Singapore Management University

Institutional Knowledge at Singapore Management University

Dissertations and Theses Collection Dissertations and Theses

7-2017

Testing and debugging: A reality check

Pavneet Singh KOCHHAR
Singapore Management University, kochharps.2012@phdis.smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/etd_coll_all

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Citation

KOCHHAR, Pavneet Singh. Testing and debugging: A reality check. (2017).
Available at: https://ink.library.smu.edu.sg/etd_coll_all/17

This PhD Dissertation is brought to you for free and open access by the Dissertations and Theses at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Dissertations and Theses Collection by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

TESTING AND DEBUGGING: A REALITY CHECK

PAVNEET SINGH KOCHHAR

SINGAPORE MANAGEMENT UNIVERSITY

2017

Testing and Debugging: A Reality Check

by

Pavneet Singh Kochhar

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

David LO (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Lingxiao JIANG (Co-supervisor)
Assistant Professor of Information Systems
Singapore Management University

Youngsoo KIM
Assistant Professor of Information Systems
Singapore Management University

Nachiappan NAGAPPAN
Principal Researcher
Microsoft Research

Singapore Management University
2017

Copyright (2017) Pavneet Singh Kochhar

Testing and Debugging: A Reality Check

Pavneet Singh Kochhar

Abstract

Testing and debugging are important activities during software development and maintenance. Testing is performed to check if the code contains errors whereas debugging is done to locate and fix these errors. Testing can be manual or automated and can be of different types such as unit, integration, system, stress etc. Debugging can also be manual or automated. These two activities have drawn attention of researchers in the recent years. Past studies have proposed many testing techniques such as automated test generation, test minimization, test case selection etc. Studies related to debugging have proposed new techniques to find bugs using various fault localization schemes such as spectrum-based fault localization, IR-based fault localization, program slicing, delta debugging etc. to accurately and efficiently find bugs. However, even after years of research software continues to have bugs, which can have significant implications for the organization and economy.

Often developers mention that the number of bugs they receive for the project overwhelms the resources they have. This brings forth the question of analyzing the current state of testing and debugging to understand its advantages and shortcomings. Also, many debugging techniques proposed in the past may ignore bias in data which can lead to wrong results. Furthermore, it is equally important to understand the expectations of practitioners who are currently using or will use these techniques. These analyses will help researchers understand pain points and expectations of practitioners which will help them design better techniques. In this thesis, I take a step in this direction by conducting large-scale data analysis and by interviewing and surveying large number of practitioners. By analysing the quantitative and qualitative data, I plan to bring forward the gap between practitioners' expectations and the research ouput. My thesis sheds light on current state-of-practice

in testing in open-source projects, the tools currently used by developers and challenges faced by them during testing. For bug localization, I find that files that are already localized can have an impact on the results and this bias must be removed before running a bug localization algorithm. Furthermore, practitioners have a high expectation when it comes to adopting a new bug localization tool. I also propose a technique to help developers find elements to test. Furthermore, through interviews and surveys, I provide suggestions for developers to create good test cases based on several characteristics such as size and complexity, coverage, maintainability, bug detection etc. In the future, I plan to perform a longitudinal study to understand the causal impact of testing on software quality. Furthermore, I plan to perform an empirical validation of good test cases based on the suggestions received from the practitioners.

Table of Contents

1 Introduction **1**

2 Literature Review **4**

2.1 Software Testing 4

2.2 Debugging 6

3 Adoption of Software Testing **9**

3.1 Introduction 9

3.2 Methodology & Statistics 10

3.2.1 Collecting the dataset 10

3.2.2 Research questions 12

3.2.3 Statistical measurements 14

3.3 Findings 15

3.3.1 RQ1: Popularity of Test Cases 15

3.3.2 RQ2: Developers and Test Cases 18

3.3.3 RQ3: Test Cases and Bug Counts 20

3.3.4 RQ4: Test Cases and Bug Reporters 22

3.3.5 RQ5: Test Cases and Programming Languages 23

3.4 Conclusion 25

4 Adequacy of Software Testing **27**

4.1 Introduction 27

4.2 Methodology & Statistics 28

4.3 Findings 30
4.3.1 RQ1: Coverage levels and Test Success Densities 30
4.3.2 RQ2: Correlations at the Project Level 32
4.3.3 RQ3: Correlations at the File Level 35
4.4 Conclusion 37

5 Understanding the Testing Culture of App Developers 39
5.1 Introduction 40
5.2 Methodology & Statistics 40
5.3 Findings 45
5.3.1 RQ1: Current State of Testing in Android Applications . . . 45
5.3.2 RQ2: Survey of Android Developers 47
5.3.3 RQ3: Survey of Microsoft Developers 53
5.4 Threats to Validity 57
5.5 Conclusion 58

6 Bug Localization: Researchers' Bias 59
6.1 Introduction 59
6.2 Biases in Bug Localization 60
6.2.1 Bias 1: Wrongly Classified Reports 61
6.2.2 Bias 2: Already Localized Reports 64
6.2.3 Bias 3: Incorrect Ground Truth Files 69
6.3 Other Evaluation Metrics 72
6.4 Conclusion 74

7 Bug Localization: Practitioners' Expectations 76
7.1 Introduction 76
7.2 Methodology 79
7.2.1 Practitioner Survey 79
7.2.2 Literature Review 83

7.3 Findings 84
7.3.1 Statistics of Responses Received 84
7.3.2 Answers to Research Questions 86
7.3.3 Respondents' Final Comments 95
7.4 Current State of Research 96
7.5 Discussion 99
7.5.1 Implications 99
7.5.2 Limitations 101

8 Learning to Test: Helping Developers Make Testing Decisions 104
8.1 Introduction 104
8.2 TestAdvisor and TestAdvisor^{CP} 105
8.2.1 Overview 106
8.2.2 Feature Extraction 107
8.2.3 TestAdvisor^{CP} 112
8.3 Experimental Setup 115
8.4 Findings 120
8.5 Conclusion 126

9 What Make Good Test Cases? 128
9.1 Introduction 128
9.2 Methodology 129
9.2.1 Open Ended Interviews 129
9.2.2 Validation Survey 132
9.3 Findings 134
9.3.1 Contents 134
9.3.2 Size and Complexity 138
9.3.3 Coverage 141
9.3.4 Maintainability 145
9.3.5 Bug Detection 148

9.3.6 Others 152
9.4 Implications 153

10 Conclusion and Future Work **157**

10.1 Summary 157
10.2 Future Direction 160

List of Figures

3.1 Distribution of Projects in Terms of Total Lines of Code 11

3.2 Test Cases and Lines of Code 16

3.3 Correlation between Test Cases and Lines of Code 17

3.4 Correlation between Test Cases per LOC and Lines of Code 18

3.5 Number of Developers in Projects with/without Test Cases 19

3.6 Test Cases and Number of Developers 20

3.7 Correlation between # of Test Cases per Developer and # of Developers 20

3.8 Correlation between # of Test Cases and # of Bugs 21

3.9 Test Cases and Bug Reporters 22

3.10 Correlation between # of Bug Reporters and # of Test Cases 23

3.11 Count of Projects and Different Languages 24

3.12 Prevalence of Test Cases for Common Languages 24

4.1 Distribution of Projects. 30

4.2 Test Success Density 31

4.3 Scatter Plots (Project Level) 33

4.4 Scatter Plots (File Level) 36

5.1 Distribution of Apps in Terms of Total Number of Lines of Code . . 44

5.2 Distribution of Apps in Terms of Number of Developers 44

5.3 Distribution of Apps in Terms of Total Number of Test Suites 45

5.4 Line Coverage (Ascending Order) 46

5.5 Block Coverage (Ascending Order) 46
5.6 Types of Testing 54
5.7 Usage of Automated Testing Tools 55
5.8 Challenges Faced by Developers 55

6.1 Example Diff of a File that is Changed to Fix a Bug in Lucene-Java Project with ID LUCENE-2616. Note: (1) The name of the file: SegmentInfo.java; (2) An empty line and an import statement are deleted; (3) An empty line is deleted and another one is added. . . . 71
6.2 Before and After Removing Bias 1 72
6.3 Before and After Removing Bias 2 73
6.4 Before and After Removing Bias 3 73

7.1 Countries Our Survey Respondents Reside 85
7.2 Importance of Fault Localization Research to Respondents of Various Demographic Groups 87
7.3 Availability of Debugging Data to Practitioners (Math-Spec = Mathematical specification, Text-Spec = Textual specification, One-Test= One test case, Multi-Tests = Multiple test cases, Suc-Tests = Successful test cases, Text-Desc = Textual description) 88
7.4 Percentages of Respondents Specifying Various Preferred Granularity Levels 89
7.5 Percentage of Respondents Specifying Various Minimum Success Criteria 90
7.6 Minimum Success Rate vs. Satisfaction Rate 90
7.7 Minimum Program Size vs. Satisfaction Rate 91
7.8 Maximum Runtime vs. Satisfaction Rate 91
7.9 Other Factors Affecting Adoption 93

8.1 Overview of *TestAdvisor* 105

8.2 Comparison between Enhanced and Standard Cross-project Model Learning. 113

List of Tables

3.1 Test Cases Distribution 15

3.2 Prevalence of Test Cases 16

3.3 Tags representing Bugs 21

3.4 Distribution of Test Cases per Project 25

4.1 Project Distribution across Coverage Levels 31

5.1 Distribution of Apps in Terms of Presence of Test Cases 43

5.2 Automated Testing Tools Usage 48

5.3 Challenges Faced by Developers while Testing 50

5.4 Automated Testing Tools Usage 54

6.1 Mean Average Precision (MAP) Scores for Reported and Actual . . 63

6.2 Mean Average Precision (MAP) Scores when Issue Reports of a Particular Misclassification Type are Omitted. Omit. = Omitted, Misclass. = Misclassification, HC = HTTPClient, JB = Jackrabbit, LJ = Lucene-Java. The last column is the MAP of all three projects. 64

6.3 Fully Localized, Partially Localized, and Not Localized Reports . . 64

6.4 Fully Localized Report: HTTPCLIENT-1078 65

6.5 Partially Localized Report: JCR-814 65

6.6 Not Localized Report: LUCENE-3721 66

6.7 Fully, Partially, and Not Localized Reports 67

6.8 MAP Scores: Fully vs. Partially vs. Not 68

6.9 Comparison: Fully vs. Partially vs. Not 68

6.10 Fisher Exact Test: Best vs. Worst Reports 69
6.11 Bug Report: LUCENE-2616 71
6.12 MAP Scores: Dirty vs. Clean Ground Truths 71
6.13 Results of Mann-Whitney-Wilcoxon Test and Cohen'd Computation for MRR. (F-P) = Fully Localized vs. Partially Localized. (P-N) = Partially Localized vs. Not Localized. (F-N) = Fully Localized vs. Not Localized. 74

7.1 Capabilities of Current State-of-Research 98

8.1 Features used for the learning model. Features appended by (f) and (m) are only calculated at the file level and method level, respectively, while others are computed at both the levels. 108
8.2 Study Subjects. 116
8.3 *TestAdvisor* versus baselines (File Level). B1 considers bug history between the *current* and the *previous* version to rank files. B2 takes the full history of the project. B3 produces a random list. 121
8.4 *TestAdvisor* versus baselines (Method Level). B1 considers bug history between the *current* and the *previous* version to rank files. B2 takes the full history of the project. B3 produces a random list. . . . 121
8.5 *TestAdvisor* versus defect prediction (File Level). DP1 is state-of-the-art defect prediction using deep learning. DP2 uses the term frequencies of AST nodes. 122
8.6 *TestAdvisor* versus defect prediction (Method Level). DP1 is state-of-the-art defect prediction using deep learning. DP2 uses the term frequencies of AST nodes. 122
8.7 Hit@5, Hit@10, MAP and MRR scores when a particular feature is used and the combination of all features (File Level). 123
8.8 Hit@5, Hit@10, MAP and MRR scores when a particular feature is used and the combination of all features (Method Level). 123

8.9 Hit@5, Hit@10, MAP and MRR scores for different classification algorithms (File Level). 124
8.10 Hit@5, Hit@10, MAP and MRR scores for different classification algorithms (Method Level). 124
8.11 Hit@5, Hit@10, MAP and MRR scores of *TestAdvisor* and *TestAdvisor*^{CP} considering cross-project setting (File Level). 125
8.12 Hit@5, Hit@10, MAP and MRR scores of *TestAdvisor* and *TestAdvisor*^{CP} considering cross-project setting (Method Level). 125

9.1 List of Hypotheses 131

Acknowledgments

I owe my subservience to The One, Primal, Omnipresent and Omniscient Lord, who manifested through various remarkable people, with whom I had an extraordinary time at SMU, filled with science, learning and fun. To these, I am forever grateful.

To my supervisor, Prof. David Lo, for accepting me as his graduate student. I am grateful to you for exposing me to a new world, teaching me to be innovative and efficient, and being supportive always. You have impressed me with your ability to communicate optimism which has helped me grow both personally and professionally. I have learnt a lot from you and that has paved the tarmac for my career ahead.

To my co-supervisor, Prof. Lingxiao Jiang, for being there to advise me whenever I needed and to persuade me to always achieve more.

To the School of Information Systems at SMU, for providing me scholarship to pursue studies at SMU and Carnegie Mellon University (CMU).

To Prof. Claire Le Goues, for being my advisor during exchange programme at CMU, and to other faculty members who advised me and gave their insights at some point or the other.

To Nachiappan Nagappan, Thomas Zimmermann and Christian Bird, for providing me an opportunity to pursue summer internship at Microsoft Research.

To SMU administration staff, Seow Pei Huan and Ong Chew Hong, for their continuous support and helping me keep track of the deadlines.

To my lab mates from SOftware Analytics Research (SOAR) group, with whom I spent numerous fun-filled hours at the lab.

To my wonderful parents, who sacrificed all their comforts and provided me with everything I wanted, so that I could pursue my goals. To my lovely brother and sister-in-law, Jaspreet and Aparna, for guiding me on innumerable aspects of my life. To my other relatives, who helped whenever I had a need. I have got the most extraordinary family; you are the true abode of My Lord.

To my grandfather, Prithipal Singh,

Without you I would be nothing today

List of Publications

Conference/Workshop Papers

Pavneet Singh Kochhar and David Lo. Revisiting Assert Use in GitHub Projects. In *21st International Conference on Evaluation and Assessment in Software Engineering Conference (EASE)*, Karlskrona, Sweden, June 2017.

Yuan Tian, **Pavneet Singh Kochhar**, and David Lo. An Exploratory Study of Functionality and Learning Resources of Web APIs on Programmable Web. In *21st International Conference on Evaluation and Assessment in Software Engineering Conference (EASE)*, Karlskrona, Sweden, June 2017.

Abhishek Sharma, Ferdian Thung, **Pavneet Singh Kochhar**, Agus Sulistya and David Lo. Cataloging GitHub Repositories. In *21st International Conference on Evaluation and Assessment in Software Engineering Conference (EASE)*, Karlskrona, Sweden, June 2017.

Yun Zhang, David Lo, **Pavneet Singh Kochhar**, Xin Xia, Quanlai Li , and Jianling Sun. Detecting Similar Repositories on GitHub. In *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Klagenfurt, Austria, February 2017.

Pavneet Singh Kochhar. Mining Testing Questions on Stack Overflow. In *The Fifth International Workshop on Software Mining (SoftwareMining)*, Singapore, Singapore, September 2016.

Pavneet Singh Kochhar, Xin Xia, David Lo, Shanping Li. Practitioners Expectations on Automated Fault Localization. In *The 25th International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrcken, Germany, July 2016.

Pavneet Singh Kochhar, Dinusha Wijedasa, David Lo. A Large Scale Study of Multiple Programming Languages and Code Quality. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

Xin Xia, David Lo, **Pavneet Singh Kochhar**, Zhenchang Xing, Xinyu Wang, Shanping Li. An Industrial Experience Report on Test Outsourcing Practices. In *The 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Gaithersburg, MD, USA, November 2015.

Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, David Lo. Understanding the Test Automation Culture of App Developers. In *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Graz, Austria, April 2015.

Pavneet Singh Kochhar, Ferdian Thung, David Lo. Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, Canada, March 2015.

Pavneet Singh Kochhar, Ferdian Thung, David Lo, Julia Lawall. An Empirical Study on the Adequacy of Testing in Open Source Projects. In *The 21st Asia-Pacific Software Engineering Conference (APSEC)*, Jeju, Korea, December 2014.

Pavneet Singh Kochhar, Yuan Tian, David Lo. Potential Biases in Bug Localization: Do They Matter?. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Vasteras, Sweden, September 2014.

Pavneet Singh Kochhar, Ferdian Thung, David Lo. Automatic Fine-Grained Issue Report Reclassification. In *19th International Conference on Engineering of Complex Computer Systems (ICECCS)*, Tianjin, China, August 2014.

Pavneet Singh Kochhar, Tien-Duy Le and David Lo. It's Not A Bug, It's A Feature: Does Misclassification Affect Bug Localization?. In *The 11th Working Conference on Mining Software Repositories (MSR)*, Hyderabad, India, May 2014.

Xiaoqiong Zhao, Xin Xia, **Pavneet Singh Kochhar**, David Lo, Shanping Li. An Empirical Study of Bugs in Build Process. In *The 29th Symposium on Applied Computing (SAC)*, Gyeongju, Korea, March 2014.

Yuan Tian, **Pavneet Singh Kochhar**, Ee-Peng Lim, Feida Zhu, David Lo. Predicting Best Answerers for New Questions: An Approach Leveraging Topic Modeling and Collaborative Voting. In *1st Workshop of Quality, Motivation and Coordination of Open Collaboration (QMC)*, Kyoto, Japan, November 2013.

Pavneet Singh Kochhar, Tegawend F. Bissyand, David Lo, Lingxiao Jiang. An Empirical Study of Adoption of Software Testing in Open Source Projects. In *The 13th International Conference on Quality Software (QSIC)*, Nanjing, China, July 2013.

Pavneet Singh Kochhar, Tegawend F. Bissyand, David Lo, Lingxiao Jiang. Adoption of Software Testing in Open Source Projects A Preliminary Study on 50,000 Projects. In *17th European International Conference on Software Maintenance and Reengineering (CSMR)*, Genova, Italy, March 2013.

Journal Papers

Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code Coverage and Post-Release Defects: A Large Scale Study on Open Source

Projects. In *IEEE Transactions on Reliability*, 2017.

Xin Xia, Lingfeng Bao, David Lo, **Pavneet Singh Kochhar**, Ahmed E. Hassan, and Zhenchang Xing. What do developers search for on the web? In *Empirical Software Engineering (EMSE)*, 2017.

Jing Jiang, David Lo, Jiahuan He, Xin Xia, **Pavneet Singh Kochhar**, Li Zhang. Why and how developers fork what from whom in GitHub. In *Empirical Software Engineering (EMSE)*, February 2017, Volume 22, Issue 1.

Tool Demos

Ferdian Thung, Tien-Duy B. Le, **Pavneet Singh Kochhar**, David Lo. BugLocalizer: Integrated Tool Support for Bug Localization. In *The 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, November 2014.

Ferdian Thung, **Pavneet Singh Kochhar**, David Lo. DupFinder: Integrated Tool Support for Duplicate Bug Report Detection. In *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Vasteras, Sweden, September 2014.

Xinyu Wang, David Lo, Xin Xia, Xingen Wang, **Pavneet Singh Kochhar**, Yuan Tian, Xiaohu Yang, Shanping Li, Jianling Sun and Bo Zhou. BOAT: An Experimental Platform for Researchers to Comparatively and Reproducibly Evaluate Bug Localization Techniques. In *36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014.

Chapter 1

Introduction

Testing and debugging are important activities during software development cycle. Testing is performed to check if the code contains errors whereas debugging is done to locate and fix these errors. Testing can be manual or automated and can be of different types such as unit, integration, system, stress etc. Debugging can also be manual or automated. These two activities are equally important and have drawn attention of researchers in the recent years.

Despite the availability of various tools to ensure quality of software through testing, it is not confirmed if large number of projects are adequately tested or not. This is important as impact of inadequate testing can consist of a substantial number of unhandled failures, which leads to poor quality of software, higher software development costs and delays in time to market the product. A study conducted by the National Institute of Standards and Technology reported that inadequate software testing costs the U.S economy $59.5 billions annually, i.e., about 0.6% of its GDP [120]. The number of bugs uncovered after the code has been shipped can overwhelm projects developers when software is not thoroughly tested. For example, a triager from Mozilla project admitted that they receive almost 300 bugs everyday that need triaging [7]. Therefore, it is important to have techniques which can help developers find buggy files quickly, which can help them resolve the bug faster. These figures reinforce the fact that software testing and debugging is paramount for

developing high quality software.

Past studies have proposed many new techniques and performed empirical research on software testing and debugging. Testing related studies in the past have proposed many techniques such as automated test generation, test minimization, test case selection, test priortization and empirical studies on test suite quality and its effectiveness in finding bugs. Studies related to debugging have proposed new techniques to find bugs using various fault localization schemes such as spectrum-based, program slicing, delta debugging etc. to accurately and efficiently find bugs. However, even after years of research software continues to have bugs, which can have significant implications. Thus, it is important to consider the current state of techniques and practitioners' expectations to understand if the practitioners find the techniques useful or not. Such quantitative and qualitative analysis can provide insights useful for researchers and practitioners. In this thesis, I intend to answer the following questions:

1. What is the current state of practice and practitioners' expectations in testing?

    1a) What is the adoption of testing in open-source projects?

    1b) What is the adequacy of testing in open-source projects?

    1c) What is the testing culture of app developers in open-source projects?

2. What are researchers' biases and practitioners' expectations in debugging?

    2a) What are potential biases in bug localization?

    2b) What do practitioners expect from bug localization tools?

Through these analyses, I plan to bring forward the gap between current state of practice and expectations of practitioners. This will help practitioners be aware of what tools and techniques researchers are building and help researchers to be aware of the needs of practitioners to build relevant tools.

My thesis sheds light on current state-of-practice in testing in open-source projects, the tools currently used by developers and challenges faced by them during

testing. For bug localization, I find that files that are already localized can have an impact on the results and this bias must be removed before running a bug localization algorithm. Furthermore, practitioners have a high expectation when it comes to adopting a new bug localization tool. I also propose a technique to help developers find elements to test. Furthermore, through interviews and surveys, I provide suggestions for developers to create good test cases based on several characteristics such as size and complexity, coverage, maintainability, bug detection etc.

Chapter 2

Literature Review

In this chapter, I will present some of the past work done by researchers in the domain of software testing and debugging.

2.1 Software Testing

Software testing is an integral part of software development lifecycle. Past studies have proposed many new techniques test generation, test minimization, test case selection, test priortization as well as empirical studies on testing.

Greiler et al. conduct a qualitative study of test practices followed by a community of people working on plug-in based applications [42]. Rehman et al. discuss several software component testing issues and classify set of testing techniques used when a component is integrated with its target system [124]. Memon et al. present their analysis to improve the current testing techniques and strategies to create new collaborative development and testing processes where developers can share tools and information repositories [84]. Cabral et al. present an analysis of testability issues and testing techniques for software product lines (SPLs) [17].

Zaidman et al. study the co-evolution between production code and test code on two open source and one industrial project [143]. Fraser et al. use search-based software testing for test data generation for open source projects [31]. They perform case study on 100 Java projects selected from SourceForge and give directions for

future research. Ceccato et al. perform an empirical study to analyse the impact of automatically generated test cases on accuracy and efficiency of debugging [20]. They compare the effectiveness of debugging between a manually designed test suite and a test suite generated by Randoop. Their results show that automatically generated test cases positively affect debugging. Stamelos et al. conduct an empirical study on open source projects to understand the implications of structural quality and the probable benefits of such analysis on software development [114].

Gopinath et al. investigate the correlation between test suite coverage and its effectiveness in killing mutants [40]. They start with more than 1,000 GitHub projects but need to remove most of them due to compilation errors, etc. They end up with around 200 GitHub projects for their analysis. Most of the projects analysed are small (less than 1000 lines of code). They find that there is a correlation between test suite coverage and effectiveness. Our work complements this work by addressing a different set of research questions. We also study a larger set of projects and most of them are of larger size (more than 10,000 lines of code). Inozemtseva and Holmes also investigate the correlation between test suite coverage and its effectiveness in killing mutants on 5 large Java programs [47]. They find that there is a weak to moderate correlation between test suite coverage and its effectiveness. Our work complements this work by addressing a different set of research questions. We also study a large set of projects instead of only 5 projects. Many of the projects that we analyse are as big as the projects that are analysed by Inozemtseva and Holmes (more than 100,000 lines of code).

Several studies have proposed new techniques and methods to increase code coverage. Thummalapenta et al. develop an approach that takes as input a user-specified intent and produces programs in the form of method sequences to produce the object state specified by user [121]. Their approach uses data from static as well as dynamic analysis and is able to produce higher coverage than existing approaches. Pandita et al. propose an approach to produce test inputs to achieve logical coverage and boundary-value coverage using existing test-generation ap-

proaches [92]. Their approach is able to increase the coverage and improves the fault detection capability of new test cases. Park et al. propose a new approach that combines random testing with techniques such as static program analysis and concolic execution [93]. They tested their approach on twelve Java applications and their results show that their approach performs better than some previous approaches such as pure random, adaptive random, and Directed Automated Random Testing (DART).

There have been many empirical studies on Android. Takala et al. reported experiences on applying model based user interface testing on Android applications [118]. Kropp and Morales investigated strengths and weaknesses of two approaches for testing mobile GUI applications: the Android instrumentation framework and Positron framework [67]. Bhattacharya et al. performed an analysis on bug reports and bug fixing process of Android applications [15]. McDonnell et al. studied the stability and adoption of APIs in Android ecosystem [82]. Syer et al. studies 15 most popular Android applications and compare them with 3 desktop applications [117]. Ruiz et al. investigated the practice of reuse in Android ecosystem [108]. Maji et al. characterize failures in Android and Symbian mobile OSes [68].

2.2 Debugging

There are many IR-based bug localization approaches that retrieve source code files that are relevant to an input bug report [6, 89, 103, 104, 110, 112, 127, 146]. Rao and Kak conducted a good comparative study on the performance of a number of general IR models on bug localization task [103]. They found that simple text models such as Vector Space Model (VSM) and Smoothed Unigram Model (SUM) perform better that more sophisticated models like Latent Dirichlet Allocation (LDA).

Zhou et al. propose an extended vector space model named rSVM to locate bug by leveraging information from similar bug reports [146]. Ali et al. proposed a

framework called LIBCOOS to combine textual information and binary-class relationships (e.g., association, aggregation, composition, etc.) for bug localization [6]. Saha et al. made use of structure information retrieved from code structure (e.g, whether a word is used as a class name or a variable name), and bug report structure (e.g., whether a word is appeared in the title or description filed of a bug report) to improve the effectiveness of IR-based bug localization [110]. Sisman et al. enhanced existing bug localization techniques by adding more textual information into bug reports to form better queries [112]. Recently, Wang et al. proposed an integrated approach by considering multiple resources (i.e, version history, similar bug reports, and structure information) [127].

Practitioners' Perception, Expectation, and Activities: Lo et al. surveyed hundreds of practitioners in Microsoft on how they perceive the relevance of 517 papers published in ICSE and FSE in 2009-2014 [74]. They asked each respondent to rate 40 randomly selected papers by answering a question: "In your opinion, how important are the following pieces of research?". In this work, we focus on *adoption* rather than relevance, and fault localization rather than all software engineering studies. Since this study is focused rather than general, we can consider more in-depth questions on thresholds for adoption, and get more respondents to comment on one topic of interest.

Perscheid et al. studied debugging practice of professional software developers [97]. Different from them, we investigate what practitioners want for a future tool, rather than the current state-of-practice. In particular, our study estimates practitioners' thresholds for adopting fault localization tools.

Empirical Study on Fault Localization: Ruthruff et al. investigated the effectiveness of a fault localization technique applied on spreadsheets [109]. Jones and Harrold performed an empirical study to evaluate Tarantula against four other fault localization techniques on programs from Siemens test suite [53]. Kochhar et al. presented a number of threats that researchers need to consider (e.g., misclassification, incorrect ground truths, etc.) when designing experiments to evaluate information-

retrieval-based techniques [65].

Parnin and Orso [94] investigated the usability of a spectrum-based fault localization technique named Tarantula [53]. They performed a user study using a defect in a Tetris application and another defect in NanoXML. They observed how participants debug with and without Tarantula. The user study highlights that (1) absolute rank should be used as the evaluation metric, (2) the combination of search and ranking should be considered, (3) a complete ecosystem for debugging is needed, (4) more studies on how "richer information" can be used to help debugging is needed.

Wang et al. [125] investigated the usability of an information-retrieval based bug localization technique named BugLocator [146]. They analyzed what information in a bug report tends to produce good results, how their user study participants used information in bug reports, and whether the participants behaved differently when they use BugLocator than without it. In their user study using 8 bugs from SWT, they find that BugLocator is only useful if bug reports come "without rich, identifiable information" and bad bug localization outputs "harm developers' performance".

Chapter 3

Adoption of Software Testing

3.1 Introduction

Software testing is an important part of software development life-cycle. Despite the availability of various tools to ensure quality of software through testing, most software products suffer from insufficient testing. The consequences of inadequate testing include a substantial number of unhandled failures, which leads to poor quality of software, higher software development costs and delays in time to market the product. A study conducted by the National Institute of Standards and Technology reported that inadequate software testing costs the U.S economy $59.5 billions annually, i.e., about 0.6% of its GDP [120].

Although a large body of research about software testing has been built, software programs continue to suffer from numerous defects. Consequently, is software testing really popular in development projects? Does it noticeably impact the quality of software code? What kind of projects are more likely to include tests? These are some of the important questions which can increase our understanding of the unexplored areas of software testing and its impact on software evolution. Our goal in this paper is indeed to fill a research gap in the importance of software testing through a large-scale empirical evaluation.

In this work, we analyse a large number of open source projects from the GitHub

hosting site. GitHub platform holds millions of software projects including important projects such as Linux and Ruby on Rails. GitHub provides various features which makes it an important platform for storing open source projects. GitHub also provides an in-house issue tracking system where users record issues and classify them as bugs, feature requests, and other self-defined categories. We investigate in this study different characteristics of software development that are related to testing: e.g., numbers of developers in projects that include test cases. We also study how the presence/absence of test cases can affect the quality of software in terms of the number of reported bugs. Finally, we investigate the programming languages in relation to the projects with test cases.

3.2 Methodology & Statistics

For our empirical study, we analyse projects downloaded using the GitHub API. GitHub does not follow a distinct ordering scheme to download the projects. Thus, the results vary every time with a new request. To ensure that most of the projects are non toy projects in our dataset, we filtered the data and selected the projects which have more than 500 lines of code (LOC). We have in total 20,817 projects of sizes 500 to 17 millions LOC. These include well-known projects such as Ruby on Rails and jQuery.

3.2.1 Collecting the dataset

a) Lines of code: GitHub uses the git software configuration management system (SCM) to store software revisions. We cloned the git repositories of the projects and used the SLOCCount[1] utility to count the lines of code of the latest revisions of these projects. Figure 3.1 shows the lines of code of different projects. We observe that 40% of our projects have LOC between 1,000 and 5,000. Around 27% of the projects lie between 500 to 1,000 LOC , while more than 23% of the projects have

[1]http://dwheeler.com/sloccount

more than 10,000 lines of code. Also, over 15,000 projects (total 20,817 projects) have more than 1,000 LOC.

