b
UNIVERSITÄT
BERN

Software Testing in Industry

Assessing Unit Testing in an Industrial Software Project

Bachelor Thesis

Markus Eggimann
from
Grosshöchstetten, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

Summer 2018

Prof. Dr. Oscar Nierstrasz
Dr. Mohammad Ghafari

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Automated testing is an important technique to ensure the quality of a software system, and there is a general consensus in industry that testing is a critical part of the development process. However, recent studies suggest that unit testing is not that widely practiced. In this thesis, we studied an industrial software project called *EPOF* with respect to testing. We tried to answer the question whether the discovery of bugs pushes the writing of tests, whether unit tests help to prevent bugs, and whether the system's architecture facilitates or impedes unit testing. To answer those questions, we studied the bug reports and associated bug fix reports of the project. Our results showed that the test coverage was rather low, and most bugs were fixed without adding or changing any tests, most bugs were detected by manual testers or customers and not by the existing tests and that the testability of the code is low in most parts of the system. In 2017, the development team decided to give unit testing higher priority. Our results show that this decision, together with other development process improvements, indeed had a positive effect on the bug rate and the testability of the system.

Contents

1 Introduction **1**

2 Case Study **5**

2.1 Description of the case 5

2.2 Insights into the development process 5

2.3 Common setup 7

2.3.1 Relevant sources of information about the EPOF project 7

2.3.2 Collection of data 8

2.3.3 Intermediate results 9

3 Inspection of Bugs **11**

3.1 Setup 11

3.2 Results 13

3.2.1 Bugs in defective classes covered by tests 13

3.2.2 Number of modified files per bug fix 14

4 Effectiveness of Unit Tests **16**

4.1 Recurring bugs 16

4.1.1 Setup 16

4.1.2 Results 17

4.2 Proneness to bugs 21

4.2.1 Setup 21

4.2.2 Results 22

4.3 Mutation testing 33

4.3.1 Setup 33

4.3.2 Results 34

5 Assessment of Testability **37**

5.1 Setup 37

5.2 Results 38

5.2.1 The bug is a usability or design issue 39

5.2.2 The erroneous component has many dependencies 40
5.2.3 The defective component violates SRP 45

6 Related Work 50

7 Threats to Validity 52

8 Conclusion and Future Work 54
8.1 Answers to research questions 54
8.2 General insights 56
8.3 Future work 56

9 Anleitung zum wissenschaftlichen Arbeiten 58
9.1 Exploiting the Jira API 59
9.1.1 The Jira API 59
9.1.2 The database model 60
9.1.3 Importing the data from Jira 62
9.1.4 Customizing the tool 63
9.1.5 Querying the imported data 64
9.2 Cloning repositories 65
9.2.1 Customizing the tool 65
9.3 Comparison of diffs on method level 65
9.3.1 The algorithm 66
9.3.2 Implementation details 66
9.3.3 Customizing and extending the tool 69

List of Figures 71

List of Tables 71

Listing 72

Bibliography 73

1
Introduction

Automated testing is an important technique to ensure the quality of large scale software systems. Especially recent trends in software development like continuous integration (CI) and continuous delivery (CD) require efficient and simple ways to test software and ensure that recent changes in the code base do not break existing functionalities.

The theory of software testing focusing on automated testing distinguishes several different categories of tests: *Unit tests*, *Integration tests* and *End-to-End tests*.

Unit tests are the most basic tests that test single components of a software system in isolation and verify the correct behavior of a component. In the context of unit testing, the notion of a component is used as a synonym for a class in object-oriented systems. Unit tests typically are only a few lines of code long, rarely need external components like a file, the hardware or a database and take just few milliseconds to execute. Large software systems usually contain hundreds or thousands of unit tests that are run often during the development process by the developers and help them to verify that the code they just wrote behaves like it should in the given scenario. Unit tests are written by the developers themselves using a programming language specific unit testing framework like *JUnit* [29] for Java or *XUnit* [6] for C#. Most integrated development environments (IDE) have native support for one or multiple such frameworks and can detect and execute tests automatically and display afterwards the results of a test run.

Integration tests are used to test the functionality and interaction of multiple components. They typically take longer to execute than unit tests, they may involve calls to external systems, and they are often run by a build server and not on the developers' computer. Integration tests are more complex to write than unit tests because they require the correct configuration and setup of notable parts of the software system under test.

Furthermore, they operate on test data that are often read from external data sources like files or a database. Integration tests are usually written by using the same testing frameworks used for unit tests.

End-to-end tests aim to test the whole system. They focus on real usage scenarios of the system and therefore require the whole system to run. These tests are the hardest to automate, and many different tools exist that can be used to create such tests like *Selenium* [24] or *Tosca* [30].

There is a general consensus in industry that software testing in general, and automated testing as one discipline of it, is a critical part of the development process. Multiple studies show that even though automated testing slows down the development process a little bit, it saves money during the operational and maintenance phase of the software project because bugs are discovered before they get into production and, as a result, debugging is less expensive. But when it comes to the question how automated testing is actually done in industry, research results show a completely different picture. Zaidman *et al.* [2] show that automated testing and test-first practices like TDD are not that widely used as developer surveys suggest. Their results also show that developers systematically overestimate the time they spend for testing. Borle *et al.* [3] show that only 16.1% of the Java Repositories on Github contain test files. Fucci *et al.* [8] tried to find out which aspects of TDD have an influence on the quality of tests and the on productivity of the developers. They found that the key aspect of TDD — test-first — seems to have no influence or only a minor one.

This obvious contradiction between the general agreement that unit testing is important and the way it is actually adopted in software projects motivated us to study this subject in an industrial context. We studied a software project in a large enterprise that provides a digital platform for receiving and sending electronic and physical mail. During the study, we had full access to the source code, the version control system, the issue tracker, and the developer team of the project. When something was unclear or if we had questions concerning the project, we could approach the developers or other project stakeholders. Our research project focused on unit tests because, when a project team uses automated testing, it usually starts developing unit tests first. Integration and end-to-end tests are more advanced kinds of tests that are usually introduced after a notable part of the code base is covered by unit tests.

The following three research questions guided this study:

RQ1: Does the discovery of bugs push the writing of tests? We wanted to find out what actions developers take after a bug has been discovered. They could add new tests to document the bug, they could modify existing tests, or they could just fix the bug without changing any tests.

To answer this question, we studied which parts of the code were modified in order to fix a bug. Our results showed that in most cases the bugs were fixed without making changes to any tests. Furthermore, we observed that for more than half of the analyzed bugs, it was necessary to change multiple files to fix them.

RQ2: Do existing tests prevent the occurrence of bugs? We wanted to find out whether the existing tests actually help the developers to find or prevent bugs. If the existing tests are not able to detect bad changes to the source code or resulting changes in the behavior of the software, they are probably useless.

To find an answer to this question, we checked whether a certain bug occurred multiple times and whether there are components that are affected by multiple different bugs and hence are more prone to bugs. Our results showed that there were only very few bugs that occurred multiple times. There were indeed components that are more prone to bugs than others. Many of those components play an important role in the system and often do too many different things.

Furthermore, we applied mutation testing to parts of the source code that was covered by tests. We wanted to see whether the existing tests detected changes to the source code. Our analysis showed that the existing tests are generally not good at detecting code manipulations.

RQ3: Is the system's architecture designed in a way that addition of new tests is easy? It might happen that a developer has to add a new unit test for an existing component. Perhaps because a bug was discovered that had not been caught by the existing tests. Or because the developer simply wants to add some tests before he changes a certain component. Using the tests, he can verify that his changes did not break any functionality. Either way, adding a test requires some effort. But it generally depends on the design of the component, its dependencies, and its complexity how much effort is needed to write a unit test for it.

We wanted to find out how test-friendly the software system under investigation is designed. Our results showed that the testability of a majority of the system's components is low. Thus, adding a test involves much work for developers. The main reasons for the low testability are the following:

- A component violates the single responsibility principle
- A component has many dependencies on other parts of the system or external components.
- The bug is a usability or design problem for which no exact definitions of what is good or what is bad exist.

The rest of this thesis is organized as follows: Chapter 2 describes the case study that was done to find answers to the research questions above for a specific industrial software project. Chapters 3, 4 and 5 describe the different analysis that were done during the case study including the results found. Chapter 6 indicates related work, chapter 7 discusses threats to validity, and chapter 8 summarizes and classifies the results of the study. Finally, chapter 9 gives some technical details about the tools and techniques used in this study and how to use them in other research projects.

2 Case Study

2.1 Description of the case

This thesis seeks to answer the research questions by doing a case study of a large industrial software project. The software project that was chosen for this research project is developed by a large logistics company of Switzerland, which will be referred to as *Company P* in this paper. The software project is called *EPOF*. It is a digital platform for sending and receiving physical as well as electronic mail. Customers of the platform are companies and also private persons. The main functionality of the platform is that customers can decide for every sender whether they want to get incoming mail of this sender in physical or electronic form. If a customer chooses to receive the mail in electronic form, Company P opens and digitalizes the customer's incoming physical mail and provides the image of the mail in the *EPOF* portal. The customer can access the platform with a mobile application or through a web interface. Also, the customer can deposit his bank account details in the portal and pay incoming bills directly within the portal.

2.2 Insights into the development process

The *EPOF* project was launched in 2011. It uses a variety of technologies for different parts of the system. The back-end uses a Microsoft SQL Server database and is written in C#. There exist mobile applications for Android (developed with Java) and iOS devices (developed with Objective-C), and the web front end uses Angular as its GUI engine.

The different parts of the application (back-end, mobile apps, web app) are developed by different developer subteams. The project uses Confluence for requirements engineering and documentation, Bitbucket for the hosting of the Git version control repositories, and Jira for issue tracking. All those three tools are developed by Atlassian, and they are hosted in-house at Company P.

The development team uses the SCRUM agile project method [23] with a development sprint duration of two weeks. The quality of the project is ensured by a quality process that includes a variety of quality assurance mechanisms:

- Automated unit tests are written to test small units of the business logic. However, the coverage varies between the various components of the system and is generally low.
- A team of testers does manual testing of the portal and the mobile apps. Manual regression tests are executed after every development sprint.
- The static analysis tool SonarQube [25] is used to find different kinds of issues in the code. The tool finds common code smells and suggests possible solutions to resolve them. The tool is applied only to the Android mobile app.
- Recently, the team is pushing towards automated UI and regression tests using Selenium [24] and Tosca [30].

There were two major changes in the development process of the *EPOF* project that are relevant to this study:

- In the early days of the project, the developers used Subversion as the version control system. The team later moved to Git and did not migrate the project's source history to the new version control system.
- The project started as a simple web application and grew larger over time. The back-end is a large monolithic system which had its origins in that web application. In summer 2017, the team decided to structure the whole system in a more modular way. New features should be added as modules, and these modules should be covered by unit tests. The team and the project management agreed to strive for 40%-80% code coverage of those new modules.

Table 2.1 shows some additional information about the *EPOF* project. The version control system related numbers in the table are not accurate because the source code was moved from Subversion to Git. However, the magnitude of those numbers is correct.

Number of developers with at least one commit	42
Number of commits in total	22’461
Number of files in repository	14’102
Number of issues in Jira	15’474
Number of reported bugs in Jira	6’531

Table 2.1: Information about the *EPOF* project

2.3 Common setup

The following sections describe the setup that served as common ground for the different analyses that are described in chapters 3, 4 and 5.

2.3.1 Relevant sources of information about the EPOF project

There are three sources of information about the *EPOF* project that are interesting for this research project.

Bug reports They are very useful because they document bugs that had occurred and provide an entry point for further investigation of the circumstances of a bug. These reports are created and maintained with labels “Bug” or “Sub-Bug” in the Jira issue tracker. The person who detects the bug usually creates an issue which contains certain information about the bug. The issue contains the steps necessary to reproduce the bug, the component affected, a description what exactly went wrong, and other details which may help the developers to locate and fix the bug. If a bug is critical, it is fixed immediately and the corrections are deployed as a hotfix. If the bug is not critical, it is moved to the backlog of the project and resolved in one of the next development sprints.

Code changes of a bug fix By examining the code changes that were done to fix a certain bug, it is possible to reconstruct what exactly the problem was. To gather this information, it is usually required to manually inspect the commits that were pushed to the version control system after a bug has been detected. Other hints like the issue key, which is perhaps mentioned in commit comments, might also be useful. In the case of the *EPOF* project, there was an easier way to gather this information. Internally Jira is linked to Bitbucket server, the software for hosting Git repositories. A developer usually creates a new branch in the Git repository for every Jira issue he works on. Jira keeps track of the branches and commits, which belong to a certain issue, and allows access to this information through its web interface and through its API.

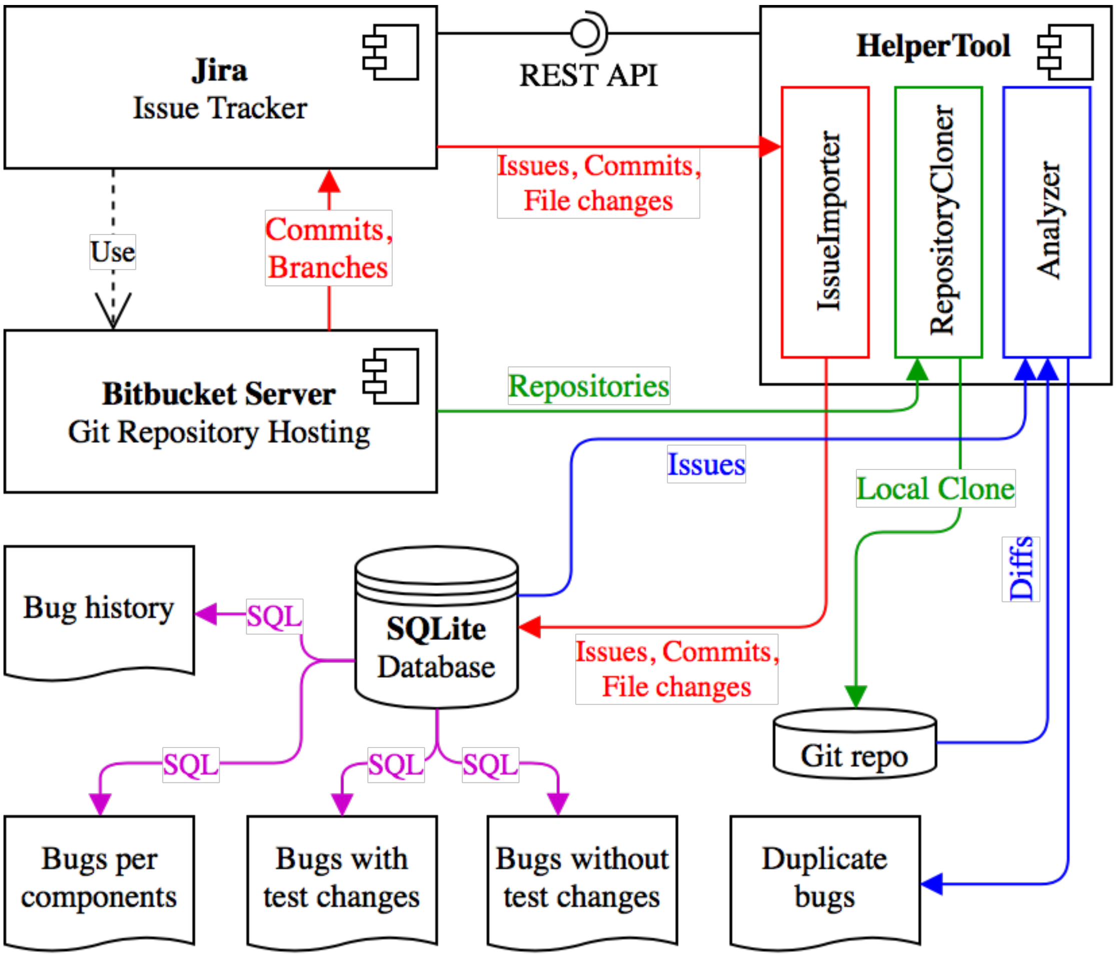


Figure 2.1: Setup for the analysis of the *EPOF* project. The colors indicate the data flow of the different kinds of analysis.

Unit tests This research project focuses on unit tests. Thus, the unit tests that are contained in the code base of the project are of special interest. It is necessary to be able to change and execute those tests to assess their quality. Futhermore, it is important to know which parts of the code the unit tests cover.

2.3.2 Collection of data

For this research project a tool was developed that collected some information about the *EPOF* project and stored them in structured form, which facilitated further analysis. A general view of the architecture of the tool, its environment, and the relevant data flows are shown in figure 2.1. The tool basically has three components.

The first component (*IssueImporter* in figure 2.1) uses the REST API provided by

Jira to collect all issues, all commits that belong to those issues and the file changes by these commits and stores them into a local SQLite database. The data flow of this process is indicated with red color in figure 2.1. Technical details of this part of the tool and the database schema that is used for the database are described in section 9.1.

The second component (*RepositoryCloner*) creates a local copy of all Git repositories that belong to the *EPOF* project by cloning them from the Bitbucket server. The data flow of this process is indicated with green color in figure 2.1. Technical details of this part of the tool are outlined in section 9.2.

The third component (*Analyzer*) seeks to find possible related bugs in the *EPOF* project. It uses information about commits and source code file diffs on method level and finally produces a list of suggestions of recurring bugs. The suggestions are then inspected and verified manually. The data flow of this process is indicated with blue color in figure 2.1. Technical details of the *Analyzer* part of the tool can be found in section 9.3.

