A Study of Bugs in Test Code and a Test Model for Analyzing Tests

by

Arash Vahabzadeh Sefiddarbon

B.Sc., Sharif University of Technology, Iran, 2014

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF APPLIED SCIENCE
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

October 2016

© Arash Vahabzadeh Sefiddarbon, 2016
Abstract

Testing has become a widespread practice among practitioners. Test cases are written to verify that production code functions as expected and are modified alongside the production code. Over time the quality of the test code can degrade. The test code might contain bugs, or it can accumulate redundant test cases or very similar ones with many redundant parts. The work presented in this dissertation has focused on addressing these issues by characterizing bugs in test code, and proposing a test model to analyze test cases and support test reorganization. To characterize the prevalence and root causes of bugs in the test code, we mine the bug repositories and version control systems of 448 Apache Software Foundation projects. Our results show that around half of all the projects had bugs in their test code; the majority of test bugs are false alarms, i.e., test fails while the production code is correct, while a minority of these bugs result in silent horrors, i.e., test passes while the production code is incorrect; missing and incorrect assertions are the dominant root cause of silent horror bugs; semantic, flaky, environment related bugs are the dominant root cause categories of false alarms. We present a test model for analyzing tests and performing test reorganization tasks in test code. Redundancies increase the maintenance overhead of the test suite and increase the test execution time without increasing the test suite coverage and effectiveness. We propose a technique that uses our test model to reorganize test cases in a way that reduces the redundancy in the test suite. We implement our approach in a tool and evaluate it on four open-source softwares. Our empirical evaluation shows that our approach can reduce the number of redundant test cases up to 85% and the test execution time by up to 2.5% while preserving the test suite’s behaviour.
Preface

The work presented in this thesis was conducted by the author, Arash Vahabzadeh, under the supervision of Professor Ali Mesbah. Second chapter of this thesis was also in collaboration with Amin Milanifard. I was responsible for devising the approach and the experiments, implementing the tools, running the experiments, evaluating and analyzing the results, and writing the manuscript. My collaborators guided me with the creation of the methodology and the analysis of results, as well as editing and writing portions of the manuscript. Parts of the results described in chapter 2 of this thesis were published as a conference paper in September 2015 in the Proceedings of the 31st International Conference on Software Maintenance and Evolution (ICSME)[52]. The results described in chapter 3 of this thesis are submitted to an IEEE software testing conference and are currently under review.
# Table of Contents

Abstract ............................................................... ii
Preface ................................................................. iii
Table of Contents .................................................... iv
List of Tables .......................................................... vi
List of Figures ........................................................ vii
Acknowledgments ....................................................... ix

1 Introduction ........................................................ 1
  1.1 Research Questions ............................................ 4
  1.2 Contributions .................................................. 4
  1.3 Thesis Organization ............................................ 5

2 An Empirical Study of Bugs in Test Code ..................... 6
  2.1 Methodology .................................................... 7
    2.1.1 Data Collection .......................................... 7
    2.1.2 Test Bug Categorization ................................ 10
    2.1.3 Test Bug Treatment Analysis ............................ 13
    2.1.4 FindBugs Study .......................................... 14
  2.2 Prevalence of Test Bugs ..................................... 15
  2.3 Categories of Test Bugs ..................................... 16
    2.3.1 Silent Horror Test Bugs ................................. 16
List of Tables

Table 2.1  Top 10 ASF projects sorted by the number of reported test bugs.  [15]
Table 2.2  Descriptive statistics of test bug reports.  . . . . . . . . . . . .  [16]
Table 2.3  Test bug categories for false alarms.  . . . . . . . . . . . . . . . . . .  [19]
Table 2.4  Test code warnings detected by FindBugs.  . . . . . . . . . . . . . .  [28]
Table 2.5  Performance of classifier for false alarm categories  . . . . . . . . . . . .  [29]
Table 2.6  Comparison of test and production bug reports.  . . . . . . . . . . . . . .  [30]
Table 2.7  Descriptive statistics of bugs reported by FindBugs.  . . . . . . . . . . . . . .  [30]
Table 3.1  Subject systems and their characteristics.  . . . . . . . . . . . . . . . . . .  [50]
Table 3.2  Partial redundancy in the test suites.  . . . . . . . . . . . . . . . . . . . . . [51]
Table 3.3  Test reduction.  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .  [51]
Table 3.4  TESTMODLER’s effectiveness in terms of execution time.  . . . . . . . . . . . . . . [54]
Table 3.5  Test suite’s coverage before and after reorganization.  . . . . . . . . . . . . . . . [56]
Table 3.6  TESTMODLER’s performance.  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . [56]
List of Figures

Figure 1.1 Different scenarios for fixing test and production bugs. . . . . [3]
Figure 2.1 Overview of the data collection phase. . . . . . . . . . . . . . [8]
Figure 2.2 Bug reports collected from bug repositories and version control systems. |(A ∪ B) − C| = 5,556 test bug reports in total (|B − A − C| = 3,849, |A − C| = 1,707). . . . . . . . . . . . . . . . [11]
Figure 2.3 Distribution of silent horror bug categories. . . . . . . . . . [17]
Figure 2.4 An example of a silent horror test bug due to a fault in for loop. [18]
Figure 2.5 An example of a silent horror test bug due to a missing assertion. [18]
Figure 2.6 Distribution of false alarms. . . . . . . . . . . . . . . . . [21]
Figure 2.7 Percentage of subcategories of test bugs. . . . . . . . . . [23]
Figure 2.8 Test bugs distribution based on testing phase in which bugs occurred. . . . . . . . . . . . . . . . [25]
Figure 2.9 Resource handling bug pattern in test code. . . . . . . . . [27]
Figure 2.10 Distribution of warnings reported by FindBugs. . . . . . . [31]
Figure 3.1 Test Cases from Apache Commons Project. . . . . . . . . [36]
Figure 3.2 Test suite and production code interaction. . . . . . . . . . . [37]
Figure 3.3 Extracted partial model for the running example. Ovals illustrate test states and rectangles illustrate labels of test statement edges. Dotted lines represent the state compatibility relations between test states and test statements. . . . . . . . . . . . . . . . [41]
Figure 3.4 Overview of our Approach. . . . . . . . . . . . . . . . . [43]
Figure 3.5 Test suite after reorganization. . . . . . . . . . . . . . . . [45]
| Figure 3.6 | Clustering test cases for reorganizing. | 46 |
| Figure 3.7 | Distribution of redundant statements and test cases in the test suite. | 52 |
| Figure 3.8 | Comparison of optimal and actual reductions for the number of test cases and test statements. | 55 |
Acknowledgments

I would like to thank my supervisor, Dr. Ali Mesbah, for his unwavering support, careful supervision and invaluable guidance throughout the course of this research. Without his critical reviews and intellectual inputs, the completion of this thesis would not have been possible for me. I am also sincerely thankful to Dr. Ivan Beschastnikh and Dr. Sathish Gopalakrishnan for accepting to be a part of my defence committee.

I would also like to thank my friends and colleagues at Software Testing and Analysis Lab who helped me with their support, encouragement and feedback. Last but certainly not least, I would like to thank my family specially my mother, for their love and endless support.
Chapter 1

Introduction

Software testing is an essential part of software development. As software systems are getting more complex in the last decades, developers rely more on software testing to ensure the quality of these systems. Developers write test cases to verify functionality of the software under test, detect bugs earlier in the software development process [58], and increase confidence and speed of software development activities [14]. Test cases are also used as regression tests to make sure previously working functionality still works, when the software evolves. Test code of a software system need to evolve alongside its production code [63]. New test cases need to be added or existing test cases need to be modified to cover new functionalities and bug fixes. Over time the quality of the test code can degrade. The test code might contain bugs, or it can accumulate redundant test cases or very similar ones with many redundant parts. The work presented in this thesis has focused on addressing these issues by characterizing bugs in test code, and proposing a test model to analyze test cases and support test reorganization to reduce redundancies in the test code.

Since test cases are code written by developers, they may contain bugs themselves. In fact, it is stated [44] and believed by many software practitioners [18, 36, 56] that “test cases are often as likely or more likely to contain errors than the code being tested”. Buggy tests can be divided into two broad categories [18]. First, a fault in test code may cause the test to miss a bug in the production code (silent horrors). These bugs in the test code can cost at least as much as bugs in
the production code, since a buggy test case may miss (regression) bugs in the production code. These test bugs are difficult to detect and may remain unnoticed for a long period of time. Second, a test may fail while the production code is correct (false alarms). While this type of test bugs is easily noticed, it can still take a considerable amount of time and effort for developers to figure out that the bug resides in their test code rather than their production code. Figure 1.1 illustrates different scenarios of fixing these test bugs.

Although the reliability of test code is as important as production code, unlike production bugs [51], test bugs have not received much attention from the research community thus far. In chapter 2 we present an extensive study on test bugs that characterizes their prevalence, impact, and main cause categories. To the best of our knowledge, this work is the first to study general bugs in test code.

We mine the bug report repository and version control systems of the Apache Software Foundation (ASF), containing over 110 top-level and 448 sub open-source projects with different sizes and programming languages. We manually inspect and categorize randomly sampled test bugs to find the common cause categories of test bugs.

Our results show that (1) around half of the Apache Software Foundation projects have had bugs in their test code; (2) the majority (97%) of test bugs result in false alarms, and their dominant root causes are “Semantic Bugs” (25%), “Flaky Tests” (21%), “Environmental Bugs” (18%), “Inappropriate Handling of Resources” (14%), and “Obsolete Tests” (14%); (3) a minority (3%) of test bugs reported and fixed pertain to silent horror bugs with “Assertion Related Bugs” (67%) being the dominant root cause; (4) developers contribute more actively to fixing test bugs and test bugs require less time to be fixed.

The results of our study indicate that test bugs do exist in practice and their bug patterns, though similar to that of production bugs, differ noticeably, which makes current bug detection tools ineffective in detecting them. Although current bug detection tools such as FindBugs and PMD do have a few simple rules for detecting test bugs, we believe that this is not sufficient and there is a need for extending these rules or devising new bug detection tools specifically geared toward test bugs.

Over time, a test suite can accumulate redundant test cases [13, 33]. Redundancy in tests increases the maintenance overhead and the test execution time, without
Figure 1.1: Different scenarios for fixing test and production bugs.

benefiting the test suite’s coverage or effectiveness. Different test-suite reduction (also called minimization) techniques [62] have been proposed for removing redundant test cases. However, test minimization techniques have two shortcomings, namely (1) they use code coverage as a guideline to remove whole redundant test cases. This can potentially remove a test case that has similar coverage as other test cases but different test statements and assertions. Assertions are known to directly affect test suite effectiveness [66], and (2) because they work at the whole test case level, they cannot target fine-grained redundancies within statements of a test case.

To be able to remove low-level redundancies, we would need an approach that can reorganize test cases at the statement level. However, reorganizing (or refactoring) test cases in general is not a straightforward task. Developers use the test suite to preserve the behaviour of the system when production code is refactored. However, there is not such a safety net when a test suite needs to go through internal reorganization. Thus, any technique for this purpose should reorganize test cases in a way that preserves the behaviour of the test suite.

In chapter 3, we propose a fine-grained analysis approach for inferring a test model. Our test model can be used for performing test reorganizational tasks while ensuring that the behaviour of the test suite remains the same. As opposed to current test reduction techniques that use coverage criteria, we model the actual behaviour of the test suite by capturing the production method calls with their inputs. Our analysis is performed on the test statement level as opposed to whole test case level. We use our fine-grained analysis and test model to remove redundancies in test cases. Our empirical evaluation shows that our approach can reduce the number of redundant test cases up to 85% and reduce the test execution time by up to 2.5%, while preserving the test suite behaviour.
1.1 Research Questions

To improve the quality of test code we designed two high-level research questions:

**RQ1.** *What are the characteristics of bugs in test code?*
We conduct the first large scale empirical study of bugs in test code to characterize prevalence, impact, and root cause categories of bugs in test code.

**RQ2.** *How can we automatically reduce statement-level redundancies in the test code?*
We propose a test suite model to support statement-level test reorganization. We use our model to reorganize test statements in test cases in a way that reduces the statement-level redundancies and execution time while preserving the test suite behaviour.

1.2 Contributions

Our work makes the following main contributions:

- We mine 5,556 fixed bug reports reporting test bugs by searching through the bug repository and version control systems of the Apache projects.
- We systematically categorize a total of 443 test bugs into multiple bug categories.
- We compare test bugs with production bugs in terms of the amount of attention received and time to fix.
- We assess whether existing bug detection tools such as FindBugs can detect test bugs.
- We propose a fine-grained test analysis method that works at the test statement level, a test model for identifying behaviour-preserving refactorings in a test suite, and a technique and algorithm that uses our test model to reduce redundancies in the test suite by reorganizing partly redundant test cases and removing the redundant parts.
- We implement our approach in a tool called TESTMODLER, which is publicly available [11].
We empirically evaluate our approach by reorganizing the test suites of four real-world open source applications in a way that reduces their redundancies and execution time.