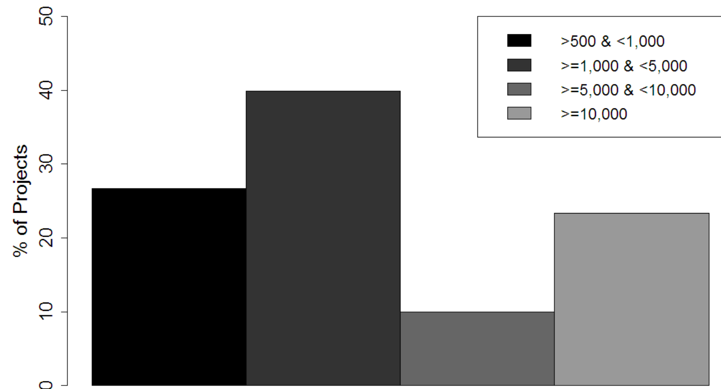


Figure 3.1: Distribution of Projects in Terms of Total Lines of Code

b) Test Cases: Test cases are an important part of a project as they help developers confirm whether their code meets the requirement laid down for the software. Collection of test cases for large number of projects is an arduous task as different languages follow different naming conventions. We perform a *lightweight* identification of test cases that can scale to thousands of projects. We notice that most test cases contain the word “test” as part of their file names. Thus we select files whose name contains the word “test”. For each project, we then count the number of such files which are treated as the number of test cases. We then investigate the relationships between the number of test cases and various project characteristics.

c) Issues & Bugs: GitHub has its own issue tracking system which provides issue trackers for each hosted project where reporters can file issue tickets, and label them with different tags. We collect all the issues (open and close) reported through the in-house tracker. We further find information such as reporter’s identity and different labels used to report the issues. In our dataset, issues are labelled as enhancement, bug, feature requests, error, fixed etc. Further, we find the issues labelled as bugs, errors or defects because they most likely represent the actual bugs in the project. We also calculate the number of bug reporters in a project, i.e., people who reported issues for the project.

d) Developer contributions: Git records store contributors name and email for

each revision of the repository. There are two types of contributors: committers and revision authors. Committers have access to main repository and commit the code contributions from revision authors. These revision authors are the end contributors of the code. We calculate the number of developers (i.e., revision authors) for each project and examine the impact of the number of developers on the presence of test cases.

3.2.2 Research questions

We examine five research questions which pertains to the importance of software testing in software development. We collect several software metrics to investigate correlations between them, which can contribute towards improvement of software testing process and overall software development. We are thus interested in analyzing the following research questions:

RQ1: *How many projects have test cases?* Testing is a crucial activity in the life-cycle of software development process. Testing is used to detect the conditions under which a program may fail and provides directions to rectify that problem. Investigating test cases in a project is important as we wish to know whether projects are properly tested or not. Although presence of test cases does not ensure that project is bug free, but it can help developers analyse the defects and provide motivation to remove those bugs.

In this research question, we examine the prevalence of test cases in open source projects. We analyse the projects containing test cases to investigate whether test cases commensurate with the lines of code of the project.

RQ2: *Does the number of developers affect the number of test cases present in a project?* Developers are the people who are main contributors of the project. They analyse requirements, prepare documents, write code and finally test the code. Usually, developers write unit test cases to test their individual modules or functions as they have better knowledge about the product or application they are developing.

They are the best people to write white box tests as they can develop multiple test cases to extensively test the application. Our dataset consist of both small and big projects where numbers of developers vary from as small as 1 to several thousands collectively working on the project.

Thus, we investigate the correlation between the number of developers working on a project and the number of test cases available for the project.

RQ3: *Does the presence of test cases correlate with the number of bugs?* A bug manifests itself as an error, failure or fault which can seriously affect the functionality of a program. The main objective of running test cases is to detect bugs in the application and find ways to fix it. Test cases can help us to find as many bugs as possible, thus, improving the efficacy of testing. Test cases can be created by analysing the bugs which can be further used to create regression test suite.

In this question, we investigate the correlation between the bug count and the number of test cases. We wish to examine whether presence of test cases has an effect on the bug count.

RQ4: *Does the presence of test cases encourage bug reporting?* Bug reports are the documents which contain details about the bugs in the program. Bug reports increases the chances of removing bugs from the software. Bug reports are also called as fault reports, problem reports, change requests etc. When a developer or tester runs test cases and find bugs, they can log this information in a bug report. Bug reports and test results can be used to analyse the quality of software.

In this research question, we examine, indirectly, whether the presence of test cases persuades users to run these test cases and report bugs. To this end, we determine the correlation between number of test cases and number of bug reporters, i.e., people who report bugs.

RQ5: *Which programming languages appear to have more test cases?* Our dataset consists of 20,817 projects written in different languages. Some people prefer writing code using their favourite language. Although we randomly selected our projects, we still want to determine if people prefer writing test cases in some par-

ticular programming language. Some of the programming languages provide unit test framework which supports writing and running of test cases. So, we investigate whether number of test cases depends upon the popularity of programming languages.

3.2.3 Statistical measurements

To the best of our knowledge, this is the first study which explores relationship of test cases with different characteristics of the project on such a large scale. We use common metrics in statistical analysis to confirm the existence of a correlation among the data and for examining the statistical significance of our figures.

a) The Mann-Whitney-Wilcoxon (MWW) test: The MWW test is a non-parametric statistical hypothesis test to assess the statistical significance of the difference between the distributions in two datasets [79]. As this test does not assume any specific distribution, we use it for our project as we collected data from different open source projects which might not be normally distributed. Given two independent samples x and y, of size n1 and n2 respectively, the MWW test allows us to evaluate whether these distributions are identical. The test first combines and arranges the data points of the two samples in ascending order of their values. Data points with identical values are assigned a rank equal to the average position of those scores in the ordered sequence. Second, the algorithm sums the ranks of data points in the first sample (x). Let us denote this sum as T. The formula for computing the Mann-Whitney U for x is :

$$U = n_1 n_2 + \frac{n_1(n_1+1)}{2} - T$$

The U value calculated above is used to determine the p-value. Given a significance level α = 0.05, if p-value $< \alpha$, then the test rejects the null hypothesis. This implies that at the significance level of α = 0.05, the two datasets have different distributions.

b) Spearman's rho: Spearman's rho (ρ), also known as Spearman's rank correlation coefficient, is a non-parametric measure used to assess statistical dependence between two variables X and Y using a monotonic function. This measure can be used when data is not normally distributed. Thus, making it a good fit for the datasets that we investigate in this study. The values of ρ are limited to the interval [-1; 1]. A perfect Spearman correlation of -1 or +1 occurs when each variable is a perfect monotone function of the other. The closer to 0 ρ is, the more independent the variables are. Equation 2 states the formula for finding this coefficient.

$$\rho = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

In this equation, x_i and y_i represent the ranks of elements Xi and Yi in X and Y respectively, while $\overline{x}$ and $\overline{y}$ represent the averages of the ranks.

3.3 Findings

3.3.1 RQ1: Popularity of Test Cases

To answer this research question, we tabulate the number of test cases in the projects. Table 3.1 shows the distribution of test cases in the projects. After curation, our dataset includes 20,817 projects of significant size, out of which 7,982 projects do not contain test cases, which represents 38.34% of the total projects. The remaining 61.65% of the projects contain one or more test cases. In total, we have 1,875,409 test cases from 12,835 projects in our dataset. We examine how presence/absence of test cases correlate with other characteristics of the projects such as lines of code (LOC).

Table 3.1: Test Cases Distribution

Projects	# of Projects	% of Projects
Without Test Cases	7,982	38.34%
With Test Cases	12,835	61.65%

Table 3.2 details the prevalence of test cases: 84.87% of the projects have less than 100 test cases. 10.7% of the projects have between 100 and 500 test cases, whereas less than 4.5% of the projects have more than 500 test cases. Only 17 projects have more than 10,000 test cases. The table also shows the mean value of the size of the projects. For eg., the mean value for the size of 17 projects, which contain more than 10,000 test cases is 2,568,813.82 LOC.

Table 3.2: Prevalence of Test Cases

# of Test Cases	# of Projects	% of Projects with Test Cases	Mean (LOC)
1-9	6,195	48.26%	12,813.55
10-49	3,769	29.36%	24,681.08
50-99	931	7.25%	47,610.31
100-249	964	7.51%	901,447.06
250-499	410	3.19%	193,629.08
500-999	303	2.36%	197,660.48
1000-4999	219	1.70%	397,159.98
5000-9999	27	0.21%	701,281.66
$>$ 10000	17	0.13%	2,568,813.82

We believe that bigger projects have higher test cases due to large number of functionalities that needs to be tested to produce a high quality software. So, we examine the correlation between the number of test cases in a project to the corresponding number of lines of code.

Figure 3.2: Test Cases and Lines of Code

Figure 3.2 [2] shows the distribution of project sizes (in terms of LOC) for projects

with and without test cases. We observe that projects with test cases have an average of 107,096 LOC (median=3549) whereas average of projects without test cases is 5,605 LOC (median=1353). We compare the LOC numbers of the set of projects with test cases and that of those without test cases using Mann-Whitney-Wilcoxon (MWW) test. Our results show that the difference between these two sets is statistically significant with p-value $< 2.2\ e^{-16}$ [3]. Thus, we can conclude that projects with test cases are bigger in size than the projects without test cases.

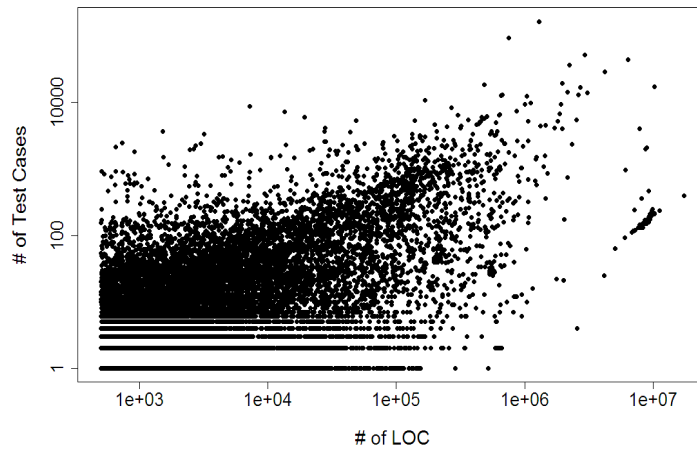


Figure 3.3: Correlation between Test Cases and Lines of Code

To verify that projects with test cases have higher LOC, we analyse the correlation between the number of LOC and the number of test cases. Figure 3.3 shows the scatter plot between the number of LOC and the number of test cases. The graph shows that there is positive correlation between these two metrics. To confirm this correlation, we use Spearman's rho which gave a value of 0.427 with p-value $< 2.2\ e^{-16}$ [4]. The result validates that there is a positive correlation between the number of test cases and the number of LOC.

[2]The line in the middle of the box represents the median. The upper part of the box represents the upper quartile, while the lower part of the box represents the lower quartile. The lines on top and below the box are referred to as whiskers. Data points above and below these whiskers are regarded as outliers – data points which are significantly different from the majority of the data points.

[3]Here, lines of code is the dependent variable and the presence/absence of test cases is the independent variable. The null hypothesis is: there is no difference in the size of projects with test cases and those without test cases. The alternative hypothesis is: projects with test cases have more LOC than those without test cases. We consider a significance level α=0.05. For this α value, if the p-value < 0.05, we reject the null hypothesis.

[4]Null hypothesis (rho is zero) is rejected

Although correlation between the number of test cases and the number of LOC is positive, we wish to examine the correlation between the number of lines of code and the number of test cases per LOC. Here, we only consider projects with test cases and divide the number of test cases by the corresponding LOC of that project. Figure 3.4 depicts the correlation between these two variables. We can observe that with an increase in the number of LOC, we see a decrease in the number of tests per LOC. The Spearman's rho for the distribution is -0.451 with p-value $< 2.2\, e^{-16}$, which confirms that there is a negative correlation between the lines of code and the number of test cases per LOC.

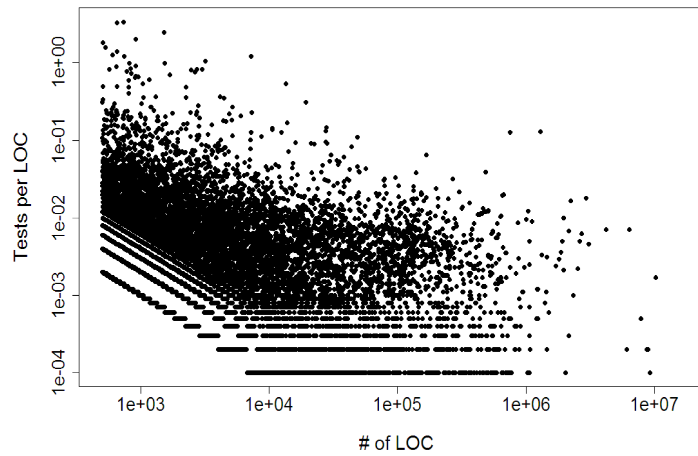


Figure 3.4: Correlation between Test Cases per LOC and Lines of Code

3.3.2 RQ2: Developers and Test Cases

Developers form an important part of the project as they contribute by writing/modifying code, developing test cases, running them and solving bugs logged in bug tracking system. So, finding a correlation between the numbers of developers and the numbers of test cases is important to understand the impact of these developers on the presence of test cases. Our dataset consists of 20,817 projects which contain a total of 2,916,105 developers who have contributed to the code bases of the projects. The projects with test cases have 2,861,031 developers whereas the projects without test cases have 55,074 developers. Thus, projects with test cases have a higher numbers of developers. We can observe from Figure 3.5 that projects with test cases

have more developers. We used MWW test between the set of numbers of developers of projects with test cases and those for projects without test cases which gave p-value $< 2.2e^{-16}$ [5]. The results signify that the difference between these two sets is statistically significant.

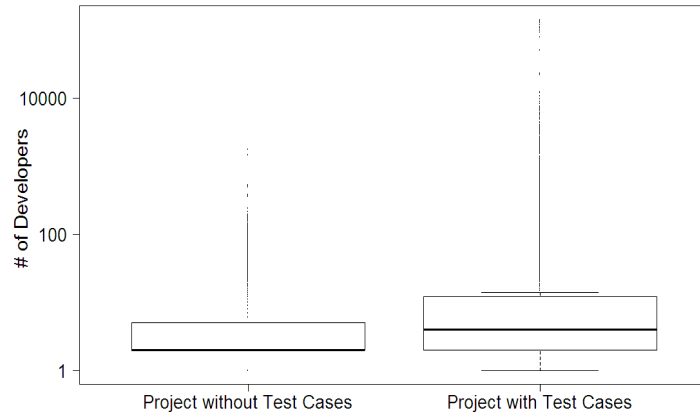


Figure 3.5: Number of Developers in Projects with/without Test Cases

We wish to examine whether increase in the number of developers leads to an increase in the number of test cases in that project. We use scatter plot (Figure 3.6) to examine the correlation between the numbers of developers and the numbers of test cases. We calculated Spearman's rho to confirm the correlation between these two variables which gave a value of 0.207 (p-value $< 2.2\ e^{-16}$). This suggests that there is a weak positive correlation between the number of developers and test cases.

We further investigate the average number of test cases contributed by each developer. For each project, we divide the total number of test cases by the corresponding number of developers in that project. Figure 3.7 depicts the correlation between the numbers of developers and the numbers of test cases per developer. We use Spearman's rho to find the correlation between these two variables. The Spearman's value is -0.444 with p-value $< 2.2\ e^{-16}$. Thus, the correlation between the number of developers and the number of test cases per developer is negative. As

[5]Here, number of developers is the dependent variable and the presence/absence of test cases is the independent variable. The null hypothesis is: there is no difference in the number of developers of projects with test cases and those without test cases. The alternative hypothesis is: projects with test cases have more developers than those without test cases. We consider a significance level α=0.05. For this α value, if the p-value < 0.05, we reject the null hypothesis.

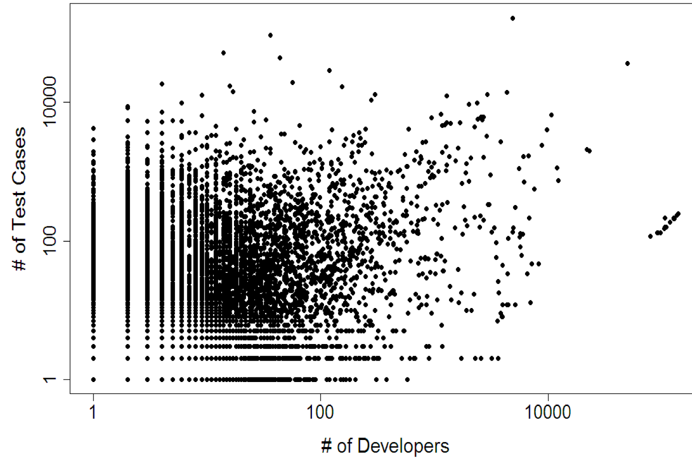


Figure 3.6: Test Cases and Number of Developers

only some of the developers write test cases, we observe a decrease in the test count per developer with an increase in the number of developers.

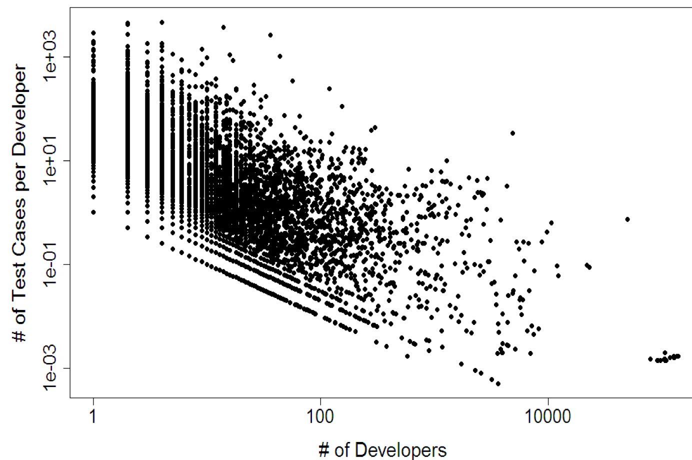


Figure 3.7: Correlation between # of Test Cases per Developer and # of Developers

3.3.3 RQ3: Test Cases and Bug Counts

In this research question we examine whether the number of bugs is correlated with the number of test cases present within a project. First, we identify the issue reports present in our dataset. GitHub provides an issue tracking system which lets users file issue tickets, tag them according to the issue and label them as the state of the issue changes. It also allows the project development team to either enable or disable the issue tracking system. Users can tag issues and categorize them. However, user-supplied tags can create a problem for developers as there can be typographical

errors while tagging. Since tags are not predetermined by GitHub, a tag can be reported in different forms. For example, a bug can be tagged as defect, type:bug, bugfix, etc. Table 3.3 depicts several representations of tags which we count as bugs for our project.

Table 3.3: Tags representing Bugs

bug	bug; T bug; Bug Confirmed; bugs; starter bug; bug fix etc.
defect	defect; Type-Defect; minor defect
error	error; Wow error; build error; error page; user error etc.

Since errors can be represented by any combination of these tags, we use these tags to account for all the bugs. In total, we have 1,081 projects which contain 24,703 bugs as represented by the tags mentioned above. These projects contain 83,576 test cases written by the project development teams.

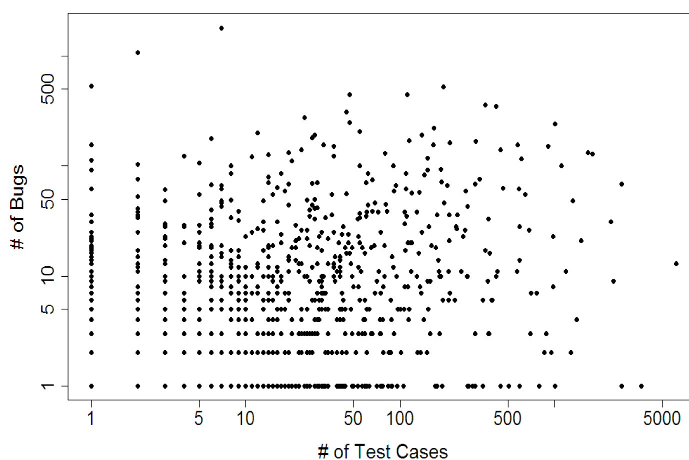


Figure 3.8: Correlation between # of Test Cases and # of Bugs

Our aim is to study and see that with increase in the number of test cases, bug count increases. Figure 3.8 shows a scatter plot to explore the correlation between the number of bugs and the number of test cases. Here, we can see that as the number of test cases increases, we see an increase in the number of bugs. We calculated the Spearman's correlation which yields rho value 0.181 (p-value $= 1.78\ e^{-09}$), suggesting a weak correlation between the number of test cases and the number of bugs.

3.3.4 RQ4: Test Cases and Bug Reporters

We wish to know if the presence/absence of test cases affects bug reporting. We examine the relationship between the number of test cases and the number of bug reporters. Bug reporters are the people who report or log bugs related to a particular application or software. Based on the user names, we collected the data about people who have reported issues in the project. As not all the projects contain issues, we identified 6,230 projects in which users logged issues. These issues were filed by 274,276 reporters.

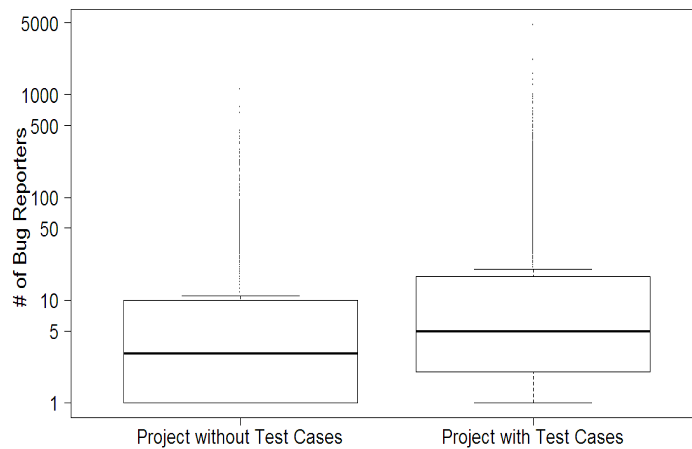


Figure 3.9: Test Cases and Bug Reporters

We can observe from the Figure 3.9 that projects with test cases have higher number of bug reporters (median=5) as compared to projects without test cases (median=3). We performed the MWW test and found that the difference between the set of bug reporters in projects without test cases and those of projects with test cases is statistically significant (p-value $< 2.2\ e^{-16}$). We can infer that if test cases are present, it can persuade users to run these test cases and if they found bugs, they can log them in issue tracking systems.

Figure 3.10 shows the scatter plot of the numbers of bug reporters and the numbers of test cases. We computed Spearman's rho for the distribution which yielded the value 0.171 (p-value $< 2.2\ e^{-16}$), suggesting a weak dependence between the number of test cases and the number of bug reporters.

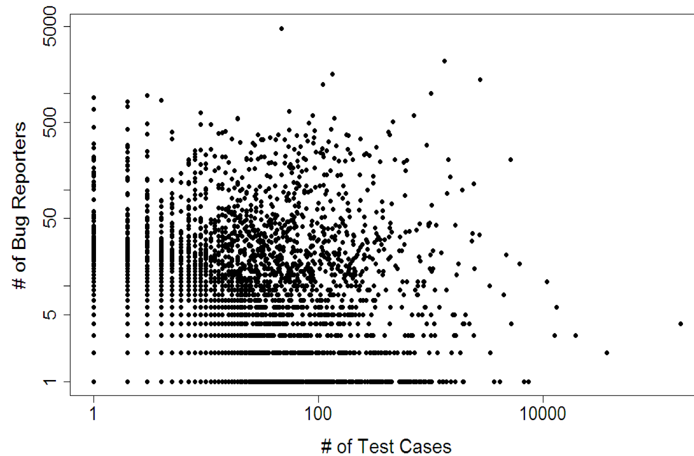


Figure 3.10: Correlation between # of Bug Reporters and # of Test Cases

3.3.5 RQ5: Test Cases and Programming Languages

With this research question, we attempt to establish whether projects written in common languages such as C#, Java, PHP or JavaScript, contain more number of test cases than other languages. We first compute the number of test cases present in projects depending on the programming language that is used. We then select projects developed in the top ten languages with the highest number of test cases.

Figure 3.11 shows the number of projects of the corresponding top ten languages in our dataset. Out of 20,817 projects in our dataset, 19,327 projects use one of these top ten languages. During the analysis, we find out that Java has 3,112 number of projects and also the highest count among all the projects. Our dataset contains 3,016, 2,902 and 2,536 projects written in ruby, PHP and Python respectively. Perl has the lowest number of projects among the projects written in the computed top ten languages. C++ has the highest number of test cases being 648,773 present in 1,920 projects. Then, we have projects written in ANSI C, PHP and Java having respective count of 286,009, 255,553 and 196,703 test cases. Perl has lowest number of test cases, i.e., 7,690 present in 630 projects.

Figure 3.12 shows the distribution of the number of tests of top-10 languages that are used in the projects of our dataset. We observe that median values of some of the pairs such as C# and Ruby, Python and Java, ANSI C and PHP, Objective-C

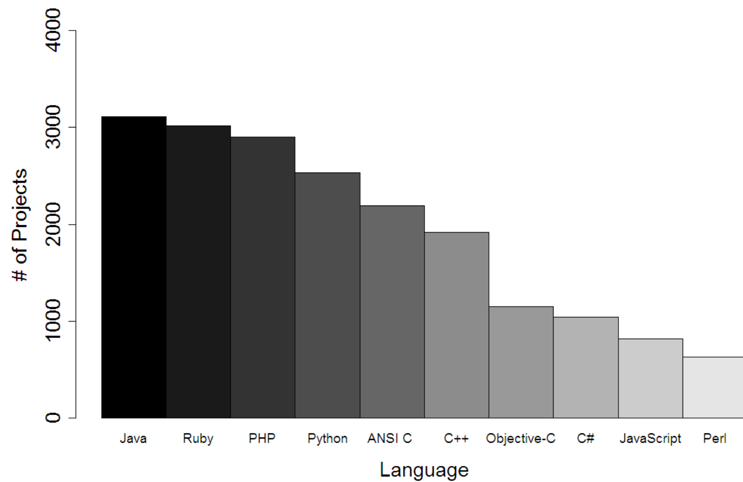


Figure 3.11: Count of Projects and Different Languages

and Perl are almost comparable to each other. JavaScript has a median value of 4 test cases, 1 less than the median value of C# and Ruby.

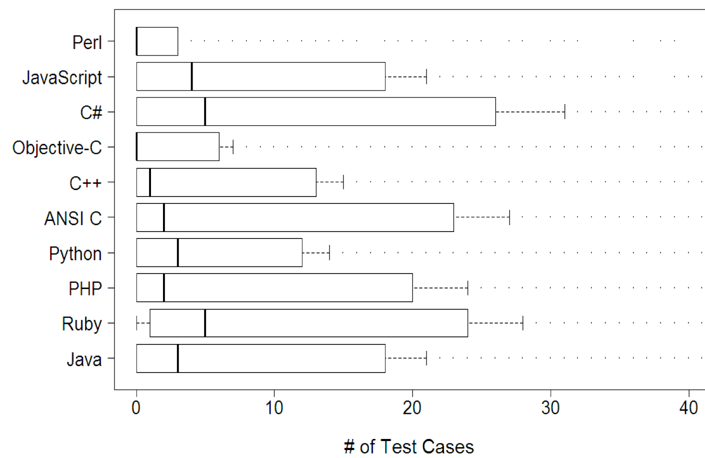


Figure 3.12: Prevalence of Test Cases for Common Languages

As most of the projects have lower number of test cases, we can observe that median line is gravitating towards the left, i.e, data is skewed towards the right. The rest of the projects having higher number of test cases are considered as outliers as they are small in number and does not have a significant impact on the box plots[6]. Thus, we can observe a big difference in the mean and median values for all the languages.

We further analyze the number of test cases per project. Table 3.4 depicts the

[6]https://github.com/isis-project/WebKit having 166488 test cases and https://github.com/chrispilot2293/CM9 having 44871 test cases

mean number of test cases per project for each language. We observe that C++ has the highest value, i.e., 337.90, whereas Perl has the lowest value among all the top ten languages. JavaScript projects has higher number of test cases per project than Python and Objective-C projects. Although the numbers of projects written in C++, ANSI C and PHP are less as compared to the numbers of projects written in Java and Ruby, they have higher mean numbers of test cases per project.

Table 3.4: Distribution of Test Cases per Project

Language	# of Projects	# of Test Cases	Test Cases/Project
C++	1,920	648,773	337.90
ANSI C	2,197	286,009	130.18
PHP	2,902	255,553	88.06
C#	1,042	81,334	78.05
Java	3,112	196,703	63.20
Ruby	3,016	173,864	57.64
JavaScript	819	39,070	47.70
Python	2,536	103,600	40.85
Objective-C	1,153	21,343	18.51
Perl	630	7,690	12.20

3.4 Conclusion

Our analysis shows the following results:

1. Projects with test cases have more LOC than those without test cases. As projects grow in size the number of test cases per LOC decreases.

2. Projects with more number of developers have more test cases. However as the number of developers grow, the number of test cases per developer decreases.

3. There is weak positive relationship between number of test cases and the number of bugs.

4. Number of test cases has a weak correlation with the number of bug reporters.

5. Projects written in popular languages, such as C++, ANSI C, and PHP, have higher mean numbers of test cases per project as compared to projects in other languages.

Chapter 4

Adequacy of Software Testing

4.1 Introduction

One metric that is commonly used to measure the adequacy of testing is code coverage, that is, a measure of the set of lines of code or code paths that are executed by a set of tests. Quality managers can use coverage information to assess test suites, to decide when to stop testing, and to focus attention on portions of the code that are not covered and thus may contain faults [142]. Judicious use of code coverage can help in finding new defects and increasing the robustness of the software [14]. Furthermore, software cost models based on coverage information can be used to estimate the cost of testing, the cost of removing faults and the potential risk caused by bugs emerging from uncovered code [98]. Measuring coverage alone, however, is not enough to obtain a complete picture of the state of testing in open-source software. To understand, and potentially improve, the state of testing in open-source software, it is necessary to correlate code coverage information with other software metrics that can characterize the software development process, such as lines of code, cyclomatic complexity, and number of developers. These easy-to-collect metrics can help characterize projects in which testing is insufficient, and thus can help developers and managers assess when more testing effort for their software may be required.

4.2 Methodology & Statistics

For our empirical study, we downloaded the projects from GitHub and the projects distributed by Debian. Our dataset includes projects developed by well-known organisations such as The Apache Software Foundation and The Eclipse Foundation.