Finally, some SQL queries are executed against the database to collect certain information about the bugs, the components they affect, and associated commits and file changes. This data flow is indicated with purple color in figure 2.1.

2.3.3 Intermediate results

Most information that could be gathered automatically from the database and the repositories consists of just intermediate results which require further manual processing. However, they provide some insight into the *EPOF* project. Table 2.2 shows some key information about the project. They were all retrieved by executing some SQL queries against the database. Some of the intermediate results in the table need some further explanation:

- “With commit” means that the association between issue and the commits which resolve it could be automatically retrieved from Jira API.
- “Without commits” means that the link between issue and commit could not be established.
- “Bug” and “Sub-Bug” are just special kinds of issues, thus the total number of issues includes bugs and sub-bugs.
- “With test change” means that at least one test file was modified during the bug fix. Test files were identified by their names: If a file’s name starts with “Test” or ends with “Test” or “Tests”, the file is considered to be a test file. This naming convention is well adhered to throughout the *EPOF* project with only few exceptions.

Category	Number of issues
Total number of issues	15’474
Issues with commits	2’009
Issues without commits	13’465
Issues with commits with test changes	347
Total number of Bugs / Sub-Bugs	6’531
Bugs / Sub-Bugs with commits	1’119
Bugs / Sub-Bugs without commits	6’531
Bugs / Sub-Bugs with test changes	122
Bugs / Sub-Bugs with commits without test changes	997

Table 2.2: Some information about the EPOF project with respect to issues and commits

There are two aspects to point out: First, it was possible to retrieve the commits that belong to an issue for 13% of all issues (or in absolute numbers: for 2’009 of 15’474 issues), which is a rather low fraction. When it comes to bugs, this fraction rises slightly to 17.1%. This research project focuses only on those 1’119 bugs for which the commit information is available. Hence, the validity of this study might be threatened because roughly 80% of the bugs are not taken into account in most of the analysis.

The second aspect to point out, and which is another threat to validity, is the fact that the linking between issues and commits, which is done by Jira, is not always complete or reliable:

- There might be commits that belong to an issue but are not linked to it. A typical example are commits that are committed after a bug fix was already merged back into the main development branch.
- There might be commits that are linked to an issue but in fact do not belong to it. A typical example for this case are merge commits from another branch, which are usually done before the bug fix is merged back into the main development branch.

The intermediate results presented above served as a starting point for further analysis, which is presented in the following three chapters. Each chapter deals with one of the research questions described in the introduction in chapter 1 and presents the different kinds of analysis that should provide answers to the research questions.

3
Inspection of Bugs

In section 2.3.3 the number of bugs with commits was presented. There are 1'119 bugs where the commits that fixed the bug could be gathered from Jira and Bitbucket automatically. To find an answer to the first research question, which covers the relation between unit tests and bugs, further analysis of the reported bugs is required.

3.1 Setup

Even though the commits, which resolved a certain bug could be gathered from Jira, it is a non-trivial task to comprehend the changes that were made by those commits. It requires manual inspection of the commits. Hence, a subset of 200 randomly chosen bugs was extracted from the 1'119 bugs and then the commits associated to those bugs were analyzed.

The analysis comprises two aspects. The first aspect is the test coverage of a bug. We checked whether the code part which caused the bug was covered by a unit test. If so, then we analyzed why these tests did not fail and, as a result, did not detect the bug. We also checked whether the developer modified, added, or removed tests during the bug fix.

The second aspect of the analysis was on the distribution of a bug fix. The main question here is how many files have to be changed for a bug fix. This information could be gathered by executing the SQL statement shown in listing 1 against the database. The query gets all file changes from the database, filters out non code file changes, groups them by bug, and finally counts them. The result is a list that contains for each bug fix the number of distinct changed files.

Listing 1: Query to retrieve number of changed files per bug fix

```sql
WITH distFileChangesPerIssue AS (
  SELECT DISTINCT
    i.Id AS `IssueId`,
    f.FilePath AS `FilePath`
  FROM Issues AS i
  INNER JOIN Commits AS c ON c.IssueId = i.Id
  INNER JOIN FileChanges AS f ON f.CommitId = c.Id
  WHERE
  (
    -- Restrict query to code files
    (
      FilePath LIKE '%.cs' AND
      -- Exclude auto generated designer files
      FilePath NOT LIKE '%.designer.cs'
    )
    OR FilePath LIKE '%.java' OR FilePath LIKE '%.ts' OR FilePath LIKE '%.m'
    OR
    (
      FilePath LIKE '%.js' AND
      -- Exclude bundles, map and minified files
      FilePath NOT LIKE '%.min.js' AND
      FilePath NOT LIKE '%.map.js' AND
      FilePath NOT LIKE '%.bundle.js'
    )
  )
  -- Exclude merge commits from main development branch
  AND c.Message NOT LIKE
    'Merge branch "develop" into %'
)

-- Number of unique code file changes per bug. Each changed file is
-- only counted once even if multiple commits affect the same file.
SELECT
  p.`KEY`, p.Name, i.Id, i.OriginalId, i.`KEY`,
  i.Summary, COUNT(dfc.FilePath) AS `FileChanges`
FROM Issues AS i
INNER JOIN IssueTypes AS t ON t.Id = i.IssueTypeId
INNER JOIN Projects AS p ON p.Id = i.ProjectId
INNER JOIN distFileChangesPerIssue AS dfc
  ON dfc.IssueId = i.Id
WHERE t.Name IN ('Bug', 'Sub-Bug')
GROUP BY i.Id
ORDER BY p.Id, i.Id
```

3.2 Results

3.2.1 Bugs in defective classes covered by tests

In only 17 out of 200 analyzed cases, the defective component had associated unit tests or tests that were added during the bug fix:

- In seven cases, tests were added to verify the correctness of the bug fix or to improve the overall coverage of the component. In the later case, the tests which were added did not necessarily cover the scenario that caused the bug to occur.
- In five cases, there existed tests for the defective component, but they did not cover the scenario in which the bug occurred. Furthermore, in those five cases, the tests were not extended during the bug fix to cover the problematic scenarios.
- In two cases, the tests were changed because some part of the code was refactored (*e.g.,* renaming of a method or a property) without changing its functionality.
- In one case, a test was introduced in an early commit of the bug fix and later removed again because it became obsolete.
- In another case, the tests were commented out during the bug fix. And they are still commented out in the current version of the code base. It seems like the developer forgot to either uncomment or remove them.
- In one case, the bug was misclassified as defect. In fact, it was a new non-functional requirement which was requested by the security responsible.

These observations provide some interesting insights: The first insight is that only few (in absolute numbers: 17) defective components are covered by unit tests. That goes along with the fact that the overall test coverage of the code base is very low. Section 5.2 will present a possible explanation why the test coverage is low.

The more interesting insight is that there exist classes that were partly covered by tests, but the part of the classes that later caused a bug was not. The question here is, which parts of the code should be covered by tests, and which parts should not. Usually, a developer must decide for which parts of the code she invests time into writing unit tests. To make this decision, she needs to take different factors into account like the criticality and importance of the code, and its testability.

Another insight is that in three cases tests are just changed because the compiler complained. The developer changed the code in a way that the test code does not compile anymore. To make the tests run and pass again, the developer has to modify them. However, the developer can just comment the tests out, which is done in one case. But if a test ever gets commented out, it is likely that this action will never be reverted. The commented test is not maintained anymore and becomes useless. So it might be a better idea to just remove the test. If it happens that the test is needed again later, it can be recovered from the version control system.

3.2.2 Number of modified files per bug fix

Figure 3.1 shows a histogram of the number of files that were changed for a bug fix. The x-axis shows the number of changed code files while the y-axis indicates the number of bugs. Consider the following reading examples: For 14 bugs, it was necessary to change 3 different code files to fix them. For 9 bugs, it was necessary to change 6 different code files to fix the bug.

A special case, which requires further explanation, are those 20 bugs where no code file changes were required to fix them. In those 20 cases, there was no need to change source code to fix those bugs. But it was necessary to change other kinds of files like configuration or resource files, which were excluded from the analysis.

There are two interesting insights about the number of changed files per bug fix. The first insight is that in 71 of 200 cases (*i.e.,* 35.5%), only one code file is modified to fix a bug. This means that quite often a bug just affects one class or component. The second insight is that in 109 of 200 cases (or in 54.5%), more than one code file is changed to fix a bug and, hence, the bug affects multiple classes or components of the software system.

The first insight suggests that those 71 bugs could be detected by unit tests that test only a single component. However, the second insight shows that those 109 bugs, which affect multiple classes, might not be detectable by simple unit tests. Those 109 bugs are rather integration issues than defects in a single component. Even though unit tests might reveal some of those integration issues, it is often necessary to write integration tests to catch such bugs in more complex scenarios. Hence, a possible advice for developers is that they should first focus on unit tests but also think about integration tests.

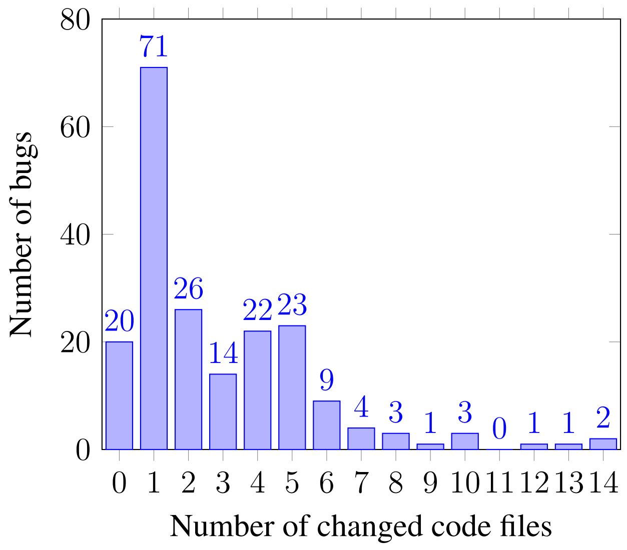


Figure 3.1: Histogram showing the number of changed files per bug fix. $n = 200$

4
Effectiveness of Unit Tests

The main goals of unit tests are the detection and prevention of bugs. The tests should fail if a developer makes a change to the system that breaks existing functionality. The problem is that just the existence of tests in the source code is not sufficient. There might be tests that do nothing useful and, as a result, do not help the developer to find and prevent bugs. Hence, the question is whether the existing tests are useful and prevent the introduction of bugs.

4.1 Recurring bugs

Software systems nowadays are very complex in structure. They have many interdependencies between their components. Therefore, it is not unusual that a certain bug occurs again after it was supposedly fixed in a previous release. Unit tests offer a way to avoid such recurring bugs by documenting a previous bug and fail if the same bug occurs again. Hence, it is reasonable to check whether there are any recurring bugs in the *EPOF* project.

4.1.1 Setup

We implemented a tool that reports possible recurring bugs. We describe it in detail in section 9.3.1. The idea behind it is sketched below:

1. Get a list of bug fix commits that changed a certain source code file. Order them

by date and group them by file. This list can be gathered with a query against the database.

2. For each file entry in this list, detect how each associated commit changed that file. The detection produces a list of methods that were changed by the commit.

3. Compare the list of changed methods with respect to the different commits that affected the same file. If two commits changed the same method, they might be related.

4. If related commits belong to different issues, those issues might be similar and, hence, are candidates for being recurring bugs. Such pairs of issues are added to a suggestion list.

5. Return the suggestion list.

The algorithm only considers changes to the method's body. The following other kinds of changes to methods are not taken into account because they are hard to detect: renaming, signature changes, removal or addition of methods.

The suggestions returned by the algorithm were then inspected manually in order to check which ones are really recurring bugs. During the inspection, the following information was considered to verify whether a suggestion is a recurring bug:

- The **description of the issues**: If the descriptions are similar or keywords like "regression" or "occurred again" are mentioned in the description, it might be a real recurring bug.

- The **code changes made during the commits**: If the changes are within the same lines of code within the method, or if the changes are similar, this might be an evidence that it is actually a recurring bug.

- The content of the **commit message** might also give some evidence.

For every suggestion, the reason why it was classified or not classified as recurring bug was recorded.

4.1.2 Results

The list of suggestions of possible recurring bugs, which was returned by the tool, contained 183 entries of which 35 suggestions were detected in C# code files and 148 in Java code files.

Of those 183 suggestions, only 8 pairs of bugs were classified as real recurring bugs or more precisely as similar bugs. We now present the details.

4.1.2.1 Identified recurring bugs

As mentioned above, there were 8 pairs of bugs that were identified as being recurring bugs. Those 8 pairs are described in detail in the following paragraphs.

Navigation problems One pair of bugs describes problems with the navigation within the mobile application. After a certain action was taken by the user, the application navigated to another screen which was the wrong one. Both bugs reported similar navigation mistakes in two different parts of the app. The bug fixes changed the same part of the code, which is not covered by unit tests. The defective code is tightly coupled to GUI components and is triggered by certain events which are hard to simulate in a unit test.

Screen orientation problems Another pair of issues both describe problems with the orientation of the screen of the mobile device. When the mobile device was rotated, the screen rotated into the wrong position. After the first bug was fixed, another problem with the screen orientation was reported, and the same code needed to be changed again to solve this second problem. The affected part of the code was also not covered by unit tests. The screen rotation depends on the sensors of the device which are hard to simulate. Furthermore, it is hard to check whether the orientation is correct in a unit test. Although detection of orientation is an easy task for a human (who needs just one glance at the screen), it is a non-trivial task for a computer.

Concurrency problems A third pair of issues describes concurrency issues that were only partly fixed by the first commit and needed final corrections with the second commit. Such concurrency issues are usually very hard to reproduce in a unit test because their occurrence is unpredictable.

Missing null checks There are three pairs of issues that are all caused by missing null checks within the code. In those three cases, the second commit added a necessary null check that had not been added with the first commit. In all three cases, there were no tests covering the defective component. In two cases, the testability of the defective component was low, but in one case it would have been easy to write a test that would have caught the bug.

Report generation problems One pair of issues is associated with the creation of a report. The corresponding source code performs some calculations and finally creates a report containing calculated key numbers. Both issues stated that the calculation of some key numbers was not correct. The second issue was reported two months after the first issue. The problem could have been avoided if the calculation of the key numbers would have been covered with a unit test. But there was no test written even after the bug occurred for the second time. The reason might be that it is not easy to write a test for this part of the code: It heavily draws on data from the database, and the calls to the database are spread all over the report generation code. Providing suitable test data and injecting them into the test requires hard work and is impeded by the code's architecture.

Password validation problems The last pair of related bugs concerns password validation on a form. The form contains a field for the password and a second field for repeating the password. The first issue reports that the password should only be validated after both fields are filled in. Some time after this issue was fixed, another issue reported that the validation does not fire anymore at all. The second commit then added a single line to the code that should have been added before with the first commit. There is no test case covering the defective component because the component has dependencies to other parts of the application and therefore it is not easy to set up a test for it.

The results above show that there are only few bugs that occurred multiple times. That might be surprising considering the fact that large parts of the code base are not covered by any unit tests. All those bugs were detected by testers who performed manual tests. Usually, the bugs were discovered quite quickly and before the release was rolled out to the customers.

The question that comes to mind is why there were no tests added after the first bug had been discovered, or at latest when a bug was discovered the second time. There are multiple possible answers to this question.

One answer is based on the fact that it is usually very hard to add tests. Many parts of the code are designed in a way that it is really hard to test single components in isolation. As a consequence, the effort required to add a test is very high and, hence, no unit tests were written. This aspect is discussed in detail in chapter 5.

Another reason why no tests are added might be the fact that some of the issues described above are usability issues that are hard or even impossible to catch with a simple unit test.

And a third possible explanation is that the developer was unaware of the fact that a specific bug had occurred before. As a consequence, she does not pay special attention to it.

4.1.2.2 Non-recurring bugs

The algorithm returned 175 pairs of possibly related bugs, which were later classified as non-recurring ones. But those 175 "false positives" were interesting as well. Assume that the algorithm reports the following pair of bugs (Bug_A, Bug_B), and further assume that the bug fix that resolved Bug_A only changed the method $methodY()$ of class $ClassX$. Then, because the algorithm reports the pair (Bug_A, Bug_B), this means that the bug fix that resolved Bug_B also changed $ClassX.methodY()$. So, even though Bug_A and Bug_B are later classified as non-recurring bugs, they are at least related because they affect similar parts of the code. Further inspection of those 175 pairs revealed the following findings:

- Many of those pairs are reported because some refactoring during one bug fix changed parts of the code that were also changed by the second bug fix. But the second bug was not caused by the refactoring.
- Another reason why bugs were reported by the algorithm are code formatting changes. Such changes are often done by the IDE automatically when the developer opens or closes a file, and often the developer is not aware of those changes.
- A third category of reported related bugs are not recurring bugs because there was a long time between their discovery. In that time, the common component may have changed considerably because new functionality was added. Hence, the two bugs might affect the same component, but that component changed so much between the discovery of those two bugs that it is in fact not the "same" component anymore. Thus, the two bugs are not related.
- The algorithm reports some pairs of bugs because their bug fixes both changed a special class, which centralizes a specific cross-cutting aspect of the system. An example for this is the registration of a component into a dependency injection (DI) container, which is done in one single method in a single class for the whole system. If a bug fix needs to register a new component, it changes that method. But if a second bug fix adds another registration, it does not mean that those bug fixes are related even though they changed the same method.