The following paper has been published in response to RQ1, and a paper submission that addresses RQ2, is currently under review at an IEEE software testing conference.


1.3 Thesis Organization

In chapter 2 of this thesis, we present the experimental methodology, results, and implications of the large-scale empirical study that we conducted to characterize impact and root causes of test bugs. In chapter 3, we describe in depth the proposed test suite model and the automated technique that uses the test model to reorganize test cases in a way that reduces statement level redundancies and execution time. Chapter 4 discusses the related work, and chapter 5 concludes the thesis and describes the possible future research directions.
Chapter 2

An Empirical Study of Bugs in Test Code

Summary\(^1\)

Testing aims at detecting (regression) bugs in production code. However, testing code is just as likely to contain bugs as the code it tests. Buggy test cases can silently miss bugs in the production code or loudly ring false alarms when the production code is correct. We present the first empirical study of bugs in test code to characterize their prevalence and root cause categories. We mine the bug repositories and version control systems of 448 Apache Software Foundation (ASF) projects and find 5,556 test-related bug reports. We (1) compare properties of test bugs with production bugs, such as active time and fixing effort needed, and (2) qualitatively study 443 randomly sampled test bug reports in detail and categorize them based on their impact and root causes. Our results show that (1) around half of all the projects had bugs in their test code; (2) the majority of test bugs are false alarms, i.e., test fails while the production code is correct, while a minority of these bugs result in silent horrors, i.e., test passes while the production code is incorrect; (3) incorrect and missing assertions are the dominant root cause of silent horror bugs; (4) semantic (25\%), flaky (21\%), environment-related (18\%) bugs are

\(^1\)This chapter is an extension of the study appeared at the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015\(^{[52]}\).
the dominant root cause categories of false alarms; (5) the majority of false alarm bugs happen in the exercise portion of the tests, and (6) developers contribute more actively to fixing test bugs and test bugs are fixed sooner as compared to production bugs. In addition, we evaluate the ability of existing bug detection tools to detect bugs in test code.

2.1 Methodology

Our goal is to understand the prevalence and categories of bugs in test code. We conduct quantitative and qualitative analyses to address the following research questions:

RQ1: How prevalent are test bugs in practice?
RQ2: What are common categories of test bugs?
RQ3: Are test bugs treated differently by developers as compared to production bugs?
RQ4: Are current bug detection tools able to detect test bugs?

All of our empirical data is available for download [1].

2.1.1 Data Collection

Figure 2.1 depicts an overview of our data collection, which is conducted in two steps: mining of bug repositories for test-related bug reports (A), and analyzing commits in version control systems (B and C).

Mining Bug Repositories

One of the challenges in collecting test bug reports is distinguishing between bug reports for test code and production code. In fact, most search and filtering tools in current bug repository systems do not support this distinction. In order to identify bug reports reporting a test bug, we selected the JIRA bug repository of the Apache Software Foundation (ASF) since its search/filter tool allows us to specify the type and component of reported issues. We mine the ASF JIRA bug repository [9], which contains over 110 top-level and 448 sub open-source projects, with various sizes and programming languages.
We search the bug repository by selecting the type as “Bug”, component as “test”, and resolution as “Fixed”.

**Type.** The ASF JIRA bug report types can be either “Bug”, “Improvement”, “New Feature”, “Test”, or “Task”. However, we observed that most of the reported test-related bugs have “Bug” as their type. The “Test” label is mainly used when someone is contributing extra tests for increasing coverage and testing new features.

**Component.** The ASF bug repository defines components for adding structure to issues of a project, classifying them into features, modules, and sub-projects [45]. We observed that many projects in ASF JIRA use this field to distinguish different modules of the project. Specifically, they use “test” for the component field to refer to issues related to test code.
Resolution. We only consider bug reports with resolution “Fixed” because if a reported bug is not fixed, it is difficult to verify that it is a real bug and analyze its root causes.

Analyzing Version Control Commits

Since our search query used on the bug repository is restrictive, we might miss some test bugs. Therefore, we augment our data by looking into commits of the version control systems of the ASF projects, Similar to [42]. We use the read-only Git mirrors of the ASF codebases [7], which “contain full version histories (including branches and tags) from the respective source trees in the official Subversion repository at Apache”; thus using these mirrors does not threaten the validity of our study. We observed that most commits associated with a bug report mention the bug report ID in the commit message. Therefore, we leverage this information to distinguish between bug reports reporting test bugs and production bugs. We extract test bugs through the following steps:

Finding Commits with Bug IDs. We clone the Git repository of each Apache project and use JGIT [8] to traverse the commits. In the ASF bug repository, every bug report is identified using an ID composed of \{PROJECTKEY\}-#BUGNUM where PROJECTKEY is a project name abbreviation. Using this pattern, we search in the commit messages to find if a commit is associated with a bug report in JIRA. Once we have the ID, we can seamlessly retrieve the data regarding the bug report from JIRA.

Identifying Test Commits. For each commit associated with a bug report, we compute the diff between that commit and its parents. This enables us to identify files that are changed by the commit, which in turn allows us to identify test commits, i.e., commits that only change files located in the test directory of a project. We refer to commits that change at least one file outside test directories as production commits. If a project is using Apache Maven, we automatically extract information about its test directory from the pom.xml file. Otherwise, we consider any directory with “test” in its name as a test directory; we also manually verify that these are test directories.

---

2We ignored auxiliary files such as .gitignore and *.txt.
This phase resulted in two sets of bug reports, namely (1) those associated with a test commit (block B in Figure 2.1), and (2) those associated with a production commit (block C in Figure 2.1). Since a bug report can be associated with both test and production commits, in our analysis we only consider bug reports that are associated with test commits but not with any production commit (set $B - C$ in the venn diagram of Figure 2.2).

2.1.2 Test Bug Categorization

Manual Test Bug Categorization

To find common categories of test bugs (RQ2), we manually inspect the test bug reports. Manual inspection is a time consuming task; on average, it took us around 12 minutes per bug report to study the comments, patches, and source code of any changed files. Therefore, we decided to sample the mined test-related bug reports from our data collection phase.

**Sampling.** We computed the union of the bug reports obtained from mining the bug reports (Section 2.1.1) and the version control systems (Section 2.1.1). This union is depicted as a grey set of $(A \cup B) - C$ in the venn diagram of Figure 2.2. We randomly sampled 499 ($\approx 9.0\%$) of the unique bug reports from this set.

**Categorization.** For the categorization phase, we leverage information from each sampled bug report’s description, discussions, proposed patches, fixing commit messages, and changed source code files.

First, we categorize each test bug in one of the two main impact classes, of *false alarms*, i.e., test fails while the production code is correct, or *silent horrors*, i.e., test passes while the production code is or could be incorrect. We adopt the terms false alarms and silent horrors coined by Cunningham [18].

Second, we infer common cause categories while inspecting each bug report. When three or more test bugs exhibited a common pattern, we added a new category. Subcategories also emerged to further subdivide the main categories.

Finally, we also categorize test bugs with respect to the location (in the test case) or unit testing phase in which they occur as follows:
1. **Setup.** Setting up the test fixture, e.g., creating required files, entries in databases, or mock objects.

2. **Exercise.** Exercising the software under test, e.g., by instantiating appropriate object instances, calling their methods, or passing method arguments.

3. **Verify.** Verifying the output or changes made to the states, files, or databases of the software under test, typically through test assertions.

4. **Teardown.** Tearing down the test, e.g., closing files, database connections, or freeing allocated memories for objects.

The categorization step was a very time consuming task and was carried out through several iterations to refine categories and subcategories; the manual effort for these iterations was more than 400 hours, requiring more than 100 hours for each iteration.

**Automatic Test Bug Categorization**

We leveraged machine learning and natural language processing techniques to further categorize false alarm test bugs into their sub-categories. For each bug report...
we used textual information of bug report such as title, description and its comments. We used other informations such as relation of this bug report to other bug reports (for example indication that this bug report is broken because of another bug report). We also used the bug report’s fixing commit and the changes that were made to the source code. We performed the automatic categorization through the following steps:

1. **Preprocessing.** *Text level features:* First, we performed a data cleaning phase, we deleted code snippets, stack traces and auto generated comments (auto generated comments about test results of commits and code quality measurements) from bug reports. We also used a synonym list and replaced each term with its synonym, for example all names of different operating system such as Windows, Ubuntu, Cent OS, etc., are synonyms of word “Operating System” and replaced by this more general term. To leverage textual information of bug reports we used bag of word approach and computed Term Frequency-Inverse Document Frequency (TF-IDF) after performing stemming.

   *Source level features:* We also used information stored in fixing commit of bug reports, we used GumTree AST differencing tools [22] to compute the changes made to the source code by each fixing commit. For each commit we collected name of method calls that one of their arguments is changed as part of commit, percentage of changes made to body of control flow statements (*if, while, for*), name of methods that their body has changed (especially if *tearDown* and *setUp* methods are changed), class accesses, instantiated classes, string arguments added and method annotation changes. Since, most of projects in our set of test bug reports use Java programming language we only consider those bug reports that their programming language is Java.

2. **Training.** For training we used Support Vector Machine with Sequential Minimal Optimization (SMO) machine learning algorithm. To train the Support Vector Machine we used the dataset of labelled bug reports collected by manual test bug categorization phase. We randomly selected 70% of bug reports for training and validation, and used the other 30% of bug reports for testing. In this way, since the testing set is not used for training, performance
of our classifier on this testing set can be representative of its performance on
data set of whole Jira bug repository’s test bug reports.

3. **Classifier Evaluation.** To evaluate our classifier we measured its perfor-
mance on the test set, we measured precision, recall and F1 for each category
classified by the classifier.

4. **Automatic Classification.** We used our classifier to automatically classify
all test bug reports we gathered by our data collection phase.

### 2.1.3 Test Bug Treatment Analysis

To answer RQ3, we measure the following metrics for each bug report:

**Priority:** In JIRA, the priority of a bug report indicates its importance in relation
to other bug reports. For the Apache projects we analyzed, this field had
one of the following values: *Blocker, Critical, Major, Minor or Trivial*. For
statistical comparisons, we assign a ranking number from 5 to 1 to each,
respectively.

**Resolution time:** The amount of time taken to resolve a bug report starting from
its creation time.

**Number of unique authors:** Number of developers involved in resolving the issue
(based on their user IDs).

**Number of comments:** Number of comments posted for the bug report. It captures
the amount of discussions between developers.

**Number of watchers:** Number of people who receive notifications; an indication
of the number of people interested in the fate of the bug report.

The metrics *priority, number of unique authors, comments, and watchers* can
be used as an indication as to what extent developers contribute for fixing a bug.
*Resolution time* metric indicates how much time is needed for fixing a particular
bug. We collected these metrics for all the test bug reports and all the production
bug reports, separately. For the comparison analysis, we only included projects
that had at least one test bug report. To obtain comparable pools of data points, the
number of production bug reports that we sampled, were the same as the number of
test bug reports mined from each project.
2.1.4 FindBugs Study

To answer RQ4, we use FindBugs [35], a popular static byte-code analyzer in practice for detecting common patterns of bugs in Java code. We investigate its effectiveness in detecting bugs in test code.

Detecting Bugs in Tests

We run FindBugs (v3.0.0)\(^3\) on the test code as well as the production code of latest version of Java ASF projects that use Apache Maven (see Figure 2.1 (D)). Compiling projects that do not use Maven requires much manual effort, for instance in resolving dependencies on third party libraries. Also we noticed that FindBugs crashes while running on some of the projects. In total, we were able to successfully run FindBugs on 129 of the 448 ASF sub-projects.

Analysis of Bug Patterns Found by FindBugs

We parse the XML output of FindBugs and extract patterns from the reported bugs. FindBugs statically analyzes byte code of Java programs to detect simple patterns of bugs in the byte code. This is done by applying static analysis techniques such as control and data flow analyses. Among patterns of bugs that FindBugs detects, we only considered reported Correctness and Multithreaded Correctness as others, such as internationalization, bad practice, security or performance, are more related to non-functional bugs.