We clone projects that use Maven as the project management tool. Out of 945 projects, 872 projects contain test suites. We analyse these projects and run test cases. For project set-up, we run the command `mvn clean install`, which clears any pre-compiled files of previous builds, builds a dependency tree for all the sub projects specified in the *pom.xml* (the root POM) and compiles all the *.java* files. Then, we run Sonar using the command `mvn sonar:sonar`, which performs dynamic analysis by running test cases and then creates reports based on the results. Unfotunately, many of the projects had compilation errors and dependencies on unavailable external libraries. This observation is consistent with the results of others [40]. We tried to resolve these issues, however, if after some effort the project still failed, we discarded the project. In the end, we have 327 projects that successfully compile, run test cases and produce coverage.

We compute these statistics to characterize the projects in our dataset and assess the suitability of these projects as representative samples to answer our three research questions. These basic statistics also describe the range of values of the various metrics for the projects in our dataset. We analyse the correlations of these metrics with code coverage in Section 4.3.

a) Lines of code (LOC): We used Sonar to count the lines of code of projects in our dataset, excluding comments, blank lines and test cases. Figure 4.1a depicts the distribution of the number of lines of code of the projects in our dataset. 90 projects have between 1 and 5,000 LOC, 56 projects have between 5,000 and 10,000 LOC, 129 projects have between 10,000 and 50,000 LOC, and 25 projects have between 50,000 and 100,000 LOC and 27 projects have more than 100,000 LOC. The largest project in our dataset is Apache Hadoop, which contains 454,137 LOC. The mean

size of the projects is 31,120.71 LOC and the median size is 11,484 LOC.

b) Test Cases: We use Sonar to collect the total number of test cases for each project. Sonar also gives information about the number of test cases that failed and the number of test cases that were skipped. Test cases can be skipped due to compilation errors, missing dependencies, etc.

Figure 4.1b shows the distribution of test cases across projects. 147 projects have fewer than 100 test cases, whereas 41 projects have more than 1,000 test cases. 96 projects have between 100 and 500 test cases and 43 projects have between 500 to 1,000 test cases. The number of test cases varies from 1 to 31,414. The mean number of test cases per project is 563.97 and the median value is 141.

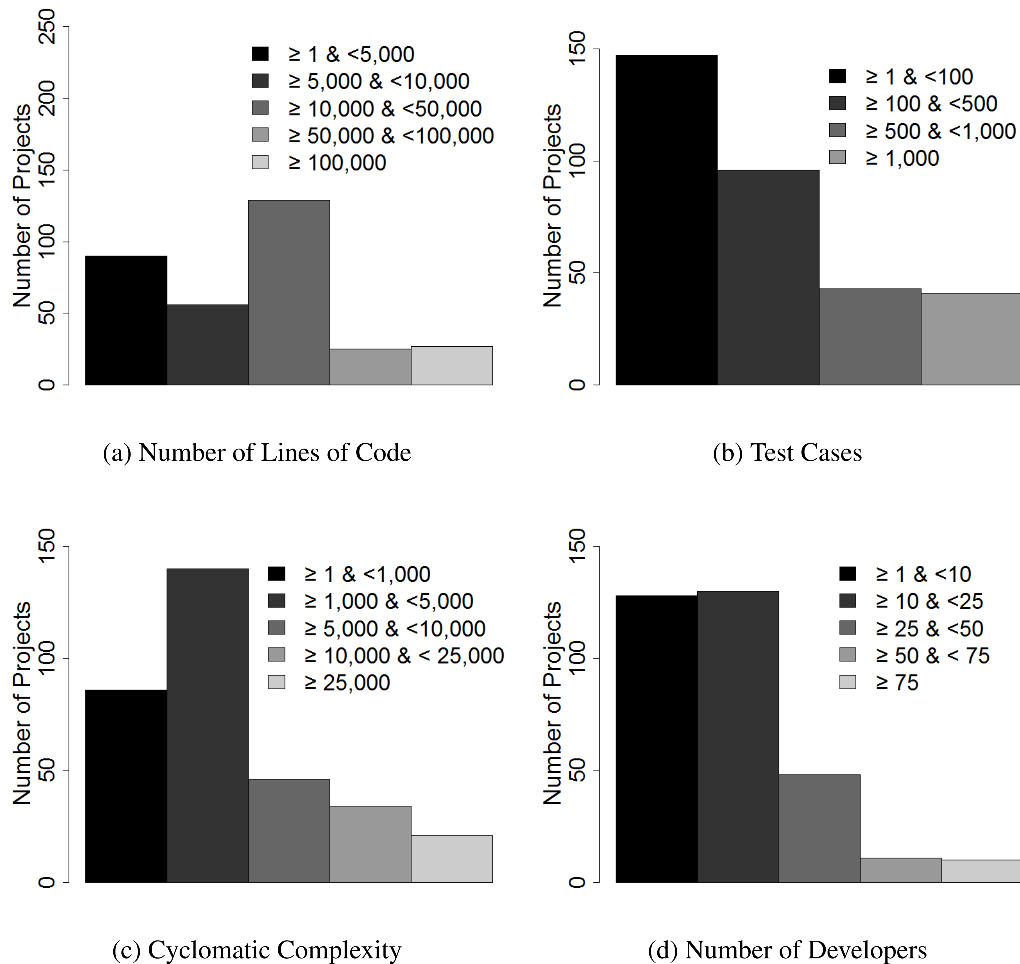
c) Cyclomatic complexity: Cyclomatic complexity is a measure of the number of linearly independent paths through the source code of a software program [81]. Cyclomatic complexity is particularly useful in approximating the number of test cases necessary to ensure exhaustive testing [129]. A program with low complexity is typically easier to maintain [36].

Figure 4.1c depicts the distribution of cyclomatic complexity of projects in our dataset. 86 projects have complexity between 1 and 1,000, 140 projects have complexity between 1,000 and 5,000, 46 projects have complexity between 5,000 and 10,000 and 34 projects have complexity between 10,000 and 25,000. 21 projects have complexity above 25,000 with the highest complexity value being 114,045. We can observe that most of the projects have complexity below 10,000.

d) Developer contributions: Our projects use different version control systems such as *git*, *svn* and *hg (mercurial)*, so we use *git log*, *svn log*, and *hg log*, respectively, to examine the commit history of all the projects and to extract the names of all of the developers working on these projects. We also collect developer information at the file level, i.e., the number of developers who have made changes to each file.

Figure 4.1d shows the distribution of the number developers of all the projects. 128 projects have between 1 and 10 developers, 130 projects have between 10 and

25 developers, 48 projects have between 25 and 50 developers and 11 projects have between 50 and 75 developers. 10 projects have more than 75 developers, with the project *Netty* having the highest number of developers i.e., 146. The mean and median numbers of developers across all the projects are 18.15 and 13, respectively.



(a) Number of Lines of Code

(b) Test Cases

(c) Cyclomatic Complexity

(d) Number of Developers

Figure 4.1: Distribution of Projects.

4.3 Findings

4.3.1 RQ1: Coverage levels and Test Success Densities

Motivation: Investigating coverage level of a project is important in understanding the reliability of the software project. A test suite with high coverage is likely to have a higher fault detection capability and to better help developers find bugs than

the one with low coverage [46].

Findings: Table 4.1 shows the distribution of coverage levels. Most of the projects exhibit low coverage levels, as the average coverage (i.e., sum of coverage of all projects divided by number of projects) is only 41.96% and the median coverage is only 40.30%. Almost one-third of the projects have coverage between 0% and 25%.

Table 4.1: Project Distribution across Coverage Levels

Coverage Level (%)	Number of Projects
0-25	105
25-50	90
50-75	92
75-100	40

Coverage indicates the amount of code touched by the test cases, but does not ensure that the program runs correctly on the tests. We thus next calculate test success density as the number of test cases that are executed successfully out of the total number of test cases. Figure 4.2 depicts the test success density of all the projects in our dataset. We observe that 254 projects have test success density greater than or equal to 98%, out of which 200 projects have 100% success density. 45 projects have test success density between 75% and 98%, and 6 projects exhibit success density between 25% and 50%. Only 9 projects in our dataset show a success rate below 25%.

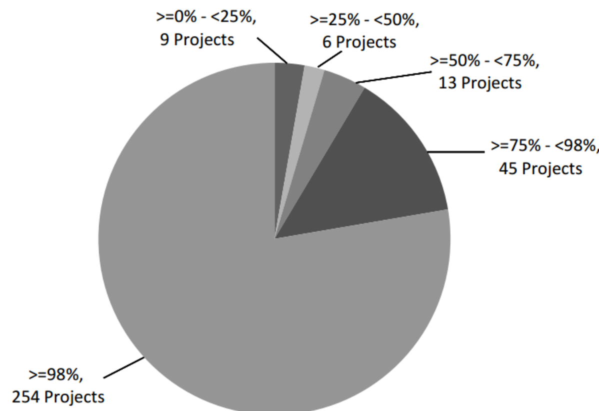


Figure 4.2: Test Success Density

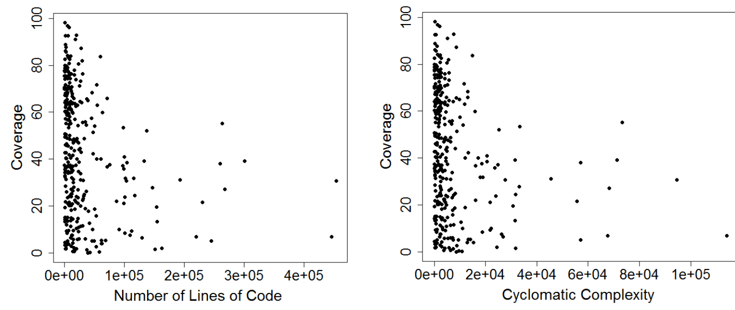
4.3.2 RQ2: Correlations at the Project Level

Motivation: Software metrics, such as lines of code, code coverage, cyclomatic complexity, etc., give quantitative measurements of the degree to which a software project and its development process exhibit a particular attribute. These metrics can be used to improve the software quality, to analyse the productivity of a software project team, to anticipate the future needs of developers and to estimate the amount of maintenance required for the project. Comparing various metrics with code coverage can help us understand which project attributes are correlated to the adequacy of testing. This understanding can help us identify characteristics of projects that are prone to inadequate testing.

Findings: First, we analyse the correlation between lines of code and amount of code coverage. As the quality of the software is related to coverage [86], we believe that the coverage should either remain the same or increase with an increase in LOC.

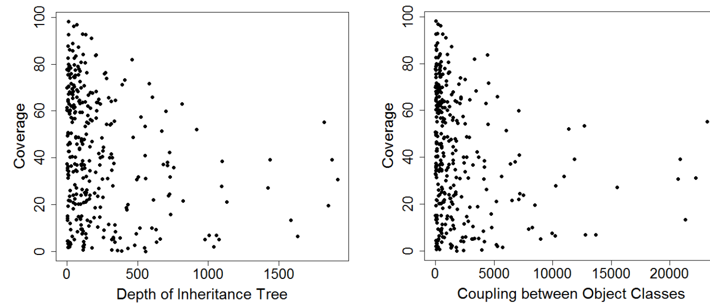
The scatter plot (Figure 4.3a) between the number of lines of code and coverage shows that as the number of lines of code increases, the coverage level actually decreases. The Spearman's ρ for the distribution is -0.306 with p-value = $1.566e^{-08}$, which shows that there is a negative correlation between number of lines of code and code coverage. This could be due to the reason that as the size of a project increases, adding new test cases become increasingly difficult. Furthermore, some parts such as getters and setters do not need testing and there is no coverage for them but they still add lines of code to the overall project.

Cyclomatic complexity of software generally increases with an increase in the number of lines of code [28, 62]. As the cyclomatic complexity of a software project increases above a threshold, the software becomes error prone [129]. Figure 4.3b depicts the scatter plot between coverage and cyclomatic complexity. The coverage level decreases with an increase in the complexity of the code. Spearman's ρ for the distribution is -0.276 (p-value = $3.665e^{-07}$), which shows a negative correlation



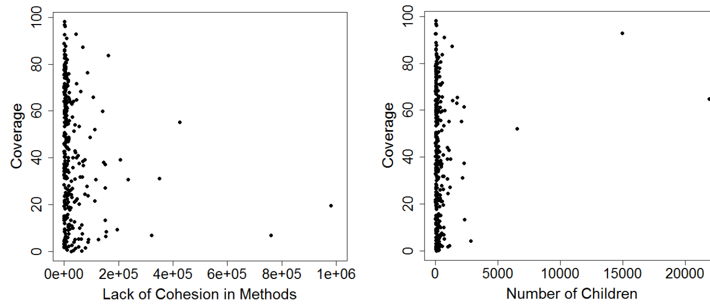
(a) Number of Lines of Code vs. Coverage
ρ = -0.306, p-value $< 1.566e^{-08}$

(b) Cyclomatic Complexity vs. Coverage
ρ = -0.276, p-value $< 3.665e^{-07}$



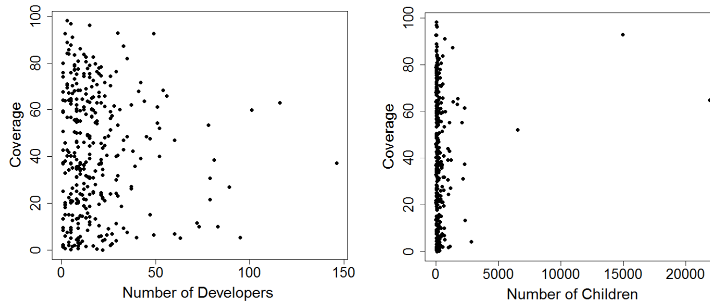
(c) DIT vs. Coverage
ρ = -0.264, p-value $< 1.337e^{-06}$

(d) CBO vs. Coverage
ρ = -0.294, p-value $< 5.825e^{-08}$



(e) LCOM vs. Coverage
ρ = -0.253, p-value $< 3.441e^{-06}$

(f) NOC vs. Coverage
ρ = -0.171, p-value = 0.002



(g) RFC vs. Coverage
ρ = -0.302, p-value = $2.583e^{-08}$

(h) Number of Developers vs. Coverage
ρ = 0.017, p-value = 0.763

Figure 4.3: Scatter Plots (Project Level)

between cyclomatic complexity and code coverage. Similar to above, adding test cases and increasing coverage for complex parts becomes difficult with an increase in size. Furthermore, test cases are not added for parts with low complexity.

Researchers in the past have shown that CK metrics are associated with system maintainability and defect proneness. Ideally, projects with higher CK metric scores need to be tested more rigorously. Thus, we want to investigate if there is a correlation between these metrics and coverage. This will inform us whether the rigor developers test their projects commensurate to the values of these metrics. Figures 4.3c, 4.3d, 4.3e, 4.3f and 4.3g show the scatter plots between the CK metrics (DIT, CBO, LCOM, NOC, and RFC) and coverage, and the Spearman's ρ values are -0.264 (p-value = 1.337e^{-06}), -0.294 (p-value = 5.825e^{-08}), -0.253 (p-value = 3.441e^{-06}), -0.171 (p-value = 0.002) and -0.302 (p-value = 2.583e^{-08}) respectively. The values show that there is a small negative correlation between the CK metrics and coverage. The above results show that with the increasing values of CK metrics, the coverage level decreases.

The above observations highlight that open-source developers need to increase the testing effort, to maintain or increase the code coverage level with the increase in size or complexity or non-maintainability or defect proneness of the software. Thus, developers who are working on large, complex, less maintainable, and more defect prone projects should put more emphasis on testing to improve the reliability of the software.

Test cases are contributed by the developers of a software project. These developers play a significant role in writing and running these test cases. Figure 4.3h depicts the scatter plot between the number of developers and the code coverage of the project. The Spearman's ρ value is 0.017, with a p-value of 0.763, which shows that the correlation between the coverage level and the number of developers is insignificant.

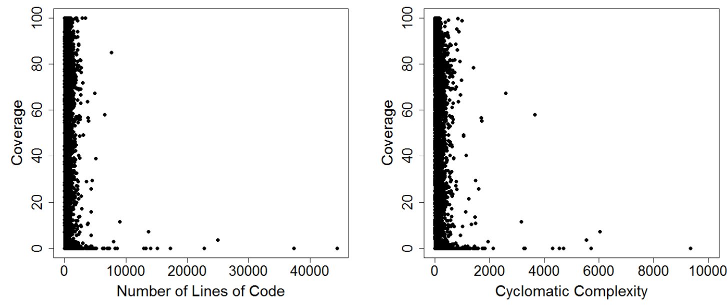
4.3.3 RQ3: Correlations at the File Level

Motivation: The coverage level of the overall software gives an idea of how well a project is tested. However, a project may consist of many files having diverse properties. So, we want to additionally examine the software metrics at the file level, which can help us to study how these metrics, which vary from file to file, are correlated to code coverage. This can enhance our understanding of the characteristics of files that are inadequately tested.

Findings: We extract the number of lines of code and coverage level for all of the files that constitute a project. In total, we have 104,797 Java class files, that compile successfully, accumulated over all the projects. Figure 4.4a shows a scatter plot of the number of lines of code and coverage. The results are contrary to the correlation between LOC and coverage at project level (Figure 4.3a). The Spearman's ρ for the distribution is 0.183 (p-value $< 2.2e^{-16}$) depicting a small positive correlation.

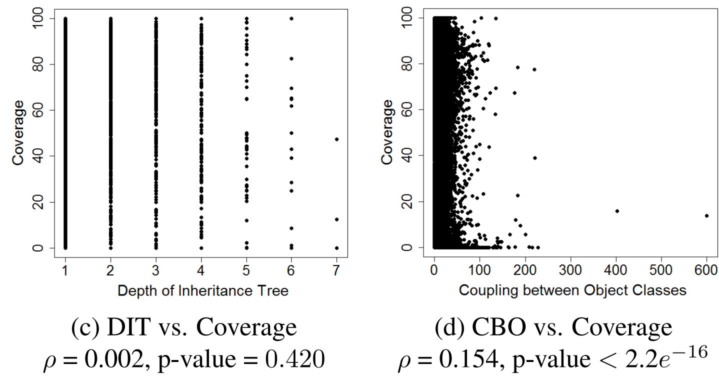
We proceed to investigate the correlation between complexity and coverage at the file level. In Figure 4.3b we observed that with an increase in the complexity of the system, the coverage of the system drops. We want to determine if more complex files are less covered than less complex files, which would lead to an overall reduction in the coverage of the software. We draw a scatter plot depicting the relationship between complexity and coverage level of source code files in Figure 4.4b. The Spearman's ρ for the distribution is 0.223 (p-value $< 2.2e^{-16}$), which shows that there is small positive correlation between cyclomatic complexity and code coverage. The results are contrary to the correlation of complexity and coverage for the overall project.

Next, we examine the CK metrics at the file level. The values of these metrics at the project level gives us an overall understanding of the system, however these values may vary from file to file. Thus, we also analyse the correlations of five metrics i.e., DIT, CBO, LCOM, NOC and RFC with code coverage. Figures 4.4c, 4.4d, 4.4e, 4.4f and 4.4g show the scatter plots between CK metrics (DIT, CBO,



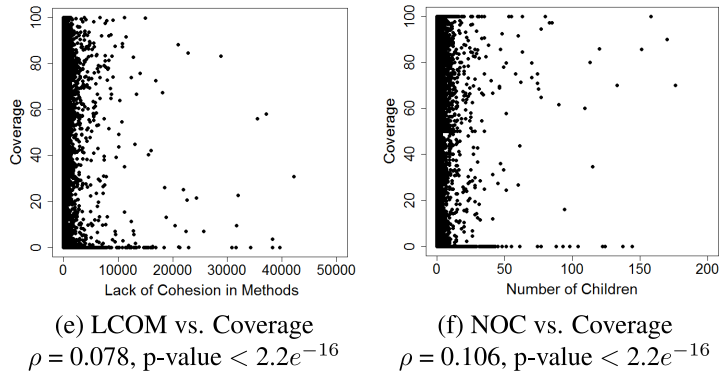
(a) Number of Lines of Code vs. Coverage
$\rho = 0.183$, p-value $< 2.2e^{-16}$

(b) Cyclomatic Complexity vs. Coverage
$\rho = 0.223$, p-value $< 2.2e^{-16}$



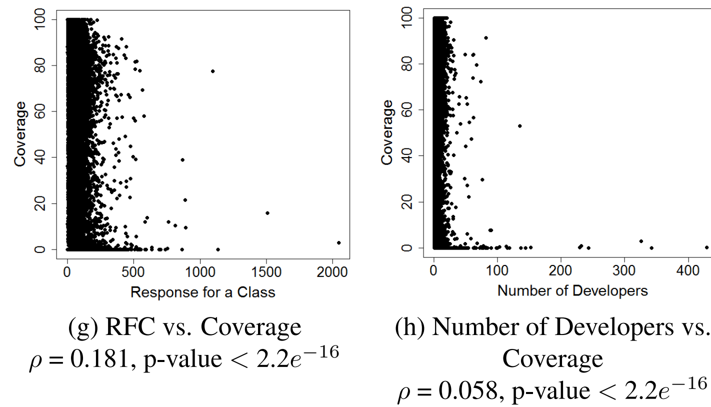
(c) DIT vs. Coverage
$\rho = 0.002$, p-value $= 0.420$

(d) CBO vs. Coverage
$\rho = 0.154$, p-value $< 2.2e^{-16}$



(e) LCOM vs. Coverage
$\rho = 0.078$, p-value $< 2.2e^{-16}$

(f) NOC vs. Coverage
$\rho = 0.106$, p-value $< 2.2e^{-16}$



(g) RFC vs. Coverage
$\rho = 0.181$, p-value $< 2.2e^{-16}$

(h) Number of Developers vs. Coverage
$\rho = 0.058$, p-value $< 2.2e^{-16}$

Figure 4.4: Scatter Plots (File Level)

LCOM, NOC and RFC) and coverage. The Spearman's ρ values are 0.002 (p-value = 0.420), 0.154 (p-value = $2.2e^{-16}$), 0.078 (p-value = $2.2e^{-16}$), 0.106 (p-value = $2.2e^{-16}$) and 0.181 (p-value = $2.2e^{-16}$), respectively. The values show that there are small positive correlations between CBO, NOC, RFC and coverage, no correlation between LCOM and coverage, and an insignificant correlation between DIT and coverage.

Finally, we examine the correlation between the number of developers and the coverage levels of files created by those developers. For each file, we consider the number of developers to be the number of people who have been the author of at least one commit that touches the file. Figure 4.4h depicts the correlation between the number of developers and coverage for all the files contained in the projects. There is no correlation between the number of developers and coverage level of the files. The Spearman's ρ value is 0.058 with p-value $< 2.2e^{-16}$.

4.4 Conclusion

Our empirical work highlights the following results:

1. Most of the projects have low coverage levels, with an average of 41.96%.
2. 254 out of the 327 projects that build successfully have test success density above 98%.
3. Code coverage of a project decreases with the increase in the size as well as cyclomatic complexity of the project.
4. However, at the file and component levels, coverage increases with the increase in the sizes of the file and component as well as their complexity.
5. The number of developers has an insignificant correlation with the coverage at the project level, no correlation with the coverage at the file level and a small positive correlation with the coverage at the component level.

6. The values of the CK metrics, i.e., DIT, CBO, LCOM, NOC and RFC, decrease with the increase in the coverage at the project level. However, these values increase with the increase in the coverage at the component level. At the file level, the values increase with coverage except DIT and LCOM values, which have insignificant and no correlation with coverage, respectively.

Chapter 5

Understanding the Testing Culture of App Developers

Smartphones have become pervasive and platforms such as Android and iOS have gained tremendous popularity recently. According to a Gartner study, worldwide sales of smartphones to end users increased by 42.3% in 2013 as compared to the previous year and Android had 78.4% of the market share of the smartphone sales in 2013 [35]. Furthermore, easy availability of app construction frameworks and dissemination through online app stores such as Google Play[1] and Apple App store[2] have attracted a large number of developers and organizations to develop and market their apps. However, low barriers to development does not ensure that apps are error free. These error-prone apps can significantly impact user experience and may cause harm to the reputation of the developers or the organizations. Therefore, it is important to adequately test these apps before releasing them to the market. A reliable app with few or no bugs is likely to have a higher chance of being well-received by the large user base of these smartphones than the unreliable ones.

[1]https://play.google.com/store?hl=en
[2]https://itunes.apple.com/us/genre/ios/id36?mt=8

5.1 Introduction

Although mobile apps use common technologies such as Java, they significantly differ from web-based and desktop-based applications. An app receives a variety of inputs from users and its environment which makes it difficult to write effective test cases. Thus, many recent studies propose new testing tools that are specifically designed for mobile applications [78, 38, 85]. Despite the growing interest in the software testing and reliability research community to build tools that can automate and improve testing of mobile apps, there has been no study that investigates how developers test these applications in practice. This study is needed to understand the "pain points" that these developers face which can be used to motivate future research that addresses concerns that matters to mobile app developers.

To address this need, we conduct an empirical study which is divided into two parts. In the first one, we analyze over 600 open-source Android apps to examine the current state of testing in the Android development community. Our dataset includes small apps to large and popular apps such as K-9 Mail[3], FrostWire - Downloader/Player[4], OsmAnd Maps & Navigation[5] and OI File Manager[6], which have more than 1,000,000 installs. In the second one, we conduct surveys with Android and Microsoft app developers to understand common testing tools used, why they are used and challenges faced during testing.

5.2 Methodology & Statistics

We collect URLs of all the applications stored on F-Droid[7] repository and select apps which are hosted on GitHub. In total, we have 627 apps in our dataset.

[3] https://play.google.com/store/apps/details?id=com.fsck.k9
[4] https://play.google.com/store/apps/details?id=com.frostwire.android
[5] https://play.google.com/store/apps/details?id=net.osmand
[6] https://play.google.com/store/apps/details?id=org.openintents.filemanager
[7] https://f-droid.org/

Test Cases & Coverage: For each app, we examine the presence of test cases by checking for the existence of files which contain the word "Test". We observe that many test files have the word "Test" either in the beginning of their names, e.g., TestUtil.java, or at the end of their names, e.g., AccentTest.java. For projects containing test files, we manually investigate them to build them and run test cases. Some projects fail to compile due to dependencies on external libraries. We try to resolve these dependencies issue by downloading libraries. However, many projects still fail to compile. For projects which compile successfully, we run the test cases present in the project repository and calculate code coverage using Emma code coverage tool[8].

Survey:

First Study - For each of the 627 apps, we collect e-mail addresses of all developers that developed these apps. In total, we sent out e-mails to 3,905 distinct e-mail addresses and ask developers questions about testing tools used by them and challenges that they face while testing their applications. Many of these developers work on both open source and commercial projects. We received a total of 83 responses (response rate of 2.13%). The unit of analysis is individual developer.

Second Study - Based on the responses from the first study, we improve our survey questions and resend the survey to Windows app developers in Microsoft. We sent out e-mails to 678 developers and received a total of 127 responses (response rate of 18.73%). The unit of analysis is individual developer.

For the first study, we use a structured survey which consists of several open-ended questions. The following are the questions that we ask as part of our survey:

[8]http://emma.sourceforge.net/

1) How do you test your app code?

Free form text

---

2) Do you use any test automation tools (e.g., monkeyrunner, robotium, robolectric, etc)? If so, what tools do you use and why do you use them (e.g., for generating test cases, for managing test suites etc.)

Free form text

---

3) What are the challenges you face during testing either manually or using automated tools (e.g., lack of documentation, limited support, unclear benefits, etc.)?

Free form text

For the second study, we also use a structured survey. However, we add additional questions, and provide multiple choices to better understand app developers testing behaviors. The questions that we ask include:

---

1) How do you test your app code?

Checkbox options: Manually, use automated testing tools, don't test, other

---

2) If you test your apps, what type of testing do you do on your apps?

Checkbox options: Unit testing, integration testing, system testing, functional testing, regression testing, acceptance testing, load testing, performance testing, beta testing, other

---

3) If you use automated testing tools for your apps, what are the names of the testing tools?

Free form text

---

4) If you use automated testing tools, why do you use testing tools?

Checkbox options: Generating test cases, executing test cases, managing test suites, creating and evaluating test execution results, analysing code coverage, finding potential bugs, reporting bugs, performing load testing, other

5) Do you face the following challenges during testing either manually or using automated testing tools and if you do how serious are they?

Challenges: Time constraints, compatibility issues, lack of exposure to tools, emphasis on development rather than testing, lack of support from employer/organization, unclear benefits of tools, poor documentation, lack of experience, steep learning curve.

Seriousness levels: Very serious, serious, insignificant, do not face, no opinion

6) Given the availability of testing tools for app development, in your opinion what are the top 2 things you look for/need/would like to see?

Free form text

Basic Statistics

We now present some statistics describing the data collected for our study in terms of number of test cases, lines of code and number of developers.

a) Test Cases: Table 5.1 shows the number of apps with and without test cases. We find that 538 (85.81%) apps do not have any test cases, whereas 89 (14.19%) apps have at least one test case. This shows that a large number of Android apps lack test cases.

Table 5.1: Distribution of Apps in Terms of Presence of Test Cases

Categories	# of Apps	% of Apps
Without Test Cases	538	85.81%
With Test Cases	89	14.19%

b) Lines of Code (LOC): We count the lines of code for all the apps in our dataset. Figure 5.1 shows the distribution of LOC. We can observe that 146 apps have sizes between 1 LOC to 1,000 LOC, whereas 234 apps have sizes between

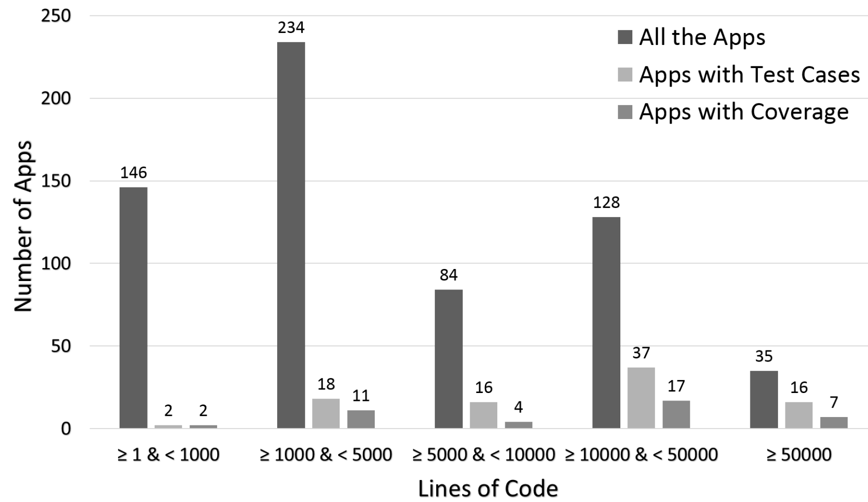


Figure 5.1: Distribution of Apps in Terms of Total Number of Lines of Code

1,000 LOC to 5,000 LOC. Furthermore, 128 apps have sizes between 10,000 LOC to 50,000 LOC and 35 apps are larger than 50,000 LOC. The largest project in our dataset is FrostWire - Downloader/Player[4], which is a native BitTorrent & Cloud file downloader with 1,070,130 LOC. Figure 5.1 shows all the apps, apps with test cases and apps for which test cases run successfully and we get coverage information.