These results show that even though two bug fixes may change similar parts of the code, this does not mean that their bugs are related. This is a contradiction to the well-known single responsibility principle (SRP). SRP states that a class should have only one single responsibility. Given the assumption that a class follows the SRP, then two different bugs that affect that class must be related. But in the *EPOF* project, there were many bugs that affected the same class but were not related. As a consequence, there must be some violations of the SRP in the *EPOF* project. This issue is discussed in chapter 5.

The results above raise another question: Why are there components that are changed by multiple different bug fixes, and hence are more prone to bugs, than other components? This question is answered in section 4.2.

4.2 Proneness to bugs

The results in section 4.1.2 suggest that not every part of a software system is equally susceptible to errors. There are some parts that contain more bugs than others. In general, a component's proneness to bugs might also change over time. In an earlier stage of the development, more bugs might be discovered, and, thus, the component may change often before it gets more stable in a later stage of development. The analysis presented in this section deals with different aspects of the *EPOF* project's proneness to bugs.

4.2.1 Setup

The information within the issue tracker served as starting point for every analysis in the following sections.

Bug report history All bugs in the *EPOF* project are reported using the Jira issue tracker. Jira tracks the dates when a bug was discovered, and when it was fixed. This information was gathered from the issue tracker and used to plot a timeline and to analyze the time it took the developers to fix a bug. This analysis should reveal whether there were times when significantly more or less bugs were discovered and whether there were changes in the bug resolution duration.

Bug fix commit history The information stored in the database could be used to gather bug fix commits. The commits were then grouped by component and plotted on a timeline, which allowed visual inspection of the development history. The resulting plot was analyzed to find conspicuous features within the development history of the system's components.

Bug fixes per component Using the information in the database, it was possible to get the number of bugs that affected a certain component of the *EPOF* system. This information was used to create a ranking of the components by their number of bugs. For the top 15 of the most error-prone components, the developers were asked why these components seem to be more susceptible to bugs than others.

In chapter 3, a subset of 200 bugs was further analyzed to identify whether a defective component is covered by a unit test. This information was reused for another analysis in this chapter: We checked how many bugs affected the components that are covered by a test. The same analysis was done for the components that are *not* covered by a test. The results were then compared to each other, and we checked whether there were some differences between the tested and the untested components.

4.2.2 Results

4.2.2.1 Bug report history

Figure 4.1 on page 24 shows when bugs were reported during the lifetime of the *EPOF* project. The vertical axis shows the number of bugs, and the horizontal axis shows the time. The red dots show the total number of reported bugs at a given point in time. The green dots show the number of resolved bugs at a given point in time. The gray vertical lines show the releases of the *EPOF* project. There was a change in the version numbering between version 2.5 and 17.10: In 2017, the team switched from incrementing version numbers to the schema *Year.Quarter*. The release dates of release 2.2 and 2.3 could not be determined anymore. That is the reason why these releases are missing in the graph. The slope of the red line indicates how frequently bugs are reported. If many bugs are discovered in a short time period, the slope is very steep. The vertical distance between the red and the green dots represents the number of open bugs at a given point in time. If the distance between the red and the green line grows, the number of open bugs in the system increases. If the distance shrinks, the number of open bugs decreases.

There are three observations to point out: Shortly before and shortly after the releases 2.10 and 2.40 the slope gets steeper which means that more bugs are reported in a short period of time. The same pattern occurs again around all 17.XX releases. Members of the development team were confronted with this observation. They said that this might be due to the fact that there were testing phases before the releases, which revealed many bugs. And right after the releases, the application and the application's error log were monitored closely and more bugs were discovered. Those issues were then resolved with a hotfix if they were critical.

The second, and more interesting observation, is the bend of the red line after release 2.5 at the end of the year 2016. This bend indicates that after this release, fewer bugs were discovered. Members of the development team explained this clear bend with drastic changes of the project's organization. In the time between release 2.10 and release 2.60, the developer team was under permanent pressure to add more features. Furthermore, the communication between the customer and the development team was not very effective. The team had already started implementing new features before all requirements were clear. This led to misunderstandings between the developers and the customers and resulted in many bug reports, which in fact were changes of the requirements. There was also a strong staff fluctuation. As a consequence, the project team neglected testing and quality assurance activities, which resulted in many bugs that got into production. To prevent the project from failure, some measures were taken at the end of 2016. In the time after December 2016, the team improved the SCRUM process and followed it more strictly. They improved the quality of communication between the customer and the development team so that it is more open and better structured. Implementation of a user story did not start before all necessary information was available. Furthermore, the responsibilities of the developers were refined, and quality assurance, especially automated unit testing, was given higher priority. The introduction of the test coverage goal in 2017 was one concrete measure that is shown in the figure. Those improvements of the development process took some time. But as a consequence, the quality of the product improved during the releases 17.XX and 18.XX, which is confirmed by feedback of the customers and developers as well. Thus, the observed decreasing number of reported bugs in figure 4.1 confirms that the described measures had a positive effect on the *EPOF* project. Because the test coverage goal was just one of many measures to improve code quality and the development process, it is not possible to distinguish its effects from those of the other measures. It is therefore not possible to say whether the test coverage goal alone had a positive effect on the code quality or the number of reported bugs.

The third observation concerns the number of resolved bugs: Most of the time the green line closely follows the red line. This means that the total number of open bugs in the system at a given point in time remains more or less constant. Thus, the developers were fixing bugs during the whole lifetime of the project. There was no period in the development history during which the developers neglected bug fixing. But after November 2016, the green line and the red line take different paths. The number of open bugs slightly increases: The average number of open bugs was 210 before November 2016 and 249 afterwards. The development team explained this with a “cleanup” measure: They observed that there were some bugs in the system that were neither reported by the quality management team nor by the customers but by the developers themselves. Those bugs usually were not critical (*e.g.,* log messages that were classified as error but in fact were just warnings), but the developers found them annoying. In spring 2017, the team started to systematically collect and report such bugs. Those additional bug reports might explain the increase of open bugs after November 2016.

The number of open bugs in a system is just one key number related to bugs within a project. Another important key number is the time it took the developers to fix them. Figure 4.2(a) on page 26 shows the results of this analysis. Each bug was assigned a unique number as an ID. The x-axis in the figure then shows the ID of the bug. The bugs are ordered by the date they were reported. So for example, the bug with ID 1 was reported before the bug with ID 45. The y-axis shows the number of days it took the developers to fix a certain bug. Table 4.2(b) shows some statistical results of the data. The statistics were evaluated for the whole lifetime of the project, the time before and the time after November 2016. In November 2016, the major process transformations happened, which should prevent the project from failure. While the number of reported bugs decreased after the transformation, there were no clear changes in the duration of bug fixes. So, it seems that the measures to ensure the quality, and the enforced usage of unit testing do not have an impact on the bug fix duration as of spring 2018. However, linear approximation of the data (indicated by the black line in figure 4.2(a)) shows a

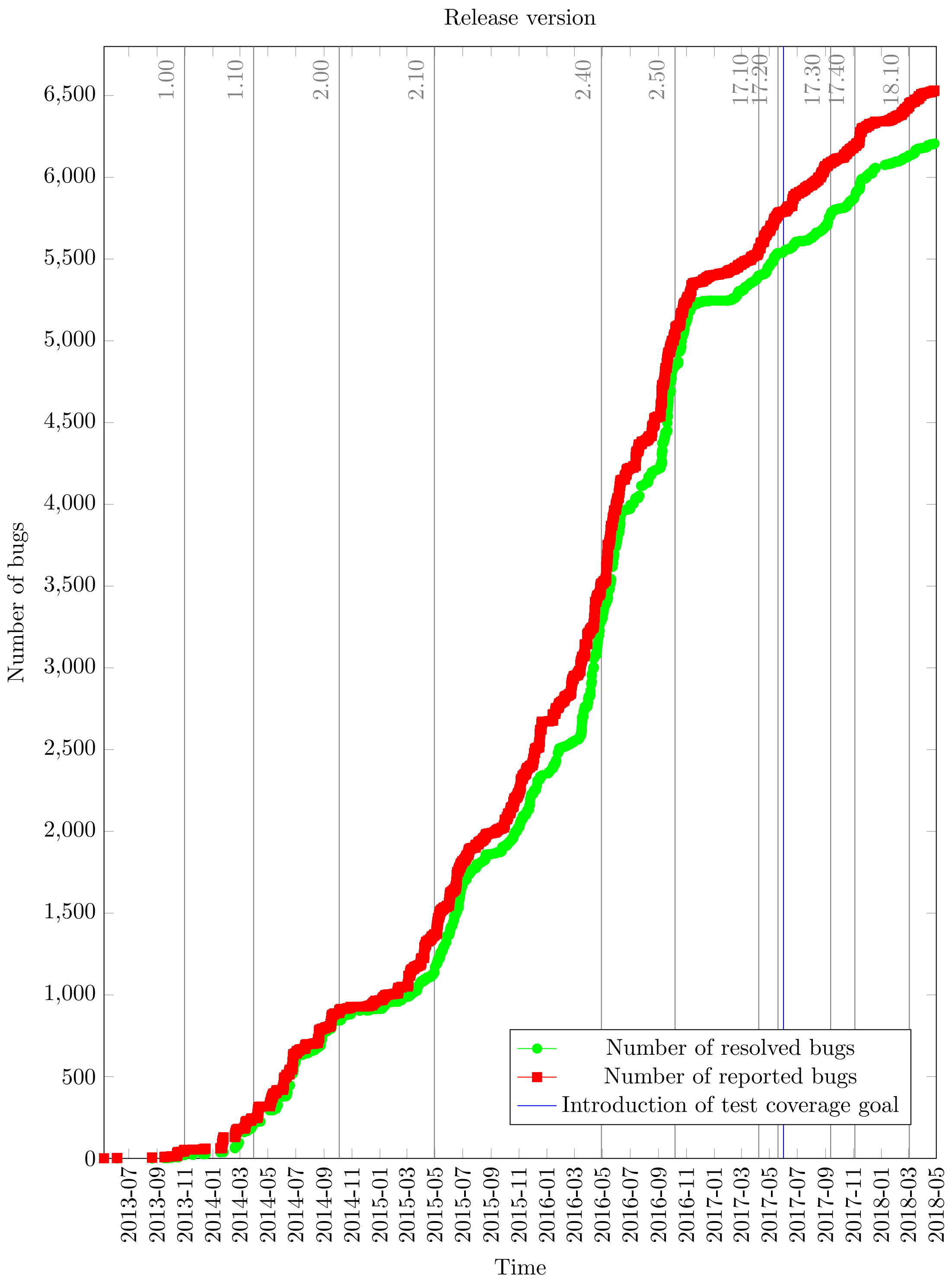


Figure 4.1: Discovery of bugs over time

slightly decreasing trend. But in absolute numbers, the average fix duration of 30 days is still rather high.

4.2.2.2 Bug fix commit history

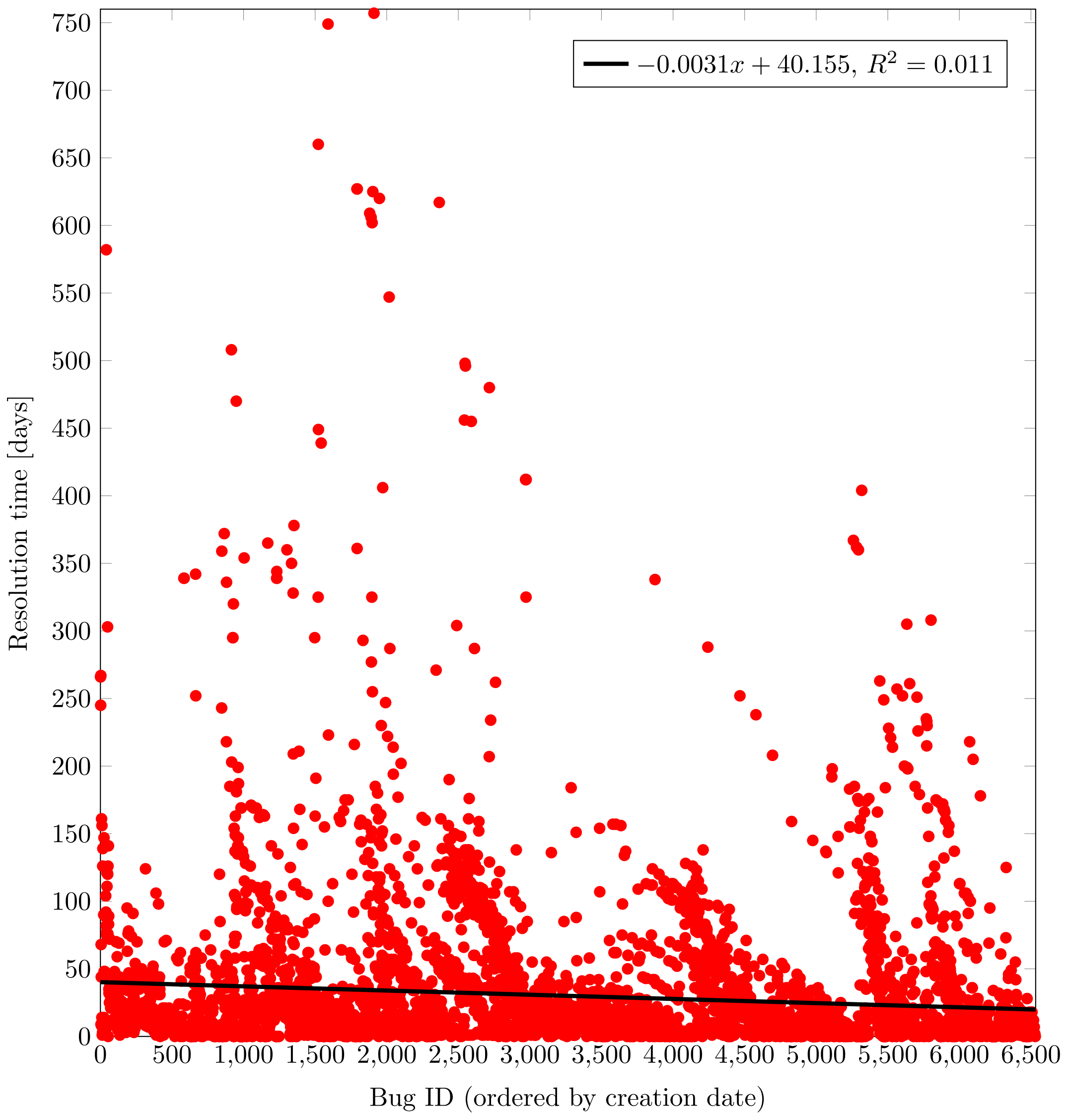
Figure 4.3 on page 28 shows a timeline of all commits of the *EPOF* project that were made in order to fix a bug. The horizontal axis shows the time. The vertical axis shows the code files which were assigned a unique ID. Whenever there is a colored mark, this means that the file was changed by a commit at that point of time. All marks on the same vertical height represent changes to the same file. The shapes of the marks are used only to distinguish neighbouring files. The colors of the marks encode the number of the commits per file: The first change to a file is marked yellow, the second a little more green, the third one even more green and so on. The vertical gray lines show the release dates of the *EPOF* project.

Two areas in the graph are shown in figures 4.4(a) and 4.4(b) and discussed below.

In the first interesting part in figure 4.4(a), there are multiple files that are changed by many commits as indicated by the green marks. Namely the files with ID 30, 85, 90, 91, 114, and 123 (marked with gray dashed lines in figure 4.4(a)). Furthermore, one can see that the files 90 and 91 are often changed at the same time. The same observation holds true for the files 114 and 123. Further inspection of those pairs of files revealed that they are closely related: Both pairs of files are part of the Android mobile app, and all files are part of the payment functionality of the app. Hence, this part of the app seems to be more prone to bugs than other parts. The files with ID 30 and 85 both deal with the display of documents in the Android mobile app, which is one of the core functionality of the app. The developers confirmed that those 6 files are rather large and play an important role within the app. They said that those components are likely to be affected by multiple bugs because of their size and complexity. The developers also confirmed that adding tests for those components is hard due to their complexity.

The second interesting part is shown in figure 4.4(b). There is a gap in the development history. In the time between December 2016 and February 2017, no bug fix commits were recorded (note the vertical broad white “gap” in the graph). This break in development history might partly be explained by the shortcomings of Jira to associate commits and bugs, but the bug history shown in figure 4.1 on page 24 shows that there are also fewer bugs reported in this time span. Team members confirmed that there was less development activity in this period because the transformations of the development process and the reorganization of the developer team happened in this period.

There is a third interesting observation in figure 4.3: The bug fix commits before and after February 2017 happen in different parts of the system. Most changes to the files 1-350 are done before February 2017 (with few exceptions). The files 350-800 are all changed after February 2017. Further inspection of those two file groups revealed that most of the files in range 1-350 are part of the Android mobile app whereas most of the



(a)

	2013-2018	before Nov 2016	after Nov 2016
Number of reported bugs	6531	5237	1294
Average fix duration [days]	31.7	31.5	32.8
Median fix duration [days]	14	14	11
Maximum fix duration [days]	757	757	404
Minimum fix duration	same day	same day	same day

(b)

Figure 4.2: Time between the reporting of a bug and its resolution.

files in range 350-800 are part of the *EPOF* back-end. It seems that there was heavy development of the Android app in 2016 and then, starting in 2017, the team focused on bug fixes in the back-end. The developers explained this change with the fact that the development of the Android app was temporarily suspended as of the end of 2017. Furthermore, the architecture of the back-end was changed slightly: New features should be implemented as micro-services to enforce low coupling and separation of concerns. Thus, the back-end was changed often in this time which resulted in the discovery and fixing of bugs.