Effectiveness in Detecting Test Bugs

To evaluate FindBugs’ effectiveness in detecting test bugs, we choose a similar approach used by Couto et al. [17]. We sample 50 bug reports from projects that we can compile the version containing the bug, just before the fix. By comparing the versions before and after a fix, we are able to identify the set of methods that are changed as part of the fix. We run FindBugs on the version before and after the fix to see if FindBugs is able to detect the test bug and could have potentially prevented it. If FindBugs reports any warning in any of the methods changed by the fix and these warnings disappear after the fix, we assume that FindBugs is able to detect the

\(^3\)http://findbugs.sourceforge.net
Table 2.1: Top 10 ASF projects sorted by the number of reported test bugs.

<table>
<thead>
<tr>
<th>Project</th>
<th>Production Code KLOC</th>
<th>Test Code KLOC</th>
<th># Test Bug Reports</th>
</tr>
</thead>
<tbody>
<tr>
<td>Derby</td>
<td>386</td>
<td>370</td>
<td>614</td>
</tr>
<tr>
<td>HBase</td>
<td>587</td>
<td>195</td>
<td>440</td>
</tr>
<tr>
<td>Hive</td>
<td>836</td>
<td>124</td>
<td>295</td>
</tr>
<tr>
<td>Hadoop HDFS</td>
<td>101</td>
<td>57</td>
<td>286</td>
</tr>
<tr>
<td>Hadoop Common</td>
<td>1249</td>
<td>380</td>
<td>279</td>
</tr>
<tr>
<td>Hadoop MapReduce</td>
<td>60</td>
<td>24</td>
<td>231</td>
</tr>
<tr>
<td>Accumulo</td>
<td>405</td>
<td>78</td>
<td>187</td>
</tr>
<tr>
<td>Qpid</td>
<td>553</td>
<td>93</td>
<td>152</td>
</tr>
<tr>
<td>Jackrabbit Content Repository</td>
<td>247</td>
<td>107</td>
<td>145</td>
</tr>
<tr>
<td>CloudStack</td>
<td>1361</td>
<td>228</td>
<td>111</td>
</tr>
</tbody>
</table>

associated test bug. Note that it is possible that the disappeared warnings are false positives or unrelated to actual fix. However, we make this assumption to make the results comparable to the results obtained by [17] for the efficacy of static analysis tools on production bugs. We also manually examine these warnings to make sure they are related to the fix and the associated test bug.

The next four sections present the results of our study for each research question, subsequently.

2.2 Prevalence of Test Bugs

Overall, our analysis reveals that 47% of the ASF sub-projects (211 out of 448) have had bugs in their tests. Our search query on the JIRA bug repository retrieved 2,040 bug reports. After filtering non-test related reports, we obtained 1,707 test bug reports, shown as $A - C$ in the venn diagram of Figure 2.2. The search in version control systems resulted in 4,982 bug reports associated only with test commits, depicted as the set $B - C$ in Figure 2.2. In total, we found 5,556 unique test bug reports ($(A \cup B) - C$). Table 2.2 presents descriptive statistics for the number of test bug reports and Table 2.1 shows the top 10 ASF projects in terms of the number of test bug reports we found in their bug repository[^4]. For additional fine-grained data per project we refer the reader to our online report [1].

[^4]: Source lines of code is for all programming languages used in project, measured with CLOC: http://cloc.sourceforge.net

15
Table 2.2: Descriptive statistics of test bug reports.

<table>
<thead>
<tr>
<th>Min</th>
<th>Mean</th>
<th>Median</th>
<th>σ</th>
<th>Max</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12.4</td>
<td>0</td>
<td>48.3</td>
<td>614</td>
<td>5,556</td>
</tr>
</tbody>
</table>

Finding 1: Around half of all the projects analyzed had bugs in their test code that were reported and fixed. On average, there were 12.4 fixed test bugs per project.

2.3 Categories of Test Bugs

We manually examined the 499 sampled bug reports; 56 of these turned out to be difficult to categorize due to a lack of sufficient information in the bug report. We categorized the remaining 443 bug reports. Table 2.3 shows the main categories and their subcategories that emerged from our manual analysis. Our results show that a large number of reported test bugs result in a test failure (97%), and a small fraction pertains to silent test bugs that pass (3%).

2.3.1 Silent Horror Test Bugs

Silent test bugs that pass are much more difficult to detect and report compared to buggy tests that fail. Hence, it is not surprising that only about 3% of the sampled bug reports (15 out of 443) belong to this category.

Figure 2.3 depicts the distribution of silent horror bug categories in terms of the location of the bug. In five instances, the fault was located in the exercise step of the test case, i.e., the fault caused the test not to execute the SUT for the intended testing scenario, which made the test useless. For instance, as reported in bug report JCR-3472, due to a fault in the test code of the Apache Jackrabbit project, queries in LargeResultSetTest run against a session where the test content is not visible and thus the resulting set is empty and the whole test is pointless. In another example, due to the test dependency between two test cases, one of test cases “is actually testing the GZip compression rather than the DefaultCodec due to the setting hanging around from a previous test” (FLUME-571). Such issues could explain why these bugs remain unnoticed and are difficult to detect.
The other 10 instances were located in the verification step, i.e., they all involved test assertions. From these, six pertained to a missing assertion and four were related to faults in the assertions, which checked a wrong condition or variable.

Interestingly, two of the silent test bugs resulted in a failure when they were fixed, indicating a bug in the production code that was silently ignored. For example, in ACCUMULO-1878, 1927, 1988 and 1892, since the test did not check the return value of the executed M/R jobs, these jobs were failing silently (ACCUMULO-1927), when this was fixed, the test failed. Figure 2.4 shows the fixing commit for HBASE-7901, a bug in the for loop condition that caused the test not to execute the assertion.

Although JUnit 4 permits to assert a particular exception through the expected annotation and ExpectedException rule, many testers are used to or prefer [26] using the traditional combination of try/catch and fail() assertion type to achieve this goal. However, this pattern tends to be error-prone. In our sampled list, four out of 15 silent bugs involved incorrect usage of the try/catch and in combination with the fail() primitive. For example, Figure 2.5 shows the fixing commit for the bug report JCR-500; the test needs to assert that unregistering a namespace that is not registered should throw an exception. However, a fail() assertion is missing from the code making the whole test case ineffective. Another pattern of this type of bug is when the SUT in the try block can throw multiple exceptions and the tester does not assert on the type of the thrown exception. It is
Figure 2.4: An example of a silent horror test bug due to a fault in `for` loop.

```java
for (int j = 0; j < cr.getFiles().size(); j++) {
    for (int j = 0; j < cr.getFiles().size(); j++) {
        assertTrue(cr.getFiles().get(j).getReader().getMaxTimestamp() < (System.currentTimeMillis() ←
            this.store.getScanInfo().getTtl()));
}
```

Figure 2.5: An example of a silent horror test bug due to a missing assertion.

```java
try {
    nsp.unregisterNamespace("NotCurrentlyRegistered");
    fail("Trying to unregister an unused prefix must fail");
} catch (NamespaceException e) {
    // expected behaviour
}
```

worth mentioning that two of these 15 bugs could have potentially been detected statically; in one case (ACCUMULO-828), the whole test case did not have any assertions, and in another (SLIDER-41) a number of test cases were not executed because they did not comply with the test class name conventions of Maven, i.e., their name did not start with “Test”.

Finding 2: Silent horror test bugs form a small portion (3%) of reported test bugs. Assertion-related faults are the dominant root cause of silent horror bugs.
Table 2.3: Test bug categories for false alarms.

<table>
<thead>
<tr>
<th>Category</th>
<th>Subcategory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic Bugs</td>
<td>S1. Assertion Fault</td>
<td>Fault in the assertion expression or arguments of a test case.</td>
</tr>
<tr>
<td></td>
<td>S2. Wrong Control Flow</td>
<td>Fault in a conditional statement of a test case.</td>
</tr>
<tr>
<td></td>
<td>S3. Incorrect Variable</td>
<td>Usage of the wrong variable.</td>
</tr>
<tr>
<td></td>
<td>S4. Deviation from Test Requirement and</td>
<td>A missing step in the exercise phase, missing a possible scenario, or when</td>
</tr>
<tr>
<td></td>
<td>Missing Cases</td>
<td>test case deviates from actual requirements.</td>
</tr>
<tr>
<td></td>
<td>S5. Exception Handling</td>
<td>Wrong exception handling.</td>
</tr>
<tr>
<td></td>
<td>S6. Configuration</td>
<td>Configuration file used for testing is incorrect or test does not consider</td>
</tr>
<tr>
<td></td>
<td>S7. Test Statement Fault or Missing</td>
<td>these configurations.</td>
</tr>
<tr>
<td></td>
<td>Statements</td>
<td>A statement in a test case is faulty or missing.</td>
</tr>
<tr>
<td>Environment</td>
<td>E1. Differences in Operating System</td>
<td>Tests in this category pass on one OS but fail on another one.</td>
</tr>
<tr>
<td></td>
<td>E2. Differences in third party libraries or JDK versions and vendors</td>
<td>Failure is due to incompatibilities that exist between different versions of JDK or different implementations of JDK by different vendors, or different versions of third party libraries.</td>
</tr>
<tr>
<td>Resource Handling</td>
<td>I1. Test Dependency</td>
<td>Running one test affects the outcome of other tests.</td>
</tr>
<tr>
<td></td>
<td>I2. Resource Leak</td>
<td>A test does not release its acquired resources properly.</td>
</tr>
<tr>
<td>Flaky Tests</td>
<td>F1. Asynchronous Wait</td>
<td>Test failure is due to an asynchronous call and not waiting properly for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the result of the call.</td>
</tr>
<tr>
<td></td>
<td>F2. Race Condition</td>
<td>Test failure is due to non-deterministic interactions of different threads.</td>
</tr>
<tr>
<td></td>
<td>F3. Concurrency Bugs</td>
<td>Concurrency issues such as deadlocks and atomicity violations.</td>
</tr>
<tr>
<td>Obsolete tests</td>
<td>O1. Obsolete Statements</td>
<td>Statements in a test case are not evolved when production code has evolved.</td>
</tr>
<tr>
<td></td>
<td>O2. Obsolete Assertions</td>
<td>Assertion statements are not evolved as production code evolves.</td>
</tr>
</tbody>
</table>
2.3.2 False Alarm Test Bugs

We categorized the 428 bug reports that were false alarms based on their root cause. We identified five major causes for false alarms. Figure 2.6 shows the distribution for each main category and also testing phase in which false alarm bug occurred.

**Finding 3**: Semantic bugs (25%) and Flaky tests (21%) are the dominant root causes of false alarms, followed by Environment (18%) and Resource handling (14%) related causes. The majority of false alarm bugs occur in the exercise phase of testing.

Semantic Bugs

This category consists of 25% of the sampled test bugs. Semantic bugs reflect inconsistencies between specifications and production code, and test code. Based on our observations of common patterns of these bugs, we categorized them into seven subcategories as shown in Table 2.3. Figure 2.7a presents percentages of each subcategory, and Figure 2.8a shows the fault location distribution in the testing phase.

The majority of test bugs in semantic bug category (33%) belongs to tests that miss a case or deviate from test requirements (S4). Examples include tests that miss setting some required properties of the SUT (e.g., CLOUDSTACK-2542 and MYFACES-1625), or tests that miss a required step to exercise the SUT correctly (e.g., HDFS-824). Test statement faults or missing statements account for 19% of bugs in this category. For example in CLOUDSTACK-3796, a statement fault resulted in ignoring to set the attributes needed for setting up the test correctly, thus resulting in a failure. The use of an incorrect variable, which may result in asserting the wrong variable (e.g., DERBY-6716) or a wrong test behaviour was observed in 9% of the semantic bugs. 7% of semantic bugs in our sampled bugs were due to improper exception handling in test code, which resulted in false test failures (e.g., JCR-505). Some tests require reading properties from an external configuration file to run with different parameters without changing the test code itself; however, some tests did not use these configurations properly or in some other cases these configurations were buggy themselves. 7% of the false alarm bugs had this issue. We categorized a bug in the wrong control flow category if the test failed due to
a fault in a conditional statement (e.g., if, for or while conditional). 5% of semantic bugs belong to this category. Another 5% of semantic bugs were due to faulty assertions (e.g., JCR-503).

**Finding 4: Deviations from test requirements or missing cases in exercising the SUT (33%) and faulty or missing test statements (19%) are the most prevalent semantic bugs in test code.**
Environment

Around 18% of bug reports pertained to a failing test due to environmental issues, such as differences in path separators in Windows and Unix systems. In this case, tests pass under the environment they are written in, but fail when executed in a different environment. Since open source software developers typically work in diverse development environments, this category accounts for a large portion of the test bug reports filed.