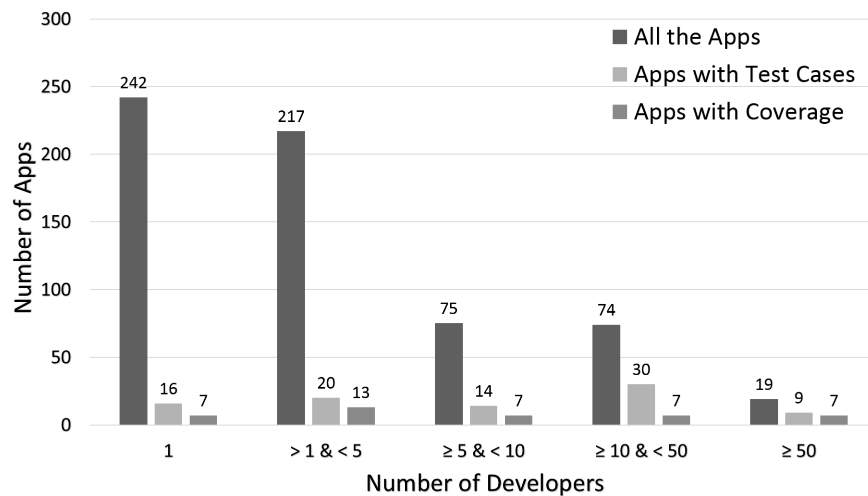


Figure 5.2: Distribution of Apps in Terms of Number of Developers

c) Number of Developers: We want to analyse number of developers involved in the development of an app. We use the information from *git logs* and collect unique e-mail addresses to count the number of developers. Figure 5.2 shows the distribution of the number of developers who worked on different apps in our dataset. We can observe that a large number of apps (242 apps) are developed by a single

developer. Also, 217 apps have more than 1 but less than 5 developers, whereas 75 apps have greater than or equal to 5 developers but less than 10 developers.

5.3 Findings

5.3.1 RQ1: Current State of Testing in Android Applications

Figure 5.3 shows the distribution of test suites for the 89 apps. We find that 19 apps have only 1 test suite each, whereas 11 apps have more than 25 test suites. Furthermore, 23 apps have more than equal to 5 but less than 10 test suites. We can observe that most of the apps have very few test suites.

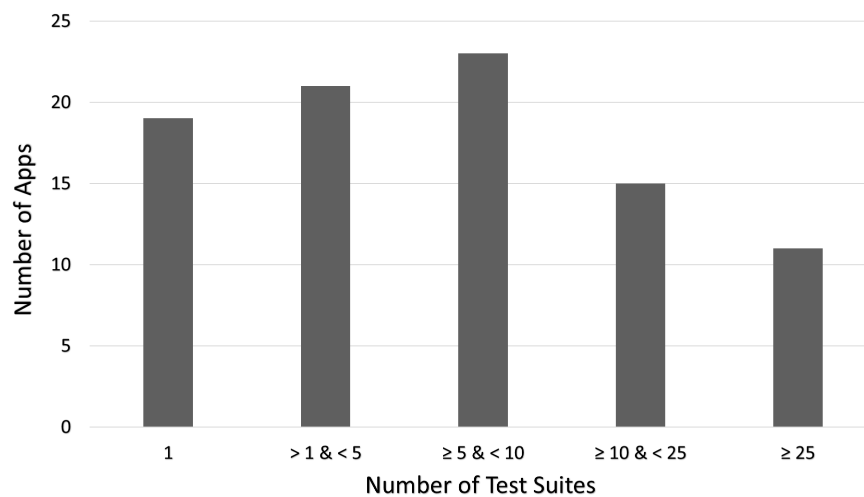


Figure 5.3: Distribution of Apps in Terms of Total Number of Test Suites

We use coverage as a measure for the adequacy of testing. We want to analyse if the projects which have test cases are thoroughly tested or not. We use two measures of coverage:

1. *Line Coverage* measures the proportion of lines executed during testing.
2. *Block Coverage* measures the proportion of code blocks covered, where each block is a sequence of statements without any jumps or jump targets which is executed as one atomic unit.

Out of the 89 apps, we have 41 apps which compile successfully and we run test cases for these apps. We then calculate code coverage for these apps.

Figures 5.4 and 5.5 show the line and block coverage of the 41 projects, respectively. We observe that 37 projects have line coverage of less than 40%, whereas 36 projects have block coverage of less than 40%. The mean and median value of line coverage is 16.03% and 9.33%, whereas the corresponding values for block coverage are 17.22% and 10.65%, respectively. The results show that most of the apps have low coverage, which shows that apps are not adequately tested.

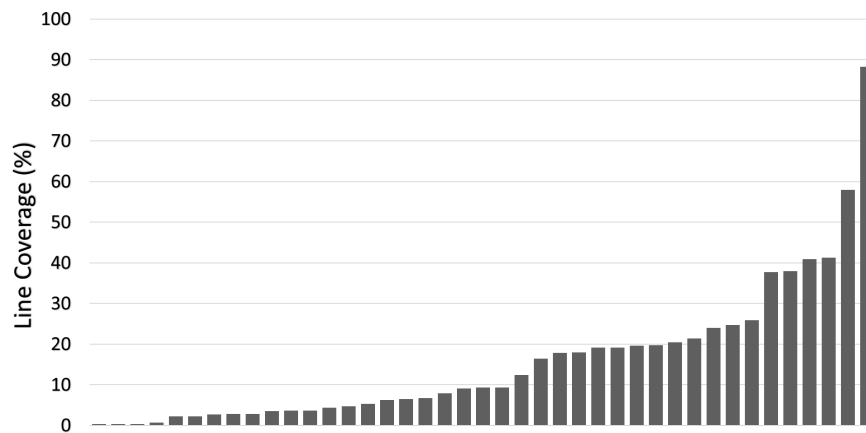


Figure 5.4: Line Coverage (Ascending Order)

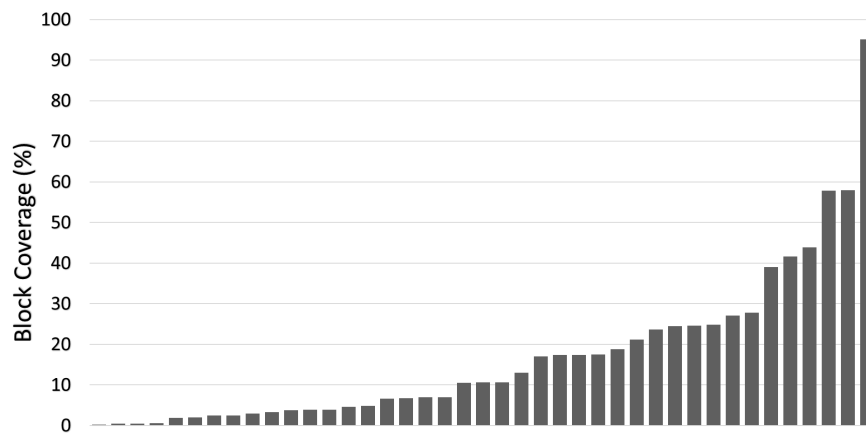


Figure 5.5: Block Coverage (Ascending Order)

5.3.2 RQ2: Survey of Android Developers

A large number of automated testing tools are available to test Android applications. Table 5.2 shows the number of respondents who use a particular tool. Some developers use more than one tool simultaneously. We briefly explain some of tools commonly used by Android developers.

a) JUnit[9] - A popular unit testing framework for Java. Since Android applications are written in Java, it can be directly used to test the parts of the code that do not call the Android API.

b) MonkeyRunner[10] - Monkeyrunner tool provides an API to write programs to control an Android device or emulator from outside of the Android code.

c) Robotium[11] - Robotium is a test automation framework, which allows developers to write black-box UI tests for Android apps. Robotium enables developers to write function, system and user acceptance test spanning multiple Android activities.

d) Robolectric[12] - It is a unit test framework for Android, which allows developers to execute test cases in Java Virtual Machine (JVM), rather than running on a mobile device or emulator.

We can observe that JUnit is the most commonly used testing framework. This could be due to the fact that JUnit is one of the mature frameworks and have been used extensively in the industry.

Some developers often leverage automated testing tools to test their apps based on the requirements of the project and the functionalities provided by the tool. One of the developers said:

"Robolectric. I use this pretty heavily for unit testing, but the scope of tests is

[9] http://junit.org/
[10] http://developer.android.com/tools/help/monkeyrunner_concepts.html
[11] https://code.google.com/p/robotium/
[12] http://robolectric.org/

Table 5.2: Automated Testing Tools Usage

Tools	Number of Respondents
JUnit	18
MonkeyRunner	8
Robotium	7
Robolectric	6
Android unit testing framework	6
Monkey	1
Espresso	1
TestNG	1
Other tools	1
No tool	35

rather limited at the moment. I run my suite of tests against my data model before checking in code. I find this to be the most mature framework at the moment, but the amount of supported features is still a bit limited as its a community driven project. There have been a number of areas (e.g. the PreferenceActivity and Preferencefragment classes) that are a bit more limited in scope.

MonkeyRunner. I run tests using this generally the night before uploading an app. My UI tends to be fairly stable at this point, so it's not that helpful, but it usually catches any serious functionality that I might have broken.

Robotium. I don't use this at the moment, but I intend to in the future. One of my limitations here right now is that there is no free "recorder" software that I'm aware of at the moment (a recorder would track a series of keystrokes for testing purposes, so I could repeat app tests rather than having to do this by hand). I need to research this a bit further."

Developers have varying opinions over usage of these tools. Some of them regularly use such tools (*"I use Jenkins as a tool for Continuous integration, for testing I use monkeyrunner, roboelectric, it's easy to integrate it with Jenkins. I also use uiautomator for testing the UI interface."*), whereas others prefer to use older frameworks like JUnit (*"I am testing my application logic (ie. service layer) with JUnit and/or TestNG as it is not dependent on Android framework. I usually*

do not use automation tools for GUI itself, in fact my experience with GUI testing frameworks is somewhat ... unbalanced."). On the other hand, some developers prefer to write their own scripts to test their applications (*"Honestly I prefer to code instead of spent my time figuring out how complex debugging tools works."*).

Our survey shows that some developers are aware of the new tools coming into the market and they express their intentions of using those tools for future projects (*"...However, I'm interested in Espresso testing tool. It can write clean test code, and runs faster than Robotium. I'll try to use it if I make a next new app."*). Furthermore, some developers who are not satisfied with some tools, plan to use new tools which provide similar functionalities (*"Robotium has been giving us a little bit of trouble by having tests flake, so I'm going to work on migrating those to espresso in the near future, as I've heard nothing but good things about it so far."*).

Several developers prefer to test applications manually because their applications are small. Developers do not find it useful to put in effort and learn something new, when the app can be tested manually in a short amount of time. One of the developers said *"because i only develop some small app. therefore, i don't need any test tool. i just write code, run, debug until it's run correct."*, while another developer mentioned that *"Most of the projects I've worked so far are simple and for short-term. So I was just fine with manual testing."*

Google Play makes it easier for users to search and install apps. Therefore, some developers do not perform much testing, rather they depend on users who download their applications to report problems. One of the respondents mentioned "*... if someone comes across a bug, they can submit on the issue tracker and I will try to fix it."*.

Our survey results also show that a large number of developers prefer testing their apps manually rather than using any automated testing tool. From our analysis, we find that such cases occur due to various reasons. The app to be tested could be simple or it could be difficult to find a tool which can meet the testing requirements of the developers. One developer stated *"I have used robotium for some UI testing,*

however I haven't found it particularly useful. The things that robotium can test are very easily verified manually and there are a lot of things it can't test AFAIK (layouting, aesthetics, etc)".

Challenges faced by Android Developers while Testing

This section discusses the challenges faced by the Android developers while testing their apps either manually or when using automated testing tools. Table 5.3 shows some of the common challenges confronted by the developers. Some developers that we survey do not mention any challenges and some mention more than one challenges. We describe each of the challenges in detail and quote responses from the developers.

Table 5.3: Challenges Faced by Developers while Testing

Challenges	**Number of Respondents**
Time constraints	20
Compatibility issues	16
Lack of exposure	11
Tool is cumbersome	9
Emphasis on development	6
Lack of organization support	5
Unclear benefits	4
Poor documentation	4
Lack of experience	4
Steep learning curve	2

Time is one of the biggest factors which hinders testing. Most of the developers want to release their applications as soon as possible before someone else develops a similar application. In such cases, developers do not want to invest time in testing but rather develop the application quickly. One developer commented *"...I work as a freelance developer. So often there are time constraints to finish the project. Designing and implementing test cases takes some extra time, which makes it difficult to finish the project in time."*.

Automated testing tools are generic in nature and are developed to suit many applications. However, several apps contain custom functionalities which make it

difficult for developers to use automated testing tools. A number of developers were of the opinion that some parts of the code are hard to test using automated testing tools, which forces them to resort to manual testing. Also, automated testing tools are designed to work on specific technology. When developers use different technology, these automated testing tools no longer work well. One developer lamented *"I tried robolectric, but ran into several issues, that were probably also related to the fact that I am using Scala on Android."*

Some developers are not aware of automated testing tools available in the market. One developer admitted *"I have not been aware of them."* Furthermore, lack of discussion about the importance of automated tests worsens the problem. One developer commented *"... but it's not a common thing to 'do' so there isn't a lot of discussion around it."*

Usability of a tool is one of the key characteristics of it being used by a large number of developers. A tool which is easier to use will appeal to more developers as compared to a tool whose usage is complex. Several developers responded that they tried to use a particular tool but due to its cumbersome usage, they discarded the tool. One respondent mentioned *"I think Monkey runner is kinda cumbersome, and breaks easily when changing layout options."*. Yet another commented *"There is some coordination problems with Robotium which can be painful to workaround sometimes"*.

Functionalities of an application are one of the key factors which decide whether the app is useful or not. If an app provides functionalities which suits the need of a large number of users, the app will be popular. For example, one of the apps in our dataset, i.e., Open Explorer[13], has between 100,000 - 500,000 installs. Therefore, developers are often more focussed on adding new features of an app. Thus, they devote most of their time towards development rather than testing. One developer commented "*... I spend most of the time I dedicate to this project to implementing features"*.

[13] https://play.google.com/store/apps/details?id=org.brandroid.openmanager

With the increasing size of an application, it becomes imperative to adequately test the application. However, larger application means significant investment in terms of cost involved in testing it, which can act as a hindrance for many developers and organizations. If organizations are not able to provide resources to the developers, it would be difficult for developers to do much testing or invest time to learn automated testing tools. One developer commented *"The advice I was given was ... not bother with trying to use the Android testing tools/frameworks"*. In several cases, clients are not willing to pay extra for doing automated testing. One developer commented "*...few clients were ready to pay more for automatic tests : they did manual tests themselves. We never used automatic tests for this reason.*"

Testing tools can play an important role during software development life cycle as they assist developers in writing and running test scripts and creating test results automatically as compared to manually testing the application. However, it is important to clarify how the tools would be beneficial to a developer or organization who wants to use it. Unclear benefits would resist developers from venturing into the arena of automated testing. One developer stated *"The pain points for me would be assessing what automated test tools are available, assessing their applicability to my applications and writing comprehensive test scripts or whatever the tools require. That is probably more effort than what went into writing the applications in the first place."*

Learning new tools and techniques requires developers to read documentation and try out examples before they can apply the tool to their app. A good documentation makes it easier for novice as well as experienced developers to grasp the functionality of a tool and get started quickly on using the tool to test their apps, whereas a poor documentation will act as a hindrance for developers to adopt the tool. Therefore, a good and easy to understand documentation is a must for a new tool. Four developers in our survey mentioned that lack of documentation is one of the challenges. One of them mentioned *"Testing is documented there, but not very well and there should be far more information (for instance, how to test interaction*

with data providers - there's only a chapter how to test OWN data providers, but that's not what we need)."

Developers who have prior experience of using automated testing tools are more likely to use new tools. Our survey responses show that developers with no experience of using automated testing tools are reluctant to use Android test automation tools. One developer mentioned *"For that, I haven't used any kind of tool for testing pourposes. The reason? Well, for starters, I have no experience with testing tools for any language/platform, so I don't really know how to tackle that..."*.

Some of the developers perceive that using testing tools involve steep learning curve. One developer mentioned *"I fear it would represent a strong learning curve."*. Another developer commented that *"I know what automated testing is how to write a test case or prepare a test suite. But I don't know how can I use automated testing effectively. Learning this will take considerable amount of time and effort."*

5.3.3 RQ3: Survey of Microsoft Developers

Types of Testing

114 out of 127 developers use manual testing whereas 68 developers use automated testing tools. Some developers use both manual and automated testing. 4 developers responded that they do no test their apps. Figure 5.6 shows number of developers who perform different types of testing while developing apps. Most of the developers i.e., 103 in our survey perform functional testing. 97 developers perform unit testing, 75 perform integration testing, 74 perform performance testing, 63 perform regression testing, 47 perform system testing, 45 perform acceptance and load testing and 43 perform beta testing.

Automated Testing Tools Usage

Table 5.4 shows some of tools used by app developers in Microsoft. The results show that developers prefer using in-house tools. We also analyze why developers

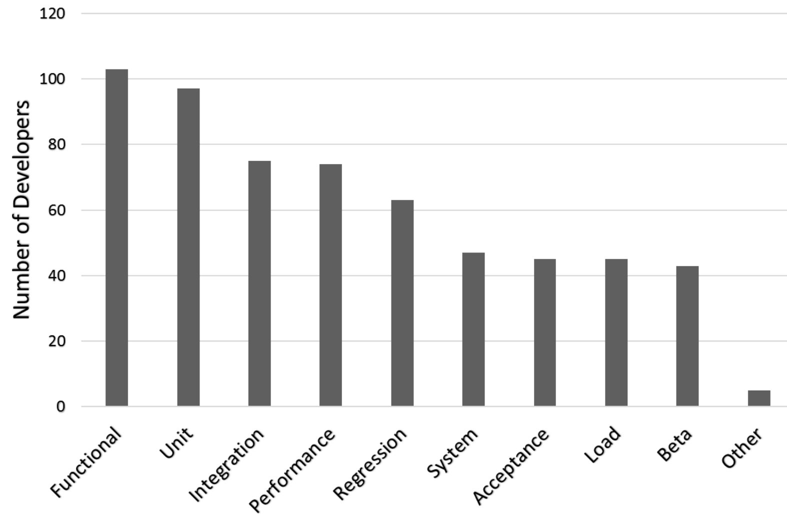


Figure 5.6: Types of Testing

use automated testing tools. Figure 5.7 shows why developers use automated testing tools and the corresponding number of developers for each type of usage. 64 developers use tools for executing test cases, 48 use them for finding potential bugs, 43 use them for analysing code coverage, 37 each use them creating & evaluating test execution results and for performing load testing, 33 each use them for generating test cases and managing test suites, whereas 27 developers use tools for reporting bugs.

Table 5.4: Automated Testing Tools Usage

Tools	Number of Respondents
Visual Studio	35
Internal Tool	8
Selenium	7
Microsoft Test Manager	5
Others (QUnit, Robotium etc.)	27

Challenges faced by App Developers

In this section, we discuss the challenges faced by the app developers at Microsoft while testing their apps either manually or using automated testing tools. Figure 5.8 shows the challenges encountered by developers along with their perceived severity levels. We can observe that 35 developers consider time constraints as a very seri-

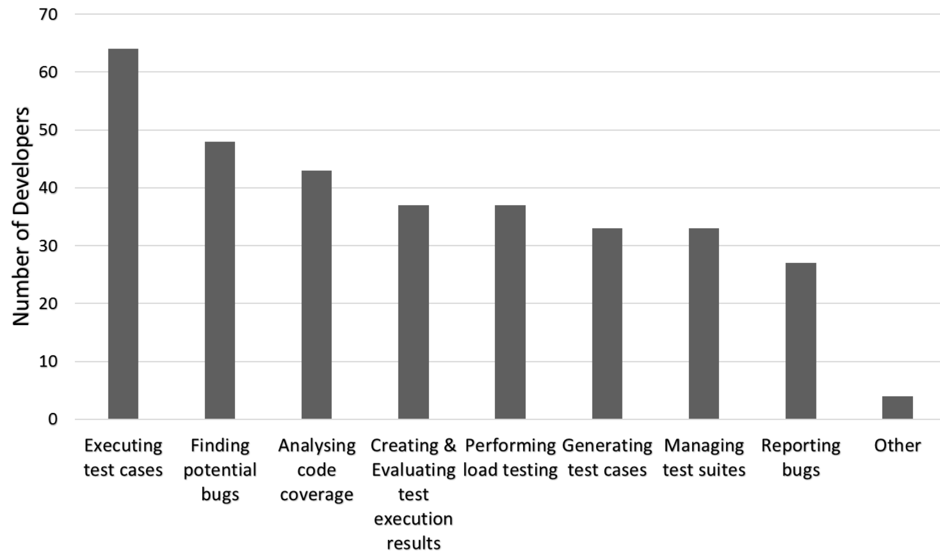


Figure 5.7: Usage of Automated Testing Tools

ous challenge and 56 consider it as a serious challenge. Poor documentation is the next big challenge which was mentioned by 19 developers as very serious and by 32 developers as serious. Lack of exposure, emphasis on development and compatibility issues were mentioned by several developers as a very serious challenge among others.

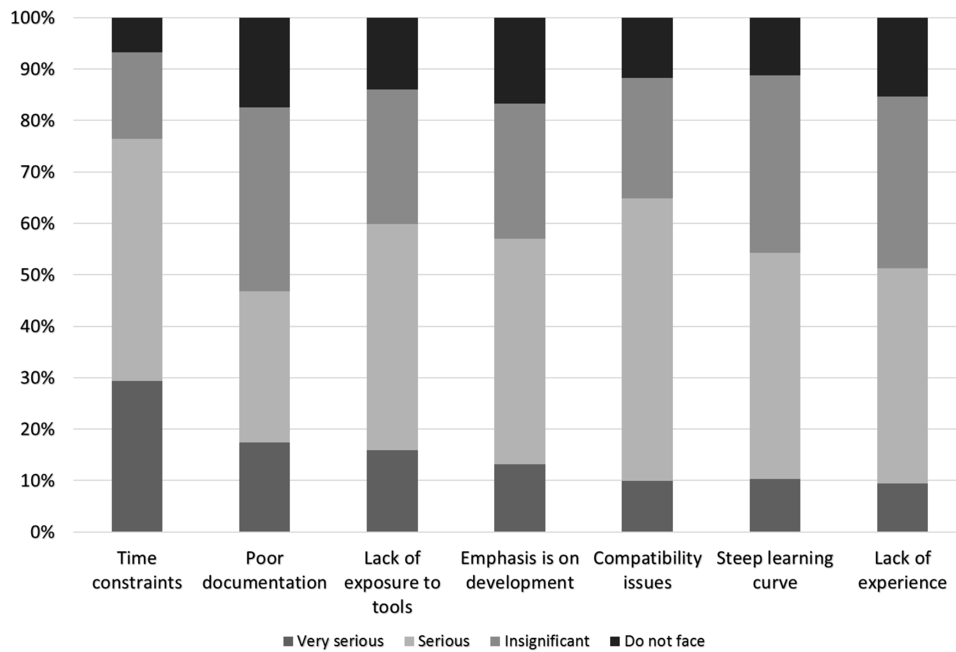


Figure 5.8: Challenges Faced by Developers

Developer Needs

In this section, we discuss the needs of the developers from the automated testing tools they use. We ask developers for two additional things they would like to see in the automated testing tools.

Poor documentation is one of the barriers for learning a new tool. Several developers expressed that a good documentation will increase their likelihood of using the tool. One of the developers commented *"Proper documentation so that a person new to the system can easily ramp up using these documents or articles."*

Developers often struggle to meet the deadlines due to the amount of the work they are assigned and the corresponding amount of time allotted for completion. To worsen the problem, developers are unaware of the testing tools which would be helpful for them. Examples of testing from successful projects would go a long way in motivating developers to use these tools. One of the developers mentioned *"We should have more internal material on proven practices about how to do testing, which Tools to use and many many samples and how-to Videos would be great. There is a lot of stuff about .NET code testing but not much about XAML App testing (at least not enough Deep digging Content)".*

Although there are lot of testing tools available, developers have to put in significant effort in activities such as generating and executing test cases. An automated testing tool which accepts the requirements and perform testing would do wonders for the developers. A developer mentioned *"Test case generation on most of the testing tools I came across needs to be generated by manually. this needs to be reduced with tools automation."*, while another one commented *"There should a tool which should accept the requirement from Dev. and should be able to develop the test suite to run test cases. It reduces lot of testing efforts."*

In general, developers expect tools which are easy to use. One of developers opined *"I would love to see testing tools that are simple to learn and straightforward to use. Most tools are cumbersome, lacking documentation and support is poor.*

Most of the existing UI Framework testing Tools for XAML feel incomplete."

5.4 Threats to Validity

Threats to External Validity. Threats to external validity relate to the generalizability of our results. We have investigated over 600 Android apps from F-Droid, which is one of the largest repositories of open-source Android apps. Our dataset consists of many kinds of apps from small ones to large and popular ones that contain more than one million lines of code or downloads. Still, it is unclear if our findings would generalize to all Android applications. In the future, we plan to reduce this threat further by analyzing more Android apps. Our respondents might not be representative of the entire population of app developers and thus our results might not generalize to all app developers. We have tried to reduce this threat to validity by surveying more than 200 developers of Android and Windows, which are the two most popular mobile app platforms. To the best of our knowledge, our study is the largest study on app developers to date.

Threats to Internal Validity. Threats to internal validity relate to the conditions under which experiments are performed. We automatically identify apps which contain test cases by using the following heuristics: we treat *.java* files whose names contain the word "test" as test files. We might miss some test files or mistakenly consider a file to be a test file when it is not. Furthermore, we manually analyse 89 apps which contain test files and calculate the coverage of test cases contained in these files. Out of the 89 apps, many failed to compile mainly due to missing dependencies. We tried our best to resolve all the dependencies by finding and downloading needed external libraries. However, we still cannot resolve many of them. We only compute coverage for 41 apps that we can successfully compile. Furthermore, some developers mention that they prefer manual testing. In such cases, test cases for these projects might not be available. To calculate the number

of developers, we use information from git logs. There may be cases where the same developer uses different e-mail addresses to commit to the same git repository and we may have wrongly counted the number of developers.

5.5 Conclusion

The following is a summary of our findings:

1. Only around 14% of the apps contain test cases and only around 9% of the apps that have executable test cases have coverage above 40%.

2. Android app developers prefer using standard framework such as JUnit, but they also use Android specific testing tools such as Monkeyrunner, Robotium and Robolectric. However, many Android developers prefer to test their applications manually without the help of any testing framework or tools. Most Windows app developers make use of Visual Studio, Coded UI, Selenium, and Microsoft Test Manager to test their apps.

3. Android and Windows app developers face numerous challenges in testing their apps and in using automated testing tools. These challenges include time constraints, compatibility issues, lack of exposure, cumbersome tools, emphasis on development, lack of organization support, unclear benefits, poor documentation, lack of experience, and steep learning curve.

Chapter 6

Bug Localization: Researchers' Bias

6.1 Introduction

Issue tracking systems, which contains information related to issues faced during the development as well as after the release of a software project, is an integral part of software development activity. Issue tracking systems such as JIRA or Bugzilla can help reporters report various kinds of issues such as bug reports, documentation update, refactoring request, addition of new feature and so on. Well-known projects often receive large number of issue reports which might be difficult for developers to handle. A mozilla developer accepted that the project receives over 300 bugs per day which needs triaging [7]. Therefore, it is important to have techniques which can help developers find buggy files quickly, which can help them resolve the bug faster.

To overcome the above issue, researchers have proposed techniques which use information given in the bug report to identify source code files that contain the bug [103, 110, 146]. These techniques often use standard information retrieval (IR) techniques to compute the similarity between the textual description of bug report and textual description of source code. Based on the similarity scores, these IR-based bug localization techniques return a ranked list of source code files which are likely to be buggy for that bug report. These techniques are evaluated using closed

and fixed issue reports marked as bugs collected from issue tracking systems. The evaluation involves comparison of files returned by bug localization techniques with the actual files changed to fix the bug. Past studies indicate that the performance of these techniques are promising – up to 80% of bug reports can be localized by just inspecting 5 source code files [110].

6.2 Biases in Bug Localization

Despite the promising results of IR-based bug localization approaches, a number of potential biases can affect the validity of results reported in prior studies. If these biases significantly affect the results of bug localization studies, researchers need to put more care in *cleaning* evaluation datasets when evaluating the performances of their techniques. In this work, we focus on investigating three potential biases:

1. **Wrongly Classified Reports.** Herzig et al. reported that many issue reports in issue tracking systems are wrongly classified [44]. About one third of all issue reports marked as bugs are not really bugs. Herzig et al. have shown that this potential bias significantly affects bug prediction studies that predict whether a file is potentially buggy or not based on the history of prior bugs. This potential bias might affect bug localization studies too as the characteristics of bug reports and other issues, e.g., refactoring requests, can be very different. Refactoring can touch a large number of files, while bug fixes are often more localized [77, 128]. Thus, there is a need to investigate whether wrongly classified reports significantly skew effectiveness results of bug localization approaches.

2. **Already Localized Reports.** Our manual investigation of a number of bug reports find that the textual descriptions of many reports have already specified the files that contain the bug. These *localized reports* do not require bug localization approaches. The buggy files are already localized and only need

to be fixed. Evaluating bug localization approaches will these localized reports will unfairly inflate the effectiveness results. Thus, there is a need to investigate how common are localized reports and whether their presence in the evaluation data set significantly skew effectiveness results of bug localization approaches.

3. **Incorrect Ground Truth Files.** Kawrykow and Robillard reported that many changes made to source code files are non-essential changes [55]. These non-essential changes include cosmetic changes made to source code which do not affect the behavior of systems. Past bug localization studies often use as ground truth source code files that are touched by commits that fix the bugs [110, 146]. However, no manual investigation was done to check if these files are affected by essential or non-essential changes. Files that are affected by non-essential changes should be excluded from the ground truth files as they do not contain the bug. Including these non-essential changes as ground truth files can unfairly inflate the effectiveness results – with more ground truth files, there is a higher chance that one of them will be identified by a bug localization tool. Thus, there is a need to investigate how common are the incorrect ground truth files and whether their presence in the evaluation data set significantly skew effectiveness results of bug localization approaches.

6.2.1 Bias 1: Wrongly Classified Reports

Methodology:

Step 1: Data Acquisition. We use Herzig et al's dataset where they manually analyzed issue reports (www.st.cs.uni-saarland.de/softevo/bugclassify). We download the issue reports from JIRA repositories and extract the textual contents of the summary and description part of the reports. After downloading, we perform the preprocessing steps described previously. In JIRA, each issue report has a unique identifier represented by project name and unique number. For example, *HTTPCLIENT-974*

represents issue number 974 of project *HTTPClient*. We use the *git* version control system of the projects to get the commit log files, which are used to map issue reports to their corresponding commits. Commit logs contain unique identifier of the issue report as part of the commit message. We use these mapped commits to checkout the source code files prior to the commits that address the issue and the source code files when the issue is resolved. For each source code files, we perform a similar preprocessing step to represent a file as a bag-of-words.