One of the measures that were introduced in 2017 was the test coverage goal for new parts of the system. This measure might explain the fourth observation that there are no components that are changed excessively often after January 2017. Such components would be shown with green commit marks, but nearly all commit marks in 2017 and 2018 are yellow, which means that most components are only changed by few commits.

4.2.2.3 Bug fixes per component

Figure 4.5(a) and table 4.5(b) on page 31 show how many components are changed by which number of bug fixes. Consider the following reading examples: 465 components are changed by only one bug fix. This means that they are affected by only one reported bug or at least changed only once during a bug fix. 229 components are changed by 2 to 4 bug fixes.

The results show that of the total 793 components that are changed by at least one bug fix, 58.6% (in absolute numbers: 465) are changed by only a single bug fix, and 41.4% (328) by more than one bug fix. 15 components are changed by at least 15 bug fixes, which might indicate that those components are more error-prone than other components.

The developers who made most changes to those 15 components were asked whether they can explain why those components are changed so often during different bug fixes. They stated that most of those components play a central role in the application and are involved in many different tasks. They contain large parts of the business logic of the application and also GUI-logic. So, they are likely to be changed often. Another reason for the numerous modifications are changes in the environment. One example is the usage of the fingerprint API of Android, which was used shortly after it was introduced. So, it took some time to use it right within the app. Another example is the API of the EPOF back-end, which also changed multiple times and, as a consequence, some errors occurred in the Android app.

The developers were asked whether they considered writing unit tests for those critical parts of the application. They said that many components of the app are designed in a way that they require a complete Android environment to run in order to test them. Hence, it is hard to write unit tests for the app and hard to test only parts of it in isolation. But starting in 2018, the Android developers introduced the Model-View-Presenter (MVP) pattern to decouple the GUI and the business logic. This allows unit testing for newer

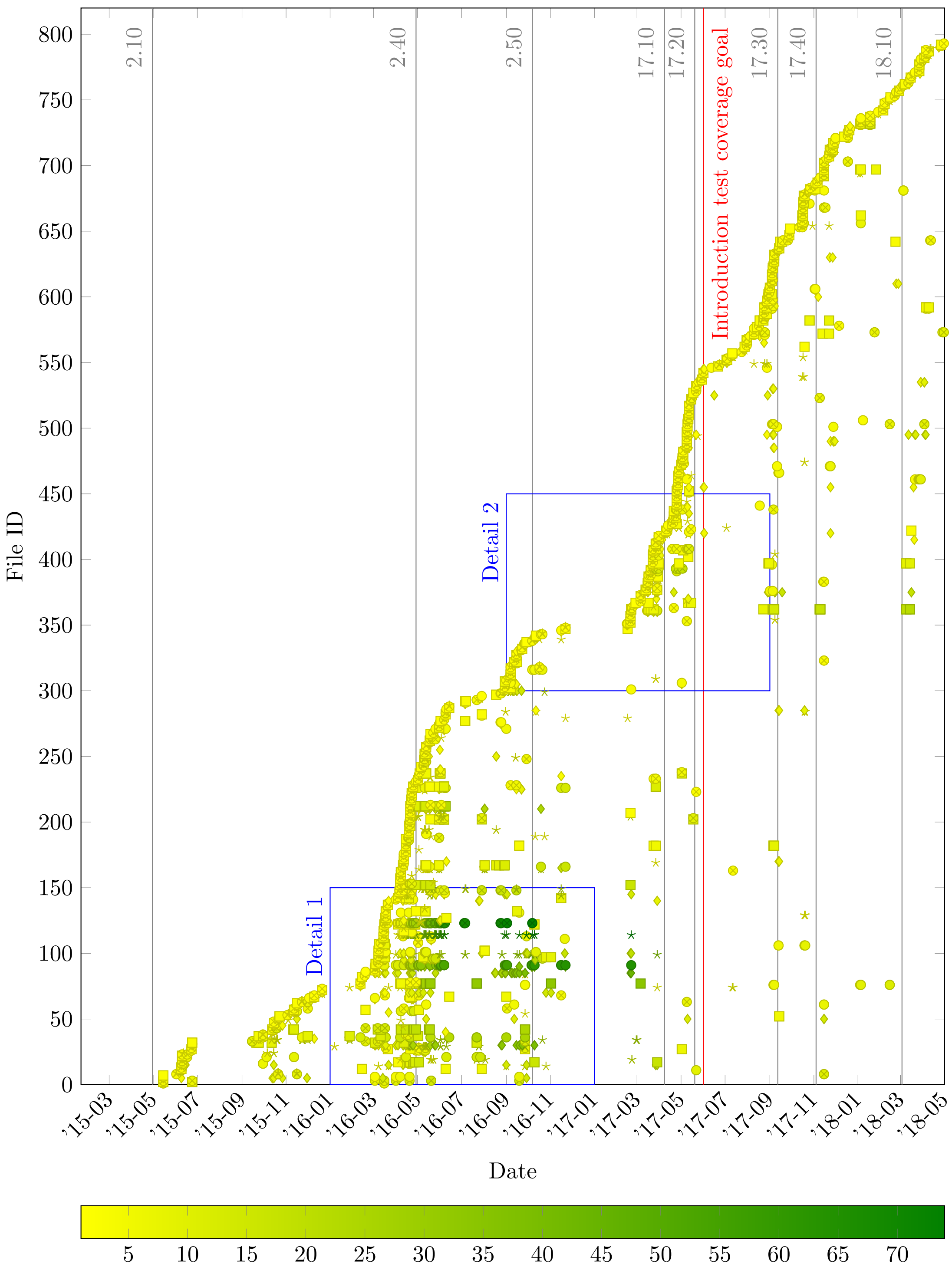
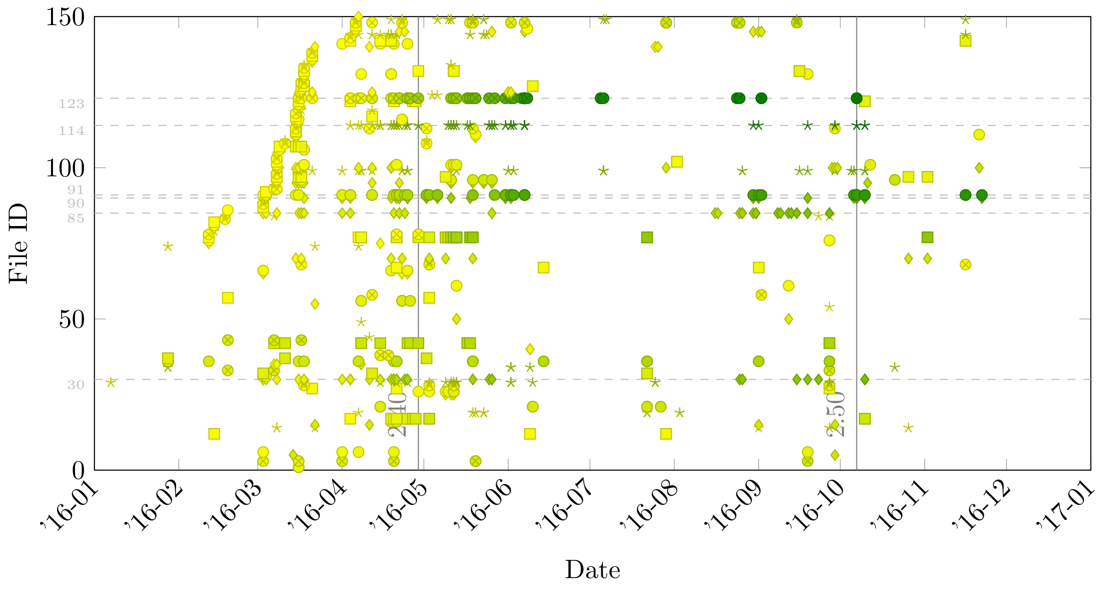
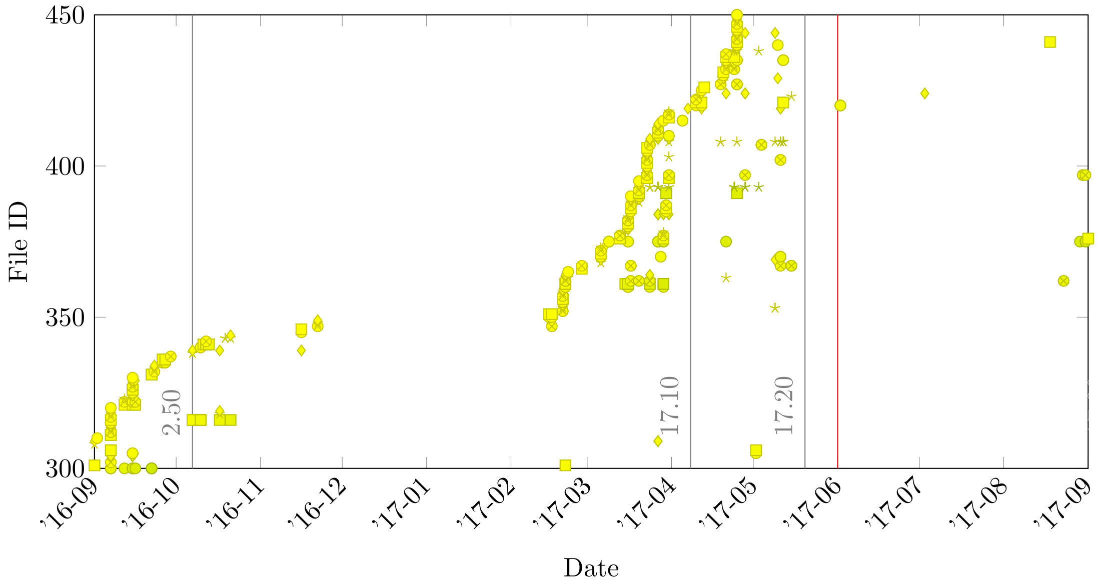


Figure 4.3: Bug fix commits to files over time



(a) Detail 1

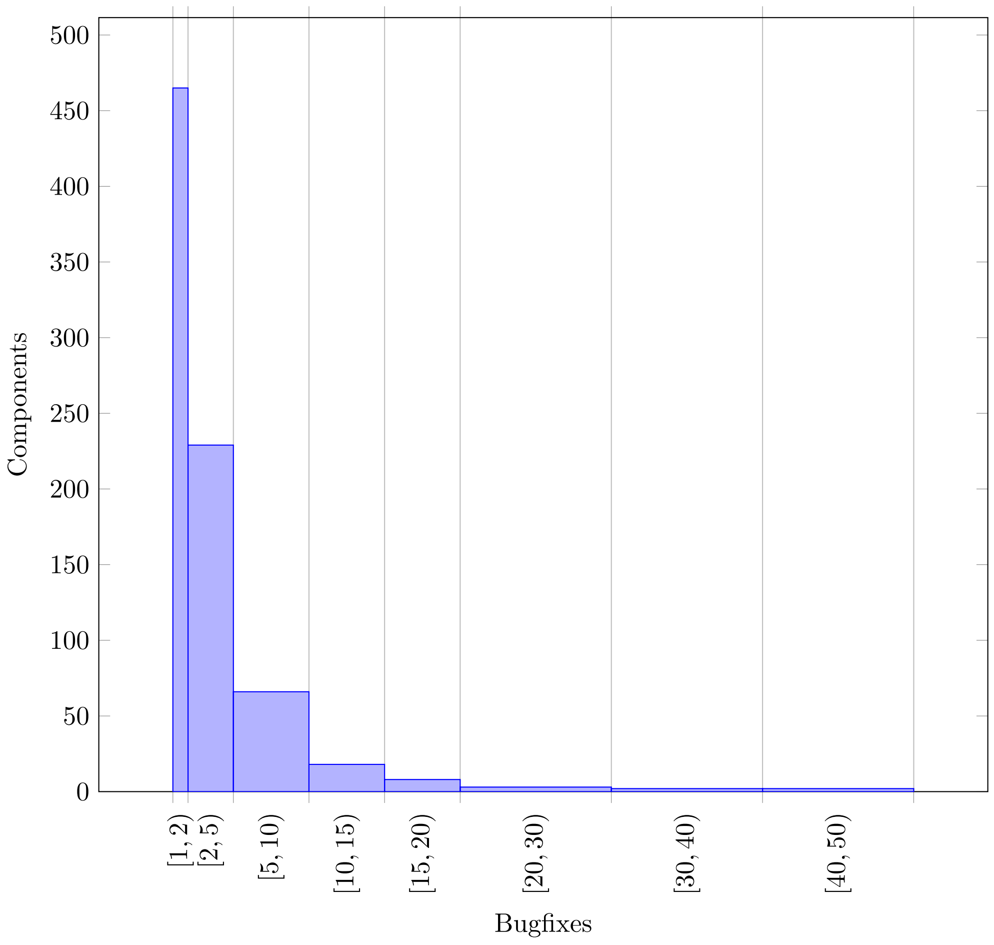


(b) Detail 2

Figure 4.4: Details of figure 4.3

parts of the Android app. It is planned to refactor older parts of the application, but due to time and budget constraints, this task has not high priority.

In chapter 3, two hundred randomly chosen bugs were analyzed in depth. From all 244 components that were affected by those bugs, there were 8 components that are covered by unit tests. It was then evaluated how many bugs affected those 8 tested and the remaining 234 untested components. The results are shown in figures 4.6(a) and 4.6(b) on page 32. The graphics are the same as in 4.5(a), but only the components that are affected by the 200 sampled bugs are considered. The figures reveal that most components under test are affected by 5 or less bugs. This suggests that tested components are less prone to bugs than untested ones. However, because the number of tested components in the sample is relatively low, this conclusion is not very reliable. Furthermore, there is one tested component that is affected by 19 bugs. That component handles the fingerprinting feature via a new Android API, and it took some time for the developers to use it in the right way. So, there is at least one component where the tests are not able to prevent bugs.

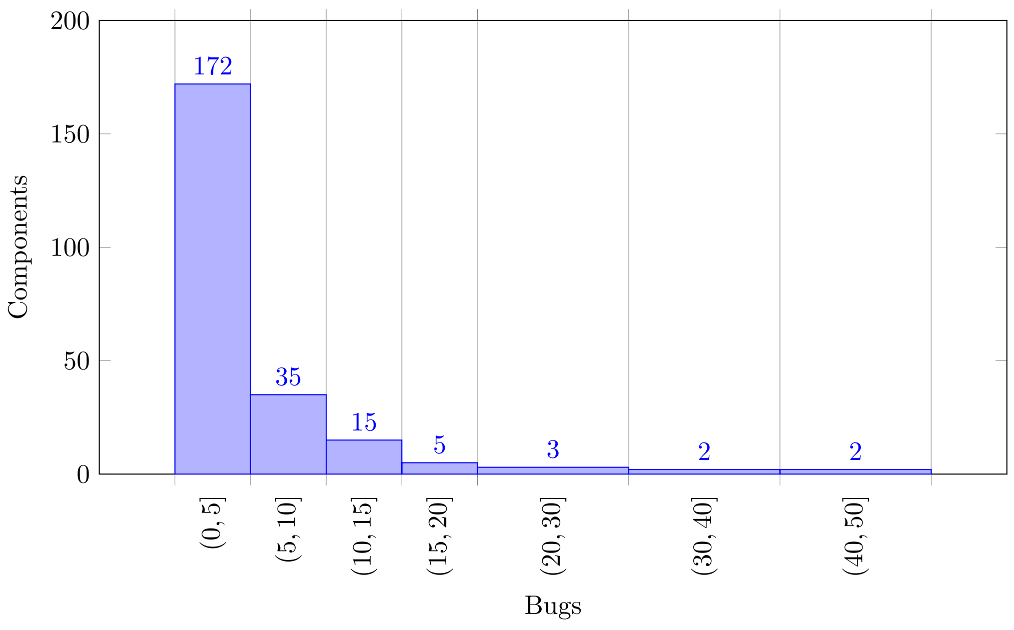


(a)

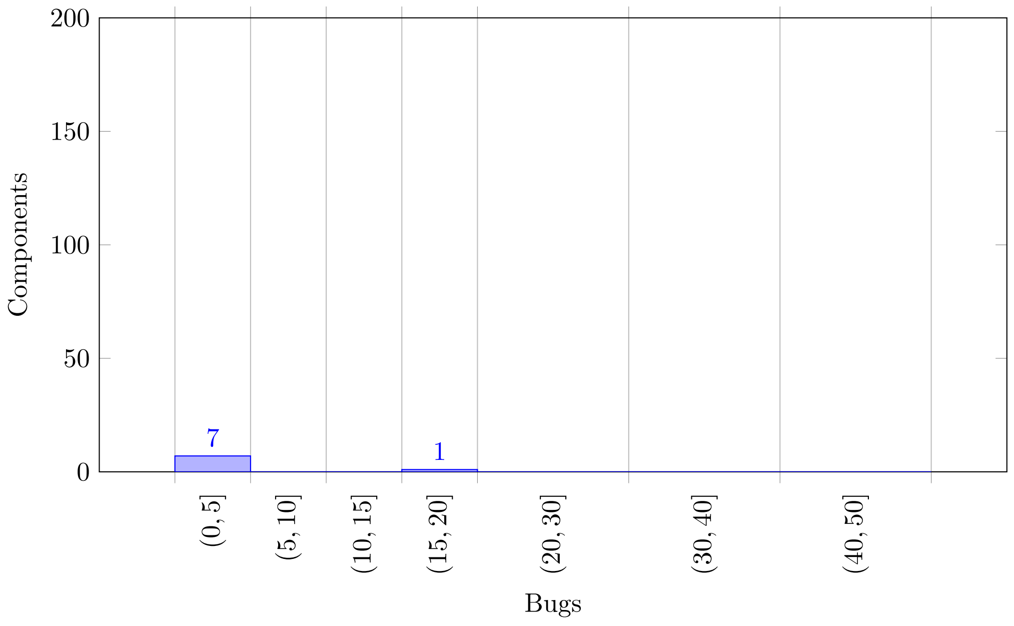
Bug fixes	Components
1	465
2-4	229
5-9	66
10-14	18
15-19	8
20-29	3
30-39	2
40-49	2
50-	0

(b)

Figure 4.5: Bug fixes per component



(a) Bugs per component for untested components, $n = 234$



(b) Bugs per component for tested components, $n = 8$

Figure 4.6: Bugs per component

4.3 Mutation testing

Mutation testing is a method to assess the quality of unit tests. The key idea behind it is to create multiple copies of the production code (called "mutants") and include some small changes into their code. Then, the existing test suite is run on those mutants and checked how many tests fail. If one or more tests fail, the test suite was able to detect the changes (the tests "killed" the mutants) and, hence, the quality of the test suite is good. If no tests fail, the changes were not detected and, as a result, the tests might also fail to catch real bugs.