Figure 2.7: Percentage of subcategories of test bugs.
Figure 2.7b and Figure 2.8b show the distribution of environmental bugs and their fault locations. About 61% of the bug reports in this category were due to operating system differences (E1), and particularly differences between the Windows and Unix operating systems. Testers make platform-specific assumptions that may not hold true in other platforms — e.g., assumptions about file path and classpath conventions, order of files in a directory listing, and environment variables (MAPREDUCE-4983). Some of the common causes we observed that result in failing tests in this category include: (1) Differences in path conventions — e.g., Windows paths are not necessarily valid URIs while Unix paths are, or Windows uses quotation for dealing with spaces in file names but in Unix spaces should be escaped (HADOOP-8409). (2) File system differences — e.g., in Unix one can rename, delete, or move an opened file while its file descriptor remains pointing to a proper data; however, in Windows opened files are locked by default and cannot be deleted or renamed (FLUME-349). (3) File permission differences — e.g., default file permission is different on different platforms. (4) Platform-specific use of environmental variables — e.g., Windows uses the \%ENVVAR\% and Unix uses the \$ENVVAR\ notations to retrieve environmental variable values (MAPREDUCE-4869). Also classpath entries are separated by ‘;’ in Windows and by ‘:’ in Unix.

Differences in JDK versions and vendors (E2) were responsible for 26% of environment related test bugs. For example, with IBM JDK developers should use SSLContext.getInstance(‘\"SSL_TLS\") instead of “SSL” in Oracle JDK, to ensure the same behaviour (FLUME-2441). There is also compatibility issues between different versions of JDKs, e.g., testers depended on the order of iterating a HashMap, which was changed in IBM JDK 7 (FLUME-1793).

**Finding 5**: 61% of environmental false alarms are platform-specific failures, caused by operating system differences.
Figure 2.8: Test bugs distribution based on testing phase in which bugs occurred.
Inappropriate Handling of Resources

Ideally, test cases should be independent of each other, however, in practice this is not always true, as reported in a recent empirical study [65]. Around 14% of bug reports (61 out of 428) point to inappropriate handling of resources, which may not cause failures on their own, but cause other dependent tests to fail when those resources are used. Figure 2.7c shows the percentage for sub-categories of resource handling bugs and Figure 2.8c shows the distribution of testing phases in which the fault occurs. About 61% of these bugs were due to test dependencies.

A good practice in unit testing is to mitigate any side-effects a test execution might have; this includes releasing locally used resources and rolling back possible changes to external resources such as databases. Most of unit testing frameworks provide opportunities to clean up after a test run, such as the tearDown method in JUnit 3 or methods annotated with @After in JUnit 4. However, testers might forget or fail to perform this clean up step properly. One common mistake is when a test that changes some persistent data (or acquires some resources), conducts the clean up in the test method’s body. In this case, if the test fails due to assertion failures, exceptions or time outs, the clean up operation will not take place causing other tests or even future runs of this test case to fail. Figure 2.9 illustrates this bug pattern and its fix. Another common problem we observed is that testers forgot to call the super.tearDown() or super.setUp() methods and this prevents their superclass to free acquired resources (DERBY-5726). Bug detection tools such as FindBugs can detect these types of test bugs.

Finding 6: 61% of inappropriate resource handling bugs are caused by dependent tests. More than half of all resource handling bugs occur in the teardown phase of test cases.

Flaky Tests

These test bugs are caused by non-deterministic behaviour of test cases, which intermittently pass or fail. These tests, also known as ‘flaky tests’ by practitioners, are time consuming for developers to resolve, because they are hard to reproduce [21]. A recent empirical study on flaky tests [42] revealed that the main root cause for flaky tests is Async Wait, which happens when a test does not wait properly for
Figure 2.9: Resource handling bug pattern in test code.

a asynchronous call, and Race Condition, which is due to interactions of different threads, such as order violations. Our results are also in line with their findings; we found that not waiting properly for asynchronous calls (46%) is the main root cause of flaky tests, followed by race conditions between different threads (Figure 2.7d). As shown by Figure 2.8d, most of flaky test bugs (51%) are due to bugs in exercise phase of tests.

Finding 7: The majority of flaky test bugs occur when the test does not wait properly for asynchronous calls during the exercise phase of testing.

Obsolete Tests

Ideally, test and production code should evolve together, however, in practice this is not always the case [63]. An obsolete test [32] is a test case that is no longer valid due to the evolution of the specifications and production code of the program under test. Obsolete tests check features that have been modified, substituted, or removed. When an obsolete test fails, developers spend time examining recent changes made
Table 2.4: Test code warnings detected by FindBugs.

<table>
<thead>
<tr>
<th>Bug Description</th>
<th>Bug Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inconsistent synchronization</td>
<td>Flaky</td>
<td>29.8%</td>
</tr>
<tr>
<td>Possible null pointer dereference in method on exception path</td>
<td>Semantic</td>
<td>17.6%</td>
</tr>
<tr>
<td>Using pointer equality to compare different types</td>
<td>Semantic</td>
<td>8.8%</td>
</tr>
<tr>
<td>Possible null pointer dereference</td>
<td>Semantic</td>
<td>7.3%</td>
</tr>
<tr>
<td>Class defines field that masks a superclass field</td>
<td>Semantic</td>
<td>3.9%</td>
</tr>
<tr>
<td>Nullcheck of value previously dereferenced</td>
<td>Semantic</td>
<td>2.9%</td>
</tr>
<tr>
<td>An increment to a volatile field isn’t atomic</td>
<td>Flaky</td>
<td>2.9%</td>
</tr>
<tr>
<td>Method call passes null for nonnull parameter</td>
<td>Semantic</td>
<td>2.4%</td>
</tr>
<tr>
<td>Incorrect lazy initialization and update of static field</td>
<td>Flaky</td>
<td>2.4%</td>
</tr>
<tr>
<td>Null value is guaranteed to be dereferenced</td>
<td>Semantic</td>
<td>2.0%</td>
</tr>
</tbody>
</table>

to production code as well as the test code itself to figure out that the failure is not a bug in production code.

As shown in Figure 2.8e, developers mostly need to update the exercise phase of obsolete tests. This is expected as adding new features to production code may change the steps required to execute the SUT, however, may not change the expected correct behaviour of the SUT, i.e., assertions. In fact, as depicted in Figure 2.7e, only 23% of obsolete tests required a change to assertions.

**Finding 8:** The majority of obsolete tests require modifications in the exercise phase of test cases, and mainly in normal statements (77%) rather than assertions.

### 2.4 Automatic Test Bug Classification

We used the training set which is randomly chosen 70 percent of our sampled set for choosing the best set of features and comparing different algorithms. For evaluation we used the remaining part of our sampled set (30 percent) as our test set. Table 2.5 shows performance of our classifier in terms of precision, recall and F1 metrics. We ran it on all of our false alarm test bug reports set and automatically categorized them into semantic, flaky, environmental, resource related and obsolete categories. Table 2.5 also shows percentage of each category obtained by automatic classification. As shown in Table 2.5, the percentages obtained by automatic
Table 2.5: Performance of classifier for false alarm categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage in Jira</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic</td>
<td>29.21 %</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>Flaky</td>
<td>18.19 %</td>
<td>0.78</td>
<td>0.7</td>
<td>0.74</td>
</tr>
<tr>
<td>Environmental</td>
<td>20.47 %</td>
<td>0.62</td>
<td>0.77</td>
<td>0.68</td>
</tr>
<tr>
<td>Resources</td>
<td>10.20 %</td>
<td>0.71</td>
<td>0.39</td>
<td>0.5</td>
</tr>
<tr>
<td>Obsolete</td>
<td>21.93 %</td>
<td>0.34</td>
<td>0.5</td>
<td>0.4</td>
</tr>
</tbody>
</table>

categorization is in line with the manual categorization. This means that our sampled data set was representative of the whole data set.

2.5 Test Bugs vs Production Bugs

Table 2.6 shows the median, mean, standard deviation, and maximum of each metric defined in subsection 2.1.3, for test bugs (TE) and production bugs (PR). We used the nonparametric Mann-Whitney U tests to compare the distribution of test bugs with that of production bugs and compute the p-values. The p-value is close to zero because of a large sample size effect; we computed effect size — standardized mean difference (d) and odds ratio (OR) — to compare the meaningfulness of differences. The results indicate that test bugs take less time to be fixed compared to production bugs. Although the priority assigned to test bugs and production bugs have a similar distribution, developers seem to contribute more to fixing test bugs as both median and mean for the number of unique authors, watchers and comments for test bugs are higher than production bugs.

Finding 9: On average, developers contribute more actively to fixing test bugs compared to production bugs and test bugs are resolved faster than production bugs.

2.6 FindBugs Study

2.6.1 Detected Bugs

FindBugs reported 205 Correctness and Multithreaded Correctness warnings in the test code of 20 out of 129 ASF projects that we were able to compile and run
Table 2.6: Comparison of test and production bug reports.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Type</th>
<th>Med</th>
<th>Mean</th>
<th>SD</th>
<th>Max</th>
<th>d</th>
<th>OR</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>PR</td>
<td>3.00</td>
<td>2.91</td>
<td>0.76</td>
<td>5.00</td>
<td>-0.13</td>
<td>0.78</td>
<td>4.9e-14</td>
</tr>
<tr>
<td></td>
<td>TE</td>
<td>3.00</td>
<td>2.80</td>
<td>0.75</td>
<td>5.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resolution Time</td>
<td>PR</td>
<td>6.39</td>
<td>10.97</td>
<td>282.04</td>
<td>2843.56</td>
<td>-0.20</td>
<td>0.69</td>
<td>&lt;2.2e-16</td>
</tr>
<tr>
<td></td>
<td>TE</td>
<td>2.77</td>
<td>58.97</td>
<td>213.72</td>
<td>2666.60</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Comments</td>
<td>PR</td>
<td>3.00</td>
<td>4.91</td>
<td>6.74</td>
<td>101.00</td>
<td>0.15</td>
<td>1.31</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TE</td>
<td>4.00</td>
<td>5.88</td>
<td>6.26</td>
<td>99.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Authors</td>
<td>PR</td>
<td>2.00</td>
<td>2.41</td>
<td>1.53</td>
<td>18.00</td>
<td>0.31</td>
<td>1.77</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TE</td>
<td>2.00</td>
<td>2.89</td>
<td>1.53</td>
<td>12.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Watchers</td>
<td>PR</td>
<td>0.00</td>
<td>1.32</td>
<td>2.04</td>
<td>24.00</td>
<td>0.25</td>
<td>1.58</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TE</td>
<td>1.00</td>
<td>1.84</td>
<td>2.06</td>
<td>16.00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the tool on. Table 2.7 summarizes descriptive statistics for the number of reported warnings. For additional fine-grained data per project we refer the reader to [1].

Table 2.7: Descriptive statistics of bugs reported by FindBugs.

<table>
<thead>
<tr>
<th>Min</th>
<th>Mean</th>
<th>Median</th>
<th>σ</th>
<th>Max</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.6</td>
<td>0</td>
<td>5.5</td>
<td>48</td>
<td>205</td>
</tr>
</tbody>
</table>

2.6.2 Categories of Test Bugs Detected by FindBugs

Table 2.4 shows the top 10 most frequent potential test bugs detected by FindBugs and their percentage. Figure 2.10 shows distribution of different categories of these warnings. We consider Multithreaded Correctness warnings reported by FindBugs as flaky test bugs since these concurrency-related warnings can potentially cause the test to pass and fail intermittently. We also consider the Correctness warnings as semantic bugs, i.e., inconsistencies between actual and intended behaviour of the test. FindBugs has also one rule to detect bugs that cause different behaviours in Linux and Windows due to path separator differences, i.e., environment related bugs.

FindBugs has six test related rules that are part of the included correctness rules. However, FindBugs did not report any warnings related to these categories. As depicted in Figure 2.10, semantic related warnings and flaky ones are the major warnings reported by FindBugs.
2.6.3 FindBugs’ Effectiveness in Detecting Test Bugs

Earlier studies \[17, 46\] report that static code analysis tools are able to detect 5-15 % of general bugs in software projects. We wanted to know how they perform on test bugs. We sampled 50 bug reports out of 623 bug reports that changed Java source files and we were able to compile their project when we checked out the version just before the fix. Among these 50 sampled bug reports, in 3 (6%) instances at least one Findbugs’ warning disappeared after the fix. We analyzed these 3 instances manually and found out one of them was actually a false positive and the other two warnings were not directly related to the bug report. This means, FindBugs was not able to detect any of the 50 test bugs.

Finding 10: FindBugs could not detect any of the test bugs in our sampled 50 bug reports.