Step 2: Bug Localization. After the data acquisition, we have the textual content of the issue reports, the textual content of each source code file in the revision prior to the fix, and a set of ground truth files that are changed to fix the issue report. We give the textual content of the issue reports and the revision's source code files as input to the bug localization technique, which outputs a ranked list of files sorted based on the similarity to the bug report.

Step 3: Effectiveness Measurement & Statistical Analysis. After Step 2, we have for each issue report, a ranked list of source code files and a list of ground truth files. We compare these two lists to compute the average precision score.

We divide the issue reports into two categories: issue reports marked as bugs in the tracking system (Reported) and issue reports that are actual bugs i.e., manually labeled by Herzig et al. (Actual). In Herzig et al.'s dataset, the set Actual is a subset of Reported. We compute the MAP scores and use Mann-Whitney U test to examine the difference between these two categories at 0.05 significance level. We use Cohen's d to measure the effect size, which is the standardised difference between two means. To interpret the effect size, we use the interpretation given by Cohen in his book [23], i.e., $d < 0.2$ means trivial, $0.20 \leq d < 0.5$ means small, $0.5 \leq d < 0.8$ means medium, $0.80 \leq d < 1.3$ means large, and $d \geq 1.3$ means very large.

Results:

Table 6.1 shows the MAP scores for the two categories: reports marked as bugs

(Reported) and manually classified bug reports (Actual). We observe that there are differences of -2.33%, 12.25% and 6.98% in the MAP scores for HTTPClient, Jackrabbit and Lucene-Java, respectively. We perform Mann-Whitney Wilcoxon test and compute Cohen's d to examine the differences between the two categories. The results are also presented in Table 6.1. From the results, we observe that, for HTTPClient and Lucene-Java, the differences are statistically insignificant and the effect sizes are trivial (i.e., less than 0.2). For Jackrabbit, the effect size is trivial, however, the difference is statistically significant.

Table 6.1: Mean Average Precision (MAP) Scores for Reported and Actual

Project	**Reported**	**Actual**	**Difference**	**d**
HTTPClient	0.429	0.419	-2.33%	0.13
Jackrabbit	0.302	0.339	12.25%	0.06
Lucene-Java	0.301	0.322	6.98%	0.04

Effect of Different Misclassification Types. In this section, we analyse the misclassification type which has the most impact on the difference of MAP scores between Reported and Actual. Herzig et al. classify issue reports into 13 categories: `BUG`, `RFE`, `IMPROVEMENT`, `DOCUMENTATION`, `REFACTORING`, `BACKPORT`, `CLEANUP`, `SPEC`, `TASK`, `TEST`, `BUILD_SYSTEM`, `DESIGN_DEFECT`, and `OTHERS`. We omit issue reports that are misclassified one category at a time and recalculate the MAP score. For example, `RFE` to `BUG` represents issue reports which are RFE (Actual) but are misclassified as BUG (Reported). Table 6.2 shows the MAP scores when we remove issue reports of particular misclassification types one at a time. Each row corresponds to a subset of reports where reports of a misclassification type is removed. We observe that `TEST` to `BUG` has the largest difference in the MAP score followed by misclassification from `IMPROVEMENT` to `BUG`.

Table 6.2: Mean Average Precision (MAP) Scores when Issue Reports of a Particular Misclassification Type are Omitted. Omit. = Omitted, Misclass. = Misclassification, HC = HTTPClient, JB = Jackrabbit, LJ = Lucene-Java. The last column is the MAP of all three projects.

Omit. Misclass. Type (Actual to Reported)	**HC**	**JB**	**LJ**	**Overall**
None	0.429	0.302	0.301	0.312
RFE to BUG	0.427	0.303	0.304	0.313
DOCUMENTATION to BUG	0.43	0.304	0.305	0.315
IMPROVEMENT to BUG	0.416	0.299	0.295	0.307
REFACTORING to BUG	0.428	0.301	0.301	0.311
BACKPORT to BUG	0.43	0.303	0.300	0.313
CLEANUP to BUG	0.429	0.303	0.303	0.314
SPEC to BUG	0.435	0.302	0.303	0.312
TASK to BUG	0.432	0.302	0.301	0.312
TEST to BUG	0.429	0.328	0.313	0.334
BUILD_SYSTEM to BUG	0.429	0.306	0.303	0.315
DESIGN_DEFECT to BUG	0.424	0.301	0.301	0.311
OTHERS to BUG	0.439	0.303	0.301	0.313

6.2.2 Bias 2: Already Localized Reports

Methodology: We first need to identify localized bug reports. We start by manual investigating of a smaller subset of bug reports and identify localized ones. We then developed an automated means to find localized bug reports so that our analysis can scale to a larger number of bug reports. Finally, we input these reports to a number of IR-based bug localization tools to investigate whether localized reports skew the results of bug localization tools.

Table 6.3: Fully Localized, Partially Localized, and Not Localized Reports

Category	**Description**
Fully	Bug reports where all the files containing the bugs are explicitly specified in the report.
Partially	Bug reports where some of the files containing the bugs are explicitly mentioned in the report.
Not	Bug reports which do not explicitly specify any of the buggy files.

Step 1: Manually Identifying Localized Bug Reports. We manually analysed 350 issue reports that Herzig et al. labeled as bug reports. Out of the 5,591 issue reports from the three projects, Herzig et al. labeled 1,191 of them as bug reports. We

randomly selected these 350 from the pool of bug reports from the three software projects. For our manual analysis, we read the summary and description fields of each bug report. We also collected the corresponding files changed to fix each bug. We classified each bug report into one the three categories shown in Table 6.3. Table 6.4, 6.5, and 6.6 show example bug reports that are fully localized, partially localized, and not localized.

Table 6.4: Fully Localized Report: HTTPCLIENT-1078

Summary:	**DecompressingEntity** not calling close on InputStream retrieved by getContent
Description:	The method **DecompressingEntity**.writeTo(OutputStream outstream) does not close the InputStream retrieved by getContent(). According to the documentation of HttpEntity.writeTo: IMPORTANT: Please note all entity implementations must ensure that all allocated resources are properly deallocated when this method returns. -> imho this is not satisfied in **DecompressingEntity**.writeTo
Buggy Files:	**DecompressingEntity.java**

Table 6.5: Partially Localized Report: JCR-814

Summary:	Oracle bundle PM fails checking schema if 2 users use the same database
Description:	When using the OracleBundlePersistenceManager there is an issue when two users use the same database for persistence. In that case, the checkSchema() method of the **BundleDbPersistenceManager** does not work like it should. More precisely, the call ”metaData.getTables(null, null, tableName, null);” will also includes table names of other schemas/users. Effectively, only the first user of a database is able to create the schema. probably same issue as here: JCR-582
Buggy Files:	**BundleDbPersistenceManager.java**, **OraclePersistenceManager.java**

Step 2: Automatic Identification of Localized Reports. In this step, we build an algorithm that takes in a set of files that are changed in bug fixing commits, a bug report, and outputs one of the three categories described in Table 6.3. Our algorithm first extracts the text that appear in the summary and description fields of bug reports. Next, it tokenizes this text into a set of word tokens. Finally, it checks whether the

Table 6.6: Not Localized Report: LUCENE-3721

Summary:	CharFilters not being invoked in Solr
Description:	On Solr trunk, all CharFilters have been non-functional since LUCENE-3396 was committed in r1175297 on 25 Sept 2011, until Yonik's fix today in r1235810; Solr 3.x was not affected - CharFilters have been working there all along.
Buggy Files:	**TokenizerChain.java**

name of each buggy file (ignoring its filename extension) appears as a word token in the set. If all names appear in the set, our algorithm categorizes the report as *fully localized*. If only some of the names appears in the set, it categorizes the bug report as *partially localized*. Otherwise, it categorizes the bug report as *not localized*. We have evaluated our algorithm on the 350 manually labeled bug reports and find that its accuracy is close to 100%.

Step 3: Application of IR-Based Bug Localization Techniques. After localized, partially localized, and not localized reports are identified, we create three groups of bug reports. We feed each of them into the VSM-based bug localization tool. We then evaluate the effectiveness of these tools for each of the three groups of reports.

Step 4: Statistical Analysis. We perform two statistical analyses. First, we compare the average precision scores achieved by VSM-based bug localization tool for the set of fully localized, partially localized, and not localized reports using Mann-Whitney-Wilcoxon test at 5% significance level. We also compute Cohen's d on the average precision scores to see if the effect size is small, medium or large.

Second, we compare a subset of bug reports where the VSM-based bug localization technique performs *the best* and another subset where the VSM-based bug localization techniques performs *the worst*. We then compare the distribution of fully, partially, and not localized bugs in these two subsets. We employ Fisher exact test [29] to see if the distribution for the first subset significantly differs with the distribution for the second subset.

Results:

Number of Fully Localized, Partially Localized, and Not Localized Reports.

The numbers of bug reports that are identified as fully, partially, and not localized are shown in Table 6.7. We can observe that out of 1,191 bug reports, 398 (33.41%) bug reports are fully localized i.e., the bug reports contains the name of all the class files changed to fix the bug. Over 50% of the bug reports are either fully or partially localized. This shows that a significant number of bug reports are already localized, and do not benefit from a bug localization algorithm. On the other hand, 546 bug reports (45.84%) are not localized at all.

Table 6.7: Fully, Partially, and Not Localized Reports

Project	Category	Number	Proportion
HTTPClient	Fully	36	3.02%
	Partially	28	2.35%
	Not	35	2.93%
Jackrabbit	Fully	299	25.10%
	Partially	132	11.08%
	Not	402	33.75%
Lucene-Java	Fully	63	5.28%
	Partially	87	7.30%
	Not	109	9.15%

Average Precision Scores of Fully vs. Partially vs. Not Localized Reports. Table 6.8 shows the Mean Average Precision (MAP) of the VSM-based bug localization technique when applied to the set of fully, partially, and not-localized reports. We can note that the MAP score differences between fully localized and not localized bug reports for HTTPClient, Jackrabbit, and Lucene-Java are 84.39%, 99.86% and 91.16% respectively. Also, the MAP score differences between partially localized and not localized bug reports for HTTPClient, Jackrabbit, and Lucene-Java are 33.05%, 66.42% and 52.71% respectively.

We also perform Mann-Whitney Wilcoxon test to examine the difference between the following categories: fully & partially, partially & not and fully & not. Table 6.9 shows the p-values between different categories. The results show that

Table 6.8: MAP Scores: Fully vs. Partially vs. Not

Project	Fully	Partially	Not
HTTPClient	0.615	0.349	0.250
Jackrabbit	0.560	0.373	0.187
Lucene-Java	0.527	0.338	0.197

Table 6.9: Comparison: Fully vs. Partially vs. Not

Project	Fully-Partially			Partially-Not			Fully-Not		
	p-value	d	Effect Size	p-value	d	Effect Size	p-value	d	Effect Size
HTTPClient	0.007	**0.94**	Large	0.007	0.53	Medium	$3.094e^{-05}$	**1.27**	Large
Jackrabbit	$4.544e^{-05}$	0.56	Medium	$< 2.2e^{-16}$	0.55	Medium	$< 2.2e^{-16}$	**1.14**	Large
Lucene-Java	0.010	0.53	Medium	$1.851e^{-05}$	0.41	Small	$3.183e^{-09}$	**1.04**	Large

there are significant differences between average precision scores of fully localized and partially localized bug reports, fully localized and partially localized bug reports, and partially localized and not localized bug reports, i.e., all the p-values are less than 0.05. We also compute Cohen's d to measure an effect size and find that the effect sizes are small to large. The effect sizes between average precision scores of fully localized and not localized bug reports are large for all three projects. This shows that there is a large substantial difference in the effectiveness of a bug localization tool when applied to bug reports which are fully localized and those which are not localized.

Best vs. Worst Bug Reports. We want to examine the difference between the proportion of bug reports that are fully, partially, and not localized in the upper and lower quartile of the bug reports based on the ability of the VSM-based bug localization tool to localize them. We simply sort the bug reports based on their average precision scores and identify the subset that appear in the top 25% of the list (upper quartile) and another subset that appear in the bottom 25% of the list (lower quartile). For Jackrabbit and Lucene-Java, we randomly select 50 bug reports from the upper quartile and another 50 from the lower quartile. For HTTPClient, we randomly select 25 bug reports from the upper quartile and another 25 from the lower quartile – since in our dataset, HTTPClient has less than 100 bug reports.

Table 6.10 shows the number of fully, partially and not localized bugs for each

Table 6.10: Fisher Exact Test: Best vs. Worst Reports

Project		Fully	Partially	Not	p-value
HTTPClient	Upper	16	5	4	0.0041
	Lower	6	4	15	
Jackrabbit	Upper	35	9	6	$2.807e^{-13}$
	Lower	7	1	42	
Lucene-Java	Upper	22	18	10	$8.724e^{-05}$
	Lower	5	18	27	

of the projects. We use Fisher exact test to examine the difference between the distribution of fully localized, partially localized, and not localized bug reports in the upper and lower quartiles. The null hypothesis is that there is no difference between the distribution of fully, partially, and not localized bug reports in the upper and lower quartiles. The alternate hypothesis is that there is a significant difference between the distribution of bug reports in the upper and lower quartiles. We find that the p-values for all the projects are very small, which shows that there is a significance difference in the distribution of fully localized, partially localized, and not localized bug reports between the best and worst bug reports.

6.2.3 Bias 3: Incorrect Ground Truth Files

Methodology:

We randomly select 100 bug reports that are not (already) localized (i.e., these reports do not explicitly mention any of the buggy files) and investigate the files that are modified in the bug fixing commits. We manually perform a *diff* that gives us the differences between the modified file and the original file. Based on these differences we manually decide if a file contains a bug or not. Files that are only affected by cosmetic changes, refactorings, etc. are considered as non-buggy files. Based on this manual analysis, for each bug report we have the set of *clean* ground truth files and another set of *dirty* ground truth files.

Thung et al. have extended Kawrykow and Robillard work [55] to automatically identify real ground truth files [123]. However the accuracy of their proposed technique is still relatively low (i.e., precision and recall scores of 76.42% and 71.88%).

Hence, we do not employ any automated tool to identify wrong ground truth files. We also cannot extend the study to investigate a large number of bug reports since the identification of wrong ground truth files is time consuming.

Step 2: Application of IR-Based Bug Localization Techniques. After the set of clean and dirty ground truth files are identified for each of the 100 bug reports, we input the 100 bug reports to a VSM-based bug localization tool. We evaluate the results of the tool on dirty and clean ground truth files.

Step 3: Statistical Analysis. We compare the average precision scores achieved by the VSM-based bug localization tool for the 100 bug reports with clean and dirty ground truth files using Mann-Whitney-Wilcoxon test at 5% significance level. We also compute Cohen's d on the average precision scores to see if the effect size is small, medium or large.

We remove all the files which were changed when the bug was fixed but these files are not buggy. For example, all the files which are refactored due to some changes in other files or change in the comments section. We randomly select 100 bug reports and manually analyse them to examine all the files changed to solve a bug report. We remove files which are not-buggy i.e., which involve addition or deletion of import statement, change in comments etc. We found that out of 498 files changed to fix the above 100 bugs, 358 files are buggy. The other 140 files only involve cosmetic changes such as adding or deleting a variable, changing the datatype of variables etc. These files actually did not caused the bug but are changed because of the changes made to the buggy files.

Results:

Number of Wrong Ground Truth Files. We found that out of 498 files changed to fix the 100 bugs, only 358 files are really buggy. The other 140 files (28.11 %) do not contain any of the bugs but are changed because of refactorings, modifications to program comments, due to changes made to the buggy files, etc. Figure 6.1 shows the diff of a file that is changed in a commit that fix bug report LUCENE-2616. The

content of the bug report with ID LUCENE-2616 is shown in Table 6.11.

```
diff --git a/lucene/src/java/org/apache/lucene/index/SegmentInfo.java b/lucene/
index 830072e..4686481 100644
--- a/lucene/src/java/org/apache/lucene/index/SegmentInfo.java
+++ b/lucene/src/java/org/apache/lucene/index/SegmentInfo.java
 import org.apache.lucene.index.codecs.DefaultSegmentInfosWriter;
-
 import java.io.IOException;
-import java.util.regex.Pattern;
    if (delFileName != null && (delGen >= YES || dir.fileExists(delFileName)))
      fileSet.add(delFileName);
    }
-
+
    if (normGen != null) {
      for (int i = 0; i < normGen.length; i++) {
```

Figure 6.1: Example Diff of a File that is Changed to Fix a Bug in Lucene-Java Project with ID LUCENE-2616. Note: (1) The name of the file: SegmentInfo.java; (2) An empty line and an import statement are deleted; (3) An empty line is deleted and another one is added.

Table 6.11: Bug Report: LUCENE-2616

Summary:	FastVectorHighlighter: out of alignment when the first value is empty in multiValued field
Description:	-
Non-Buggy File:	SegmentInfo.java

MAP Scores: Dirty vs. Clean. We compare the Mean Average Precision (MAP) scores of these 100 bug reports when evaluated on dirty and clean ground truths. Table 6.12 shows that the differences in the MAP scores are between 0 to 0.036. We also ran Mann-Whitney Wilcoxon test and compute Cohen's d to check if each difference is significant or substantial. We find that the difference is not statistically significant and the effect size is trivial (< 0.2).

Table 6.12: MAP Scores: Dirty vs. Clean Ground Truths

Project	Dirty	Clean	Difference	d
HTTPClient	0.207	0.171	0.036	0.08
Jackrabbit	0.115	0.115	0.000	0.08
Lucene-Java	0.271	0.239	0.032	0.17

6.3 Other Evaluation Metrics

Beside Mean Average Precision (MAP) which we used in the previous sections, HIT@N and MRR have also been used to evaluate bug localization studies [103, 110, 146]. HIT@N and MRR are presented below:

- **HIT@N:** This metric counts the percentage of bug reports with at least one buggy file found in the top N (e.g., 1) ranked results.
- **MRR (Mean Reciprocal Rank):** The reciprocal rank of a bug report is the inverse of the rank of the first buggy file in the ranked results. The mean reciprocal rank takes the average of the reciprocal ranks of all bug reports. For a set of bug reports Q, MRR is defined as:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{Q} \frac{1}{rank_i} \tag{6.1}$$

where $rank_i$ is the rank of the first buggy file in the output ranked list.

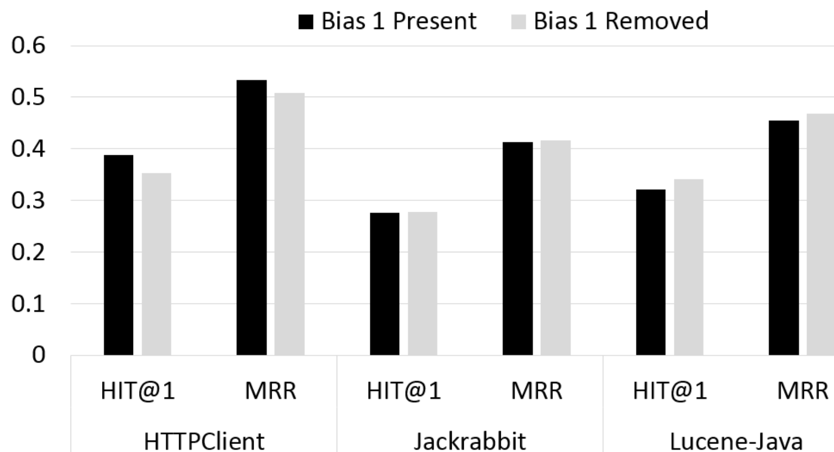


Figure 6.2: Before and After Removing Bias 1

The effect of bias 1, bias 2, and bias 3 measured by HIT@1 and MRR are shown in Figures 6.2 to 6.4. Figure 6.2 shows that for bias 1, its effect in terms of HIT@1 and MRR scores is minimal. Figure 6.3 shows that for bias 2, its effect in terms of HIT@1 and MRR score is substantial. Figure 6.4 shows that for bias 3,

for Jackrabbit, its effect is minimal. For HTTPClient and Lucene-Java, its effect is more apparent albeit not as substantial as the effect of bias 2.

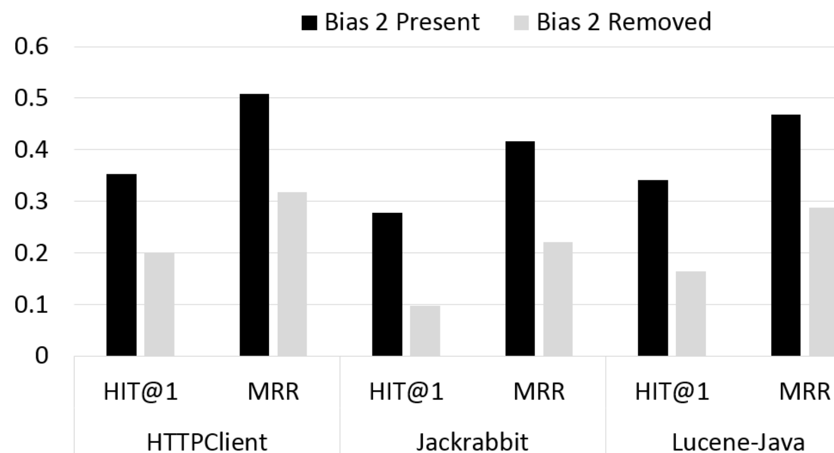


Figure 6.3: Before and After Removing Bias 2

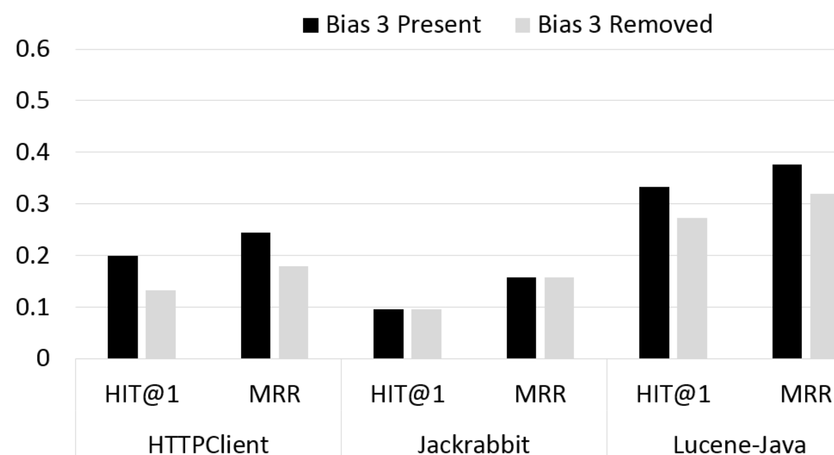


Figure 6.4: Before and After Removing Bias 3

For MRR since it is a mean of a distribution, we also run Mann-Whitney-Wilcoxon test and compute Cohen's d values. The results are shown in Table 6.13. We find that for bias 1, its effect is not statistically significant for all projects. For bias 2, its effect is both statistically significant and substantial when comparing the results of (fully or partially) localized bug reports with results of not localized bug reports. For bias 3, its effect is not statistically significant for all projects.

To conclude, the above results show that bias 2 has substantial effect on the performance of bug localization techniques. The effects of bias 1 and 3 are more minor

or even negligible. These results are in line with the findings of Sections 6.2.1, 6.2.2, and 6.2.3.

Table 6.13: Results of Mann-Whitney-Wilcoxon Test and Cohen'd Computation for MRR. (F-P) = Fully Localized vs. Partially Localized. (P-N) = Partially Localized vs. Not Localized. (F-N) = Fully Localized vs. Not Localized.

Bias Type	Project		p-value	d
Bias 1	HTTPClient		0.6667	0.241
	Jackrabbit		0.7855	0.050
	Lucene-Java		0.7336	0.043
Bias 2	HTTPClient	(F-P)	0.5465	0.142
		(P-N)	**0.0008925**	0.364
		(F-N)	**0.0003381**	0.634
	Jackrabbit	(F-P)	0.075	0.128
		(P-N)	**<2.2e-16**	**1.421**
		(F-N)	**<2.2e-16**	**0.962**
	Lucene-Java	(F-P)	0.2024	0.097
		(P-N)	**8.201e-08**	**0.944**
		(F-N)	**3.805e-06**	0.775
Bias 3	HTTPClient		0.6464	0.163
	Jackrabbit		0.9404	0.088
	Lucene-Java		0.7449	0.137

6.4 Conclusion

In this study, I analyze the impact of these potential biases on bug localization results. This empirical study highlights the following results:

1. Wrongly classified issue reports do not statistically significantly impact bug localization results on two out of the three projects. They also do not substantially impact bug localization results on all three projects (effect size $<$ 0.2).

2. (Already) localized bug reports statistically significantly and substantially impact bug localization results (p-value $<$ 0.05 and effect size $>$ 0.8).

3. Existence of non-buggy files in the ground truth does not statistically significantly or substantially impact bug localization results (effect size $<$ 0.2).

These findings suggest that future bug localization researchers need to at least remove (already) localized bug reports from their evaluation dataset since they have significant and substantial impact on the performance of bug localization techniques.

Chapter 7

Bug Localization: Practitioners' Expectations

7.1 Introduction

Despite numerous studies on bug localization, unfortunately, few studies have investigated the expectations of practitioners on research in fault localization. It is unclear whether practitioners appreciate this line of research. Even if they do, it is unclear whether they would adopt fault localization techniques, what factors affect their decisions to adopt, and what are their minimum thresholds for adoption. Practitioners' perspective is important to help guide software engineering researchers to create solutions that matter to our "clients".

To gain insights into practitioners' expectations on bug localization, we surveyed thousands of practitioners from various companies spread across the globe and obtained 386 responses. To get these thousands of practitioners, we sent emails to our contacts in IT industry (Microsoft, Google, Cisco, LinkedIn, ABB, Box.com, Huawei, Infosys, Tata Consultancy Services and many other small to large IT companies in various countries) to disseminate our survey form to their colleagues. We also sent emails to practitioners contributing to open source projects hosted on GitHub. In our survey, we first collected demographic information from respon-

dents, e.g., whether they are professional software engineers, whether they have contributed to open source projects, their experience level, their job roles, their English proficiency level, and their country of residence. Next, we gave a brief overview of research in bug localization, and asked our respondents about their views of the importance of this research area. We allowed respondents to answer "I don't understand" to filter out those with insufficient background knowledge. Next, we investigated practitioners' willingness to adopt fault localization techniques, and their thresholds for adoption measured in terms of various factors: debugging data availability, granularity level, success criterion, success rate, scalability, efficiency, ability to provide rationale, and IDE integration.

After the survey, we performed a literature review. We went through papers published in ACM/IEEE International Conference on Software Engineering (ICSE), ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC-FSE), ACM International Symposium on Software Testing and Analysis (ISSTA), IEEE Transactions on Software Engineering (TSE), and ACM Transactions on Software Engineering Methodology (TOSEM) in the last 5 years and identified those that proposed bug localization techniques. We then compared the techniques proposed in the papers against the criteria that practitioners have for adoption.

We investigated the following research questions in our survey and literature review:

RQ1 *Do practitioners value research on fault localization?*

RQ2 *What debugging data are available to practitioners during their debugging sessions?*

RQ3 *Which granularity levels (e.g., components, classes, methods, basic blocks, statements) should a fault localization technique work on?*

RQ4 *When would a practitioner view a fault localization technique to be successful in localizing bugs?*

RQ5 *How trustworthy (reliable) must a fault localization technique be before a practitioner will consider its adoption?*

RQ6 *How scalable must a fault localization technique be before a practitioner will consider its adoption?*

RQ7 *How efficient must a fault localization technique be before a practitioner will consider its adoption?*

RQ8 *Will a practitioner adopt a trustworthy, scalable, and efficient fault localization technique?*

RQ9 *What additional criteria aside from trustworthiness, scalability, and efficiency, must a fault localization technique meet before some practitioners will consider its adoption?*

RQ10 *How close are the current state-of-research to satisfy practitioner needs and demands before adoption?*

We investigated *RQ1* to understand the general views of practitioners on research in fault localization. In *RQ2* to *RQ7*, we probed the practitioners to better understand their minimum thresholds for adopting a fault localization technique considering different factors. We considered *availability of debugging data* and *preferred granularity level* in *RQ2* and *RQ3*. Prior studies have considered a variety of data and focused on different granularity levels. Unfortunately, none has checked with practitioners whether they are available or preferred. In *RQ4* and *RQ5*, we considered *success criterion* and *success rate* (i.e., the proportion of time the success criterion is met) since they were measured in various ways to evaluate past fault localization techniques [130, 132, 49, 76]. We considered *scalability* in *RQ6* due to a recent shift in fault localization studies that analyze larger programs, beyond those

in Siemens suite [46]. We considered efficiency, i.e., the amount of time a technique takes to produce results, in *RQ7*, since it is often used as a criterion to evaluate program analysis tools (e.g., [111]). We considered *RQ8* to understand the willingness of practitioners to adopt a tool which satisfies a set of desirable properties. We investigated additional criteria aside from trustworthiness, scalability, and efficiency in *RQ9*. We considered *RQ10* to evaluate the extent current state-of-research matches practitioners' expectations.

7.2 Methodology

7.2.1 Practitioner Survey

Respondent Recruitment

Our goal is to get a sufficient number of practitioners from diverse backgrounds. We followed a multi-pronged strategy to get respondents:

- First, we contacted professionals from various countries and IT companies and asked their help to disseminate our survey within their organizations. We sent emails to our contacts in Microsoft, Google, Cisco, LinkedIn, ABB, Box.com, Huawei, Infosys, Tata Consultancy Services and many other small to large companies in various countries to fill up the survey and disseminate it to some of their colleagues. By following this strategy we can get respondents from diverse organizations.

- Second, we sent emails to 3,300 practitioners contributing to open source projects on GitHub, out of which around 150 were not delivered, and around 50 emails received automatic replies notifying the receiver's absence. By sending to GitHub developers we get respondents who are open source practitioners in addition to professionals working in industry.

We included practitioners working on open source and closed source projects,

those working in small as well as large organizations, and those from different nationalities across the globe. A similar methodology of collecting responses through contacts in industry has been used in previous studies, e.g., [131].

Survey Design

We collected the following pieces of information.

Demographics:

- *Professional software engineer*: Yes / No
- *Involvement in open source development*: Yes / No
- *Role*: Software development / Software testing / Project management / Other (Pick all that apply)
- *Experience in years* (decimal value)
- *English proficiency*: Very good / Good / Mediocre / Poor / Very poor (Pick one)
- *Current country of residence*

The demographic information is used to: 1) filter respondents who may not understand our survey (i.e., respondents with less relevant job roles, respondents with poor/very poor English proficiency), 2) break down results by groups (e.g., by roles, by experience levels, etc.).

Practitioners' Expectations:

Importance. We provided respondents a brief description of research in fault localization and asked them how they perceive the importance of such line of research. We described fault localization as an approach that generates a ranked list of suspicious program locations given debugging data (e.g., a crash or a program failure). We asked respondents to pick one of the following ratings: "Essential", "Worthwhile", "Unimportant", "Unwise", and "I don't understand". The ratings are the same as those used in prior studies by Begel and Zimmermann [11] and Lo et

al. [74]. We included the category "I don't understand" to filter respondents who do not understand our brief description. For respondents who selected "Unimportant" or "Unwise", we asked why they think research in fault localization is unimportant/unwise. They may or may not provide answers to this optional question.