Mutation testing is an expensive process, mainly because the process involves test suite execution for every mutant. There are multiple techniques available that should speed up the whole process:

One technique called "goal-oriented mutation testing with focal methods" restricts the number of methods that are mutated. This approach was proposed by Vercammen *et al.* [31]. The idea behind it is to make a connection between a unit test and the actual method it tests (the "focal method" under test). During the mutation testing process, only those focal methods are mutated.

Another technique to speed up the mutation testing process is the restriction of the set of applied mutations. In a project that performs many numerical calculations it is generally more interesting to apply mutations to the numerical operators (*e.g.,* replace $+$ with $/$ or $-$ with $*$) whereas in a project with many logical decisions, it is more useful to replace $true$ with $false$ or $=$ with $\neq$.

The test coverage in the *EPOF* project is rather low. But the test coverage is a key number with a low expressive power when it comes to assessing the quality of the test suite. Hence, mutation testing was applied to parts of the *EPOF* project to get some information about the quality of its unit tests.

4.3.1 Setup

There exist different tools to automate the mutation testing process. VisualMutator [4] and NinjaTurtles [20] are two tools that are available for C# projects. However, neither of them are used on a large scale, and NinjaTurtles is not actively developed anymore. Both tools were tested together with the *EPOF* source code. But neither was able to deal with the large size of the code base and, therefore, both tools failed to create mutants. Hence, the mutation testing was done manually. To keep the manual work at a reasonable level, the mutation testing was only applied to parts of the code where bugs were located and that were under test control.

15 classes matched this criterion. But four of those classes did not exist anymore in the current version of the source code and, for different reasons, it was not possible to execute tests on older versions of the project. For one class, there were tests that still existed in the latest version of the code but were commented out.

Hence, mutation testing was applied to the 10 remaining classes. Both optimizations described above in section 4.3 were used to speed up the whole process:

Focal methods under test were identified for every test that covered the 10 selected classes. Ghafari *et al.* [9] presented a technique to perform this linking between test and production code automatically. However, for C# projects there is no implementation available. As a consequence, the identification of focal methods under test was done manually.

Furthermore, because most of the selected classes perform boolean operations, if-else-decisions, and numerical calculations, the set of applied mutations was restricted to the following operations:

- Replace $==$ with $! =$ and vice versa
- Replace a boolean expression with its negation
- Replace $true$ with $false$ and vice versa
- Replace $+$ with $/$ and vice versa
- Replace $-$ with $*$ and vice versa
- Replace $+ =$ with $- =$ and vice versa
- Replace $++$ with $--$ and vice versa

Only one mutation was applied at a time. As such, each modification yielded a new mutant. The mutant was then compiled, and afterwards the tests covering the mutated code were executed.

4.3.2 Results

Table 4.1 on page 36 shows aggregated results of the mutation testing process while table 4.2 shows more detailed results of it. The results are grouped by the components of the *EPOF* projects that the mutated classes belong to. The score in table 4.1 is calculated by dividing the number of killed mutants by the total number of mutants.

The lowest score is achieved by the component *PkPortal*. This is in line with the fact that this component is a large monolithic subsystem with very low test coverage. *PkPortal* was the starting point of the whole *EPOF* project and constantly grew larger and more complex over time. The few existing tests only cover small parts of the classes (usually the main code path) and they do not cover border conditions. This is reflected in their inability to detect mutations.

The other components achieve higher scores. That goes along with the fact that they are all components that were developed after the project team committed itself to the goal of achieving a certain test coverage percentage in newer parts of the system. So, the classes in those newer components are usually covered by more than one unit test and, consequently, those tests are able to detect more mutations.

There is one class of the *PkPortal* that seems to be rather well tested: the class *Price*. This is a class that has no dependencies to other components and performs some financial calculations. This class was added after the test coverage goal was defined by the project team. Furthermore, it is very easy to write tests for this class. Those two factors might explain why this class is unusually well tested compared to other classes in the *PkPortal* component.

Component	# Mutated classes	# Mutants	# Killed	Score
DelOrd	2	11	9	81.8%
GkPortal	2	24	21	87.5%
PkPortal	3	66	44	66.7%
RecPref	2	31	26	83.9%
Transfer	1	2	2	100.0%
Total	10	134	102	76.1%

Table 4.1: Aggregated results of mutation testing

Component	Class	# Tests	# Mutants	# Killed
DelOrd	ProtocolDeliveriesController	3	2	2
DelOrd	DeliveryCommitProcessor	11	9	7
GkPortal	KlpUtils	4	5	4
GkPortal	KurepoUtils	5	19	17
PkPortal	ShippingJobService	1	23	7
PkPortal	ProfileService	1	12	6
PkPortal	Price	84	31	31
RecPref	SendersController	6	17	15
RecPref	MigrationsController	6	14	11
Transfer	CleanupDeliveriesJob	2	2	2

Table 4.2: Detailed results of mutation testing

5 Assessment of Testability

The testability or “test-friendliness” of code is a concept to describe how much effort is necessary to write unit tests for that code. The testability is determined by multiple factors of the system like its architecture, the dependencies within the system, the coupling between the components, and their complexity and size. The testability of the code might give an answer to the question why a certain component is not covered by unit tests.

5.1 Setup

The analysis in this chapter focusses on the question why a certain component is not under test control and how complicated it would be to cover it with unit tests. To answer this question, the 200 randomly chosen bugs from chapter 3 were analyzed again: For each bug, the defective code was identified. Then the testability of that code was classified into three categories: good, normal and bad. Good testability means that it does not require much effort to write a test for the code. Bad testability means that a significant effort is required to write unit tests. Normal testability means something in between: Adding tests to cover the code is not trivial but also not very complicated. The criteria that we suggest for classification are outlined in table 5.1. They are all rather soft criteria based on project knowledge and testing experience and focus primarily on the aspect how easy a component can be instantiated and configured within a unit test. This approach was chosen over the usage of common code metrics (like the maintainability index [21] or the cyclomatic complexity [17]) due to practical reasons. The *EPOF* project uses a wide variety of technologies and, hence, calculating code metrics for different types of code files at different points in development history is very expensive. Furthermore, such metrics usually do not take domain and project knowledge into account.

Category	Criteria
Good	The SUT can be easily instantiated in a unit test.
	The SUT has no or only very few dependencies to other components.
	Verification of correct behavior is possible by direct inspection of the state of the SUT.
Normal	The SUT has only few dependencies which can be mocked easily.
	Verification of correct behavior is possible by direct inspection of the state of the SUT or indirectly by its calls to the environment.
Bad	The SUT has many dependencies to other components.
	The SUT depends on complex data from the database which are hard to replace with test data.
	The SUT has dependencies to GUI components or device properties.
	The defective method that causes the bug is not directly accessible by the test (*e.g.,* private or protected).
	The defective method is very complex (*e.g.,* nested structures)

Table 5.1: Categories and heuristics used for classification of testability of source code. SUT = system or component under test.

5.2 Results

Figure 5.1 shows the results of the testability analysis. To clarify the graphics, consider the following reading example: In 16 cases it would have been easy to add unit tests that cover the defective class that caused a bug. A special case, which requires further explanation, is the category “n/a” (not available). Those bug fixes involved only changes to non-code files like resource files, images or to static data in the database (*e.g.,* missing translations). Therefore, they are irrelevant for this analysis which focuses on code files.

There are two observations that are interesting:

- There are 16 cases where the testability of the code is good. In 7 of those 16 cases, there are indeed tests that at least partly cover the defective component. Some of

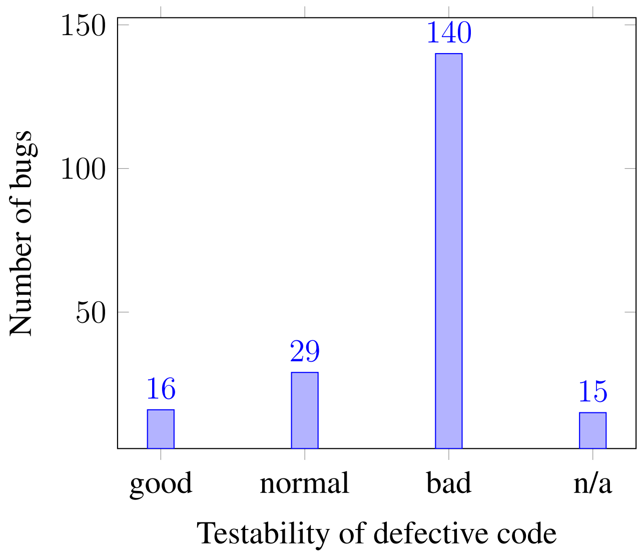


Figure 5.1: Testability of code affected by bugs. $n = 200$

those tests existed before the bug was discovered, and some of them were added when the bug was corrected.

- In 140 cases, it would be very hard to add tests for the defective components. In only 3 of those cases, the defective components are partly covered by tests.

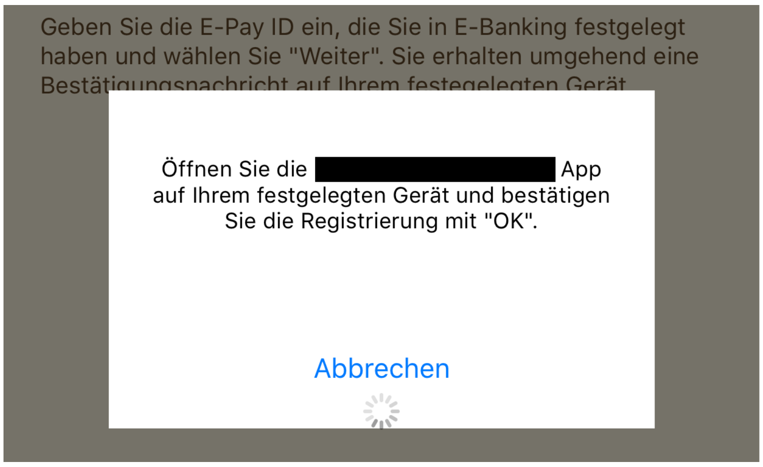
Further inspection of the second observation above revealed that there are some few main reasons why testability is low.

1. The bug is a usability or design issue
2. The erroneous component has many dependencies to other components
3. The defective component violates the single responsibility principle (SRP)

The remainder of this section presents for each of these main reasons a typical example, describes what the problem is, and offers possible solutions.

5.2.1 The bug is a usability or design issue

Example There was one bug report that reported the following problem: There was a popup dialog in the mobile app where an animation of a wheel was used to indicate that some background activity was going on. The issue reported that this animation was not displayed at the correct location. It should have been centered vertically. Figure 5.2 shows the state before the issue was solved. It is obvious that the animation was positioned too low.


Geben Sie die E-Pay ID ein, die Sie in E-Banking festgelegt haben und wählen Sie "Weiter". Sie erhalten umgehend eine Bestätigungsnachricht auf Ihrem festgelegten Gerät

Öffnen Sie die [redacted] App auf Ihrem festgelegten Gerät und bestätigen Sie die Registrierung mit "OK".

Figure 5.2: Wrong position of spinner

Problem description The problem with issues like the one in this example is that they concern usability or graphical design problems where an exact definition of what is right and what is wrong is missing. There exist best practices that may guide the developers and are sometimes subject of ongoing controversial debates between different experts in the same field. As a consequence, it is hard or impossible to catch such issues with automated tests.

Solutions Manual acceptance testing, sometimes by the end-users themselves, is to date the common approach to address such issues. Other approches like automated GUI tests with Selenium or Cucumber require exact specification of the desired behaviour and look of the application in advance. Furthermore, such tests are likely to break on even small changes made to the GUI. Another approach is followed by Moran *et al.* [19] and Mahajan *et al.* [15] who both use computer vision techniques to detect differences between the expected look of a screen (as defined by mockups) and the actual look of the screen (when the application runs). The solution of Moran *et al.* is already used in industrial contexts.

5.2.2 The erroneous component has many dependencies

Example The code in listing 2 is an excerpt from a C# API controller of the *EPOF* back-end. The class is called *DocumentsController* and implements a REST API for accessing documents of a customer. The method shown in the listing does the following:

1. Validates the input parameter, returning an error code if it is invalid
2. Gets the document from the database and marks it as read

3. Checks whether the document was changed since the last time that it was accessed. If it was not modified, returns a NOT MODIFIED status code.

4. Checks whether the document is an email. If this is the case, returns it as a structured document.

5. If the document is not an email, returns its raw content as a binary stream.

There was a bug-report that stated that the format of the data that had been returned from the method *GetDocument* was wrong. To fix the bug, some lines of code had to be deleted from the controller. The controller was not covered by any test code.

Listing 2: Excerpt from class *DocumentsController*

```
PUBLIC VIRTUAL IHttpActionResult GetDocument(STRING id)
{
  VAR validationResult = _globalValidaters.CheckDocumentId(id);
  IF (!validationResult.IsValid)
  {
    RETURN ResponseMessage(Request.CreateResponse(
      (HttpStatusCode)validationResult.Errors
        .First().AttemptedValue,
      STRING.Join(";", validationResult.Errors)));
  }

  INT isNumber;
  VAR tryParse = INT.TryParse(id, out isNumber);
  IF (!tryParse)
  {
    RETURN ResponseMessage(Request.CreateResponse(
      HttpStatusCode.BadRequest,
      "DocumentId is not a number."));
  }

  VAR content = _documentFileService
          .RenderDocumentFile(id);

  _documentService.MarkDocumentAsRead(content.Entity);

  IF (Request.Content.Headers.LastModified.HasValue
    && Request.Content.Headers.LastModified.Value
      .CompareTo(content.Entity.LastModified) >= 0)
  {
    RETURN ResponseMessage(Request.CreateResponse(
      HttpStatusCode.NotModified, "Not Modified"));
  }

  HttpContext.Current.Response.AppendHeader("FolderStructureEtag",
    _folderService.GetFolderStructure().Etag);
```

```

HttpContext.Current.Response.AppendHeader("DocumentEtag",
    content.Entity.Version.ToEtag());

HttpContext.Current.Response.AddHeader("DocumentType",
    "application / octet – stream");


HttpContext.Current.Response.ContentType =
  NEW MediaTypeHeaderValue("application/octet–stream")
     .ToString();

VAR documentEmail = content.Entity AS DocumentEmailDto;

IF (documentEmail != null)
{
  // Deserialize the email document
  VAR emailContent = _documentFileService.GetDocumentFile(id);
  VAR emailMessage = _emailService
    .DeserializeMessage(emailContent.BinaryContent);

  // GetEmailMessage, set the document as Read => eTag changes,
  // so we return the new ones in the header
  HttpContext.Current.Response.AppendHeader("DocumentEtag",
      documentEmail.Version.ToEtag());

  // Get the dto object
  VAR emailDto = EmailMessageDto.MapToDto(emailMessage);

  //Set the date format handling to "\/Date(1234656000000)\/"
  VAR microsoftDateFormatSettings = NEW JsonSerializerSettings
  {
    DateFormatHandling = DateFormatHandling.MicrosoftDateFormat
  };

  // Serialize in JSON
  STRING emailJson = JsonConvert
      .SerializeObject(emailDto, microsoftDateFormatSettings);

  RETURN Ok(emailJson);
}

HttpContext.Current.Response.AppendHeader("DocumentEtag",
  content.Entity.Version.ToEtag());
HttpResponseMessage result =
  NEW HttpResponseMessage(HttpStatusCode.OK);
result.Content = NEW ByteArrayContent(content.BinaryContent);
result.Content.Headers.ContentType =
  NEW MediaTypeHeaderValue("application/octet–stream");
```

```
 RETURN ResponseMessage(result);
}
```

Problem description The main reason why it is hard to write tests for the method *GetDocument* is the large number of other components it depends on:

- In line 3, a validator object is used to validate the parameters.
- At various locations within the code, static references to *Request* or *HttpContext.Current* are used (*e.g.,* line 6, 16, 44).
- In lines 21 and 53, a *DocumentFileService* is used to gather the document from the database.
- In line 24, a *DocumentService* is used.
- In line 35, a *FolderService* is used.
- In line 54, a *EmailService* is used.

For all those dependencies, a stub or mock object has to be injected that behaves exactly the way it is required for a unit test. Due to the large number of dependencies, this process is expensive and rather complicated. The static reference to *HttpContext.Current* is particularly hard to replace with a mock object because the part of the .Net framework, which provides this object is not designed in a test-friendly way.