2.7 Discussion

Our study has implications for both testers and developers of bug detection tools. The results of this study imply that test code is susceptible to bugs just like production code (Finding 1). Test code is supposed to guard production code against potential (future) bugs and thus should be bug free itself. However, current bug detection tools are mainly geared towards production code. For instance, FindBugs has only six bug detection patterns dedicated for testing code among its 424 bug patterns [6]. Similarly, PMD, another popular static bug detection tool for Java, has only 12 bug pattern rules for bugs in JUnit [10]. Moreover patterns of environmental
bugs, flaky tests, and resource handling bugs in test code differ from production code, making current bug detection tools unable to detect them in test code. For example, the latest version of FindBugs detects run-time error handling mistakes based on the method proposed by [57]. We identified a similar pattern of this bug in test code (Figure 2.9). However, because of a slight change of pattern in the test code, FindBugs was not able to detect this.

In our study, FindBugs generated an average of 1.6 warnings in the test code of 129 open source projects, most of which fall into semantic and flaky test categories, the most prevalent categories of bug reports (Findings 2 and 4). However, many of the reported bugs cannot be simply detected using current static bug detection tools. This is particularly true for the silent horror bugs, which are mostly due to assertion related faults (Finding 3). Finding automated ways of detecting silent horror test bugs could be of great value to developers, since such bugs are extremely difficult to detect.

Our results show that a large portion of bugs in test code belongs to semantic bugs, i.e., test code does not properly test production code (Finding 4). Any method that can enhance developers’ understanding of the requirements, the software under test, and its valid usage scenarios, can help to reduce the number of semantic bugs in test code.

Compared to production bugs, we find that test bugs receive more attention from developers and are fixed sooner. This might be because the majority of the test bugs result in a test failure, which is difficult to ignore for developers. Another explanation could be that bugs in test code might be easier to fix than bugs in production code.

**Threats to Validity.** An internal validity threat is that the categorization of bug reports was made by two of the co-authors, which may introduce author-bias. To mitigate this, we conducted a review process in which each person reviewed the categorization done by the other person. Regarding the test bugs detected by FindBugs, we did not manually inspect each to see if is indeed a real bug. However, since we chose only the *Correctness* categories of FindBugs, we believe the reported bugs are issues in the test code that need to be fixed.

In terms of external threats, our results are based on bug reports from a number of experimental objects, which calls into question the representativeness. How-
ever, we believe that the chosen 448 ASF projects are representative of real-world applications as they vary in domains such as desktop applications, databases and distributed systems, and programming languages such as Java, C++ and Python. In addition, we focus exclusively on bug reports that were fixed. This decision was made since the root cause would be difficult to determine from open reports, which have no corresponding fix. Further, a fix indicates that there was indeed a bug in the test code.

2.8 Conclusions
This work presents the first large-scale study of test bugs. Test bugs may cause a test to fail while production code is correct (false alarms), or may cause a test to pass, while the production code is incorrect (silent horrors). Both are costly for developers. Our results show that test bugs are in fact prevalent in practice, the majority are false alarms, and semantic bugs and flaky tests are the dominant root causes of false alarms, followed by environment and resource handling related causes. Our evaluation reveals that FindBugs, a popular bug detection tool, is not effective in detecting test bugs.
Chapter 3

Reducing Fine-Grained Test Redundancies

Summary

Developers write tests to ensure the correct behaviour of production code. Test cases need to be modified as the software evolves. Over time, tests can accumulate redundancies, which in turn increase the test execution time and overhead of maintaining the test suite. Test reduction techniques identify and remove redundant test cases of a given test suite. However, these techniques remove whole test cases and do not address the issue of partly redundant test cases. In this chapter, we propose an approach for performing fine-grained analysis of test cases to support test case reorganization, while preserving the behaviour of the test suite. Our analysis is based on the inference of a test suite model that enables test reduction at the test statement level. We evaluate our technique on the test suites of four real-world open source projects. Our results show that our technique can reduce the number of partly redundant test cases up to 85% and the test execution time up to 2.5%, while preserving the test suite’s behaviour.
3.1 Approach

To enable a fine-grained analysis of test cases, we first present a model that captures the relationship between test states and test statements. We then describe how we automatically infer this model from a given test suite. Finally, we present a technique that uses this inferred model to remove fine-grained redundancies by reorganizing test cases.

3.1.1 The Test Suite Model

There are a number of properties that our model needs to exhibit. First, the model should capture how the test suite essentially tests the production code; this is important to preserve the behaviour of the test suite after any refactoring activity. Second, the model should capture dependencies at the test statement level to support test reorganization; since a test statement might have dependencies on previous statements, it is not possible to freely move test statements between test cases. Finally, the model should facilitate the discovery and removal of redundancies in test cases.

Figure 3.1 shows four test cases of the test class ComplexTest scraped from the Apache Math Commons [3], which test add, subtract, multiply and divide functionality for the Complex class. We use Figure 3.1 as a running example.

We refer to each statement in a test case as a test statement (st). In this case each test case is a sequence of test statements. For example, we refer to each line inside the test cases of Figure 3.1 as a test statement. Note that assertions are also a particular type of test statements.

A unit test case typically creates a set of variables (e.g., objects) and assigns values to their (member) variables, then it calls the production method under test using those variables as inputs, and finally it asserts the method’s returned value. Our test model needs to capture these three entities, namely, variables and their values, production method calls, and test assertions.

Definition 1 (Variable Value). The value of a variable $x$ ($Val(x)$) is defined as:
```java
@Test
public void testAdd() {
    Complex x = new Complex(3.0, 4.0);
    Complex y = new Complex(5.0, 6.0);
    Complex z = x.add(y);
    assertEquals(8.0, z.getReal(), 1.0e-5);
}

@Test
public void testSubtract() {
    Complex x = new Complex(3.0, 4.0);
    Complex y = new Complex(5.0, 6.0);
    Complex z = x.subtract(y);
    assertEquals(-2.0, z.getReal(), 1.0e-5);
}

@Test
public void testMultiply() {
    Complex x = new Complex(3.0, 4.0);
    Complex y = new Complex(5.0, 6.0);
    Complex z = x.multiply(y);
    assertEquals(-9.0, z.getReal(), 1.0e-5);
}

@Test
public void testDivide() {
    Complex dividend = new Complex(3.0, 4.0);
    Complex divisor = new Complex(5.0, 6.0);
    Complex q;
    q = dividend.divide(divisor);
    assertEquals(39.0 / 61.0, q.getReal(), 1.0e-5);
}
```

**Figure 3.1:** Test Cases from Apache Commons Project.

\[ Val(x) = \begin{cases} 
\text{primitive\_value} & : \text{Type}(x) \in P \\
\{(x, Val(x))| x \in Fields(x)\} & : \text{Type}(x) \notin P 
\end{cases} \]

\textit{Type}(x)\) denotes the type of the variable \(x\), \(P\) is the set of all primitive types, and \(Fields(x)\) denotes the set of all member variables of the object \(x\). If the variable \((x)\) is an object, its value is a set of \((x, Val(x))\) pairs where \(x_i\) is the name of \(i\)th member variable in \(x\); this includes Private, Protected and Public member variables of the object in Java. Otherwise, if the variable is of a primitive type, \(Val(x)\) is the variable’s primitive value. For example the value of the object \(x\) in Figure 3.1 at line 5 is \(Val(x) = \{(\text{Complex}, r, 3.0), (\text{Complex}, i, 4.0)\}\) given that the \texttt{Complex} class has two member variables of type \texttt{int} named \(r\) and \(i\).
In order to preserve a test suite’s behaviour, we look at the test suite as a black box and capture all the externally observable behaviours of the test suite. We refer to methods in the production code that are under test as production methods. The external behaviour of the test suite can be modelled by capturing the production methods that it calls along with their inputs.

**Definition 2 (Production Method Calls (PMC)).** The Production Method Calls of a test statement (PMC(st)) is the set of production methods that the statement calls with their inputs. The PMC of a test statement is a set of (MethodNamei, InputSeti) pairs, in which MethodNamei is the called production method’s qualified name, and InputSeti is the ordered set of (Type(xj), Val(xj)) pairs for each input variable xj of the method starting with this object for non-static member functions.

For example, the PMC for the test statement of line 5 in Figure 3.1 is \{(Complex.add, [(Type(x), Val(x)), (Type(y), Val(y))])\} since as part of the test statement’s execution, the method Complex.add is called with the two inputs x and y. Figure 3.2 illustrates the interaction between test code and production code for the execution of this test statement. Note that the PMC of a test statement that does not call any production methods is an empty set.

Further, our model needs to accommodate the ability of moving test statements from a source position in one test case to a destination position potentially in another test case. In order to preserve what the test statement does after the movement, we need to provide the same data and control dependencies the test statement had in the source position. If we know which variables are used as part of the execution of a test statement, we can determine if we can safely move it to another destination position in the test suite.
Definition 3 (Used Variables of Test Statements (UVS)). The Used Variables of a test statement (UVS(st_j)) is a set of (Type(x_i), Val(x_i)) pairs where each variable x_i is used in the execution of st_j.

For example, the used variables set of line 5 in Figure 3.1 contains \{(Type(x), Value(x)), (Type(y), Value(y))\}.

For assertions, we also keep track of the method calls that create the value of the variables. Since assertions check the output of particular production method calls, we need to capture this information as part of the test model. For example in line 6 of Figure 3.1 the assertion checks the output of Complex.add with specific inputs. It is possible to retrieve the whole chain of method calls that the assertion checks in this way. For example, the assertion checks the output of Complex.add and in turn checks the output of two constructor calls of Complex.Complex with their inputs.

Definition 4 (Used Variables of Assertions (UVA)). The Used Variables of an assertion (UVA(as)) is a set of (Type(x_i), Val(x_i), Meth(x_i)) tuples where as part of the assertion’s execution the variable x_i is used and Meth(x_i) is the PMC of the test statement that assigns the value of x_i.

For instance, the UVA set of line 6 is \{(Type(z), Value(z), Meth(z))\}; in this case Meth(z) = \{(Complex.add, [(Type(x), Val(x)), (Type(y), Val(y))])\} since variable z is being used as part of the execution of the assertion and its value is created by the Complex.add production method call (PMC of the test statement at line 5).

In addition to data dependencies, a test statement can have definition dependencies on its previous test statements. For example, in line 27 of Figure 3.1 the test statement does not depend on the value of q. We can replace the variable q with any other variable of type Complex given that the data dependency of the test statement is satisfied. In this case, the test statement needs three variables of the type Complex defined in the previous test statements, in addition to its data dependencies, in order to be executed. The Defined Variable Set (DV) of a test statement captures this definition dependency.
**Definition 5 (Defined Variables (DV)).** The Defined Variables of a test statement is a bag of the variable types that are referenced in the test statement and need to be defined in the previous test statements.

Consider Figure 3.1 again. The defined variable set of line 27 is \{Complex, Complex, Complex\} since the variables dividend, divisor and q need to have been defined for the test statement.

To perform data and definition dependency analysis, we maintain a test state.

**Definition 6 (Test State).** A Test State encompasses information regarding the defined variables, their values, and the PMC that created those values at a specific test statement in the test case. Formally, the Test State \((S_j)\) is a set of \((\text{Type}(x_i), \text{Val}(x_i), \text{Meth}(x_i))\) tuples for each variable \(x_i\) referable from \(j\)th test statement in the test case.

In the Java programming language and JUnit testing framework, the test state includes information about local variables, static field of loaded classes, and member variables of the test class. For example in Figure 3.1, the test state before the execution of line 5 is \{\((\text{Type}(x), \text{Value}(x), \text{Meth}(x)), (\text{Type}(y), \text{Value}(y), \text{Meth}(y))\)\}; since the two variables \(x\) and \(y\) are referable at line 5, \(\text{Meth}(x) = \{(\text{Complex}, \text{Complex}, \{(\text{double}, 3.0), (\text{double}, 4.0)\})\}\) since \(x\) is created by the production method call of the constructor Complex.Complex with two input values of 3.0 and 4.0 of the type double.

We do not consider variables’ identity (such as its memory address) or name as part of the test state since most tests do not depend on object’s identity or variable names in the test code. However, field names in objects are part of the test state, since those variables are defined in the production code.

It is possible to move a test statement only if the test statement is compatible with the test state at the destination position.

**Definition 7 (Compatible States).** A test state is compatible with a test statement if it satisfies the test statement’s data and definition dependencies. In this case, the test statement can be executed on the test state while preserving its behaviour. Formally, a test state \((S_i)\) is compatible with a test statement \((st_j)\) iff its used variables \((UVS(st_j))\) and defined variables \((DV(st_j))\) are subsets of the test state \((UVS(st_j) \subset S_i)\)
\( S_i \land (DV(s_j) \subset Def(S_i)) \). \( Def(S_i) \) denotes the set of defined variables in the test state \( S_i \).

Note that the compatibility relation for an assertion is defined similarly.

Now that we have all the required information, we can define a test suite model that enables our fine-grained test analysis at the test state and test statement levels.

