enhancing change information." In: *12th Asia-Pacific Software Engineering Conference (APSEC'05)*. IEEE, 2005, 8 pp. ISBN: 0-7695-2465-6. DOI: 10.1109/APSEC.2005.95.
- [8] R. Niedermayr, E. Juergens, and S. Wagner. "Will My Tests Tell Me If I Break This Code?" In: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery - CSED '16* (2016), pp. 23–29. DOI: 10.1145/2896941.2896944. arXiv: 1611.07163.

- [9] G. Rothermel and M. J. Harrold.
“A safe, efficient regression test selection technique.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6.2 (1997), pp. 173–210.
ISSN: 1049-331. DOI: 10.1145/248233.248262.
- [10] G. Rothermel and M. J. Harrold.
“Analyzing Regression Test Selection Techniques.”
In: *IEEE Transactions on Software Engineering* 22.8 (1996), pp. 529–551.
ISSN: 00985589. DOI: 10.1109/32.536955.
- [11] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold.
“Prioritizing Test Cases For Regression Testing.”
In: *IEEE Transactions on Software Engineering* 27.10 (2001), pp. 929–948.
ISSN: 0098-5589. DOI: 10.1145/347324.348910.
- [12] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry. “An information retrieval approach for regression test prioritization based on program changes.”
In: *Proceedings - International Conference on Software Engineering*. Vol. 1. 2015.
ISBN: 9781479919345. DOI: 10.1109/ICSE.2015.47.
- [13] Vectorcast. *Vector Software kündigt VectorCAST/QATM an.*
<https://www.vectorcast.com/de/news/vector-software-press-releases/2016/vector-software-kuendigt-vectorcast-qa>.
2017.
- [14] D. Willmor and S. M. Embury.
“A safe regression test selection technique for database – driven applications.”
In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*.
IEEE, 2005, pp. 421–430. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.15.
- [15] Y. Wu, M.-H. Chen, and H. M. Kao.
“Regression testing on object-oriented programs.” In: *Proceedings 10th International Symposium on Software Reliability Engineering (Cat. No.PR00443)*.
IEEE Comput. Soc, 1999, pp. 270–279. ISBN: 0-7695-0443-4.
DOI: 10.1109/ISSRE.1999.809332.