Solutions There are many solutions to overcome the problem of too many dependencies. They usually involve a refactoring of the existing code to reduce coupling between the components. One typical refactoring is the introduction of abstraction layers to make the code less dependent on other external or internal components. It depends on the concrete case which refactoring techniques are applicable. Feathers [5] and Fowler [7] both show some techniques, and when one should apply them.

The following section describes a possible solution for the example in listing 2. A technique called “Dependency Injection” (DI) was already used to inject components like *DocumentService* and *EmailService* through the constructor of the controller class. Therefore, it is easy to provide fake implementations for those required components to the *DocumentsController* when executing a unit test. A more difficult case is the static reference to *HttpContext*. A technique that can be used to resolve this problem is called “Extract and override method” and is described by Feathers [5]. The application of the technique is sketched in listing 3. The key steps are as follows:

1. Write a wrapper interface *IHttpContext* for *HttpContext* that defines only the methods and properties that are required by the code in the controller.

2. Add a class that implements this interface and just forwards the method calls to the real *HttpContext*.

3. In the controller, add a method *GetCurrentHttpContext* that returns an object that implements *IHttpContext*. Make the method protected.

4. Replace all calls to *HttpContext* in the controller with calls to the method created in the previous step.

5. Create a class *MockDocumentsController* that inherits from the real *DocumentsController*

6. Override the method *GetCurrentHttpContext* and return a stub that implements *IHttpContext* and does not use the real *HttpContext*

7. Use the *MockDocumentsController* for all test cases and use the real *DocumentsController* in production code.

Listing 3: Possible solution to avoid tight coupling.

```
public interface IHttpContext {
  void AddResponseHeader(string header, string value);
  void SetContentType(string contentType);
}

public class RealHttpContext : IHttpContext {
  public void AddResponseHeader(string header, string value){
    HttpContext.Current.Response.AppendHeader(header, value);
  }

  public void SetContentType(string contentType){
    HttpContext.Current.Response.ContentType = contentType;
  }
}

public class MockHttpContext : IHttpContext {
  public void AddResponseHeader(string header, string value){
    // Mock implementation goes here
  }

  public void SetContentType(string contentType){
    // Mock implementation goes here
  }
}

public class DocumentsController {

  protected VIRTUAL IHttpContext GetHttpContext() {
    RETURN NEW RealHttpContext();
  }

  PUBLIC VIRTUAL IHttpActionResult GetDocument(STRING id){
    // some code omitted

    IHttpContext httpContext = GetHttpContext();

    httpContext.AddResponseHeader("FolderStructureEtag",
      _folderService.GetFolderStructure().Etag);

    httpContext.AddResponseHeader("DocumentEtag",
      content.Entity.Version.ToEtag());

    httpContext.AddResponseHeader("DocumentType",
      "application / octet − stream");


    httpContext.SetContentType(
      NEW MediaTypeHeaderValue("application/octet−stream")
        .ToString());

    // some code omitted
  }
}

PUBLIC class MockDocumentsController : DocumentsController {
  // Inherited from base class
  protected override IHttpContext GetHttpContext(){
    RETURN NEW MockHttpContext();
  }
}
```

5.2.3 The defective component violates SRP

Example The code in listing 4 is an excerpt from the Android mobile app of the *EPOF* project. The containing class is called *SetupEPofPayListFragment* and represents one specific screen which is displayed to the user. The class contains many other methods besides the ones shown in the listing. The methods shown in the listing basically prepare loaded data to be displayed on the screen. The method *onLoadFinished* is executed when some data loading task is completed and then calls the private method *normalizeForGui* that performs some processing on the previously loaded data. The method aggregates some of the loaded data to improve the readability of the result.

Listing 4: Excerpt from class *SetupEPofPayListFragment*

```
PUBLIC CLASS SetupEPofPayListFragment {

 // Other instance variables and methods omitted

 @Override
 PUBLIC VOID onLoadFinished(
  Loader<AsyncTaskResult<PaymentContractCompositionDto[]>> loader,
  AsyncTaskResult<PaymentContractCompositionDto[]> data) {
   IF (loader.getId() == EPOF_PAY_LOADER_ID) {
       IF (data.isSucceeded()) {
           paymentContracts = data.getResult();
           dataLoaded = TRUE;
           ePofPayListAdapter.setData(
             NEW EPofPayAccounts(
               normalizeForGui(paymentContracts))
                 .getAccounts());
       } ELSE {
           showException(data.getException());
       }
   }
   mRecyclerView.hideLoading();
   mSwipeRefreshLayout.setRefreshing(FALSE);
 }

 PRIVATE List<EPofPayItem> normalizeForGui(
  PaymentContractCompositionDto[] payContrDtos) {
  List<EPofPayItem> items = NEW ArrayList<>();
  IF (payContrDtos.length == 0) {
   items.add(NEW EPofPayNoItems());
  } ELSE {
   FOR (INT cur = 0; cur < payContrDtos.length; cur++) {
     BOOLEAN nextLineIsEquals = (cur + 1 >= payContrDtos.length)
       ? FALSE
       : itemsClearingContractNumberEq(payContrDtos[cur],
           payContrDtos[cur + 1]);
     PaymentContractCompositionDto curDto = payContrDtos[cur];
     PaymentContractCompositionDto nextDto = NULL;
     IF (cur + 1 < payContrDtos.length) {
       nextDto = payContrDtos[cur + 1];
     }
     IF (cur == 0) {
       items.add(
         NEW EPofPayGroup(
           curDto.getFinancialInstituteWithContractName(),
           "" + curDto.getId(), "", "", curDto, ""));

       FOR (UserBankAccountDto account : curDto.getAccounts()) {
         String accountState = ((account.isDefault())
            ? getActiv().getString(R.string.epof_default_account)
            : "");
          items.add(
            NEW EPofPay(curDto.getFinancialInstitute(),
              "" + account.getId(),
              account.getDescription() + accountState,
              account.getIban(), account.isDefault(),
              account.getId(),
              curDto, account.getDescription(),
              account.getOwnerName(),
              account.getOwnerStreet(),
              account.getOwnerZip(),
              account.getOwnerCity()));
        }

        // more nested structures omitted

      } ELSE {
        IF (!nextLineIsEquals && nextDto != NULL) {
          items.add(
            NEW EPofPayGroup(
              nextDto.getFinancialInstituteWithContractName(),
              "" + nextDto.getId(), "", "", nextDto, ""));
        }

        // more nested structures omitted
      }
    }
  }

  RETURN items;
 }
}
```

Problem description The single responsibility principle (SRP) is a design principle for object oriented systems that states that each class should only have one responsibility within a system and should encapsulate all aspects of this responsibility. Other formulations state that a class should only do one thing or should only have one reason to change [16]. If a class violates this principle, this usually means that it does too many different things. The main symptoms of a violation of the SRP are usually large classes, long and complex methods, and many private methods within a class. Those symptoms are a hint that a class does in fact more things than it should. The class might encapsulate multiple domain concepts within its implementation that should be modelled by multiple classes. The code that should be tested is often hidden in private methods, which are not directly accessible in unit tests. Furthermore, such large classes usually have many dependencies.

As a consequence, it is very difficult to write a unit test for such classes.

The class *SetupEPofPayListFragment*, which contains the methods shown in listing 4, is 564 lines of code long and contains 27 methods of which 14 are private. A violation of the SRP is shown in the listing. The main responsibility of the class *SetupEPofPayListFragment* is the rendering of a screen to show certain information to the user. It should not be its responsibility to aggregate the data that should be presented. But exactly this is actually done in the method *normalizeForGui*. As a consequence, it is hard to test whether this aggregation is performed correctly because the aggregation code is only reachable through the public method *onLoadFinished*. And the class *SetupEPofPayListFragment* is generally hard to instantiate within a test due to its complexity and its many dependencies. Those are all factors that make testing of this class a very difficult and expensive task.

Solutions The general solution to fix violations of the SRP involves again refactoring of such large classes. First, one has to clearly define the responsibility of the class. Then, the code of the class has to be inspected, and all tasks that do not belong to this responsibility have to be identified. Finally, those parts of the code have to be refactored out into separate classes. As a result, many smaller classes are created, and the original class shrinks dramatically. It is then easier to test the smaller classes than the original large class. Multiple techniques that support such refactoring tasks are presented in [5] and [7].

In the example of listing 4, the method *normalizeForGui* could be moved to a new class *PaymentAggregator*. A solution is sketched in listing 5. Writing unit tests for the extracted class *PaymentAggregator* does not require complex setup and is therefore easier to do.

Listing 5: Possible solution to avoid violations of SRP.

```
// Define new class
PUBLIC CLASS PaymentAggregator {

 PUBLIC List<EPofPayItem> aggregate(
 PaymentContractCompositionDto[] payContrDtos,
 Activity activity)
 {
  List<EPofPayItem> items = NEW ArrayList<>();
  IF (payContrDtos.length == 0) {
   items.add(NEW EPofPayNoItems());
  } ELSE {
   FOR (INT cur = 0; cur < payContrDtos.length; cur++) {
    // rest of code omitted
   }
  }
  RETURN items;
 }
```

```

  //Further methods on which aggregate() depends on
}

PUBLIC CLASS SetupEPofPayListFragment {
  // rest of code omitted

  PRIVATE List<EPofPayItem> normalizeForGui(
    PaymentContractCompositionDto[] payContrDtos)
  {
    // Change original normalizeForGui method to use the new class to do the work.
    PaymentAggregator agg = NEW PaymentAggregator();
    RETURN agg.aggregate(payContrDtos, getActiv());
  }
}
```

6
Related Work

In a large-scale field study, Zaidman *et al.* [2] used an IDE plugin to record the development behavior of 2'443 software engineers. Their results show that half of the developers in their study do not test, tests are run rarely in the IDE, and most programming sessions end without any tests being executed. Furthermore, their results show that developers strongly overestimate the time they spend on testing. These results are in line with our observation that automated testing is not widely practiced in the *EPOF* project.

Levin *et al.* [14] studied the co-evolution of production and test code by analyzing commits of 61 open source projects. They used machine learning techniques to classify commits into maintenance categories (adaptive, corrective and perfective) and analyzed which commits included test changes. Their results show that, more often than not, developers perform code fixes without changing any tests (in $< 24.7\%$ of corrective commits, $< 30.4\%$ of adaptive commits, in $< 35.0\%$ of perfective commits). In the *EPOF* project, there are 2'009 issues where the associated commits could be reconstructed. Only 347 (17.3%) of them involved test changes. The statement from Levin *et al.* that most maintenance activities are done without changing tests is also valid for the *EPOF* project.

Klammer *et al.* [13] discuss the challenges and experiences of the transformation from a classical development process, which involves manual testing, to an agile process, which largely uses automated testing. They accompanied an industrial software development team, recorded certain events during the project, and analyzed the test coverage of public methods in the code base after every development iteration. They discovered multiple evolution patterns like increasing and decreasing coverage, constant low and constant high coverage, and flaky coverage. A reason for constant low coverage according to their study are technical challenges that impede the addition of tests. Furthermore, they state that humans are a very important factor when it comes to automated testing: The development team must commit itself to testing, and testing activities have to be honored by the management. Both findings could be observed in the *EPOF* project. The testability of the code is generally very low and, therefore, many components are not tested. But after the introduction of the test coverage goal by the management, the developers put more efforts into unit testing.

Fucci *et al.* [8] studied the aspects of test-driven development (TDD), that have an influence on the quality of the written source code and the productivity of the developers. They recorded the behavior of professional developers while they solved a specific programming task and built predictive models based on these data. Their results show that it is not the test-first characteristics of TDD which improves quality and productivity, but rather the fact that TDD encourages fine-grained, steady steps which enhance focus and development flow. TDD is not practiced in the *EPOF* project. But the introduction of TDD might have a positive effect on the development process and might also improve the testability and the test coverage of the system.

Borle *et al.* [3] analyzed open source Java projects on Github. They tried to determine how many of them use TDD and whether TDD has some effects on commit velocity, on the number of commits or on the number of reported bugs. Their results show that only 16.1% of Java repositories on GitHub have test files, and only 0.8% repositories practiced TDD. Furthermore, they found no statistically significant support of their hypothesis that TDD affects commit velocity, number of bug fixes, and number of issues. As already mentioned, TDD is not practiced in the *EPOF* project. So, the results from Borle *et al.* are not directly transferable to *EPOF*. However, our findings support their observation that automated testing is not that widely practiced.

7 Threats to Validity

There are multiple issues that might affect the validity of this research project. They are organized into four categories as suggested by Runeson *et al.* [22].

Construct validity The results of this research project largely depend on the correct association of issues in the issue tracker Jira and commits in Git. The association between those two entities is done by Jira through naming conventions of feature branches (*e.g., feature-EPO-1234*) in the Git repository and the key of the issue (*e.g., EPO-1234*) and references to Jira issues in commit messages (*e.g.,* Commit message: *This commit fixes EPO-1234*). While the association using feature branches is very reliable, the association using commit messages is not. The latter depends on the developer's commit messages for which no general conventions exist. As a consequence, there might be commits that, in fact, belong to a certain issue, but Jira does not link them together. It is also possible that a commit does not belong to a specific issue but is associated with it by Jira. Both of these threats may impede the traceability of a bug fix.

Another similar problem is imposed by merge conflicts. Before a developer can send a pull request to the main repository, he needs to merge the current state of the main repository back into his own feature branch. This merge usually changes many files but is not directly related to the changes the developers made to fix a certain bug. However, such merge commits are associated with the bug fix and, therefore, the traceability of the bug fix is hampered. This issue was addressed by excluding such merge commits from the possibly impaired analysis.

As mentioned in chapter 2, the project was migrated from Subversion to Git without keeping the development history. So, it is not possible to reconstruct bug fixes that were done before this migration.

In chapter 4, a tool was used to detect possibly recurring bugs. This tool classifies two issues as possibly related when their respective bug fixes change the same method in the same code file. The tool does not consider method additions, removals, signature changes, and renaming when it compares two files. So, there might be some possibly related issues that are not detected by the tool. The tool might also indicate false positives: bug fixes from different bugs that change the same method but are not related. Those false positives were manually removed from the data set as described in section 4.1.1.

Internal validity To answer the question why many bug fixes do not modify tests, 200 bugs were randomly chosen and further analyzed as described in chapter 3. They represent only 17.8% of those 1'119 bugs for which the bug fix could be reconstructed automatically and only 3.1% of all 6'531 reported bugs. Thus, the findings, which were observed on this sample, might not generalize to the full set of bugs. However, the programming style and the architecture do not vary significantly between the components of the *EPOF* project, and the analyzed bugs are spread over all parts of the system. Hence, the reported findings (low testability, bug fixes affecting many files) are likely to be valid for unanalysed parts of the software too.

Due to technical difficulties, the mutation testing described in section 4.3 was applied to only 10 classes. The *EPOF* project contains more than 10'000 classes, thus, those few mutated classes are not representatives for the whole project. Hence, generalizing the findings from the mutation testing process to the whole *EPOF* project suffers from a severe bias.

External validity This research project investigated just one large industrial software project. So, basically, the findings are valid for at least the *EPOF* project. However, related work from Zaidman *et al.* [2] and Levin *et al.* [14] (see chapter 6) suggest that the most important findings — automated testing is not that widely used and most commits do not affect tests — are also valid for other projects. Furthermore, Klammer *et al.* [13] state that technical difficulties (like low testability) might be an explanation for low test coverage.

Reliability The classification of testability presented in table 5.1 on page 38 is based on our testing experience rather than objective software metrics (*e.g.,* cyclic dependency, class couping). The choice of this approach was justified with the technical complexity of gathering software metrics of the *EPOF* project, and because software metrics lack project and domain knowledge. However, the chosen approach introduces a threat to validity because the classification depends only on our experience.

8 Conclusion and Future Work

8.1 Answers to research questions

There were three research questions that guided this study. The first research question investigates the relation between unit tests and bugs. It asks whether the discovery of bugs pushes the writing of tests. In chapter 3, we showed that there are only very few bug fixes that introduced new tests or modified existing ones. Thus, in most cases the developer just fixed the bug in the production code and did not add or change any test code. There were few bug fixes that included changes to tests. However, only roughly half of those test changes actually covered the bug. Some of the other test changes were necessary because the compiler complained. The bug fix changed the code in a way that the tests did not compile anymore. Thus, the developer had to make them compile again.

More than half of the studied bug fixes affected more than a single file. This indicates that the majority of bugs were not caused by a single component but rather by the cooperation between multiple components of the system. This suggests that developers should also consider writing integration and end-to-end tests to ensure the quality of the software. However, writing only integration tests is not advisable because roughly one third of the analyzed bug fixes changed only one single component. Thus, unit tests alone could catch numerous bugs.

The second research question deals with the quality of the existing unit tests and whether they help to prevent bugs. All reported bugs were detected either by a test team, which performed manual tests as well as by customers who used the system, or by the developers themselves. Evidence whether the unit tests helped to discover bugs could not be gained from the commits. A sub-commit level analysis, like the one performed by Zaidman *et al.* [2], would yield more insights to answer this question.

However, the results presented in chapter 4 showed that there were only very few bugs that occurred again or that were similar to each other even though the test coverage in the *EPOF* project is low. But the results presented in section 4.1.2 also show that there are usually no tests written after a bug occurred for the first time and also not after the second time.

We showed that many components of the system are affected by more than one bug. There was one component that was affected by nearly 50 different bugs. But we could observe some differences between the parts of the system. Some components are more prone to bugs than others. Those components usually contain large parts of the business logic and do many different things or play a central role in the application. The developers explained that those bug-prone components are huge and have many dependencies to other components, which has a negative impact on their testability. The low testability of the components and the time pressure prevented the developer from writing tests for those critical parts of the application. Furthermore, bad communication between the customer and the development team led to many bug reports that were in fact changes of the requirements and not real bugs.