**Definition 8 (Test Suite Model).** A Test Suite Model is a directed graph denoted by \(<r,V,E>\) where \( V \) is a set of vertices representing test states, \( E \) is a set of directed edges representing test statements and assertions, and \( r \) denotes the root of the graph, which is the initial empty state.

Figure 3.3 depicts the test suite model of the test suite of Figure 3.1. Ovals illustrate test states and rectangles illustrate labels of test statement edges. Dotted lines represent the state compatibility relations between test states and test statements. Note that we have illustrated only a subset of compatibility relations to avoid cluttering the graph. For example, the statement of line 12 is compatible with the states \( S_2, S_3, S_4, S_5, S_6 \) and \( S_7 \) since the read variable set at line 12 \( \{(\text{Complex}, \{(\text{Complex}.r,3.0),(\text{Complex}.i,4.0)\}), (\text{Complex}, \{(\text{Complex}.r,5.0), (\text{Complex}.i,6.0)\})\} \) is a subset of these test states. With the notion of compatible states, we can determine possible valid reorganizations of test statements in test cases. For example we can relocate the test statement of line 12 in Figure 3.3 to the location after \( S_3, S_4, S_5, S_6 \) or \( S_7 \).
Figure 3.3: Extracted partial model for the running example. Ovals illustrate test states and rectangles illustrate labels of test statement edges. Dotted lines represent the state compatibility relations between test states and test statements.
To detect redundancies in the test suite, we look at the external behaviour of each test statement to detect those that have identical external behaviour. These equivalent test statements are identical as far as testing the production code is concerned.

**Definition 9 (Equivalent Test Statements).** Equivalent Test Statements are the set of test statements that have the same production method calls (PMC) set.

To preserve the coverage of the production code, we need to call at least one of the test statements in each set of equivalent test statements. For example in Figure 3.1, the set of equivalent test statements is \{\{st_3, st_{10}, st_{17}, st_{24}\}, \{st_4, st_{11}, st_{18}, st_{25}\}, \{st_5\}, \{st_6\}, \{st_{12}\}, \{st_{13}\}, \{st_{19}\}, \{st_{20}\}, \{st_{27}\}, \{st_{28}\}\}. In this case, we only need to include one of the test statements from the set of \{st_3, st_{10}, st_{17}, st_{24}\} to maintain the coverage.

### 3.1.2 Inferring The Model

We now describe how we create the model given a test suite. Figure 3.4 shows the overview of our approach.
Figure 3.4: Overview of our Approach.
**Equivalent Test Statements.** To capture the test state at each test statement level, we store the type and value of all referable variables through code instrumentation. This includes all local variables, member variables of the test class, and static fields of loaded classes. To capture production method calls (PMCs), we instrument the production code to log the entry point, input values (including the `this` object for non-static methods), and exit points of each method in the production code. This way, we can trace the call stack for each method. Methods with a call stack void of any other production methods are those that are called directly by the test code. This enable us to capture the production methods that are directly called (with their input values) for each test statement. We execute the instrumented test cases against the instrumented production code, and use the traces to compute sets of equivalent test statements that have the same PMC.

**Used and Defined Variables.** For each method invocation, we assume all the input variables and all of their properties are *used* as part of the test statement execution. This also includes all referable variables, such as static variables and member variables of an object that are part of a method invocation on the object. For example, in line 6 of Figure 3.1 we assume that all of the properties of the variable `z` (i.e. `z.r` and `z.i`) will be used as part of the test statement’s execution. Although in this case `z.i` is not actually used as part of the test statement’s execution. This is a conservative assumption in terms of detecting compatible states for test statements. In this case, we might not be able to detect some compatibility relations but all the relations that we detect are correct. To compute the *defined* variables set, we check for the type of the variables that are referenced in the test statement.

**Compatible States.** To compute compatible states for a test statement `st`, we check the states in which the variables used in `st` have the same value. We also check whether the test states satisfy the test statement’s definition dependency. For assertions, additionally, we check for the PMC that defined the most recent value for the used variables. Although it is possible to make sure that the whole chain of method calls that an assertion checks remains the same, in this work, we only require that the direct method calls that an assertion checks remain the same. The reason behind this decision is that constraining a deeper level of method calls can restrict our options for reorganizing test cases.
Reducing Redundancy in the Test Suite. We use the inferred model to identify and remove redundant test statements. For example, by reorganizing the four test cases of Figure 3.1, we can create a reduced test case shown in Figure 3.5, which has the same coverage with six fewer statements.

To maintain the test suite coverage, we basically need to call each production method once. Each test case in our test model is a path starting from the initial state. For example in Figure 3.3, the test `testAdd` is the path `(st3, st4, st5, st6)` in which `st_i` is `i`th test statement edge in Figure 3.3. Thus, to maintain the test suite coverage, we need to find a set of paths, starting from the initial state, that visits at least one test statement from each set of equivalent test statements. To find such paths, we propose a greedy algorithm.

Identifying Clusters of Redundant Test Cases. The test suite model for the whole test suite can become large (e.g., the graph for Apache Commons has 22K nodes). Since we are interested in reorganizing test cases that share some common equivalent test statements, we can construct a test suite model for each set of test cases that share at least `k` equivalent test statements, in which `k` is a cutoff value. To that end, we create a new cluster graph in which nodes are test cases and there is an edge between two nodes if there is at least `k` common test statements between the two test cases. Then, we create a test suite graph for each connected component of this cluster graph and perform the reorganization on each of these test suite graphs, independently. For example in Figure 3.6, nodes are test cases and edges are weighted by the number of common equivalent test statements two nodes contain.
With a cutoff value of $k = 10$, we get three clusters, namely for \{T1, T2, T4\}, \{T3, T6\}, and \{T7\}.

![Diagram showing clusters T1 to T7 with connections](image)

**Figure 3.6:** Clustering test cases for reorganizing.

**Reorganization Algorithm.** The intuition behind our algorithm, shown in algorithm 1, is to extend a path to cover as many unique test statements and assertions as possible. In this step, we reorganize and merge test cases in each cluster. We maintain a set of equivalent test statements and assertions that we need to cover (uncoveredEqStmts). Each test statement operates on a compatible test state and transforms it to another test state. If $S_i$ and $S_{i+1}$ are the test states before and after the execution of the test statement $st_i$, the function $\text{apply}(st_i, S_i) = S_{i+1}$ applies the effect of executing the test statement $st_i$ on the test state $S_i$ and returns the changed test state. We have the test state before and after the execution of each test statement. We assume that the test statement could potentially change all of its used variables. So we can compute the effect of running the test statement on each of its compatible states. Essentially, we need to update the value of the used variables of the compatible test state to the values of variables in $S_{i+1}$. If $S_j$ is a compatible state of the test statement $st_i$, then $\text{apply}(st_i, S_j) = \text{updateValues}(USV(st_i), S_{i+1}, S_j)$. We maintain a running state for the reorganized test case so that at each point we know what would be the test state at the end of reorganized test case up to this point (runningState). We update this running state at each iteration of the algorithm.
AlGORITHM 1: Reorganization

(Line 9). With the information from the test states, we update the graph to include possibly new compatibility edges (Line 10). We start from the initial state and find the nearest test statement node that we still need to cover. We find the shortest path from the initial state to that node and do the same from that node until we have covered at least one statement from each set of the equivalent test statements and all the assertions (Line 7). To find the shortest path, we use a variant of the best-fit search algorithm that also maintains the running state. We maintain the running state for each path that is being examined during the algorithm. For example for the test suite model of Figure 3.3, our algorithm returns the path \((s_{13}, s_{12}, s_{19}, s_{20}, s_{27}, s_{28})\). This path is highlighted (thick-line) in Figure 3.3.

Composing Reduced Test Cases. Algorithm 1 gives us a set of paths that minimizes the number of test statements executed while maintaining the test suite’s coverage. Since each test statement in the path can be originating from a different test case, the variables with the same value can have different names. For example in Figure 3.1 variables \(x\) and \(y\) have names \(\text{dividend}\) and \(\text{divisor}\) in \text{testDivide}. Therefore, to generate the reduced test cases, we might need to rename the variables. Test statements can also define variables that are defined with the same name previously; for example in the reorganized test case the variable \(z\) is
defined in each test case, so we need to rename variable definitions as well. Also, test statements can use member variables and member functions of the source test class, so we need to include those in the destination test class as well. Because of polymorphism in object oriented programs, we also need to cast a variable to its sub or super class if the static type of the variables with the same value is different in source and destination state. Algorithm 2 shows the pseudocode for the algorithm responsible for composing reorganized test case. We use a bidirectional map of variable values to variable names and their type to maintain the state. As we go through the test statements in the reorganized test case path, for each test statement, we check if we have the value for each variable in the test statement; if there is a value in the state but with different name, we rename the variable in the test statement to the name of the variable in the state (Lines 6-11). If the type of the variable is different, we cast the variable to the destination type. We also check the variable definitions for name duplicates and rename the duplicates (Lines 12-16). Finally, we need to update the bidirectional state map with the changed values from test statement execution (Line 17).

**Preserving Test Suite Behaviour.** Assume that we reorganize a set of test cases $x$ into the reorganized set of test cases $y$, we show that $y$ preserves the fault revealing behaviour of $x$. $PMC(x)$ denotes the set of production method calls that the set of test cases $x$ call with their inputs. Since $PMC(x) = PMC(y)$, each production method $m_i$ that is called as part of the execution of $x$, will be called with the same inputs in $y$. Hence, we preserve the coverage and thus the soft oracles of $x$. We also preserve the hard oracles of $x$, because our approach includes all the corresponding test assertions in the reorganized test cases. Assume that in $x$, assertion $as_i$ checks the return value of the production method $m_i$ with the input $in_i$. Let $as_j$ be the same assertion $as_i$ that is included in $y$. Since $UVA(as_i) = UVA(as_j)$, assertion $as_j$ will check the return value of $m_i$ with the same input $in_i$. If a fault $f$ in $m_i$ affects the return value of $m_i(in_i)$ and is detectable by $as_i$, it is also detectable by $as_j$.

**Implementation.** We implemented our approach in a tool called TESTMODLER, which is publicly available [11]. The tool is written in Java, although TESTMODLER only supports Java programs with JUnit tests, our approach is applicable to other programming languages and testing frameworks. It gets as input the path to a
input: ordered list of statements in the composed test case
output: renamed compilable list of statements for the composed test case

\[
\text{renameStatements(\text{statementsPath}) begin}
\]

\[
\text{stateBiMap} \leftarrow Value, Set < Name, Type > \leftarrow \emptyset
\]

\[
\text{foreach stmt \in statementsPath do}
\]

\[
\text{nameValueMapPreq} \leftarrow \text{getValueValuePreqVarsInStatement}(\text{stmt})
\]

\[
\text{renameMap} \leftarrow OldName, NewName \leftarrow \emptyset
\]

\[
\text{castMap} \leftarrow \text{varName, oldType, newType} \leftarrow \emptyset
\]

\[
\text{foreach (varName, varValue) \in nameValueMapPreq do}
\]

\[
\text{varNamesInState} \leftarrow \text{stateBiMap}[\text{varValue}]
\]

\[
\text{else if varName \notin varNamesInState then}
\]

\[
\text{renameMap}[\text{varName}] \leftarrow \text{pickName}(\text{varNamesInState})
\]

\[
\text{castMap} \leftarrow \text{checkForTypes(stateBiMap, stmt)}
\]

\[
\text{leftHandSideVars} \leftarrow Name \leftarrow \text{getVarsInLeftHandSide(stmt)}
\]

\[
\text{foreach varName \in nameValueMapDef do}
\]

\[
\text{varNameInState} \leftarrow \text{stateBiMap}[\text{varName}]
\]

\[
\text{else if varNameInState! = null then}
\]

\[
\text{renameMap}[\text{varName}] \leftarrow \text{generateNewName(varName)}
\]

\[
\text{end}
\]

\[
\text{stmt} \leftarrow \text{renameStatement(stmt, renameMap, castMap)}
\]

\[
\text{updateStateMap(stateBiMap, stmt)}
\]

\[
\text{end}
\]

\[
\text{end}
\]

**Algorithm 2: Test Case Composition**

Java project, instruments the test and production code, and runs the instrumented test code against instrumented production code to obtain traces. TESTMODLER reorganizes the partly redundant test cases i.e. test cases that have at least one redundant test statement, and generates a new reduced test suite.