Adoption. Next, we asked respondents factors that affect their likelihood to adopt a fault localization technique. We elicited the following pieces of information:

- *Availability of debugging data*: mathematical specification, textual specification, one failing test case, multiple failing test cases, passing test cases, textual description of a defect. (Options: all the time, sometimes, rarely, never)
- *Preferred granularity levels*: pinpoint buggy components, pinpoint buggy classes, pinpoint buggy methods, pinpoint buggy basic blocks, pinpoint buggy statements (Pick all that apply)
- *Minimum success criteria*: Top 1[1] / Top 5 / Top 10 / Top 50 / Other (Pick one)
- *Minimum success rate*: at least 5% / 20% / 50% / 75% / 90% / Other (Pick one)
- *Minimum scalability*: Programs of size 1-100 / 1-1,000 / 1-10,000 / 1-100,000 / 1-1,000,000 lines of code (LOC) / Other (Pick one)
- *Minimum efficiency*: Return result in less than 1 second / 1 minute / 30 minutes / 1 hour / 1 day / Other (Pick one)

We then asked respondents whether they will adopt a fault localization technique which is trustworthy (i.e., satisfies a minimum success rate), scalable, and efficient. If a respondent answered "No", we asked the respondent his/her reason to not adopt such a technique. The respondent may or may not answer this optional question.

Next, we asked respondents to indicate their level of agreement (disagreement) with the following statements:

[1] A buggy program element exists in the top 1 position of a ranked list returned by a fault localization technique.

- A fault localization technique must provide a rationale why some program locations are marked as suspicious. (Options: Strongly agree, Agree, Neutral, Disagree, Strongly disagree)
- I will *still adopt* an efficient, scalable, and trustworthy fault localization technique, even if it cannot provide rationales. (Options: Strongly agree, Agree, Neutral, Disagree, Strongly disagree)
- A fault localization technique must be integrated well to my favourite IDE. (Options: Strongly agree, Agree, Neutral, Disagree, Strongly disagree)
- I will *still adopt* a an efficient, scalable, and trustworthyfault localization technique, even if it is not integrated well to my favorite IDE. (Options: Strongly agree, Agree, Neutral, Disagree, Strongly disagree)

We considered the above statements to validate the observations that "more context [is] needed" for debugging and there is a need for a "complete ecosystem for debugging" [94]. If a respondent chose "Disagree" or "Strongly Disagree" for either the second or fourth statement above, we asked their reasons to disagree. A respondent may or may not answer these optional questions.

At the end of the survey, we allowed respondents to provide free-text comments, suggestions, and opinions about fault localization and our survey. A respondent may or may not provide any final comment.

To support respondents from China, we translated our survey to Chinese before distributing it to them. We chose to make our survey available in Chinese and English as the earlier is the most spoken language and the latter is an international lingua franca. A large number of our survey recipients are expected to be fluent in one of these two languages. Moreover, prior to sending our survey to a large number of potential respondents, we asked a few practitioners that we know to take a preliminary version of our survey and give comments. They found that overall the survey was easy to understand and gave some feedback to improve it further. We made some minor modifications to the survey based on their feedback. We

discarded responses that we received from these pilot respondents. The full text of this survey is publicly available [1].

Data Analysis

Based on our survey responses, we set out to answer the first 9 research questions described in Section 5.3.2. We plotted practitioners' responses as charts and used them to answer the research questions. Considering different factors (e.g., trustworthiness, scalability, etc.), we identified thresholds to achieve 50%, 75%, and 90% satisfaction rates (i.e., 50%, 75%, and 90% of respondents are happy with a fault localization technique if the thresholds are met). Moreover, we summarized respondents' reasons for their unwillingness to adopt and their final comments.

7.2.2 Literature Review

We went through full research papers published in ICSE, FSE, ESEC-FSE, ISSTA, TSE, and TOSEM from 2011 to 2015. We have a total of 417, 255, 169, 350, and 137 ICSE, FSE/ESEC-FSE, ISSTA, TSE, and TOSEM papers to consider, respectively. We selected papers from the above conferences and journals as they are premier publication venues in software engineering research community and state-of-the-art latest findings are published in these conferences and journals.

We read the titles and abstracts of these papers and judged whether each of the papers proposes a new fault localization technique that can help practitioners pinpoint the root cause of a failure. We included papers on spectrum-based fault localization (e.g., [130]), information-retrieval-based fault localization (e.g., [146]), and specialized fault localization techniques (e.g., [80]). We excluded papers on automatic repair (e.g., [57, 72]), empirical study on debugging (e.g., [101]), theoretical analysis of existing debugging techniques (e.g., [137]), failure reproduction (e.g., [49]), debugging comprehension (e.g., [57, 106]), failure clustering (e.g., [43]), bug prediction (e.g., [102]), and bug detection (e.g., [83]). Debugging

comprehension and failure clustering techniques do not produce a *ranked list* of potential buggy program locations. Bug prediction focuses on *future* bugs, while bug detection focuses on detecting *unknown* bugs that have not manifested as failures.

For each fault localization paper, we read its content and analyzed the capabilities of the proposed technique in terms of the following factors: debugging data required, granularity level, success rate, scalability, efficiency, ability to provide rationale, and IDE integration. We compared the capabilities of techniques proposed in the papers with practitioners' thresholds for adoption. To check for IDE integration, we also searched if the authors publish any tool papers based on the original papers. If they do, we checked the contents of the tool papers (and accompanying videos, if any) too. We then identified discrepancies between the current state-of-research and practitioners' needs and demands.

This study is a first cut in assessing the extent existing research studies match up to practitioners' expectations. In-depth assessments and comparisons of success rate, efficiency or scalability require a more comprehensive and head-to-head evaluation of the techniques over a representative bug collection, which we leave as future work.

7.3 Findings

7.3.1 Statistics of Responses Received

In total we received 403 responses. These responses were made by respondents from 33 countries across five continents – see Figure 7.1. The top two countries where the respondents reside are China and the United States.

We excluded 3 responses made by respondents who are neither professional software engineers nor open source developers, and whose job roles are neither software development, software testing, or project management. These respondents have the following roles: Linux operation and maintenance, business analyst, and

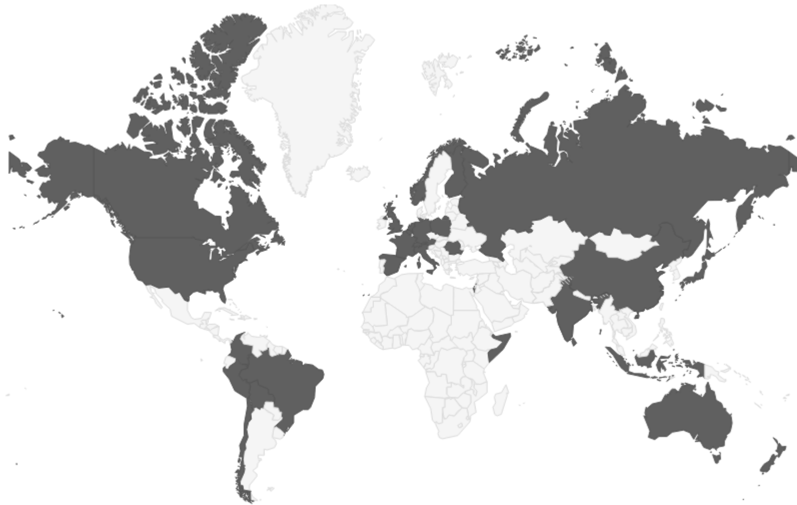


Figure 7.1: Countries Our Survey Respondents Reside

cloud migration support. We also excluded 8 responses made by respondents who did not understand our description of fault localization (i.e., he/she chose the "I don't understand" option). Moreover, we excluded 6 responses from respondents who participated in the English version of our survey but indicated their English language proficiency level as "Poor" or "Very Poor". At the end, we had a set of 386 responses.

Out of the 386 respondents, 80.83%, 30.05%, and 17.10% described software development, software testing, and project management as their job role respectively. Note the percentages do not add up to 100% since some respondents perform multiple roles (especially for respondents in small to medium sized companies, or from open-source projects). Based on their experience level, we grouped respondents into three categories: low, medium, high. We first sorted respondents based on their experience in years. Respondents in the bottom and top quartile were put in the low and high categories respectively, while the others were put in the medium category. Out of the 386 respondents, 78.13% and 44.24% are professional and open-source software developers, respectively. Note that the percentages do not add up to 100% since some respondents are both professional and open-source software developers.

7.3.2 Answers to Research Questions

RQ1: Importance of Fault Localization. Figure 7.2 shows the percentages of ratings of various categories (i.e., Essential, Worthwhile, Unimportant, Unwise) given by respondents from the following demographic groups:

- All respondents (All)
- Respondents with software development role (Dev)
- Respondents with software testing role (Test)
- Respondents with project management role (PM)
- Respondents with low experience (ExpLow)
- Respondents with medium experience (ExpMed)
- Respondents with high experience (ExpHigh)
- Respondents who are open source practitioners (OS)
- Respondents who are professional software engineers (Prof)

From Figure 7.2, we can notice that most respondents gave "Essential" and "Worthwhile" ratings. Only a minority gave "Unimportant" and "Unwise" ratings (less than 10%) across all demographic groups. Around 20-35% of respondents across demographic groups rated fault localization as an "Essential" research topic.

We notice that testers value fault localization techniques *slightly more* than developers and project managers (less percentage of testers marked fault localization as "Unimportant" or "Unwise"). To check whether this difference is statistically significant, we performed the Fisher's exact test [29] and found no significant difference (p-value = 0.265).

As experience level increases, less percentage of respondents view fault localization as "Essential". We can especially notice a sharp drop in the percentage of respondents rating fault localization as "Essential" between ExpMed and ExpHigh groups. Again, we performed the Fisher's exact test and this time we found that the

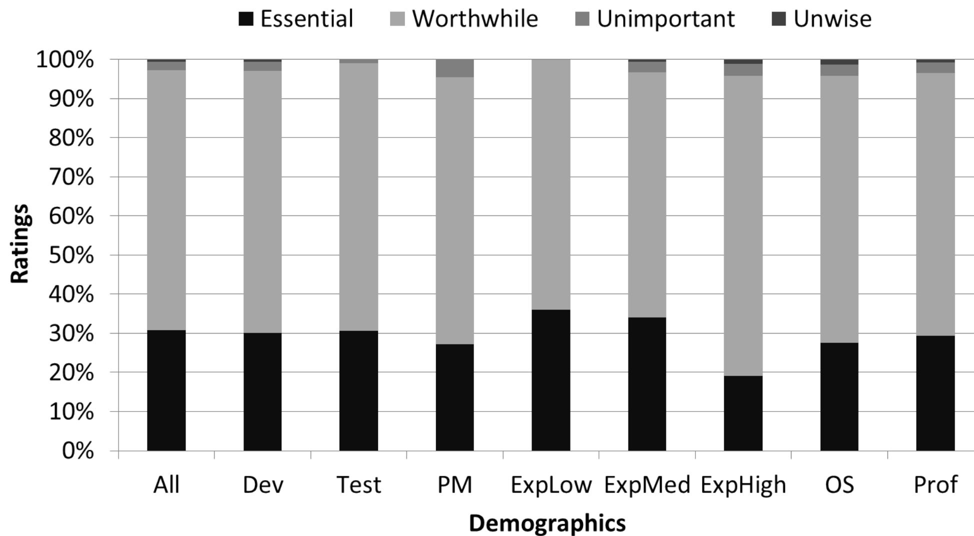


Figure 7.2: Importance of Fault Localization Research to Respondents of Various Demographic Groups

difference is statistically significant (p-value = 0.014). We also performed the Spearman correlation test [113] and found that there is a significant (p-value = 0.007) yet small negative correlation (ρ = -0.14) between experience (in years) and ratings (mapped to a value between 1 ("Unwise") to 4 ("Essential")). These results suggest that more experienced developers perceive fault localization to be less "Essential" than less experienced ones.

For respondents who rated "Unimportant" and "Unwise", some of them described their reasons, as follows:

- Disbelief that fault localization techniques can deal with difficult bugs, e.g.,
  - "*Hairy bugs hide in interaction between various components and I don't think automated tools help much. I'm well aware of what static analysis can do and very few hard bugs would be solved with it.*"
  - "*My opinion is scoped by the web development, but still: different frameworks, different technologies and for each one you'll need to adapt your potential tool to solve specific bugs ...*"
- Disbelief that fault localization techniques can provide rationale, e.g.,
  - "*I doubt any automated software can explain the reason for things such*

as broken backwards compatibility, unclear documentation, what really should happen etc. They require human analysis."

- Belief that the status quo is good enough, e.g.,
  - "... *And even if you will succeed, I don't think personally I would pay for it, because for my cases usual stack trace is over than enough."*

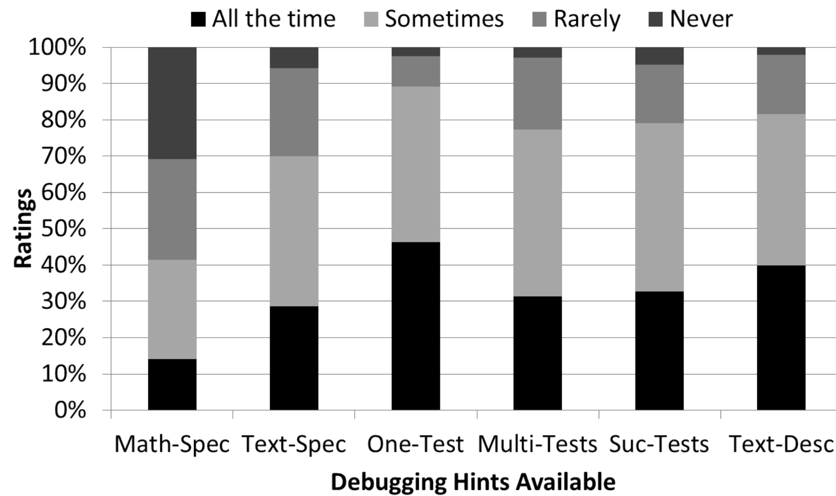


Figure 7.3: Availability of Debugging Data to Practitioners (Math-Spec = Mathematical specification, Text-Spec = Textual specification, One-Test= One test case, Multi-Tests = Multiple test cases, Suc-Tests = Successful test cases, Text-Desc = Textual description)

RQ2: Availability of Debugging Data. Figure 7.3 shows practitioners' feedback on availability of different debugging data, which were assumed to be available by prior fault localization studies: specification (e.g., [39]), single failing test case (e.g., [99]), multiple failing test cases (e.g., [10]), passing test cases (e.g., [10]), and bug reports (e.g., [146]). The following are our findings:

- Most respondents indicated that mathematical specifications are rarely or never available. Textual specifications are more common with almost 70% of the respondents indicated that they are available "all the time" or "sometimes".
- Test cases are more commonly available than specifications. More than 70% of the respondents mentioned that these debugging data are available "all the time" or "sometimes".

- Bug reports are also commonly available with close to 80% of the respondents mentioned that they are available “all the time” or “sometimes”.

RQ3: Preferred Granularity Level. Different fault localization techniques pinpoint bugs at different granularity levels, e.g., class (file) [146], method [138], basic block [76], statement [53]. Figure 7.4 shows practitioners’ preferred granularity levels. Note that the percentages do not add up to 100% since a respondent can indicate more than one preferred granularity level. We notice that the top-3 preferred granularity levels are: method, statement, and block, respectively. There is no clear winner among these three granularity levels, with method being slightly preferred by practitioners. Class and component are too coarse granularity levels to many respondents. A technique that can pinpoint the right buggy component or class may still require practitioners to manually check a large chunk of code.

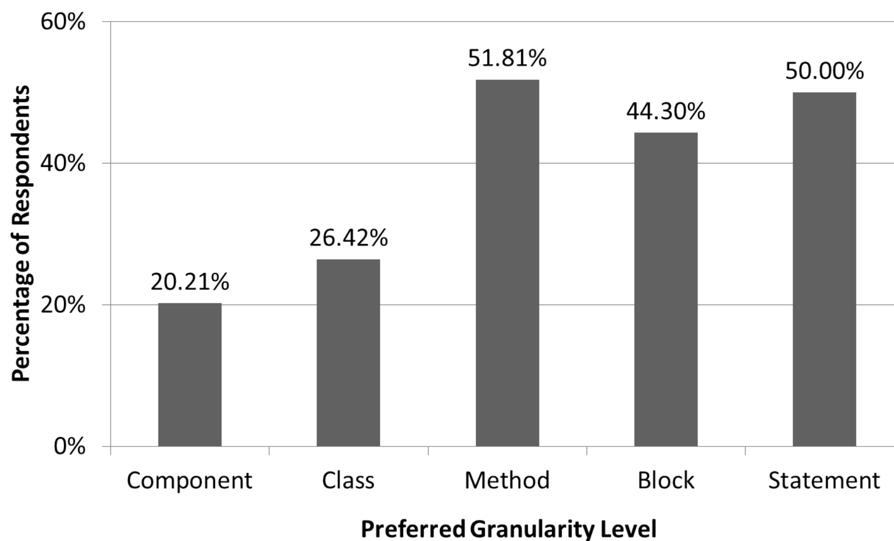


Figure 7.4: Percentages of Respondents Specifying Various Preferred Granularity Levels

RQ4: Minimum Success Criterion. Fault localization techniques return a list of suspicious program elements. If buggy program elements appear at the end of a long list, practitioners may be better off doing manual debugging. Figure 7.5 shows percentages of respondents with their minimum success criteria. Around 9 percent of respondents did not consider a fault localization session that requires him/her to

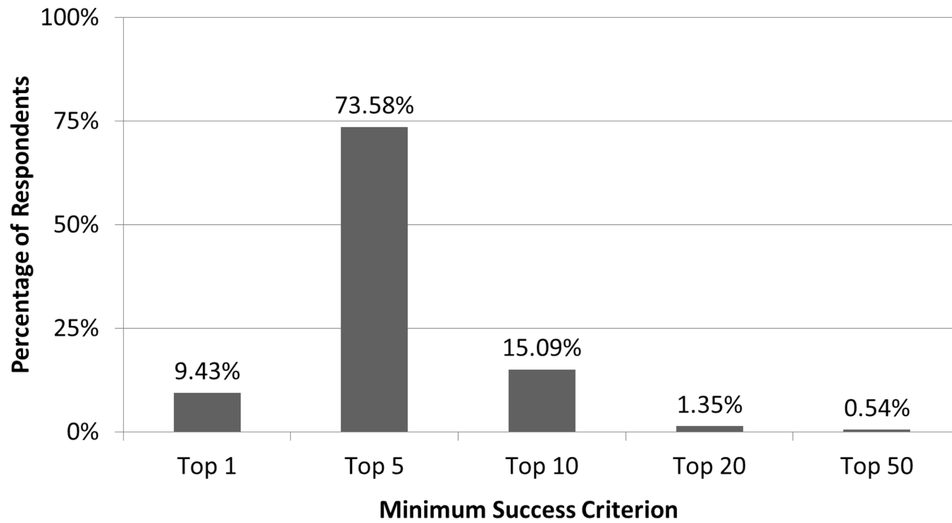


Figure 7.5: Percentage of Respondents Specifying Various Minimum Success Criteria

inspect more than one program element to find a bug as successful. The threshold was 5 program elements for 73.58% of the respondents. Moreover, almost all respondents (close to 98%) agreed that inspecting more than ten program elements is beyond their acceptability level.

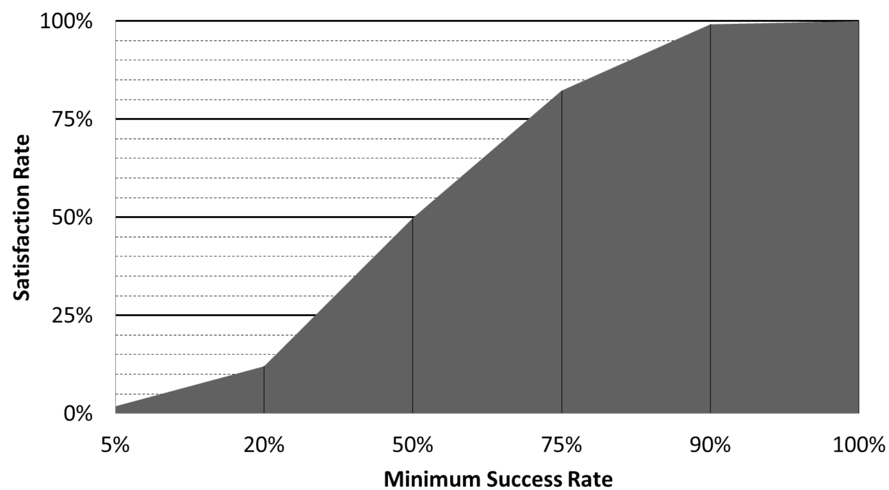


Figure 7.6: Minimum Success Rate vs. Satisfaction Rate

RQ5: Trustworthiness. Intuitively, a technique that is unsuccessful most of time will be considered as untrustworthy (unreliable) and is less likely to be used. Figure 7.6 shows the percentages of respondents who were satisfied with different success rates. A very small proportion of respondents can tolerate a fault localization

technique that is only successful 5% of the time. Around twelve percent of respondents were satisfied with a technique that has a 20% success rate. To achieve a satisfaction rate of 50%, 75%, and 90%, a fault localization technique needs to be successful 50%, 75%, and 90% of the time, respectively.

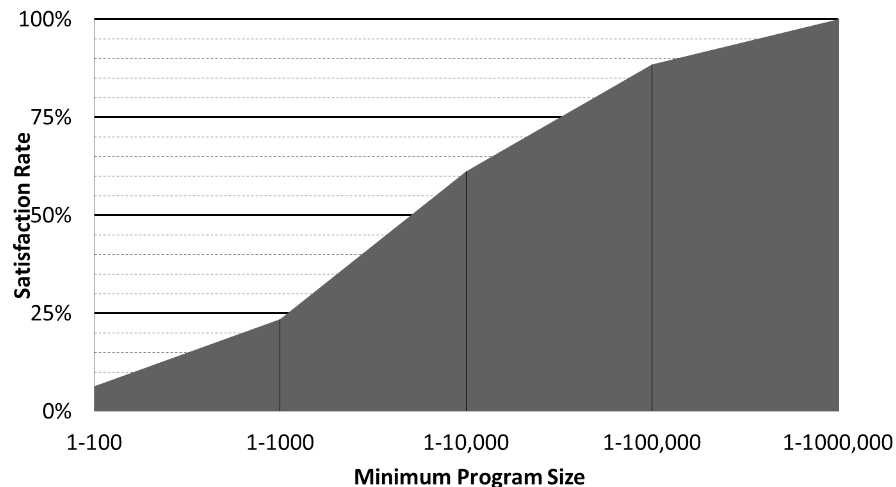


Figure 7.7: Minimum Program Size vs. Satisfaction Rate

RQ6: Scalability. Figure 7.7 shows the minimum program sizes that fault localization techniques need to support before practitioners consider them useful. To achieve a satisfaction rate of 50%, 75%, and 90%, a fault localization technique needs to be scalable enough to deal with programs of size 10,000 LOC, 100,000 LOC, and 1,000,000 LOC, respectively.

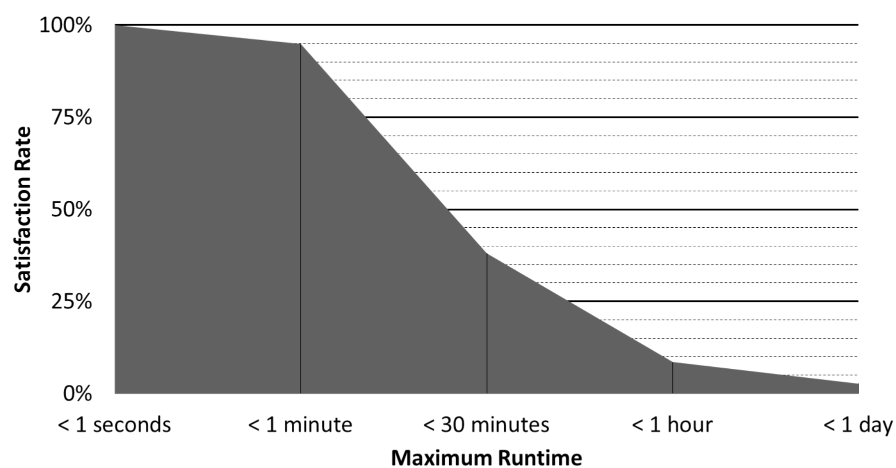


Figure 7.8: Maximum Runtime vs. Satisfaction Rate

RQ7: Efficiency. Figure 7.8 shows the maximum amount of time practitioners are

willing to wait for a fault localization technique to provide a recommendation. Few respondents were willing to wait more than an hour for a fault localization technique to do its job (less than 9%). To achieve a satisfaction rate of at least 50%, a fault localization technique needs to finish its computation in less than a minute. This efficiency threshold satisfied more than 90% of the respondents.

RQ8: Willingness to Adopt. We find that almost all the respondents (except less than 2 percent) were willing to adopt a trustworthy, scalable, and efficient fault localization technique. The main reasons why some of the respondents were still unwilling to adopt are as follows:

- Resistance to change
  - "*Since I already have one and to use another would require training time and time to get used to it*"
  - "*I would probably prefer traditional breakpoint / single stepping debugging watching what the program does. This of course depends on the kind of bugs. If it could find difficult to locate bugs*"
- More information needed
  - "*would it be open source? Would it work with my main programming language? Would it work with distributed environments? These are important aspects and I cannot commit to adoption without the answers.*"
- Disbelief of possibility of success
  - *"I don't think you can do it."*

RQ9: Other Factors. After asking respondent willingness to adopt a trustworthy, scalable, and efficient fault localization technique, we ask about two additional factors: ability to provide rationale and IDE integration. We provided practitioners with four statements (listed in Section 5.3.2) and asked respondents to indicate their

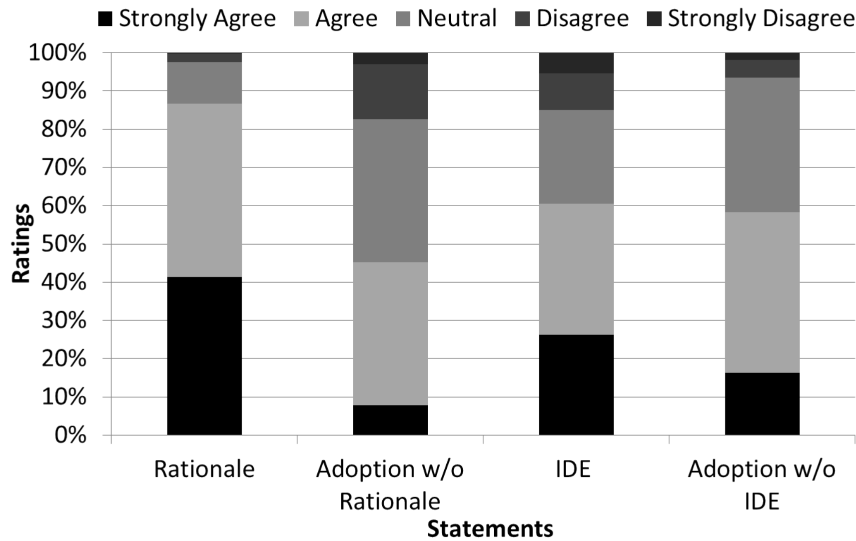


Figure 7.9: Other Factors Affecting Adoption

levels of agreement or disagreement with the statements. Figure 7.9 shows respondents' agreement levels for the four statements.

From the figure, we find that more than 85% of our respondents strongly agreed or agreed that ability to provide rationale is important. Adoption rate reduces for fault localization techniques that cannot provide rationale – more than 15% disagreed or strongly disagreed that they will *still use* a trustworthy, scalable, and efficient fault localization technique if it cannot provide rationale why some program locations are marked as suspicious, and many were on the fence (around 40% chose "Neutral"). Reasons why they chose not to adopt (i.e., they picked "Disagree" or "Strongly disagree") include:

- Lack of trust due to possibilities of false positives
  - *"False positives are worst than false negatives in my opinion. That is, if the tool tells me where the bug is but that's not actually true, that annoys me greatly."*
  - *"I need to know why the debugger considers code faulty, otherwise I will consider it a false positive and ignore. Not providing a rationale also means I have to investigate code that might be a false positive, which is a waste of my time."*

- *"Software development is all about logic. Debugging is done logically and rationally. Therefore any tool should facilitate in the rational thinking of the developer and not intuitive thinking"*

- Rationale is needed for bug fixing and code quality improvement
  - *"Because to make a decisions about bug fixing I want to *exactly* know why the automated tool "thinks" that the code have a bug."*
  - *"... I would also need to provide the fix, so I feel some rationale would also help with that."*
  - *"Rationale gives understanding which will help in improving the code quality for future"*
- Rationale is needed to incorporate practitioners' own domain knowledge
  - *"So that I can filter the results through my own knowledge ..."*

Furthermore, we find that IDE integration is less important than ability to provide rationale – only less than 65% agreed or strongly agreed that IDE integration is necessary. Without IDE integration, adoption rate is likely to reduce (albeit less substantially than when rationale is not provided) – more than 5% disagreed or strongly disagreed that they will *still use* a trustworthy, scalable, and efficient fault localization technique if it is not integrated to their favourite IDE, and many were on the fence (around 40% chose "Neutral"). Reasons why they chose not to adopt (i.e., they picked "Disagree" or "Strongly disagree") include:

- Extra steps are needed which affects debugging speed
  - *"Testing is awkward and should be made as easy as possible. No integration means extra steps which means testing will be more cumbersome and hence less used."*
  - *"debugging needs to be fast and efficient"*
- Developers have a strong reliance on IDE

- *"Currently Visual Studio 2013 provides all the tools required to build, test and deploy and application. It is not worthwhile attempting to use a different tool for debugging."*
- *"IDE is our environment. If I can't add something into my environment, it's useless."*

- Developers refuse to change personal workflow for convenience reason
  - *"If it doesn't fit into my workflow, then it's more trouble than it's worth."*
  - *"Personal habits, or feel inconvenient."*
  - *"Convenience."*

7.3.3 Respondents' Final Comments

Some respondents provided additional comments and suggestions:

- Integration with continuous integration tool would be a plus
  - "*I would be interested in running an automated debugging tool as part of continuous integration, so that rather than the test just failing, it gives a report on what the likely cause of the problem is.*"
  - "*Would be nice if it will be pluggable to the build systems such as Gradle, Maven, SBT, etc. For example, auto-run after failed test on the CI.*"
  - "*... Can I also run it offline i.e. CLI, CI server, via SonarQube or SonarGraph, etc..? ...*"
- Need to support multiple languages and workflows
  - "*Should be able to run from cmd-line. Doesn't need rational, just needs to give me suggestions of where to look at in the code with many objects interacting, it is sometimes hard to determine the cause. Should work with most programming languages.*"

  - “*A debugging or bug-finding tool should be easily integrable into other workflows. Portability and ability to be used with other tools is the most important characteristic for me when choosing the tools to integrate into a development process.*”

- Extension of work to evaluate claim of bug existence is needed

  - “*... Having an automated tool would be useful to not only locate the source of a bug, but to evaluate the original claim of a bugs existence. Having a tool automatically confirm a bug and perhaps where to look to fix it would easily convince a maintainer that this bug report is worth looking at. ...*”

7.4 Current State of Research

At the end of our literature review process, we identified 2, 5, 3, 2, and 4 fault localization papers from ICSE, FSE/ESEC-FSE, ISSTA, TSE, and TOSEM, respectively. Jin and Orso presented their technique F3 in ISSTA 2013 [50] and TOSEM 2015 [51]. In this study, we considered the journal version. Table 7.1 shows the capabilities of state-of-the-art fault localization techniques in terms of seven factors.