Many of those problems were recognized by the project team in late 2016, and some measures were taken to improve the development process and the collaboration between the developer team and the customer. Those measures aimed to prevent the project from failure. Our results showed that the number of discovered bugs decreased after those organizational changes. But we could not observe a change in the time it took the developers to fix a certain bugs.

The results of the mutation testing show that a class should be covered by multiple test cases to effectively detect mutations of the class. Each test case should test the class under different conditions. If the test coverage of a class is low, or if only the main program flow is tested, many mutations of the class are not detected. As a consequence, bugs within such classes might not be discovered by the test suite.

The project team decided to give unit testing higher priority as of the beginning of 2017. They formulated a code coverage goal to reach for new components. Futhermore, they changed the architecture of the system in a way that new features can be added as microservices. The intention behind those changes was to reduce coupling between the components and to improve their testability. The results of our mutation testing show that the quality of the unit tests that were introduced after January 2017 is better than the quality of older tests. Thus, the measures to improve the architecture and testability of the system achieved their intended goals.

The third research question deals with the system's testability. Our analysis revealed that most parts of the source code of the *EPOF* project are designed in a way that makes it very hard to write unit tests for them. Section 5.2 presented three main problems that cause the low testability: firstly, the violation of the single responsibility principle; secondly, the large number of dependencies between the components; and thirdly the bug being a usability or design issue. The first two causes could be resolved by refactoring parts of the system and by designing the system in a more modular and, consequently, more testable way. The resolution of the third problem is subject to further research.

8.2 General insights

This study showed that when the testability of a certain part of source code is low, no tests are written for those parts. Thus, it is important to design the architecture of a software system with testability in mind. Techniques like Test Driven Development (TDD) or other test-first development processes might help to ensure testability of the code that is developed.

Another insight is that the test team which performed manual tests has a very important role when it comes to quality assurance. The testers caught many bugs, and they also caught many non-functional issues that might disturb the users or, in the worst case, prevent them from using the system.

The third general insight is that humans are an important factor when it comes to unit testing. If a developer or the whole development team commits itself to automated testing, and if this decision is supported by the project management, it is likely that the test coverage and test quality of the system increase. This effect could be observed in the *EPOF* project after the transformation and improvement of the development process in 2016/2017.

8.3 Future work

It would be interesting to apply the methods used in this study to other projects of Company P that use the same toolset (Jira and Git Bitbucket) and see whether they yield similar or completely different results.

Another direction of research would be to find an answer to the question which parts of the code need to be tested to catch most bugs. For which part of a system is it worth investing time into writing unit tests and for which parts of a system is it less critical?

Closely related to the former question is the following aspect: A developer must not only decide *whether* he should write a test for a component but also *when* he should write one. Should he write a test as soon as the component is created? Should he add tests even before the component exists (as TDD suggests)? Or should he wait until the component is not changed often anymore?

An interesting question would be whether a development team should focus on integration testing rather than on unit testing. Our work showed that many bugs in the

EPOF project affected more than one component. So, the question is whether integration tests would have helped to prevent those bugs.

Another follow-up question is how usability and graphical design aspects can be covered with automated tests, and how such tests could detect usability or design problems.

9 Anleitung zum wissenschaftlichen Arbeiten

The following sections describe the helper tool that was developed during the research project in more detail. The helper tool automates some repeating tasks that needed to be done during the project. In particular, the tool is able to perform the following tasks:

- Exploit the API of the Jira issue tracker and the Bitbucket server to gather information about issues and related commits and store them into a local SQLite database.
- Clone all Git repositories which belong to a certain Jira project into a local directory.
- Given two commits that modified the same source code file, check whether those commits made changes to the same methods within these files.

The tool is a cross-platform console application written in C# and builds upon the .Net Core framework. It uses the Entity Framework Core for database access.

Each of the following sections describes one of the points from above in more detail and also gives hints for adapting the tool to other research projects that analyze software projects built upon the same toolchain (Jira server and Bitbucket).

9.1 Exploiting the Jira API

To answer the research question of this project, it was important to be able to link issues and commits which resolved them together. The project under analysis used Jira as issue tracker and Bitbucket Server as code repository host. Those two products are developed by the same company (Atlassian) and, therefore, are tightly integrated. It is possible to gather commits for a specific issue from the system. The standard way of perceiving this information is through the web interface of Jira. However, the web interface is not convenient for quantitative analysis and complex evaluations which are necessary for this research project. A more structured and faster approach to query this information is required.

Jira exposes an extensive and well-documented REST application programming interface (API). This API is better suited to access the information in both the issue tracker and the code repository, for the purposes of this project. To avoid an excessive network load and for query performance reasons, we decided to gather all relevant information from the issue tracker and to store it in a local SQLite database. The use of a local database allows the usage of standard SQL queries for analysis.

9.1.1 The Jira API

9.1.1.1 API methods

Jira exposes an extensive REST API that allows access to a wide range of information about the projects in the issue tracker and the issues related to those. The API is generally well documented [11]. Only few exposed interfaces are interesting for the purposes of this project. In particular, just four exposed REST methods were used to gather all required information. Those four methods are shown in table 9.1. The first two are used to get information about projects and their issues while the third and the fourth methods are used to get commit related information of the issues.

9.1.1.2 Security

The Jira API uses standard HTTP basic authentication to authenticate a user. The credentials are therefore passed in the HTTP header of all requests to the API. Jira has an internal permission system to restrict access of users to the information in the issue tracker. Those permissions apply to the web frontend as well as to the REST API. Therefore, a user needs to be granted permissions by a Jira server administrator to access a given project. The minimal required permission level is “Developer” because the last two API methods described in 9.1 are not accessible to users with lower permissions.

`https://{urltojira}/rest/api/2/project/{projectKey}`	
GET	Get detailed information about the Jira project identified by the projectKey.
`https://{urltojira}/rest/api/2/search?jql=project={projectKey}&maxResults={maxResults}&startAt={startAt}`	
GET	Get detailed information about all issues belonging to the project identified by the projectKey. The result is paged.
`ttps://{urltojira}/rest/dev-status/latest/issue/detail?issueId={issueId}&applicationType=stash&dataType=repository`	
GET	Get detailed information about the commits that belong to a specific issue. This is an undocumented API method.
`https://{urltojira}/rest/dev-status/latest/issue/detail?issueId={issueId}&applicationType=stash&dataType=pullrequest`	
GET	Get detailed information about the pull requests that belong to a specific issue. This is an undocumented API method.

Table 9.1: Jira REST API methods used in this project

9.1.2 The database model

The information gathered through the Jira API are stored in a local relational SQLite database. In order to allow easy querying of the data, the database schema depicted in figure 9.1 was developed. The tables are described in further detail in this section.

The table *Projects* holds information about the Jira project. This includes an alphanumeric key, which is unique per project and is used extensively throughout Jira and its API. There is also an integer ID, which is used internally by Jira and is necessary for some API calls.

The table *Components* holds information about so-called components of a Jira project. Components are used within Jira to structure large projects into smaller parts. As an example, consider a large business application with a desktop frontend, a back-end, and a mobile app. Then, there might be the components "Desktop app", "Back-End", and "Mobile app". Issues can be assigned to a component which allows efficient filtering.

The table *Issues* is the most important table. It holds information about the issues of a Jira project. The table holds technical information like the issue key, the issue ID internally used by Jira, and the creation date and also business information like the title, the description, the reporter, and the state of the issue. The table references other tables within the database. For instance, there is a reference to the table *Projects* due

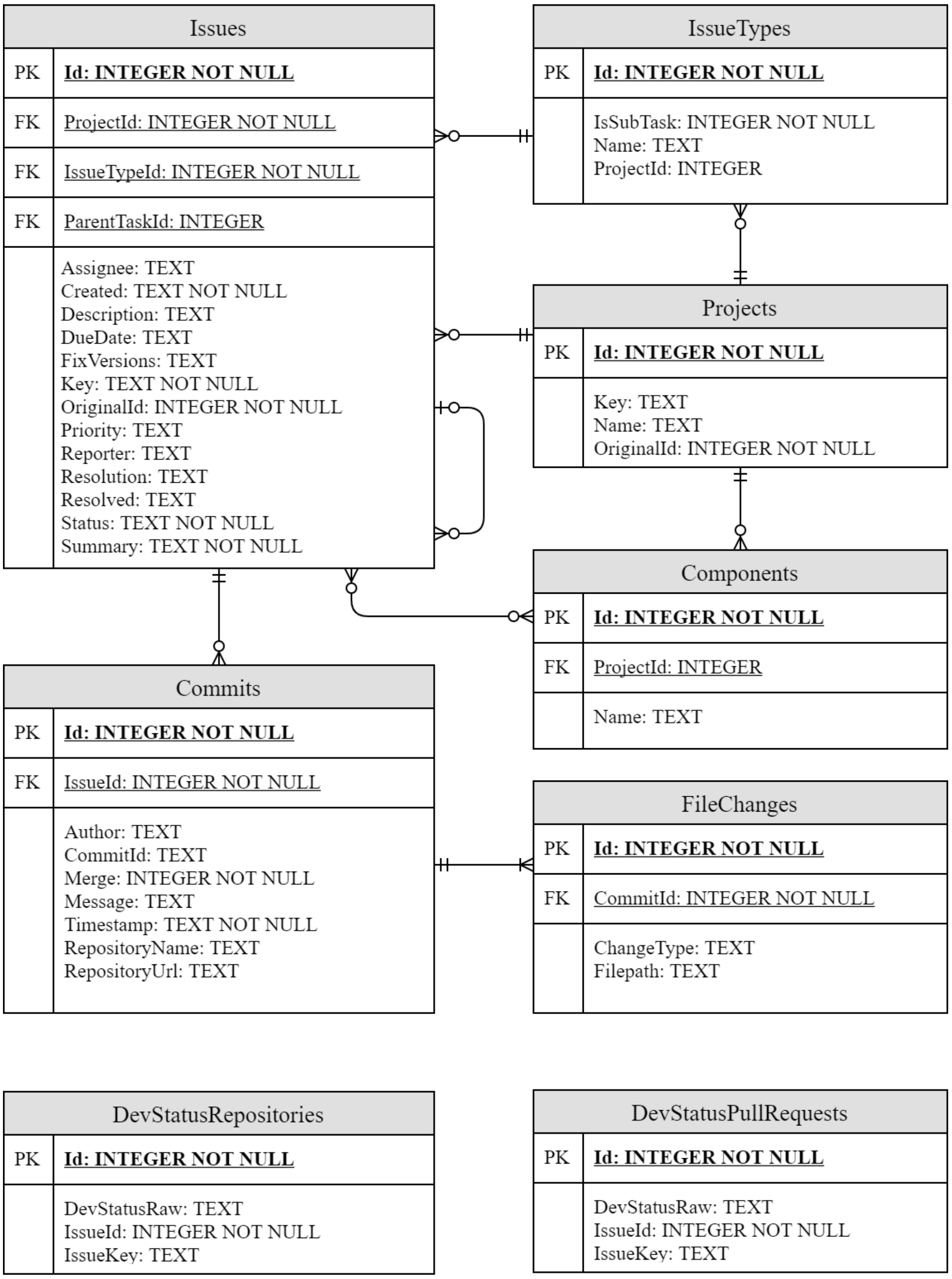


Figure 9.1: The entity relationship diagram of the database

to the fact that each issue belongs to exactly one Jira project. There is also a many-to-many relationship to the table *Components* because each issue belongs to zero or more components.

The table *IssueTypes* holds information about the issue types used in a specific Jira project. Issue types are for instance “Bug”, “Improvement”, “New feature”, and many others. There are also sub-types like “Sub-Bug” or “Sub-Improvement” because Jira has the notion of sub-tasks — issues that are subordinates to another issue. In this research, especially the categories “Bug” and “Sub-Bug” respectively are interesting.

The tables *DevStatusPullRequests* and *DevStatusRepositories* are intermediate tables that hold the raw result of some Jira API calls. The data in these tables is then later transformed into a more structured form and stored in the other tables in the database.

The table *Commits* holds information about commits of the Git repository of the Jira project. The table holds some standard information about a commit like the author, the timestamp, and the SHA1 hash. The most precious information is the reference to an issue the commit belongs to. Using this association, it is possible to relate issues and commits together.

The table *FileChanges* finally holds information about changes to files that where made by a certain commit. The relevant information there is the type of change (“Add”, “Delete”, “Modify”) and the path to the changed file. There is a foreign key reference back to the commit the file change belongs to.

9.1.3 Importing the data from Jira

The tool performs multiple steps to retrieve the relevant pieces of information from the Jira API and store them in the local database.

1. Import the Jira project and related information (components, issue types)
2. Import issues for each project
3. Import raw development information for each issue
4. Extract commits and file changes from the raw development information

The order specified in this list has to be strictly followed because the steps depend on each other. Regarding the architecture of the tool, each step is modelled by a separate class in the code.

The first step is performed by the class *ProjectImporter*. To import a certain Jira project, the class needs to be passed the project key. A request for gathering the project information is then sent to the Jira API. The answer in JSON format is then parsed and converted to entities that are then stored in the database. The *ProjectImporter* stores the project as well as its components and issue types.

The second step is performed by the class *IssueImporter* which imports all issues of a certain Jira project and links them to the project information previously imported. It first gathers the total number of issues from the API and then imports them in batches of equal size to avoid running into timeouts. This batch mode is possible because the Jira API supports paging of the result data. Even though the database supports the modelling of the relation between issues and their sub-issues, this information is not imported by the tool in its current version.

The third step is performed by the class *DevStatusImporter*. This is an intermediate step to gather just the raw data from the third and fourth Jira API call described in table 9.1. The class sends requests to those API methods for every issue of a certain project and simply stores the result, which is in JSON format, in the database. The reasons that caused the introduction of this intermediate step are performance and efficiency considerations. The import of the raw data needs to be done only once and, afterwards, there is no need to query the API again as the data are stored locally. This allows faster access to the data than over the network.

The fourth and final step is done by the class *CommitImporter*. The class takes the data imported in the third step, parses them, and extracts commits and associated file changes.

All direct communication with the Jira API is encapsulated by the class *JiraAccessor*. This class provides public methods to access the API. Those methods are then called by the other classes mentioned above. This class is also responsible for authentication against the API. The necessary credentials are stored inside a text file which is then embedded as resource into the tool's executable. The credentials are expected to have the form *username:password*.

9.1.4 Customizing the tool

In this section, a step-by-step instruction is provided to import issues and related commit information for a certain Jira project. Table 9.2 provides a scenario that is used throughout the instructions.

Name of project	TeaTime
Jira project key	TEA
Jira base url	`https://jira.teacompany.com/rest/api/2/`
Jira username	janedoe
Jira password	secretpass1234

Table 9.2: Scenario information for customization of the tool

The .Net Core framework and all necessary tooling should be correctly installed and configured. Instructions for this process can be found on the Microsoft website [12].

The following steps need to be taken in order to adapt the tool to the scenario above:

1. Ask a Jira server admin to give the user *janedoe* “Developer” permissions on the project *TeaTime*.

2. Modify the class *Model/AppDbContext* by adjusting the connection string to the SQLite database in the method *OnConfiguring* to point to a suitable location. The database will be created automatically on startup of the tool. Pay attention to the fact that subdirectories aren’t created by the application but need to be created manually.

3. In the main source code directory of the tool (where the **.csproj* file is located) add a folder *Exclude*. Within this folder create a file *pwd.txt*.

4. Add the line *janedoe:secretpass1234* to the file *pwd.txt*.

5. In the source code class *JiraAccessor* modify the field *BaseUrl* to `https://jira.teacompany.com/rest/api/2/`.

6. In the source code class *Program* in the method *ImportProjectsAndIssues()* add the project key *TEA* to the local variable *projectKeys*. You might need to remove keys that are already present there.

After all those steps are completed, the tool is ready for being compiled and run. Depending on the speed of the internet connection and the size of the Jira project, the import may take a while.

9.1.5 Querying the imported data

After import, the data are stored in a normal SQLite database. There are multiple options to query the data:

- Use the SQLite command line shell program to access the database. The shell program as well as the documentation of SQLite can be found on the SQLite website [27] [28].

- Use the application *DB Browser for SQLite*, an open-source app that allows manipulation, querying and managing SQLite databases. The app provides a graphical user interface and functions for importing and exporting of data. The documentation and compiled binaries for different platforms can be found on the website of the application [26]. The source code is hosted on Github [10]

9.2 Cloning repositories

This part of the tool builds upon the previously imported data. During import of commits, the name and the url of the repository to which the commit belongs is stored in the database. The tool exploits this information to gather all repositories that are used and clones each repository into a local directory. These operations are performed by the class *GitRepositoryCloner*. Internally, it executes Git's clone command in a separate process. Therefore, the tool requires Git to be installed.

9.2.1 Customizing the tool

It was already mentioned above that the tool assumes that the information of Jira has already been imported as described in section 9.1.3. So, the behavior depends on this data. However, there are some options that one must adapt to make the tool work. The following modifications all have to be done in the class *GitRepositoryCloner*:

1. Modify the constant *BaseCheckoutPath* to point to the correct location where all repositories should be cloned to. The tool won't create new folders. So, the provided path in this constant has to be valid.
2. Modify the constant *BitbucketBaseUrl* to point to the correct Bitbucket host.
3. Modify the constant *PathToGit* to point to the Git executable.