### 3.2 Evaluation

To assess the real-world relevance and efficacy of our approach, we address the following research questions:

**RQ1:** How prevalent are partly-redundant tests in practice?

**RQ2:** Can TESTMODLER reduce redundancy and test execution time while preserving the test suite coverage?

**RQ3:** What is the performance of running TESTMODLER?
Table 3.1: Subject systems and their characteristics.

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Production Code (KLOC)</th>
<th>Test Code (KLOC)</th>
<th># Tests (Static)</th>
<th># Tests (Dynamic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>45.2</td>
<td>59.1</td>
<td>3,990</td>
<td>6,174</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>12.3</td>
<td>20.3</td>
<td>1,264</td>
<td>16,069</td>
</tr>
<tr>
<td>AssertJ</td>
<td>6.4</td>
<td>24.7</td>
<td>4,620</td>
<td>5,802</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>16.6</td>
<td>27.0</td>
<td>1,865</td>
<td>1,956</td>
</tr>
<tr>
<td>Total</td>
<td>80.5</td>
<td>131.1</td>
<td>11,739</td>
<td>30,001</td>
</tr>
</tbody>
</table>

3.2.1 Subject Systems

We selected four open-source Java programs that have JUnit test cases. Table 3.1 shows our subject systems and their characteristics. Apache Commons Math is a light-weight mathematics and statistics library [3]. Apache Commons Collections is a library for more powerful data structures in Java programming language [2]. AssertJ is a library that provides richer typed, easy to use assertions [4]. CheckStyle is a static analysis tool that helps developers enforce a coding standard [5]. Note that the number of actual written static test cases in a test suite might differ from the number of test cases that are dynamically executed. For example, testers can create a common abstract test class with test cases and inherit this abstract test class to test the production code under different input data and scenarios. Another example is JUnit 4 Parameterized test classes, through which testers can run multiple tests with different input data using the same test class/methods.

3.2.2 Procedure and Results

Prevalence (RQ1). To answer RQ1, we measure the number of test cases that have at least one common equivalent test statement with another test case. To do this, we use our test suite model to identify classes of equivalent test statements in the test suite (see Section 3.1.2). Table 3.2 shows the number of partly-redundant tests, number of clusters of test cases that have at least one common test statement (3.1.2), total number of common test statements in the whole test suite, total number of unique classes of equivalent test statements, and the number of redundant
Table 3.2: Partial redundancy in the test suites.

<table>
<thead>
<tr>
<th>Subject System</th>
<th># redundant tests</th>
<th># clusters</th>
<th># common test statements</th>
<th># unique test statements</th>
<th># redundant test statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>1,354</td>
<td>235</td>
<td>4,052</td>
<td>1,305</td>
<td>2,747</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>472</td>
<td>122</td>
<td>1,856</td>
<td>629</td>
<td>1,227</td>
</tr>
<tr>
<td>AssertJ</td>
<td>947</td>
<td>131</td>
<td>516</td>
<td>178</td>
<td>338</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>164</td>
<td>55</td>
<td>258</td>
<td>96</td>
<td>162</td>
</tr>
<tr>
<td>Total</td>
<td>2,937</td>
<td>543</td>
<td>6,682</td>
<td>2,208</td>
<td>4,474</td>
</tr>
</tbody>
</table>

test statements in the test suite. Figure 3.7c shows the distribution of number of redundant test statements in each test case for those test cases that have at least one redundant test statement. Figure 3.7b shows the distribution of number of test cases in each redundancy clusters, and Figure 3.7a shows the distribution of number of test statements in each equivalent test statements set.

**Finding 11:** Our results show that 2,937 (25%) out of the total number of 11,739 test cases in the test suite of our subject systems are partly redundant. This means that it is possible to reduce the execution time of 25% of test cases by removing redundancies and repeated production method calls.

Table 3.3: Test reduction.

<table>
<thead>
<tr>
<th>Subject System</th>
<th># tests reorganized</th>
<th># tests reorganized into</th>
<th># tests reduced</th>
<th># statements reduced</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>1,149</td>
<td>203</td>
<td>946</td>
<td>1,623</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>186</td>
<td>62</td>
<td>124</td>
<td>250</td>
</tr>
<tr>
<td>AssertJ</td>
<td>392</td>
<td>139</td>
<td>253</td>
<td>207</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>124</td>
<td>47</td>
<td>77</td>
<td>122</td>
</tr>
<tr>
<td>Total</td>
<td>1,851</td>
<td>451</td>
<td>1400</td>
<td>2,202</td>
</tr>
</tbody>
</table>
Effectiveness (RQ2). To assess the efficacy of our approach in reducing fine-grained redundancies in test cases, we ran TESTMODLER on the subject systems. TESTMODLER reorganizes redundant test cases to avoid repeated production method calls, which reduces the number of redundant test statements. We measured
the number of partly-redundant test cases and test statements before and after running the tool on the test suite. Table 3.3 presents the number of redundant test cases that were reorganized, the number of test cases that TESTMODLER reorganized these test cases into, the number of test cases reduced, and the number of redundant test statements reduced. Note that in some tests we need to reorganize test cases that are in the same test cluster into more than one test case. For example, for the subject system AssertJ, we reorganized 947 test cases that belonged to 131 clusters into 139 test cases. In this case, some test cases change the value of a member variable of the test class, while other test cases depend on the old value of the variable; although TESTMODLER tries to minimize the number of reorganized test cases, in this case, it is not possible to reorganize all test cases in the cluster into one test case. In another case, some of test classes use a custom runner and we only reorganize together the test cases that have the same test runner (see Section 3.3.3).

Figure 3.8a and Figure 3.8b illustrate the comparison between the actual number of redundant test cases and statements that exist in the test suite of our subject systems with the number of test cases and statements that our approach was able to reduce.

Finding 12: In total, TESTMODLER reorganized 1,851 out of 2,937 (63%) partly-redundant test cases, which removed 2,202 out of 4,474 (49%) redundant test statements.

To assess the effects of reducing redundancy on test suite execution time, we measure the execution time of individual test cases. Non-deterministic test cases can have variable execution times. In some cases, such test cases retry several times until they pass, which can affect the measured test suite run time.

To mitigate this variability and compare the execution time of the tests before and after reorganization, we measure the execution time of individual test cases before and after reorganization. We sum up the execution time of the test cases that TESTMODLER reorganized, before and after reorganizing the test suite. This way, we obtain more stable test execution results. We perform the measurements 10 times and report the averages. Table 3.4 shows the execution time of the whole test suite, the test cases that TESTMODLER reorganized, and the execution time.
reduction for these test cases. In total, TESTMODLER was able to reduce test execution time of the four test suites by 2.4% (Average=2.1, Median = 2.3, SD=0.61, Variance=0.36). We measured the statement coverage and branch coverage, using EclEmma 2.3.3, for the test suite before and after reorganization to assess whether TESTMODLER preserves the statement and branch coverage. As shown in Table 3.5 TESTMODLER preserves the statement and branch coverage for all the subject systems. Note that since the subject systems Apache Commons Math and Collections have many undeterministic tests, the coverage varies from run to run, and the coverage of reorganized test suite differ slightly from the original test suite for these subject systems.

**Finding 13:** In total, TESTMODLER reduced the execution time of our four subject systems by 2.4%, while maintaining the coverage.

**Table 3.4:** TESTMODLER’s effectiveness in terms of execution time.

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Execution Time (Total)</th>
<th>Execution Time (reorganized)</th>
<th>Execution Time Reduction</th>
<th>% reduced (Total)</th>
<th>% reduced (reorganized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>83.1</td>
<td>17.1</td>
<td>2.1</td>
<td>2.5</td>
<td>12.3</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>16.1</td>
<td>4.8</td>
<td>0.37</td>
<td>2.3</td>
<td>7.7</td>
</tr>
<tr>
<td>AssertJ</td>
<td>3.4</td>
<td>0.23</td>
<td>0.04</td>
<td>1.2</td>
<td>17.4</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>26.3</td>
<td>3.1</td>
<td>0.6</td>
<td>2.4</td>
<td>20.1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>128.9</strong></td>
<td><strong>25.23</strong></td>
<td><strong>3.13</strong></td>
<td><strong>2.4</strong></td>
<td><strong>12.4</strong></td>
</tr>
</tbody>
</table>

**Performance (RQ3).** To assess the performance of TESTMODLER, we measured the execution time for different steps of running the tool. Table 3.6 shows the execution time for instrumentation step, running the instrumented test suite against the instrumented production code, and reorganizing and recomposing algorithms. On average, it takes TESTMODLER 1,027 milliseconds to reorganize and recompose each test case. Taking into account all three steps for running the tool, on average it

\[1\] All measurements are performed on a Mac OS X machine, running on a 2.7GHz Intel Core i5 with 8 GB of memory. Values reported are in seconds.
Figure 3.8: Comparison of optimal and actual reductions for the number of test cases and test statements.

takes 1,933 milliseconds for each redundant test case to be identified, reorganized and recomposed.

Finding 14: On average, TESTMODLER takes 1,026 milliseconds to reorganize each partly redundant test case.
Table 3.5: Test suite’s coverage before and after reorganization.

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Statement Coverage Before (%)</th>
<th>Branch Coverage Before (%)</th>
<th>Statement Coverage After (%)</th>
<th>Branch Coverage After (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>92.73</td>
<td>85.72</td>
<td>92.71</td>
<td>85.71</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>84.46</td>
<td>77.27</td>
<td>84.43</td>
<td>77.23</td>
</tr>
<tr>
<td>AssertJ</td>
<td>95.56</td>
<td>92.19</td>
<td>95.56</td>
<td>92.19</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>95.44</td>
<td>96.82</td>
<td>95.44</td>
<td>96.82</td>
</tr>
</tbody>
</table>

Table 3.6: TESTMODLER’s performance.

<table>
<thead>
<tr>
<th>Subject System</th>
<th>Instrumentation (m:s)</th>
<th>Running instrumented code (m:s)</th>
<th>Reorganization (m:s)</th>
<th>Reorganization per test case (ms)</th>
<th>Total time per test case (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Math</td>
<td>1:41</td>
<td>6:41</td>
<td>18:23</td>
<td>960</td>
<td>1,398</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>0:40</td>
<td>0:58</td>
<td>3:18</td>
<td>1,062</td>
<td>1,588</td>
</tr>
<tr>
<td>AssertJ</td>
<td>4:04</td>
<td>0:37</td>
<td>6:23</td>
<td>976</td>
<td>1,693</td>
</tr>
<tr>
<td>CheckStyle</td>
<td>2:52</td>
<td>10:25</td>
<td>3:36</td>
<td>1,749</td>
<td>8,174</td>
</tr>
<tr>
<td>Total/Average</td>
<td>9:17</td>
<td>18:41</td>
<td>31:40</td>
<td>1,027</td>
<td>1,933</td>
</tr>
</tbody>
</table>

3.3 Discussion

3.3.1 Applications

Our technique can be run offline to reduce the redundancy in the test suite. For projects with large (and slow) test suites, the reduction in test number and execution time can be especially beneficial.

Our tool keeps both the original and reduced versions of the test suite and links them together. This enables tracing back any failed reorganized test case to the test case in the original test suite. Since the reorganization might reduce the readability of the test suite, the reorganized test suite can be used for running test cases and regularly kept up to date with the original test suite. An important feature is that the
reorganized test suite does not need to be updated if there are no changes made to
the reorganized tests, as part of a commit. However, if parts of a reorganized test
case change, the changed test case needs to be analyzed by TESTMODLER again.
As discussed in RQ3, this whole process takes less than two seconds to complete
for each test case.

Our test suite model can also be used for other test analysis activities. For in-
stance, it can be used to analyze tests for detecting potential bugs [52] or smells [53].
One of the tasks performed during test refactoring is to reorganize test cases to
remove eager and lazy test smells [53]; our model in this case can help with the
refactoring task, since it is not straight forward to manually reorganize test cases
in a way that preserves the behaviour of the test suite. Our model can be used to
identify test cases that have common test statements and are small, and merge them
to remove lazy test smell. It can also be used to reorganize large test cases into
smaller ones, to remove the eager test smell while keeping the incurred redundancy
at a minimum.

3.3.2 Relation to test suite reduction techniques
In our approach, we consider two test statements equivalent if they call exactly the
same production methods with exactly the same inputs. However, it is possible to
use different criteria to identify these equivalent test statements. For example, a
substitute might be to compute production method’s coverage for each test statement
and consider test statements with exactly the same coverage as equivalent test
statements. By using the coverage information, our approach can act like a fine-
grained test reduction technique. Existing test suite reduction techniques remove a
test case that has the same coverage as other tests. Our approach on the other hand
only removes redundant parts of test cases.