Debugging Data: From Table 7.1, we notice most of the papers use test cases as debugging data, followed by bug reports. From Section 7.3.2, we find that most of the respondents mention that these data are available “all the time” or “sometimes” during their debugging sessions. No paper relies on manually created specification as debugging data which are often unavailable. The work by Mariani et al. used *automatically* generated specifications to help automated debugging [80].

Granularity Level: From Table 7.1, we notice that only two papers (i.e., [71, 133]) work at method level granularity – which is the most preferred option. Most papers work at statement level granularity, which is the second most preferred option. There are several papers that work at class (file) level granularity which most re-

spondents found to be too coarse-grained.

Trustworthiness: We analyzed the papers using the most popular success criterion indicated by our respondents, i.e., buggy program elements must appear in the top-5 positions (Top 5). Using this criterion, we read the papers and checked the success rates of the techniques proposed in them. By comparing a technique's success rate with our survey results, we can derive a satisfaction rate. Our survey results point out that a fault localization technique with a success rate of 50%, 75%, and 90% satisfies at least 50%, 75%, and 90% of our respondents, respectively. From Table 7.1, we can note that none of the papers can satisfy at least 75% of our respondents. Five papers can satisfy at least 50% of our respondents. These papers are those that use bug reports as debugging data instead of test cases. Unfortunately, they work at a coarser level of granularity (i.e., class (file)) that is not preferred by a large majority of our respondents. We put some papers in category "?" since we cannot ascertain the success rates of the fault localization techniques presented in those papers.

Scalability: Our survey results point out that a fault localization technique that supports at least 1,000,000 LOC, 100,000 LOC, and 10,000 LOC satisfies at least 90%, 75%, and 50% of the respondents, respectively. Table 7.1 shows that 6 papers can satisfy at least 75% of our respondents, while 7 can satisfy at least 50% of our respondents. We put the work by Kim et al. [58] in category "?" since the paper does not mention the number of lines of code of programs used to evaluate their work (i.e., various components of Mozilla Firefox and Core programs).

Efficiency: Our survey results point out that a fault localization technique that can produce output in less than a minute satisfies at least 90% of the respondents. From Table 7.1, we find that 5 papers can satisfy at least 90% of our respondents. Some papers do not describe the runtime of their proposed techniques and thus we put them in the "?" category.

Table 7.1: Capabilities of Current State-of-Research

Factor	Type	Papers
Debugging Data	Specification	-
	Test Cases	[9],[10], [71], [80], [99], [107], [116] [139], [141], [144]
	Bug Reports	[51], [58], [71], [133][5], [140], [146]
Granularity	Method	[71], [133]
	Statement	[9], [10], [80][6], [99], [116], [139], [141] [144]
	Basic Block	[51]
	Other	[58], [107], [140], [146]
Factor	**Sat. Rate**	**Papers**
Success Rate	90%	-
	75%	-
	50%	[51], [58], [99], [107], [133], [140], [144], [146]
	?	[9][7], [10][8], [80][9], [139][7], [141][8]
Scalability	90%	[80], [133]
	75%	[51], [71], [140], [144], [146]
	50%	[9], [10], [99], [107], [116], [139], [141]
	?	[58]
Efficiency	90%	[9], [71], [107], [116], [140]
	?	[51], [58], [80], [133], [141], [146]
Factor	**Support?**	**Papers**
Rationale	Yes	[80][10], [116][10]
IDE Integration	Yes	-

Provide Rationale: Most fault localization techniques only highlight potentially buggy program elements. Practitioners can understand why these program elements are highlighted by reading the description of the heuristics employed by the techniques, e.g., they are highlighted because they are executed more often by failed test cases, but rarely or never by successful test cases (e.g., [10, 139]), they are

[5]The technique proposed in the paper uses crash traces.

[6]The technique identifies faulty method invocations.

[7]Most likely its satisfaction rate is below 50%. The mean number of program elements to check to locate bugs is substantially larger than 5.

[8]Only relative evaluation scores are shown in the paper.

[9]The technique returns connected components containing method invocations. The size of each component is not reported.

[10]To some extent (see paragraph).

highlighted because they contain contents that are textually similar to the content of the input bug report (e.g., [146, 140]), etc. Unfortunately, these basic rationales are not likely to be sufficient to help practitioners separate false positives from real bug locations or fix bugs – c.f., [94].

We highlight two papers by Sun and Khoo [116] and Mariani et al. [80] which go an extra mile. Both papers provide a graph-based structure that a practitioner can inspect to better understand why a program element is flagged as potentially buggy – which is referred to as a *bug signature* by Sun and Khoo. However, since no user study has been conducted to evaluate the graph-based structures that are returned by these approaches, it is unclear whether these graph-based structures can help practitioners to debug better.

IDE Integration: None of the fault localization techniques proposed in the 15 papers that we have reviewed has been integrated into a popular IDE. We find that the work by Zhou et al. [146] has been integrated into Bugzilla by Thung et al. [122], however, Bugzilla is not an IDE. IDE integration requirement is expressed as one of the prerequisites for adoption by some of our survey respondents.

7.5 Discussion

7.5.1 Implications

Large demand for fault localization solutions. Devanbu et al. recommended disseminating empirical findings and giving attention to practitioner beliefs, in particular where results are preliminary [26]. Fault localization tools are currently research prototypes. Thus, participants may not have used them before. Our survey is a practical way to reach out to a large number of practitioners and get their feedback. It is similar to a requirement elicitation phase in a typical software project where a developer tries to understand client's requirement (without a system being completed). Several studies have also tried to understand the adoption factors

of tools in a similar way [131]. Our survey highlights the importance of research in fault localization. More than 97% marked this field of research as “Essential” or “Worthwhile”. Almost all respondents indicated that they are willing to adopt a fault localization technique that satisfies some criteria. Thus, although there are challenges in this research area, we encourage researchers to continue innovating since there is still a wide “market” awaiting working solutions.

High adoption barrier exists. Despite practitioners’ enthusiasm in this field of research, they have high thresholds for adoption. More than eighty percent of respondents indicated that they view a fault localization session as successful only if it can localize bugs in the top 5 positions. To satisfy 75% of our respondents, a fault localization technique needs to be successful 75% of the time, be able to process programs of size 100,000 LOC, and complete its processing in less than a minute. Inability to provide rationales of why program elements are marked as potentially buggy and poor integration to practitioners’ favorite IDEs are likely to reduce practitioners’ willingness to adopt (with around 5-15% of respondents indicated that they would withdraw their willingness to adopt, and about 40% of respondents sat on the fence).

Large improvement in trustworthiness (reliability) of existing techniques is needed. Our literature review highlights that the most crucial issue with existing fault localization techniques is their trustworthiness. Without this quality, practitioners may ignore outputs of fault localization techniques. The best performing studies cannot satisfy 75% of the respondents or more. Even many of those that can satisfy at least 50% of the respondents work at a granularity level that is considered too coarse by most of the respondents (i.e., class (file)). One of the studies by Qi et al. [99] work at a preferred granularity level and can satisfy more than 50% of the respondents (its success rate is beyond 50%) – however its effectiveness has only been tested on 5 different bugs from small to medium sized programs (less than 100 kLOC). Recent efforts have mitigated this issue by developing techniques that can help practitioners estimate reliability of a fault localization output [69, 70].

Some improvement in scalability is needed. Another issue with existing fault localization techniques is their scalability. To achieve 90% satisfaction rate, such techniques need to work on programs of size 1,000,000 LOC. Among the papers we used in our literature review, only 2 papers [80, 133] have demonstrated that the proposed techniques are able to satisfy such requirement.

Research on ways to provide suitable debugging rationale is needed. Among the papers that we have investigated, there are only 2 papers proposing techniques that can offer some explicit rationales behind their recommendations in the form of graph-based bug signatures. However, more user studies are needed to check if these signatures are useful to help debugging. Future research should be devoted on designing more advanced fault localization techniques that can provide explicit and useful rationales to help practitioners debug better.

Community-wide effort to integrate state-of-the-art fault localization techniques to popular IDEs is needed. None of the papers investigated in our literature survey describe integration to a popular IDE. There is a need for a community-wide effort to encourage the integration of state-of-the-art fault localization techniques to popular IDEs. Campos et al. [18] and Pastore et al. [95] have released Eclipse plugins that implement two existing fault localization techniques, i.e., [4] and [96], respectively. However, many latest techniques (including those analyzed in Section 7.4) have not been integrated to IDEs yet.

7.5.2 Limitations

Noisy Responses. It is possible that some of our survey respondents do not understand fault localization or our questions well, and thus their responses may introduce noise to the data that we collect. To reduce this threat to validity, we drop responses that are submitted by people who are neither professional software engineers nor participants of open source projects, and whose job roles are none of these: software development, testing, and project management. We also drop responses by

respondents who select the “I don’t understand” option, or declare to have “Poor” or “Very poor” English proficiency level. We also translate our survey to Chinese to ensure that respondents from China can understand our survey well. Still, we cannot *fully* ascertain whether participant responses are accurate reflections of their beliefs. This is a common and tolerable threat to validity in many past studies about practitioners’ perceptions and expectations, e.g., [60], which assume that the majority of responses truly reflect what respondents truly believe.

Generalizability. To improve the generalizability of our findings, we have surveyed 386 respondents from more than 30 countries across 5 continents working for various companies (including Microsoft, Google, Cisco, LinkedIn, ABB, Box.com, Huawei, Infosys, Tata Consultancy Services and many more) or contributing to open source projects hosted on GitHub, in various roles. Still, our findings may not generalize to represent the expectations of all software engineers. For example, practitioners who are not proficient in either English or Chinese are not represented in our survey.

Overall Expectation. We consider practitioners’ overall expectation for “all spectrum” of bug types. Practitioners’ expectations for a particular type of bugs (e.g., concurrency bugs) may differ. We also consider “all spectrum” of practitioners. In the future, we plan to collect, and even control for practitioners’ prior experience with automated debugging tools, or even automated test generation or automated bug finding tools. Such exposure, may bring down the expectations of users, while making them realize the utility of such tools.

Adoption Factors. We have only considered several factors that may affect the adoption of a fault localization technique: debugging data availability, preferred granularity level, success criterion, success rate, scalability, efficiency, ability to provide rationale, and IDE integration. There could be other factors that contribute to adoption that we have not investigated. We plan to consider these factors in a future study.

Willingness to Adopt vs. Actual Adoption. Our survey can only estimate practitioners' willingness to adopt. Actual adoption is a complex process which involves not only individual attitudes (e.g., perceived usefulness) but also organizational support (e.g., training, incentives) and social influence (e.g., support by peers/colleagues) – c.f., [5, 73, 119]. Still, individual attitudes is one factor that leads to actual adoption and our survey measures such factor. When state-of-the-art fault localization techniques achieve practitioners' perceived thresholds for adoption, it would be interesting to perform industrial studies to let practitioners use such techniques for a substantially long period of time (to overcome their resistance to change) and under various settings for a thorough evaluation, and collect further feedback.

Chapter 8

Learning to Test: Helping Developers Make Testing Decisions

8.1 Introduction

Companies spend thousands of dollars and developers often spend significant amount of time in testing their software. Even though projects spend almost 40% of the time during testing, more than 80% of the open-source developers agree that their projects lack testing plans [145]. In another study, developers mention that they spend 50% of their time on testing [12]. Even though it is hard to ascertain how much time developers spend on testing, it is known that complete testing is often not possible due to limited availability of time and resources. Open source developers and industry professionals mention that they often face schedule constraints, due to which they cannot perform testing [64]. Thus, developers need to prioritize parts they need to test, which is not a trivial task.

To aid developers make testing decision, we propose a "learning to test" framework named *TestAdvisor*. It automatically learns a model from program elements that are tested in a previous version to provide suggestions on program elements to test in a newer version. We extract a comprehensive set of features from source code and version control system. The features can be grouped into six categories: impact,

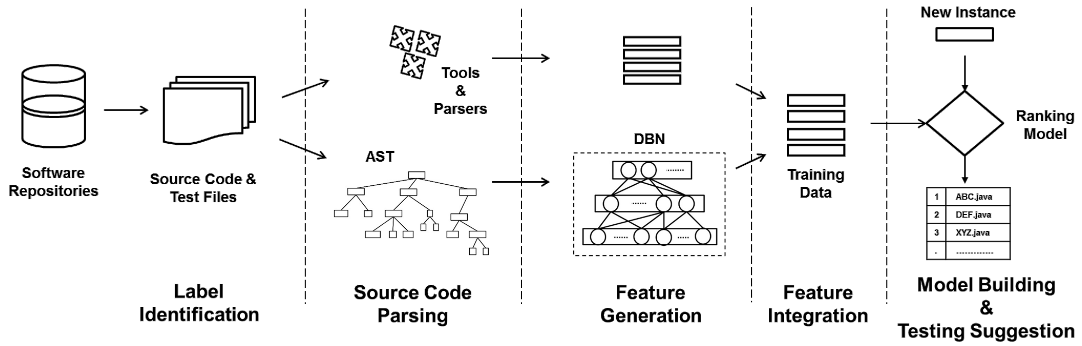


Figure 8.1: Overview of *TestAdvisor*

functionality, process, ownership, code and semantic. We combine the various features to build a ranking model by using a machine learning technique namely Naive Bayes. Our choice is based on the fact that Naive Bayes has been successfully used in many past studies that require ranking or classification step [126, 147].

Moreover, to deal with cold start problem and make our approach applicable to projects with limited training test cases, we also consider the cross-project setting. Data from other projects are used to learn a model that is then applied to a target project. We propose an enhanced learning strategy to allow *TestAdvisor* to work better on cross-project setting. Our adapted *TestAdvisor* (referred to as *TestAdvisor*^{CP}) estimates the utility of a project in a training data for cross project setting. Data from some projects may be better than others in creating a generalizable model that can work well across projects. Based on the utility estimates, *TestAdvisor*^{CP} learns multiple models, each being trained on a randomly selected subset of projects in the training data having high utility. These models are then integrated into a unified composite model.

8.2 TestAdvisor and TestAdvisor^{CP}

In this section, we first present an overview of our approach. Next, we describe the features we use to build a ranking model.

8.2.1 Overview

The goal of this study is to guide developers to make informed testing decisions using a machine learning model built by considering a comprehensive set of features. Figure 8.1 presents an overview of our framework. Below, we explain its steps:

1. ***Label Identification:*** For each data file, we consider it as "tested" if it is covered by at least one test case, or "untested" otherwise. Similarly, each method in the project is considered "tested" if a test case touches that method. We use heuristics to identify test classes, i.e., files which contain "test" in the name. For identifying calls to classes and methods by test cases, we build a call graph for each project. First, we leverage Maven automated build system to build projects and use *java-callgraph* [41] to construct call graphs. We then parse the call graphs to get tested and untested program elements.

2. ***Source Code Parsing:*** We leverage open-source tools to collect metrics such as Lines of Code (LOC), Cyclomatic Complexity from the source code. We write a parser to collect process and ownership features from the history of these projects. Our parser extracts information about each file and method changed as well as number of lines added, deleted and changed in both the *previous* and *latest* version. We also write a parser to collect information about test cases from call graphs.

   For collecting the semantic features, we extract syntactic information from the source code using Java Abstract Syntax Tree (AST). We make use of Eclipse Java Development Tools (JDT) [30], which provides tools to build Java applications and to generate AST from the source code. We extract three types of AST nodes: a) method invocations and class instance creation nodes, b) control-flow nodes such as for loops, while loops, if statements, break statements, switch statements etc., and c) method, enum and type declarations. For each file and method, we extract these nodes as a vector of tokens.

3. ***Feature Generation:*** We extract a comprehensive set of features characterizing program elements. We describe the features in detail along with their intuition and how we extracted them in Section 8.2.2.

4. ***Feature Integration:*** To build a ranking model, we combine the different features generated, which are used in the next step, i.e., model building.

5. ***Model Building and Testing Suggestion:*** Using the above steps we generate feature vectors for each project. Each vector is a set of features collected for each file or method. We use these feature vectors and labels to build a machine learning model to learn about code previously tested by developers. We model the decision on whether to test a code as a ranking problem. We use Naive Bayes as our default technique for model building and ranking.

   Using the above learning process, we get a ranking model which we use on a newer version to rank program elements to be tested. We can build this model offline and can use it to provide suggestions to developers on elements to test when they want to create new test cases.

8.2.2 Feature Extraction

Feature extraction is an important part of "learning to test" as the quality of features extracted from the dataset will determine the performance of the model. We extract different features from the source code and version control system. We divide these features into six categories: impact, functionality, process, ownership, code and semantic. Table 8.1 shows the features we collect and their respective definitions. We briefly explain the intuition behind each set of features and how we collect them in the following sub-sections.

Table 8.1: Features used for the learning model. Features appended by (f) and (m) are only calculated at the file level and method level, respectively, while others are computed at both the levels.

Metric	Definition
Impact Features	
HS	Hub Score
AS	Authority Score
CT	Between Centrality
Functionality Features	
CS	Cosine Similarity
JC	Jaccard Index
JW	Jaro-Winkler
NG	N-Gram
SD	Sorensen-Dice coefficient
LCS	Longest Common Subsequence
DL	Damerau-Levenshtein
Process Features	
NC	Total number of Commits
TLD	Total number of LOC deleted
ALD	Average number of LOC deleted
MLD	Maximum number of LOC deleted
TLA	Total number of LOC added
ALA	Average number of LOC added
MLA	Maximum number of LOC added
TLC	Total number of LOC added, deleted or modified
ALC	Average number of LOC added, deleted or modified
MLC	Maximum number of LOC added, deleted or modified
Ownership Features	
TDO	Total number of developers
MO	Highest proportion of commits
MiC	Total number of developers who have made less than 5% of the total commits
MaC	Total number of developers who have made more than 5% of the total commits
Code Features	
LOC	Lines of Code
CC	Cyclomatic Complexity
CA	Afferent Couplings
CE	Efferent Couplings
SC	Total count of statements
CLOC	Total count of comment LOC
CD	Comment Density
NL	Nesting Level
DIT (f)	Depth of Inheritance Tree
CBO (f)	Coupling between Object Classes
RFC (f)	Response For Class
LCOM (f)	Lack of Cohesion between Methods
NOC (f)	Number of Children
CBOI (f)	Coupling between Object Classes Inverse
TNM (f)	Total number of methods
TNPM (f)	Total number of public methods
TNLM (f)	Total number of local methods
TNS (f)	Total number of setters
TNG (f)	Total number of getters
TNA (f)	Total number of attributes
NP (m)	Number of Parameters
Semantic Features	
MC	Method Invocations & Class Instances
CFN	Control-flow nodes
DN	Declaration nodes

Impact Features

These features are collected from the static call graph, which contains the caller-callee relationships for different files and methods. These relationships can be helpful to understand the dependencies between various program elements. Some program elements are highly impactful; change in these elements can induce changes in its dependant elements. Thus, these elements may need to be thoroughly tested. To identify these impactful elements, we compute a number of scores proposed by social network researchers, which includes hub score, authority score and betweenness centrality. For example, hubs act as important aggregators and dispensers of information, thus, developers tend to perform more testing for hub methods [19].

We compile each project in our dataset and use the generated bytecode to construct call graphs using *java-callgraph*[1]. We input this call graph structure to Jung [54] to compute various impact features at both the file and method level. The call graph gives information about different method calls. We aggregate these method calls to generate calls between files.

Functionality Features

Developers often specify the main functionalities of an application in a README file which serves as an introduction for novice as well as experienced developers to understand the goal of a project. Developers may want to test program elements implementing these main functionalities rather than those implementing secondary functionalities.

For these features, we compute similarity between source code files or methods and the README file. We consider several similarity measures such as Cosine similarity, Jaccard index, Jaro-Winkler distance, etc. We chose these measures as they consider lexical similarity and represent both term-based (Cosine similarity, Jaccard index) and character-based (Jaro-Winkler, N-gram) similarity [37]. We compute these features using an open-source library - *java-string-library* [25].

Process Features

These features describe the changes of source code i.e., how many changes are made to program elements (e.g., files or methods). These features can be useful for developers to understand which program elements are changed often and might need investigation as changes can often induce bugs. Thus, such program elements might need to be tested more rigorously.

We use the version control system of each project to calculate these features. At the file level, we use `git log --numstat` to get all the files changed and the number of lines changed in each commit. For the method level, we use `git log -U1 -w`, to download the commit patches for each commit. Similar command has been used in a previous study to get commit patches [19]. We write a parser which reads the commit patches and extract methods changed in each commit and the number of lines modified.

Ownership Features

These features describe the ownership of different program elements (e.g., files or methods) by developers, e.g., how many developers modify a particular program element, what are their respective contributions, etc. Ownership features can help developers gauge how much testing effort should be put in as these features can serve as proxy for software quality [87]. We use the git logs retrieved from the version control to get information about contributors and their respective contributions for each file or method.

Code Features

These features describe the code in terms of size, complexity, cohesion, coupling, comment availability, and depth of inheritance. These features can help developers identify elements that need to be tested. For example, with increasing size and complexity of a file, more testing effort may needed as the file is more likely to be

buggy.

To calculate these features, we use SourceMeter [75], which is a static source code analyis tool available for various languages such as Java, C, C++, Python etc. SourceMeter computes features at different levels of granularity such as files and methods.

Semantic Features

Programs have well-defined syntax represented by Abstract Syntax Trees (AST). Semantics represent the meaning of various elements in a program, which can be used to differentiate between different files and methods. These features are different from the above proposed features as there can be cases where features such as lines of code can be the same across files, however, their semantics may differ, e.g., the order of program constructs may differ corresponding to different program logic.

Following Wang et al. [126], we leverage deep learning to automatically generate semantic features from the Abstract Syntax Tree of the source code. We use Deep Belief Network (DBN) [45], which is a deep neural network, composed of multiple layers of Restricted Boltzmann's Machines (RBM) [13]. The RBM, used to learn representation from input data, consists of a two-layer network. The first layer is called *visible layer* containing the input node, wheras the second layer is the *hidden layer* including several hidden nodes. DBN is formed by stacking multiple RBMs where the hidden layer of the former RBM is the visible layer of the next RBM. By applying deep architecture with more RBM layers, DBN is able to learn more meaningful features. In this paper, we use number of hidden layers as 10, the number of nodes in each hidden layer as 100, and the number of iterations as 200 since they are well-tuned parameters [126].

We first create input vectors after traversing the AST based on the position of the elements in the tree. Similar to [126], we filter noisy instances by using Closest List Noise Identification (CLNI) [61], which identifies k-nearest neighbors and if a

certain number of neighbors have opposite labels, then that instance is considered as noise. As our feautres are semantic tokens, similar to [126], we apply edit distance similarity algorithm [88] to detect and eliminate mislabeled data. After filtering the noise, we use DBN on the filtered instances. DBN automatically learns features based on the difference between two vectors. DBN accepts only numerical vectors and all the vectors of a project must be same in length. We first create mapping between tokens and integers, and create integer vectors. We assign a unique value to each token i.e., different method names will be assigned different values. As our integer vectors have different lengths due to different files having various semantic features, we append zeroes to the vectors to make them of uniform length. Adding zeroes makes the vectors acceptable by DBN and it does not impact the results.

8.2.3 TestAdvisor^{CP}

Cross-project model learning is helpful when training data within a single project is too few to create an effective discriminative model. It addresses this cold start problem by borrowing training data from other projects. However, it remains a challenge to learn a cross-project model due to the diversity of data across projects, which causes low accuracy. To deal with this challenge, we propose a new learning strategy to improve the effectiveness of *TestAdvisor* in cross-project setting. We refer to *TestAdvisor* with the new learning strategy as *TestAdvisor*^{CP}.

In a nutshell, *TestAdvisor*^{CP} estimates the utility of various projects in a training set for cross project model learning. Data from some projects may be better than others in creating a generalizable model that can work well across projects. The utility estimates are then used to create randomly selected samples of projects of high utility. These samples are in turn used to learn multiple learning models that are then integrated to create a unified model. Figure 8.2 demonstrates the model learning steps of *TestAdvisor*^{CP} and the standard cross-project solution. Compared to standard cross-project model learning, *TestAdvisor*^{CP} randomly samples the train-

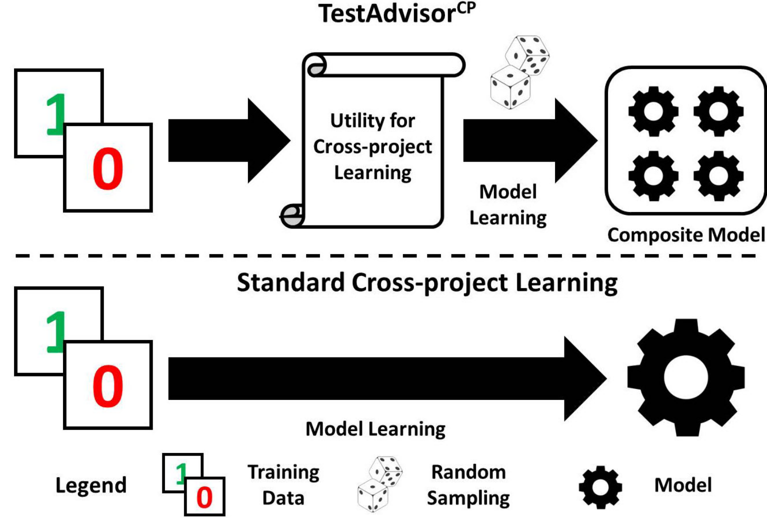


Figure 8.2: Comparison between Enhanced and Standard Cross-project Model Learning.

Algorithm 1: Enhanced Cross-Project Model Learning

Input : $T = \{T_p \mid 1 \leq p \leq N\}$: Set of training data where N is the number of projects
CLS: A classification algorithm
SP:Sampling percentage
NP:Number of samples

Output: ECP: Composite cross-project model

1 **for** $T_p \in T$ **do**
2 | Randomly select $T_{x_1}, \ldots, T_{x_{10}} \in T$ ($T_{x_i} \neq T_p$)
3 | Run CLS on T_p to train model M
4 | Deploy M on $T_{x_1}, \ldots, T_{x_{10}}$
5 | $EvaScore[T_p] \leftarrow$ *HIT@10* score
6 **end**
7 $ECP \leftarrow \{\}$
8 **for** $1 \leq i \leq NP$ **do**
9 | $n \leftarrow SP \times N$ `// sample size`
10 | Randomly pick n projects without replacement from T where projects with higher $EvaScore[T_p]$ scores have higher chance to be selected
11 | Run CLS on the selected n projects to train SM_i
12 | $ECP \leftarrow ECP \cup SM_i$
13 **end**
14 **return** ECP

---

ing data to filter out projects that are best for cross-project setting.

Algorithm 1 describes how *TestAdvisor*^{CP} works. The algorithm takes as input a training data $T = \{T_p \mid 1 \leq p \leq N\}$ consisting of N projects, a classifica-

tion algorithm CLS, sampling percentage SP, and number of samples NP. Algorithm 1 samples the training data to retain projects that are likely to be good for cross-project model learning. To achieve that, for every project p in T, Algorithm 1 randomly picks up 10 other projects x_i ($1 \leq i \leq 10$) (line 1). At lines 3 to 4, it runs CLS on all instances of p to construct a model M, and deploy M to predict labels (i.e., "test" or "not") of program elements in x_i ($1 \leq i \leq 10$). At line 4, we compute *Hit@10* score to assess the effectiveness of M. The *Hit@10* score estimates the utility of using p in cross-project setting. A low value of *Hit@10* indicates that the feature values of program elements in p are unique and solely represent the specific characteristics of p. Therefore, p is not suitable to be used in constructing a model to predict for labels of program elements in other projects. On the other hand, higher values of *Hit@10* indicate features of program elements of p are potentially good for cross-project setting. We store the *Hit@10* scores of various p in $EvaScore$.

In the next steps, *TestAdvisor*^{CP} builds multiple models from randomly selected samples of projects from T. The scores stored in $EvaScore$ are used as the criteria to include a project to the samples. At lines 8 to 12, *TestAdvisor*^{CP} creates NP samples of size $SP \times N$. For every sample, we randomly pick a number of projects from the input N projects, where the probability of a project being selected is proportional to its score. Thus, projects with higher scores stored in $EvaScore$ have more chance to be included to the sample. For every sample, we run CLS to construct a prediction model SM_i. The algorithm returns a collection of all models trained from the NP samples.

We apply Equation 8.1 to estimate the ranking score of an instance x (denoted as ECP(x)) using the collection of models as follows:

$$ECP(x) = \frac{\sum_{i=1}^{NP} SM_i(x)}{NP} \tag{8.1}$$

In the equation above, NP is the number of samples, $SM_i(x)$ is the ranking score returned by SM_i model. The final ranking score of x is the average of all ranking

scores generated by the NP SM_i models. The higher the final ranking score, the more likely the label of x is "test". By default, we set $NP = 10$ and $SP = 10\%$.

8.3 Experimental Setup

In this section, we describe the dataset we use, the evaluation metrics and baselines, and the research questions we investigate.

Dataset

We perform several steps to create our benchmark dataset. Firstly, we fetch top 2,500 most popular open-source Java projects from GitHub (sorted by the sum of their number of stars and number of forks). GitHub contains many toy projects, thus, we only consider popular projects similar to prior studies [105, 66]. We only clone the git repositories of projects which use Maven as we leverage Maven to automatically build the projects and construct call graphs from compiled classes. Out of the 2500 projects, 831 projects use Maven. Secondly, we collect 342 Apache Java projects which use Maven and are hosted on GitHub. After removing the overlapping projects, our dataset contains 1,143 projects.

Next, we ignore projects with less than 10 Java files as these projects are too small to analyse. We also filter out projects which have less than 5 tested files. For each project, we have two versions: *current* version (i.e., latest version as of June 2016) which serves as a test dataset, and *previous* version (i.e., version one year prior to *current* version) which serves as a training set. We compile these two versions using *mvn compile:compile*. We ignore projects whose current and/or previous version cannot be compiled. In the end, our dataset has 103 projects.

Table 8.2 shows the statistics of our dataset, which contains 103 projects having a total of 46 million SLOC, 83 thousand source code files, 0.9 million methods, 280 thousand commits and over 45 thousand test files contributed by more than 5 thousand developers spanning over period of 15 years, i.e., 2001-2016. We consider

Table 8.2: Study Subjects.

Number of Projects	103
Source LOC	46,700,733
Number of Source Code Files	83,694
Number of Methods	965,474
Cyclomatic Complexity	5,743,959
Test LOC	12,628,328
Number of Test Files	45,718
Test Cyclomatic Complexity	1,265,859
Number of Commits	280,101
Number of Developers	5,307
Total Period	04/2001 - 06/2016

the file that contains word ”test” in its name as a test file. Our dataset contains popular projects such as Apache Commons IO [8], which is a library of utilities for IO functionalities and Joda-Time [52], which is date and time library for Java.