Finally, some configuration tasks have to be done before the tool can be compiled and executed.

- Ask a Bitbucket server administrator to grant you at least *read* permissions on all repositories that will be cloned.
- Make sure that the correct credentials are contained in the file *Exclude/pwd.txt* as described in section 9.1.4.

9.3 Comparison of diffs on method level

This part of the tool builds upon the information imported from Jira (see section 9.1.3) and also on the cloned repositories (see section 9.2). During the research project, a method was needed to recognize similar or recurring bugs in the code. The main idea was to find commits that belong to different bugs but change the same method in the same code file. Hence, normal file differences, which are produced by a program like *diff*, are not sufficient. An algorithm was developed to solve this task.

9.3.1 The algorithm

The main idea of the algorithm is as follows: For a given commit, check out the current version and the previous version of a certain file from the version control system, then parse both files, extract all method declarations from those files, and finally compare the method declarations of the current version of the file to those of the previous version of the file. This yields a list of changed methods of a given commit. This list can be compared to the list of changed methods of another commit. If there are methods that are changed by both commits, this might be a hint that the commits and the issues they belong to are possibly related. The pseudocode of the algorithm is shown in algorithm 1 and algorithm 2.

Algorithm 1 Procedure that suggests possibly recurring bugs.

```
 1: procedure GetCommits
 2:     $possibleRecurring \leftarrow$ CreateList( )
 3:     $commitTuples \leftarrow$ GetCommitTuples()        ▷ Get a list of commits that
    changed the same code file and belong to different bugs. This information can be
    retrieved from the database.
 4:     for all commitTuple $ct$ of commitTuples do
 5:         $c1 \leftarrow ct$.FirstCommit
 6:         $c2 \leftarrow ct$.SecondCommit
 7:         $file \leftarrow ct$.CommonChangedFile                    ▷ Path to file
 8:         $cm1 \leftarrow$ GetChangedMethodsOfCommit($c1$, $file$)
 9:         $cm2 \leftarrow$ GetChangedMethodsOfCommit($c2$, $file$)
10:         $commonChangedMethods \leftarrow$ Intersect($cm1$, $cm2$)
11:         if $commonChangedMethods$ not empty then
12:             Add($possibleRecurring$, ($c1$, $c2$, $commonChangedMethods$))
13:         end if
14:     end for
15:     return $possibleRecurring$
16: end procedure
```

9.3.2 Implementation details

9.3.2.1 Language specific aspects

The algorithm described in section 9.3.1 works for any programming language that has the notion of methods or functions or procedures. However, there are two language specific parts of this algorithm:

The first is the call to the database in algorithm 1 on line 3. This call should only return code files for a single language. For instance, it should return only Java files, or it

Algorithm 2 Function that compares the current and the previous version of a certain file and extracts all changed methods.

```
 1: function GETCHANGEDMETHODSOFCOMMIT($commit$, $file$)
 2:     $cur$ ← GETFILEFROMSOURCECONTROL($commit$, $file$)
 3:     $prev$ ← GETFILEFROMSOURCECONTROL($commit.precedent$, $file$)
 4:     $syntaxTreeCur$ ← PARSEFILE($cur$)
 5:     $syntaxTreePrev$ ← PARSEFILE($prev$)
 6:     $methodsCur$ ← GETMETHODS($syntaxTreeCur$)
 7:     $methodsPrev$ ← GETMETHODS($syntaxTreePrev$)
 8:     $changedCommonMethods$ ← CREATELIST( )
 9:     $commonMethods$ ← INTERSECT($methodsCur$, $methodsPrev$)
10:     for all $method$ in $commonMethods$ do
11:         $methodCur$ ← $methodsCur$($method$)            ▷ Current version of method
12:         $methodPrev$ ← $methodsPrev$($method$)          ▷ Previous version of method
13:         if $methodCur.MethodBody \neq methodPrev.MethodBody$ then
14:             ADD($changedCommonMethods$, $method$)                ▷ Add to list
15:         end if
16:     end for
17: end function
```

should return only C# files. The second programming language specific part is obviously the actual parsing of the source code file in algorithm 2 on lines 4 and 5.

9.3.2.2 Architecture

The class diagram in figure 9.2 gives an overview of the architecture of the part of the tool that implements the algorithm. The architecture is designed to allow easy extension of the algorithm to other programming languages. The classes and how they collaborate is explained in the remainder of this section.

The class *SameMethodChangeFinder* contains the largest part of the algorithm. It loads the necessary data about commits that change the same file from the database, runs the algorithm based on these data, and writes the result to a JSON file which can be processed further. The class delegates some aspects of the algorithm to other classes. It is an abstract class because the loading of the information of the database is language-specific as mentioned above. The call to the database must therefore be implemented by language-specific subclasses of *SameMethodChangeFinder*. Implementations for Java and C# are given.

The class *CodeFileComparer* is responsible for the extraction of changed methods from code files. It takes two versions of the same file as argument and returns a list of methods that changed between the two versions. The class *CodeFileComparer* is

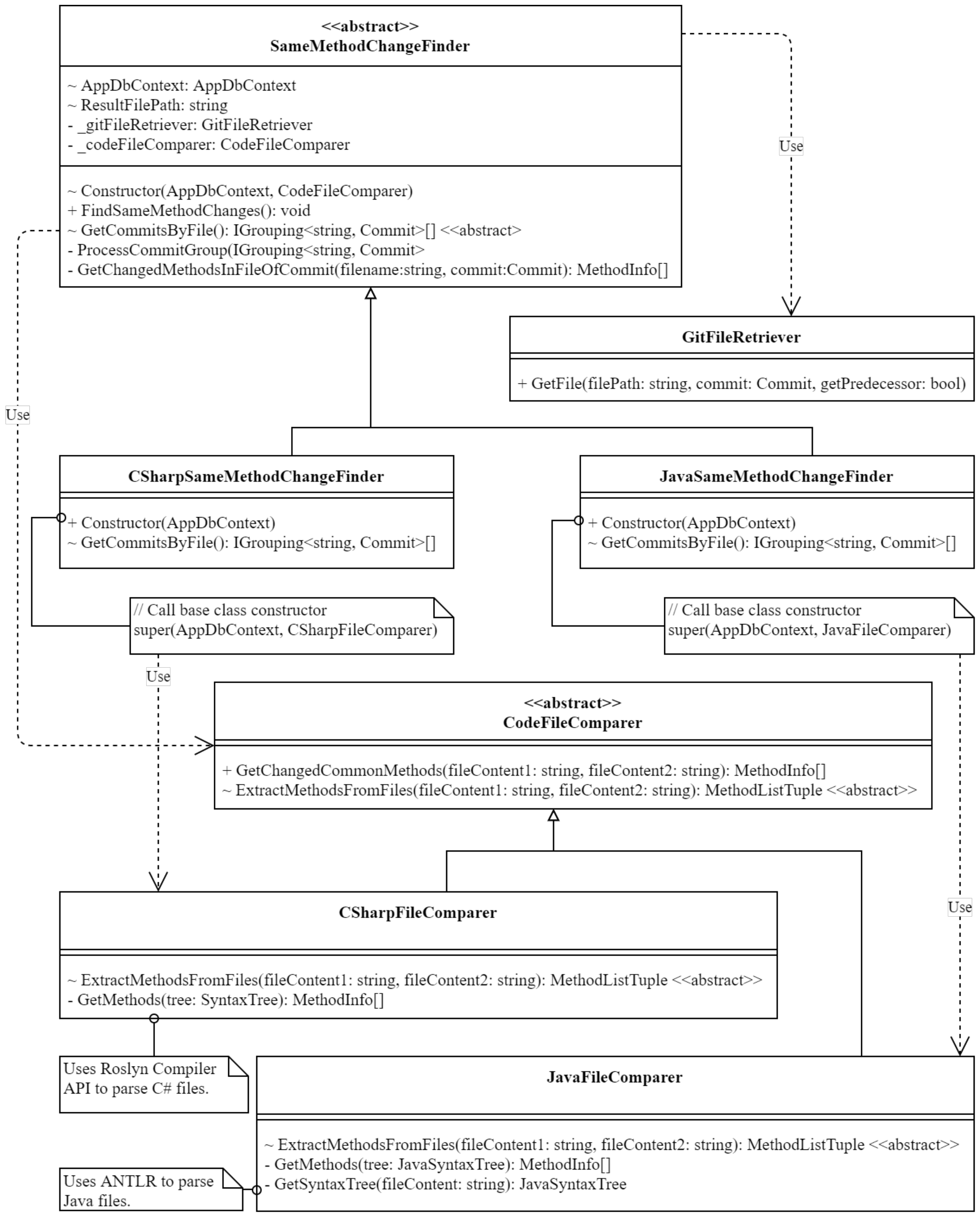

<<abstract>>
SameMethodChangeFinder
~ AppDbContext: AppDbContext
~ ResultFilePath: string
- _gitFileRetriever: GitFileRetriever
- _codeFileComparer: CodeFileComparer
~ Constructor(AppDbContext, CodeFileComparer)
+ FindSameMethodChanges(): void
~ GetCommitsByFile(): IGrouping<string, Commit>[] <<abstract>
- ProcessCommitGroup(IGrouping<string, Commit>
- GetChangedMethodsInFileOfCommit(filename:string, commit:Commit): MethodInfo[]
Use
GitFileRetriever
+ GetFile(filePath: string, commit: Commit, getPredecessor: bool)
Use
CSharpSameMethodChangeFinder
+ Constructor(AppDbContext)
~ GetCommitsByFile(): IGrouping<string, Commit>[]
JavaSameMethodChangeFinder
+ Constructor(AppDbContext)
~ GetCommitsByFile(): IGrouping<string, Commit>[]
// Call base class constructor
super(AppDbContext, CSharpFileComparer)
// Call base class constructor
super(AppDbContext, JavaFileComparer)
Use
<<abstract>>
CodeFileComparer
+ GetChangedCommonMethods(fileContent1: string, fileContent2: string): MethodInfo[]
~ ExtractMethodsFromFiles(fileContent1: string, fileContent2: string): MethodListTuple <<abstract>>
Use
CSharpFileComparer
~ ExtractMethodsFromFiles(fileContent1: string, fileContent2: string): MethodListTuple <<abstract>>
- GetMethods(tree: SyntaxTree): MethodInfo[]
Uses Roslyn Compiler
API to parse C# files.
JavaFileComparer
~ ExtractMethodsFromFiles(fileContent1: string, fileContent2: string): MethodListTuple <<abstract>>
- GetMethods(tree: JavaSyntaxTree): MethodInfo[]
- GetSyntaxTree(fileContent: string): JavaSyntaxTree
Uses ANTLR to parse
Java files.

Figure 9.2: All classes that support finding recurring bugs.

abstract because this task is language-specific. To implement the procedure for a given language, a subclass of *CodeFileComparer* has to be created to do the language-specific parsing of the code files. Implementations for Java and C# files are already provided. For parsing they use ANTLR [1] with a Java grammar and the Roslyn Compiler Platform [18] respectively. Language specific instances of *CodeFileComparer* are passed to the *SameMethodChangeFinder* in its constructor.

All version control system related tasks are handled by the class *GitFileRetriever*. It can load the content of a specific file for a specific commit from Git. The class is also able to load the previous version of a specific file from Git. *GitFileRetriever* requires Git to be installed on the system because it uses the Git executable internally.

9.3.3 Customizing and extending the tool

9.3.3.1 Customizing the tool

As already mentioned above, the tool supports Java and C# files out of the box. Java support is implemented by the classes *JavaSameMethodChangeFinder* and *JavaFileComparer* whereas C# support is implemented by the classes *CSharpSameMethodChangeFinder* and *CSharpFileComparer*. There are only few options that need to be changed to make the tool work in an environment similar to the one of this research project:

- In class *GitFileRetriever* set the constant *BaseCheckoutPath* to point to the base directory where all repositories have been cloned into as described in section 9.2.
- In class *GitFileRetriever* change the constant *PathToGit* to point to the Git executable.
- In the class *GitFileRetriever* modify the constant *ErrorFilePath* to point to a valid location for an error log file.

9.3.3.2 Extending the tool to support another programming language

Adding language support for a programming language other than Java or C# requires the creation of two language-specific classes. Assuming that Python language support should be added to the tool, the following steps need to be done:

1. Add a class *PythonFileComparer* that derives from *CodeFileComparer*. It is good practice to add it to the directory *Analysis* where the classes for the other supported languages reside.
2. Implement the abstract method *ExtractMethodFromFiles(string fileContent1, string fileContent2)*. It should extract all methods from the files passed as arguments and

return a tuple containing lists of those methods per file. It is left up to the reader how the parsing of the python code is implemented.

3. Add a new class *PythonSameMethodChangeFinder* that derives from the class *SameMethodChangeFinder* to the project. It is good practice to add it to the directory *Analysis* where the classes for the other supported languages reside.

4. Override the method *GetCommitsByFile()*. It should return all commits that changed a python source code file, grouped by file. You might want to add additional restrictions to the query. For instance, you could restrict the result to return only commits that belong to bugs.

5. In the constructor of *PythonSameMethodChangeFinder* set the constant *ResultFilePath*. The results of the analysis will be stored in this file.

6. Make sure that the base class constructor is being called and passed an *AppDbContext* and an instance of class *PythonFileComparer* created in step 1.

List of Figures

2.1 Setup for the analysis of the *EPOF* project. 8

3.1 Number of changed files per bug fix 15

4.1 Discovery of bugs over time 24
4.2 Time between the reporting of a bug and its resolution. 26
4.3 Bug fix commits to files over time 28
4.4 Details of figure 4.3 29
4.5 Bug fixes per component 31
4.6 Bugs per component 32

5.1 Testability of code affected by bugs 39
5.2 Wrong position of spinner 40

9.1 The entity relationship diagram of the database 61
9.2 All classes that support finding recurring bugs. 68

List of Tables

2.1 Information about the *EPOF* project 7
2.2 Some information about the EPOF project with respect to issues and commits 10

4.1 Aggregated results of mutation testing 36
4.2 Detailed results of mutation testing 36

5.1 Criteria to classify testability of code 38

9.1 Relevant Jira REST API methods 60
9.2 Scenario information for customization of the tool 63

Listing

1 Query to retrieve number of changed files per bug fix 12
2 Excerpt from class *DocumentsController* 41
3 Possible solution to avoid tight coupling. 44
4 Excerpt from class *SetupEPofPayListFragment* 46
5 Possible solution to avoid violations of SRP. 48

List of Algorithms

1 Procedure to detect recurring bugs 66
2 Function to compare file diffs on method level 67

Bibliography

[1] ANTLR. Antlr: Another tool for language recognition. `http://www.antlr.org/`, June 2018.

[2] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: Patterns, beliefs and behaviour. *IEEE Transactions on software engineering, Vol. 14, No. 8*, 2015.

[3] Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in GitHub. *Empirical Software Engineering*, 23(4):1931–1958, Aug 2018.

[4] Anna Derezińska and Piotr Trzpil. VisualMutator. `https://visualmutator.github.io/web/index.html`, July 2018.

[5] Michael C. Feathers. *Working effectively with legacy code*. Prentice Hall, 2005.

[6] .Net foundation. Xunit. `https://xunit.github.io/`, June 2018.

[7] Martin Fowler. *Refactoring - improving the design of existing code*. Addison-Wesley, 1999.

[8] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, July 2017.

[9] M. Ghafari, C. Ghezzi, and K. Rubinov. Automatically identifying focal methods under test in unit test cases. *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 61–70, Sept 2015.

[10] Github. DB browser for SQLite. `https://github.com/sqlitebrowser/sqlitebrowser`, June 2018.

[11] Atlassian Inc. Jira REST APIs. `https://developer.atlassian.com/server/jira/platform/rest-apis/`, June 2018.

[12] Microsoft Inc. Getting started with .Net Core. `https://www.microsoft.com/net/learn/get-started`, June 2018.

[13] Claus Klammer and Geort Buchgeher. A retrospective of production and test code co-evolution in an industrial project. 2018.

[14] Stanislav Levin and Amiram Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. 2017.

[15] S. Mahajan and W. G. J. Halfond. Detection and localization of HTML presentation failures using computer vision-based techniques. *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, April 2015.

[16] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2005.

[17] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[18] Microsoft. Roslyn compiler platform. `https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/`, June 2018.

[19] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. Automated reporting of GUI design violations for mobile apps. *CoRR*, abs/1802.04732, 2018.

[20] NinjaTurtles. .Net mutation testing. `http://www.mutation-testing.net/`, July 2018.

[21] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. pages 337–344, Nov 1992.

[22] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131, Dec 2008.

[23] Ken Schwaber and Jeff Sutherland. *The Scrum Guide*. 2016.

[24] Selenium. Selenium. `https://www.seleniumhq.org/`, June 2018.

[25] SonarQube. SonarQube. `https://www.sonarqube.org/`, June 2018.

[26] sqlitebrowser.org. DB browser for SQLite. `http://sqlitebrowser.org/`, June 2018.

[27] SQLite.org. SQLite documentation. `https://www.sqlite.org/docs.html`, June 2018.

[28] SQLite.org. SQLite downloads. `https://www.sqlite.org/download.html`, June 2018.

[29] The JUnit team. Junit. `https://junit.org/`, June 2018.

[30] Tricentis. Tosca. `https://www.tricentis.com/software-testing-tools/`, June 2018.

[31] Sten Vercammen, Mohammad Ghafari, Serge Demeyer, and Markus Borg. Goal-oriented mutation testing with focal methods. *Proceedings of 9th Workshop on Automated Software Testing, Florida, United States*, November 2018.