3.3.3 Limitations
We investigated why our tool cannot reduce all redundant test cases. First, we
cannot reorganize test cases that terminate abruptly. For example, test cases with
JUnit’s @Test (expected=SomeException.class) annotations throw an
exception and terminate abruptly. Also, we have seen test cases that try to achieve
similar behaviour with the use of return and fail statements. Although the use of inheritance in test code is debatable [37], all of our subject systems use inheritance in their test code heavily. We chose not to reorganize test cases in test classes that are subclassed by another test class, since in this case the subclass might override some of the test cases and render the reorganized test cases useless. TESTMODLER cannot reorganize test case inside a parameterized test class with test cases of other classes, since in this case, the test case will be run with different inputs and can only be merged with another test case that has the same inputs. Some test classes also use custom test runners to run their test cases. For example, in one case, test cases would be retried several times with a custom runner until they pass. In this case, we can only reorganize test classes that have the same custom runner. We also chose not to reorganize test statements inside conditionals such as for-loops, try-catch and ifs. Further, since we do not store a variable’s identity as part of our test state (Definition 6), we do not support reorganizing test cases with \texttt{assertSame} assertions. Since we depend on dynamic values of variables in test cases, we do not support reorganizing nondeterministic test cases.

We used test statement as the smallest unit of computation for the test model, using a smaller unit, such as bytecode operation, increases the model size and the algorithm complexity. On the other hand, using a larger unit, such as blocks of statements, decreases our granularity in reorganizing test cases and detecting partly-redundant test cases with common test statements.

3.3.4 Threats to validity

Similar to any other experiment using a limited number of subject systems, an external validity threat to our results is the generalizability of our results. We tried to mitigate this threat by choosing subject systems with various sizes, domains, and tests, although we need more subject systems to fully address the generalization threat. With respect to reproducibility of our results, the source code of our tool and all subject systems is available online [11], making the experiments reproducible.
3.4 Conclusions

In this chapter, we proposed a test suite model that facilitates test code analysis at the test statement level. We used the proposed model to present an automated technique and tool, called TESTMODLER, for reducing fine-grained redundancies in test cases, while preserving the behaviour of the test suite. We empirically evaluated our technique on four subject systems and overall, TESTMODLER was able to reduce the number of partly-redundant test cases up to 85% and test execution time 2.5%, while preserving the original test suite coverage and production method call behaviour.
Chapter 4

Related Work

Empirical Bugs and Smells studies. Test smells were first studied by van Deursen et al. [53] and later other works defined types of test smells, such as test fixture [29], eager test, and mystery guest, and proposed methods to detect these test smells [13, 35, 54, 55]. Test smells are, however, not bugs. In this study, we focus on bugs that change the intended behaviour of the test.

Zhang et al. [65] found that the test independence assumption does not always hold in practice. They observed that the majority of dependent tests result in false alarm and some of these dependencies result in missed alarms. In this case a test which should reveal a fault passes accidentally because of the environment generated by another dependent test case. Test dependency (I1) is one of the 16 cause subcategories for test bugs emerged in our empirical study.

Lu et al. [41] studied real world concurrency bugs, and found that most of concurrency bugs belong to order or atomicity violations. Luo et al. [42] categorized and investigated the root cause of failures in test cases manifested by non-determinism, known as flaky tests. Flaky tests are one of the main cause categories emerged in our categorization study. Our results are inline with the findings of Lou et al in terms of the root causes of such test bugs.

Li et al. [40] mined the software bug repositories to categorize types of bugs found in production code. Their work is similar to ours in case of categorization but we looked and categorized types of bugs in test code instead of production code.
Herzig and Nagappan [34] proposed an approach to identify false alarms. They use association rule learning to automatically identify these false alarms based on patterns learned from failing test steps in test cases that lead to a false alarm. The authors aim at identifying test alarms to prevent development process disruption, since a test failure halts the integration process on the code branch that test failure occurred. Our work, however, aims at providing insights into patterns of faults in test code to help detect them by static analysis tools.

**Test quality.** Athanasiou et al. [12] proposed a model to assess test quality based on source code metrics. They showed that there is a high correlation between the test quality as assessed by their model and issue handling performance. Zaidman et al. [63] investigated how production code and test code co-evolve. They introduced three test co-evolution views, namely change history view, growth history view, and test quality evolution view. It would be interesting to see how test bugs would fit into these views, for instance, are test bugs introduced when they are first added or when they are modified later as test code co-evolves with production code?

**Test Refactoring.** Fang et al. in [23] used assertion fingerprints to detect similar test cases that can be refactored into one single test case. They performed static analysis on test code and by analyzing CFG they computed branch count, merge count, exceptional successor count for each assertion. Based on these attributes they detect refactoring candidate test cases. Unlike their approach, our approach finds refactoring candidates based on common redundant statements that they have. Guerra et al. in [30] visually represent test cases with a graphical notation before and after test refactoring to help developers verify that the behaviour of the test case has been kept unchanged. Our approach on the other hand makes sure that reorganizing test statements in the test suite preserve its behaviour.

Xuan et al. in [60] split test cases into smaller fragments to enhance dynamic analyses. Xuan and Monperrus in [59] perform test purification to improve fault localization. They decompose failing test cases with several assertions to multiple test cases in which each test cases has only one assertion. They perform test purification in three steps namely, test case atomization, test case slicing and refinement. First, they generate k copies for a test case which has k test case and disable all assertions but one in each test case. Then they perform dynamic slicing and slice the test case
from the broken assertion statement in the test and its variables. By this approach they eliminate the unnecessary statement from the purified test cases. Finally, they rank the suspicious statements with the purified test cases which improves fault localization accuracy of fault localization techniques.

Davaki et al. \cite{19} merge web applications GUI test cases to reduce test execution time. They use a combination of browser’s DOM state and database state to define the state of the program. They do not merge test statement and assertions of test cases instead they focus on merging test steps. Each test case in their approach is comprised of several test steps and each test step is an event exercised on GUI elements. Unlike our approach that can reorganize and interleave all valid refactorings of a unit test case, their approach can only interleave chunks of test steps that result in the same browser’s DOM state. Fraser and Wotawa in \cite{24} merge test cases generated by a model checker, they compare state of the application for different tests and merge only those test cases that one of the test cases is a prefix of the other. Our approach on the other hand can reorganize all valid refactorings of test cases, and as opposed to their approach that operates on models, our approach operates on real code and unit tests.

Our test state representation is closely related to heap representation of \cite{31}. They store the portions of concrete heap accessible from static fields of test classes in order to find test cases that pollute the program state. While they only consider static fields of test classes for their state representation, in order to support test reorganizing, in addition to static fields, we need to include local variables and also member variables of the test class. Our state also include information about the static type of the variables and the production method calls that create a specific value in the test state, since for assertions we need to make sure that after reorganizing the assertion checks the output of the same function with the same inputs.

**Test Suite Reduction.** There is a large body of work on test suite reduction and test selection techniques \cite{62}. Different techniques \cite{15,43,50,61} are proposed for removing redundant test cases. These techniques use different coverage criteria such as statement coverage or branch coverage to detect redundant test cases. Although it is possible to use coverage criteria with our approach, we chose to preserve the test suite’s behaviour and find redundant parts of test cases that call the same production...
method calls with exactly the same input. Smith et al. in [49] construct call trees of
the program and identify a subset of test cases that cover the call tree paths.

**Test Suite Selection.** Many techniques [20] are proposed for regression test selec-
tion. These techniques use different levels of granularity for tracking dependencies,
such as file dependency [25] and class dependency [47], to detect affected test cases
as part of a change to production code. Gligoric et al. in [25] proposed a light
weight test selection technique based on dynamic dependencies of tests of files and
integrated their technique into JUnit testing framework. They showed that since
they use a very lightweight test selection technique the end-to-end testing time is
lower than the prior techniques.

More recently, there are techniques [39, 48, 64] that combine test reduction,
test selection and test prioritization techniques. Korel et al. in [39] combine test
reduction with test selection techniques. They identify the differences between
original EFSM model and the modified model as a set of elementary model modifi-
cations and use EFSM model dependence analysis to reduce regression test suite
and remove the test cases that are redundant respect with the modifications that
are made to the model. Shi et al. [48] combine test reduction and test selection
technique to further reduce the number of tests executed for each commit.

**Test Comprehension and Visualization.** Greiler et al. in [27] interviewed 25
eclipse developers and incorporated the finding in creating five architectural views
namely, plug-in modularization view, extension initialization view, extension and
service usage view, and test suite modularization view to help developers in test
suite understanding for plug-in architectures.

Greiler et al. in [28] compute similarities in test execution traces to detect
similar high level end-to-end tests and fine grained unit tests. With this approach
they were able to restore traceability links between unit tests and requirements.

Kamimura et al. in [38] generate human-oriented summaries for test cases.
Their approach is based on static analysis of test cases’ source code. They identify
unique method invocations for each test case and find the verification statements
related to these method invocations. Based on these information they generate
human readable sentences describing the test case.
Cornelissen et al. in [16] try to get understanding about software under test by analyzing unit tests. They perform dynamic analysis on test cases and construct a sequence diagram based on traces of test executions.
Chapter 5

Conclusion and Future Work

This thesis aims at improving test code quality by (1) characterizing bugs in test code (2) reducing redundancies in test code. In the first part of the thesis, we present the first large scale empirical study of bugs in test code to characterize their prevalence, impact and root cause categories. We mine bug repositories and version control systems of 448 Apache Software Foundation (ASF) projects, which are from a broad spectrum of domains, with various sizes and programming languages. We find 5,556 test bugs that were reported and fixed in the test code of these projects. The focus our study was to get insight into different categories of test bugs and their root causes. We (1) qualitatively study a total of 443 randomly sampled test bug reports in detail and categorized them based on their impact, root cause and fault location; (2) used our manually sampled data and applied machine learning techniques to automatically categorize rest of test bug reports based on their root cause; (3) compare properties of test bugs with production bugs, such as active time and fixing effort needed; (4) investigated if FindBugs[6], a popular static bug detection tool for Java, is effective in detecting test bugs.

The results of our study show that (1) around half of all the projects analyzed had bugs in their test code that were reported and fixed; (2) the majority of test bugs are false alarms i.e., test fails while the production code is correct, while a minority of these bugs result in silent horrors i.e., test passes while the production code is incorrect; (3) incorrect and missing assertions are the dominant root cause of silent horror bugs; (4) semantic (25%), flaky (21%), environment-related (18%) bugs are
the dominant root cause categories of false alarms; (5) the majority of false alarm bugs happen in the exercise portion of the tests; (6) developers contribute more actively to fixing test bugs and test bugs are fixed sooner compared to production bugs, and (7) FindBugs is not effective in detecting test bugs.

Our study has implications for both developers and researchers. The characterization of root causes of test bugs has practical implications for developers and testers on how to avoid these bugs. The results of our study indicate that test code is susceptible to bugs just like production code. However, current bug detection tools are mainly geared toward production code. Our results imply that there is a need to devise new bug detection tools for detecting bugs in test code. We believe that the results of our study will be useful to researchers and developers in building new bug detection tools for detecting bugs in test code.

In the second part of the thesis, we focus on improving test code quality by reducing redundancies in test code. While the current test reduction techniques operate at the test case level to detect and remove redundant test cases, we propose an approach for performing fine-grained analysis of test cases to perform test reduction at the test statement level. Our analysis is based on the inference of a test suite model that enables test case reorganization, while preserving the behaviour of the test suite. We use our test suite model to reorganize test statements in the test cases in a way that removes the redundant test statements and reduces the redundancy. We implemented our approach in a tool called TESTMODLER and evaluated it on four open source projects. Our empirical results show that (1) 25% of all test cases in the test suite of our subject systems are partly redundant; (2) Our technique can reduce the number partly redundant test cases by 63% and the number redundant test statements by 49%; (3) Our technique can reduce the execution time of the test suites by 2.4% while preserving the test suite’s behaviour.

5.1 Future Work

For future work, we plan to build on top of our empirical study and analyze correlations between test bugs and various software metrics. We plan to use the results of the empirical to design a bug detection tool with test bug patterns capable of detecting bugs in test code. We also plan to extend our test model described in
chapter 3 and use coverage instead of production method calls (Definition 9) to
detect redundancies. In this case we might not actually preserve test suite behaviour,
but we can preserve the test suite’s coverage while reducing the test suite even
more. This way, we can perform fine-grained test suite reduction while preserving
the coverage of the test suite. We also plan to investigate the use of test selection
techniques on the reduced test suite. Since we only reorganize and merge test cases
that have common test statements, it is likely that most of the reorganized test
cases can be chosen together as part of a test selection technique. For example,
all test selection techniques that operate on the Class level likely choose all of the
reorganized test cases together. Therefore, it might be possible to use test selection
techniques on the reorganized test suite to further reduce the test suite.
Bibliography


[37] P. Kainulainen. Three reasons why we should not use inheritance in our tests. [https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/](https://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/). → pages [58]