Evaluation Metrics and Baselines

Metrics. Our approach outputs a ranked list of files or methods. A number of metrics can be used to calculate the accuracy of approaches producing ranked lists. We use the actual files or methods that developers test as ground truth. A good ranked list includes these ground truth files or methods early in it. We use several popular metrics:

Hit@N: This metric counts the percentage of projects with at least one ground truth file or method found in the top N (e.g., 5) of the ranked list produced by a technique.

Mean Average Precision (MAP): To compute MAP, first we compute the Average Precision (AP) as follows:

$$AP = \sum_{i=1}^{M} \frac{P(i) \times rel(i)}{\text{\#All tested files}} \tag{8.2}$$

where M is the number of retrieved files or methods, $rel(i)$ is a binary value that represents whether this file or method is tested or not. $P(i)$ is the precision at

position i of the retrieved list, which is defined as:

$$P(i) = \frac{\text{\#Ground truth files/methods at top i positions}}{\text{i}} \tag{8.3}$$

MAP is the mean of the average precisions over all the lists for the projects.

Mean Reciprocal Rank (MRR): The reciprocal rank of a project is the inverse of the rank of the first ground truth file in a ranked list. The mean reciprocal rank is the average of the reciprocal ranks for all the projects. For a set of projects Q, MRR is defined as:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{Q}\frac{1}{rank_i} \tag{8.4}$$

where $rank_i$ is the rank of the first ground truth file in a ranked list of the ith project. *Baselines.* We compare *TestAdvisor* with three baselines and two state-of-the-art defect prediction techniques at both the file and method level. Our **first baseline (B1)** considers that a file or a method should be tested if it was changed in a bug-fix commit between the current version and the previous version. We compute the number of times a file or a method has been changed in a bug-fix commit and produce a ranked list with files changed more often ranked higher. The **second baseline (B2)** is similar to B1 except we consider the complete history of the project. The **third baseline (B3)** randomly creates and returns a ranked list of files or methods. For comparison with defect prediction, we consider state-of-the-art defect prediction technique proposed by Wang et al. [126] which uses deep learning to learn semantic representation from source code. We refer to this technique as **DP1**. Similar to Wang et al. [126] we use post-release bugs i.e., bugs found after the release, for the two different versions of each project. We count all the bug fix commits from commit logs by matching keywords such as 'bug', 'fix', 'issue' etc., a similar heuristic used in the past studies [19, 105]. For our second defect prediction technique **(DP2)**, for each instance, we use term frequencies of the AST nodes. This baseline was also used by Wang et al. [126] to compare their technique. Similar to

Wang et al. [126], we use re-sampling technique Synthetic Minority Over-sampling Technique (SMOTE) [21], which has been used in many previous studies, on DP1 and DP2.

Research Questions

RQ1: How effective is TestAdvisor compared to the baselines?

In this question, we examine the effectiveness of *TestAdvisor* compared with the baselines considering within-project setting. Our aim is to investigate whether *TestAdvisor* can leverage the features we collect to learn a model that can accurately prioritize files and methods for testing. If our technique performs well, it will be able to rank ground truth files and methods higher than other files and methods. To answer this question, for each of the 103 projects, we use the previous version to train a model, and employ the model to rank files or methods in the current version. The ranked lists of files or methods which are generated are then evaluated to compute Hit@5, Hit@10, MAP, and MRR.

RQ2: How effective is TestAdvisor compared to state-of-the-art defect prediction techniques?

In this question, we examine the effectiveness of *TestAdvisor* compared with two defect prediciton techniques. Firstly, we consider state-of-the-art technique proposed by Wang et al. [126] that use deep learning (Deep Belief Network) to learn semantic features from the source code. After checking with the authors [126], we could not get the source code. We, thus, reimplemented their technique and used the same parameter values as used by the authors. Secondly, we use AST nodes that were used in our technique and each instance is represented as a vector of term frequencies of these nodes. This baseline was also used by Wang et al. [126].

RQ3: What is the effect of using different sets of features?

TestAdvisor combines six different sets of features: impact, functionality, process, ownership, code and semantic. In this question, we analyze each of the feature

sets individually. We also evaluate the value of using all the feature sets together. In particular, we investigate whether *TestAdvisor* with all feature sets performs better than if only one set is used. For each project, we create feature vectors for each feature set and follow a similar methodology as in RQ1 to evaluate the different sets of features.

RQ4: What is the effect of using different classification algorithms?

By default, we use Naive Bayes as the classification algorithm. Many other algorithms are available to produce a ranking model. In this question, we investigate the effectiveness of other classification algorithms when each of them is used inside *TestAdvisor*. We consider six popular classification algorithms: Logistic Regression, Decision Table, ADTree, J48, Random Forest, and Bayes Net.

Logistic Regression is used to model a dichotomous outcome variable by estimating probabilities using a logistic function on independent variables [24]. Decision Table consists of a hierarchical table with rows corresponding to possible actions that can be taken for each relevant condition [63]. Alternating Decision Tree (ADTree) is represented as an alternation of decision nodes, which specify a predicate condition and prediction nodes, and classification is done by traversing the paths for which these decision nodes are true [33]. J48 is an open source implementation of C4.5 algorithm [100], which uses entropy information to build decision trees. Random Forest is an ensemble learning method and aggregates the predictions made by a multitude of decision trees constructed on a training set [16]. Bayesian Network (Bayes Net) is a graphical model that represents a probabilistic relationship between between a set of random variables using a directed acyclic graph [34].

We use the implementations of these algorithms made available in Weka toolkit. We follow a similar methodology as in RQ1 to evaluate the different classification algorithms.

RQ5: How do TestAdvisor and TestAdvisor^{CP} perform in cross-project setting?

TestAdvisor^{CP} is designed to boost the effectiveness of *TestAdvisor* for cross-project setting. In this question, we evaluate the performance of these two approaches and investigate whether and to what extent *TestAdvisor*^{CP} outperforms *TestAdvisor*. For each of the 103 projects, we use the remaining 102 projects as training data. We repeat the process 103 times using different project as test data and report the average scores of the evaluation metrics.

8.4 Findings

In this section, we describe findings which answer each of our research questions.

RQ1: Effectiveness of Our Approach

Tables 8.3 and 8.4 show the Hit@5, Hit@10, MAP and MRR scores of our approach and the three baselines at file and method level, respectively.

From Table 8.3, we observe that at file level, B1, B2 and B3 achieve Hit@10 scores of 0.864, 0.942 and 0.786, respectively, whereas the corresponding score for *TestAdvisor* is higher, i.e., 0.990. The improvements in Hit@10 score of *TestAdvisor* as compared to the three baselines are 14.58%, 5.09% and 25.95%, respectively and improvements in MAP scores are 82.75%, 68.24% and 96.52%, respectively. Similarly, we find improvements in the value of Hit@5 and MRR scores for *TestAdvisor*. Since MAP is a mean of scores, we can perform Mann-Whitney Wilcoxon (MWW) test [79] to compare the MAP score of our approach with those of the three baselines. As we run the MWW test multiple times, we perform Bonferroni correction [3] to counteract the results due to multiple comparisons. We find that the difference is significant (p-value$<$0.05). We also compute Cohen's d and find that the effect size is large when the MAP score of *TestAdvisor* is compared against the three baselines.

From Table 8.4, at the method level, we observe that *TestAdvisor* outperforms the three baselines in terms of Hit@5 by 140.21%, 89.43% and 75.63% and Hit@10

by 100.00%, 52.81% and 71.43%, respectively. We observe similar improvements for MAP and MRR scores. We also run Mann-Whitney Wilcoxon test to compare the MAP scores and find that the differences are statistically significant (p-value<0.05), after performing the Bonferroni correction. The effect size using Cohen's d is large when we compare *TestAdvisor* against each of the three baselines.

Table 8.3: *TestAdvisor* versus baselines (File Level). B1 considers bug history between the *current* and the *previous* version to rank files. B2 takes the full history of the project. B3 produces a random list.

Models	Hit@5	Hit@10	MAP	MRR
B1	0.825	0.864	0.371	0.627
B2	0.845	0.942	0.403	0.680
B3	0.650	0.786	0.345	0.482
TestAdvisor	**0.961**	**0.990**	**0.678**	**0.875**

Table 8.4: *TestAdvisor* versus baselines (Method Level). B1 considers bug history between the *current* and the *previous* version to rank files. B2 takes the full history of the project. B3 produces a random list.

Models	Hit@5	Hit@10	MAP	MRR
B1	0.291	0.408	0.145	0.217
B2	0.369	0.534	0.149	0.288
B3	0.398	0.476	0.137	0.274
TestAdvisor	**0.699**	**0.816**	**0.426**	**0.623**

TestAdvisor is significantly and substantially more effective than the baselines in ranking files and methods to be tested.

RQ2: Comparison with Defect Prediction

Table 8.5 and 8.6 show the comparison of *TestAdvisor* with state-of-the-art defect prediction techniques.

From Table 8.5, we observe that *TestAdvisor* can achieve significant improvement over the two defect prediction techniques at the file level. The improvement in MAP score of *TestAdvisor* over the two baselines are 66.58% and 62.98%, respectively and improvement in MRR score are 47.55% and 46.32%, respectively. We

perform Mann-Whitney Wilcoxon (MWW) test to compare the MAP score of our approach with those of the two defect prediction techniques. Similar to RQ1, we perform Bonferroni correction for multiple-comparison correction and find that the difference is signficant (p-value<0.05). We also compute Cohen's d and find that the effect size is large when the MAP score of *TestAdvisor* is compared against the two approaches.

Table 8.5: *TestAdvisor* versus defect prediction (File Level). DP1 is state-of-the-art defect prediction using deep learning. DP2 uses the term frequencies of AST nodes.

Models	Hit@5	Hit@10	MAP	MRR
DP1 (Wang et al.)	0.689	0.816	0.407	0.593
DP2 (AST Based)	0.718	0.816	0.416	0.598
TestAdvisor	**0.961**	**0.990**	**0.678**	**0.875**

Similarly, from Table 8.6, we observe that *TestAdvisor* can achieve significant improvement at the method level. The improvement in MAP score of *TestAdvisor* over the two baselines are 204.29% and 230.23%, respectively and improvement in MRR score are 222.80% and 229.63%, respectively. After performing the Bonferroni correction, we find a significant difference (p-value<0.05) in MAP scores of our approach and the two techniques at the method level and the effect size, computed using Cohen's d, is large.

Table 8.6: *TestAdvisor* versus defect prediction (Method Level). DP1 is state-of-the-art defect prediction using deep learning. DP2 uses the term frequencies of AST nodes.

Models	Hit@5	Hit@10	MAP	MRR
DP1 (Wang et al.)	0.282	0.369	0.140	0.193
DP2 (AST Based)	0.282	0.388	0.129	0.189
TestAdvisor	**0.699**	**0.816**	**0.426**	**0.623**

TestAdvisor is significantly and substantially more effective than the baselines in ranking files and methods to be tested.

RQ3: Different Feature Sets

Table 8.7 shows the scores of various evaluation metrics when different feature sets are used to rank files. We can observe that using a combination of all feature sets performs better than if only one set is used. Among the six categories of features, code and impact features perform the best. Table 8.8 shows the corresponding scores for method level. We can observe a similar trend: The combination of all features perform best, and impact features perform better than other features at the method level.

Table 8.7: Hit@5, Hit@10, MAP and MRR scores when a particular feature is used and the combination of all features (File Level).

Features	Hit@5	Hit@10	MAP	MRR
Impact	0.893	0.951	0.604	0.795
Functionality	0.786	0.913	0.460	0.699
Process	0.847	0.913	0.513	0.715
Ownership	0.748	0.825	0.434	0.630
Code	0.970	0.990	0.637	0.867
Semantic	0.806	0.913	0.483	0.678
Combined	**0.961**	**0.990**	**0.678**	**0.875**

Table 8.8: Hit@5, Hit@10, MAP and MRR scores when a particular feature is used and the combination of all features (Method Level).

Features	Hit@5	Hit@10	MAP	MRR
Impact	0.641	0.699	0.373	0.544
Functionality	0.311	0.388	0.163	0.202
Process	0.350	0.505	0.194	0.259
Ownership	0.311	0.398	0.169	0.201
Code	0.417	0.505	0.152	0.272
Semantic	0.485	0.573	0.258	0.410
Combined	**0.699**	**0.816**	**0.426**	**0.623**

At both file and method levels, the combination of all features perform the best. Among the individual feature sets, code and impact features perform the best.

RQ4: Different Classification Algorithms

Tables 8.9 and 8.10 compare the evaluation scores of different classification algorithms at file and method levels. *TestAdvisor* uses Naive Bayes to build a ranking model and produce a ranked list. We observe that Bayes-based learning models (i.e., Naive Bayes and Bayes Net) and Random Forest, which constructs many decision trees, achieve the highest performance. Next in the line are Logistic Regression, which is a linear classifier and ADTree, whcih consists of an alternation of decision nodes. The worst performing ones are J48 and Decision Table.

Table 8.9: Hit@5, Hit@10, MAP and MRR scores for different classification algorithms (File Level).

Models	Hit@5	Hit@10	MAP	MRR
Logistic Regression	0.893	0.961	0.653	0.830
Decision Table	0.893	0.961	0.622	0.825
ADTree	0.932	0.971	0.669	0.813
J48	0.854	0.932	0.612	0.756
Random Forest	0.951	0.971	0.747	0.888
Bayes Net	0.951	0.990	0.677	0.872
Naive Bayes	**0.961**	**0.990**	**0.678**	**0.875**

Table 8.10: Hit@5, Hit@10, MAP and MRR scores for different classification algorithms (Method Level).

Models	Hit@5	Hit@10	MAP	MRR
Logistic Regression	0.709	0.816	0.443	0.632
Decision Table	0.641	0.699	0.401	0.546
ADTree	0.680	0.757	0.441	0.580
J48	0.602	0.670	0.386	0.512
Random Forest	0.718	0.806	0.473	0.651
Bayes Net	0.709	0.816	0.427	0.631
Naive Bayes	**0.699**	**0.816**	**0.426**	**0.623**

TestAdvisor can be coupled with different classification algorithms and among the algorithms investigated, the best performing ones are Naive Bayes, Bayes Net and Random Forest.

RQ5: Cross-Project Setting

Tables 8.11 and 8.12 compare the effectiveness of *TestAdvisor* and *TestAdvisor*^{CP} for the cross-project setting at file and method level respectively. At the file level, *TestAdvisor* can achieve Hit@5 and Hit@10 scores of 0.860 and 0.910, whereas the corresponding values for *TestAdvisor*^{CP} are 0.930 and 0.970. Similarly, we observe that *TestAdvisor*^{CP} can outperform *TestAdvisor* in terms of MAP and MRR scores by 21.14% and 15.20% respectively. We perform Mann-Whitney Wilcoxon test and find that the difference between MAP scores of *TestAdvisor*^{CP} and *TestAdvisor* is significant at the confidence level of 95%. We also compute Cohen's d and find that the effect size is small (but not negligible).

At the method level, *TestAdvisor*^{CP} can achieve Hit@5 and Hit@10 scores of 0.840 and 0.920, which are higher than the corresponding values of *TestAdvisor*. Comparing the MAP and MRR scores of *TestAdvisor*^{CP} and those of *TestAdvisor*, we observe that the earlier can outperform the latter by 82.51% and 71.93% respectively. We perform Mann-Whitney Wilcoxon test and find that the difference between the MAP scores is significant at the confidence level of 95%. We also compute Cohen's d and find that the effect size is medium.

These results show that our enhanced cross-project strategy is effective. This is true for both file and method levels, considering all evaluation metrics. The Hit@5, Hit@10, MAP, and MRR scores can be increased by up to 80%.

Table 8.11: Hit@5, Hit@10, MAP and MRR scores of *TestAdvisor* and *TestAdvisor*^{CP} considering cross-project setting (File Level).

Models	Hit@5	Hit@10	MAP	MRR
TestAdvisor	0.860	0.910	0.492	0.684
TestAdvisor^{CP}	0.930	0.970	0.596	0.788

Table 8.12: Hit@5, Hit@10, MAP and MRR scores of *TestAdvisor* and *TestAdvisor*^{CP} considering cross-project setting (Method Level).

Models	Hit@5	Hit@10	MAP	MRR
TestAdvisor	0.630	0.750	0.223	0.399
TestAdvisor^{CP}	0.840	0.920	0.407	0.686

TestAdvisor^{CP} can outperform TestAdvisor in the cross-project setting by a statistically significant and substantial margin.

8.5 Conclusion

In this study, I propose a "learning to test" framework named *TestAdvisor*, which automatically extracts a comprehensive set of features which can be grouped into 6 categories, i.e., impact, functionality, process, code, ownership and semantic. These features are then used to build a ranking model trained using files and methods which have been tested in a previous version or in another project. These ranking models are used to identify files and methods that require testing in a new version or a new project. I empirically evaluate *TestAdvisor* and its extension that is designed for cross project setting (i.e., *TestAdvisor^{CP}*) on a large dataset of 103 open-source Java projects collected from GitHub. Some of the findings of this study are:

1. For within-project setting, *TestAdvisor* can improve the performance of several baselines by 13.73%-140.21%, 5.10%-100.00%, 68.24%-210.95%, 28.68%-187.10% in terms of Hit@5, Hit@10, MAP and MRR, respectively. Compared with state-of-the-art defect prediction techniques, *TestAdvisor* can improve MAP scores by 62.98%-230.23% and MRR scores by 46.32%-229.63%.

2. At both file and method levels, the combination of all features perform the best. Among the individual feature sets, code and impact features perform the best.

3. *TestAdvisor* can be coupled with different classification algorithms. Among the seven classification algorithms investigated, the best performing ones are Naive Bayes, Bayes Net and Random Forest.

4. *TestAdvisor^{CP}* can outperform *TestAdvisor* in the cross-project setting by a statistically significant and substantial margin.

Dataset

Our dataset is made publicly available and it can be downloaded from: *https://github.com/smusis/learning-test*.

Chapter 9

What Make Good Test Cases?

9.1 Introduction

Test cases are a central piece in testing and practitioners put in a significant amount of time writing and maintaining them. However, despite testing effort, it is often seen that bugs appear in programs. Moreover, for many projects, testing effort continues to be high as systems evolve. These bring forward the issue of test case quality and prompt us to investigate the question of what make good test cases. Past studies mostly analyze artifacts that practitioners make (e.g., code and bug reports) rather than surveying or interviewing practitioners. The latter is often needed to get deeper insights into rationales behind practitioner actions.

In this study, we complement the existing empirical studies that investigate test case quality by conducting interviews with industrial and open-source practitioners to understand the characteristics of good test cases. We validate the hypotheses that we formulate from the interviews by doing a survey on 254 practitioners from Facebook, Microsoft, Google, LinkedIn, Salesforce, other small to large companies, and top 650 projects (ranked based on their popularity[1]) on GitHub. Our study produces 29 validated hypotheses on characteristics of good test cases in several dimensions: test case contents, size and complexity, coverage, maintainability, bug

[1] Number of stars + number of forks

detection, and others.

9.2 Methodology

Our study consists of two parts: open-ended practitioner interviews and a validation survey. The goal of the first part (described in Section 9.2.1) is to get insights into practitioner views to help us formulate a set of hypotheses. These hypotheses are then checked by the validation survey (described in Section 9.2.2) which is sent to a large number of practitioners (i.e., hundreds of them).

9.2.1 Open Ended Interviews

Participants

We contact the top 42 practitioners who contributed the most to Apache projects hosted on GitHub[2] and practitioners from our industry partner in China (i.e., Hengtian[3]) to find practitioners who are willing to spend a block of their time to get interviewed. Many Apache practitioners are highly experienced and many Apache projects are well-known. This motivates us to pick Apache practitioners as our candidate interviewees. Insigma Hengtian is a large software outsourcing provider in China. Its service include delivering test cases (i.e., test outsourcing) and solutions for its clients which include Fortune 500 companies. We pick Hengtian due to its long experience as a test outsourcing provider and our prior experience conducting research with them – c.f., [136, 134, 135]. Many practitioners in the company have created a large number of test cases for many external systems belonging to many clients coming from different industries and parts of the globe.

Following Opdenakker [90], to get more participants, we use several ways to conduct interviews: face-to-face, via Skype, via email, and via an online form. At the end, we get 5 participants from Apache who are willing to be interviewed –

[2]Ranked based on their number of commits.
[3]`http://www.hengtiansoft.com/`

either via Skype, email, or online form. These participants include members of Apache Hadoop, Hive, SystemML, Commons-Math, Sling, etc. with an average professional experience of 20 years. We also get 16 participants from Hengtian who are willing to be interviewed face-to-face or via online form. The average experience of these practitioners is 4 years. In total, we interview 13 practitioners face-to-face or via Skype, and 8 practitioners via email or online form.

Protocol

Asynchronous: via email or online form. We first ask participants some demographic questions (e.g., their number of years of professional experience, etc.). Next, we ask a set of open-ended questions including:

a) How would you define a good and a bad test case?

b) What criteria do you use to characterize a test case quality?

c) What factors do you consider while writing test cases?

d) What kinds of issues do you face in the creation and management of test cases?

The participants respond to these questions in writing via an online form or through email.

Synchronous: face-to-face or via Skype. We start the interview by describing our study and asking for permission to record the interview. Then, initial questions which are related to the participant demographics are asked. Next, we start our discussion which is *loosely* guided by a set of open-ended questions which we prepare in advance. These questions are the same questions that we ask participants who prefer to provide their responses via email or online form. We encourage the practitioners to talk in detail about any relevant topic which our questions do not cover. We ask follow up and clarification questions for answers we find interesting. At the end of the interview, we allow the participants to provide suggestions, comments, and opinions about writing better test cases. The interviews typically last between 30 minutes to 1 hour.

Table 9.1: List of Hypotheses

	Contents
H1	A good test case is specific or atomic, i.e., one test case should be testing one aspect of a requirement.
H2	Test cases in a test suite should be self-contained, i.e., independent of one another.
H3	A good test case should check for normal and exceptional flow.
H4	Test cases must perform boundary value analysis i.e., take as input values at the extreme ends of an input domain.
H5	Test cases should serve as a good reference documentation.
	Size and Complexity
H6	Most test cases should be small in size (in terms of its lines of code).
H7	Large test cases are often hard to understand and maintain.
H8	Large test cases may be needed to detect difficult bugs.
H9	A good suite contains lots of small test cases (with fewer LOC) and few large test cases.
H10	Increased complexity in a test case can lead to bugs in the test code itself.
	Coverage
H11	Code coverage is necessary but not sufficient.
H12	Code coverage should be used to understand what is missing in the tests and create tests based on that.
H13	Higher coverage does not mean that a test suite can detect more bugs.
H14	Each test case should have a small footprint, i.e., the amount of code it executes.
H15	A test case that is designed to maximize coverage is often long, not understandable and brittle (i.e., breaks easily).
H16	Designing test cases to cover different requirements is often more important than designing test cases to cover more code.
	Maintainability
H17	A good test case should be well-modularized.
H18	A good test case should be readable and understandable.
H19	Test cases should be simpler than the code being tested.
H20	Test code should be designed with maintainability in mind since evolution of code often requires changing of test code.
H21	Traceability links should be maintained between test code, requirements, and source code.
	Bug Detection
H22	A good test case should attempt to break functionality to find potential bugs.
H23	Test even the simplest things that cannot go wrong.
H24	During maintenance, when a bug is fixed, it is good to add a test case that covers it.
H25	Test assertions can help detect subtle errors that might otherwise go undetected.
H26	Adding common errors and possible causes as comments in test code is helpful to debug failures.
	Others
H27	A good test case should be designed such that its results are deterministic.
H28	Test cases in a test suite should not have side effects so running a test before or after another should not change the results.
H29	Test cases should use tags or categories, such as slow tests, fast tests etc., so as to be able to run a specific set of tests easily at a time.

Data Analysis

At the end of the interviews, we create interview transcripts manually by replaying the recordings. These transcripts are then analyzed to create a set of hypotheses. We group similar hypotheses into a small set of dimensions. Tables 9.1 lists the hypotheses that we have created divided into seven dimensions. We choose to create hypotheses that can hold true for testing at various levels of granularity (unit, integration, or system). These hypotheses are the input of the second part of our

study (i.e., validation survey).

9.2.2 Validation Survey

Respondents

In the validation survey, we try to get as many practitioners as possible to support or refute our hypotheses. We follow a multi-pronged approach to get survey respondents:

- First, we contact professionals in our network who are working for various organizations such as Facebook, Microsoft, Google, Box.com, LinkedIn, Salesforce, Infosys, Tata Consultancy Services (TCS) and many other small to large companies in various countries. We ask them to fill in our survey and distribute it to their friends and colleagues. Doing this helps us in getting diverse set of responses from industrial practitioners around the world.
- Second, we invite people working on the top 650 most popular open source projects in GitHub (based on the sum of their number of stars and number of forks). Many projects in GitHub are "toy" projects and thus similar to prior studies, e.g., [105], we only consider highly popular ones. We analyze the commit history of practitioners and rank them based on the number of commits in which a test file was added or edited. Following Zaidman et al. [143], we heuristically identify test files by looking for the occurrence of the word "Test" in the file name. We send invitations to the top 1,000 practitioners who have committed at least 10 commits in which at least a test file was changed in each commit. Doing this helps us in getting diverse set of responses from open source practitioners around the world. Of the 1,000 invitations, 64 of these are not successfully delivered and we receive 1 automatic reply notifying the receiver's absence.

In total, we receive 254 responses. The top two countries where the respondents

come from are China and United States. The professional experience of these 254 respondents vary from 0.2 years to 30 years, with an average of 6.01 years.

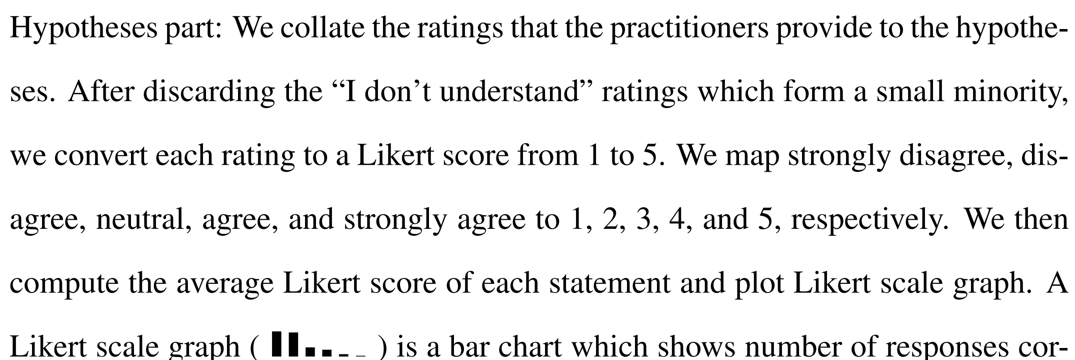
Protocol

Our validation survey consists of two parts: *hypotheses* and *rationales*. We describe them below:

1. In the first part, we present our hypotheses as statements that we ask our respondents to rate. Each respondent can rate each statement as: *strongly agree*, *agree*, *neutral*, *disagree*, *strongly disagree*, and *I don't understand*. We include the option *I don't understand* to prevent respondents providing arbitrary ratings to hypotheses that they are not clear about. Respondents can also choose not to provide any rating to any question.

2. Although ratings help us to understand respondent positions on the hypotheses, they are not sufficient for us to understand respondent reasonings. Thus, in the second part, we ask a few additional questions. First, we randomly select two statements that a respondent has rated as *strongly agree* or *agree*. We then ask the respondent the reason why he/she has provided such ratings. Second, we randomly select two statements that a respondent has rated as *strongly disagree* or *disagree* and ask he/she to provide his/her reasons. Answering these questions is optional.

Data Analysis

Hypotheses part: We collate the ratings that the practitioners provide to the hypotheses. After discarding the "I don't understand" ratings which form a small minority, we convert each rating to a Likert score from 1 to 5. We map strongly disagree, disagree, neutral, agree, and strongly agree to 1, 2, 3, 4, and 5, respectively. We then compute the average Likert score of each statement and plot Likert scale graph. A Likert scale graph () is a bar chart which shows number of responses cor-

responding to strongly agree, agree, neutral, disagree, strongly disagree, and N/A or I don't understand, respectively.

Rationale part: We collect arguments that practitioners have provided to support or refute each hypothesis. We then summarize these arguments.

9.3 Findings

In this section, we describe characteristics of good test cases. We divide the characteristics into six dimensions: test case contents, size and complexity, coverage, maintainability, bug detection, and others. For each dimension, we describe a list of hypotheses (described in Section 9.2) and their ratings. We then describe arguments that support or refute the hypothesis as provided by our interview participants and survey respondents.

9.3.1 Contents

Intuitively, the contents of a test case would significantly affect its quality. In this dimension, we investigate practitioners agreement on some hypotheses that describe characteristics of good test cases based on their contents.

Specific (H1). ▮▮▪▪_

In general, practitioners advice that a test case should be specific, i.e., it should try to test only one functionality. Out of the responses that we receive, 93 indicate strong agreement and 87 agreement with hypothesis H1. The overall Likert score is 3.94 (i.e., close to "agree"). The following are some of the comments that support or refute the hypothesis:

- ☝ *"I prefer atomic things or smaller test cases that test one thing if possible. It's easy to understand, easy to manage."*
- ☝ *"One test case should be testing one aspect of a use case."*

- *"...If you are in a scenario where test is actually taking little extra time then at least I do not see a problem in verifying multiple different things in the same test or testing multiple scenarios in the same test."*

From the above comments, we note that many practitioners support this hypothesis since specific (or atomic) test cases are easier to understand. However, in cases where tests take longer to run, testing multiple things in one test case may be a more efficient alternative.

Self-Contained (H2).

Most respondents express that test cases should be self-contained with no or minimal dependency on other test cases present in a suite. The average Likert score for this hypothesis is 3.94 (i.e., mostly "agree"). Interestingly, 47, 17, and 6 respondents neither agree/disagree (i.e., they are neutral), disagree, or strongly disagree with this hypothesis, respectively. The following are some comments that support or refute the hypothesis:

- *"The more isolated the tests, the better. You might create a library of things, the tests need to use [a] library of utilities but apart from that I prefer the test to be isolated. "*
- *"I try to be maybe have 3 or 4 instance variables within the setup and I am using the nested classes to minimize the scope so you are not sharing, not a lot of globals floating around, fairly localized to where they are used but a bit of reuse is fine I think."*
- *"Some test cases may share commonalities."*
- *"There may be inherent relationships or dependencies between test cases."*

From the above comments, again we note that respondents prefer self-contained test cases since they are easier to understand. On the other hand, we note that there is a trade-off between the simplicity achieved by self-contained test cases and reuse potentials. Respondents that disagree with this statement often highly value reuse over simplicity. Some test cases are inherently related or dependent on one another

and keeping them self-contained may mean a lot of duplication.

Consider Different Flows (H3).

Almost all of our respondents strongly agree (133 respondents) or agree (103 respondents) that it is important for test cases to check both normal and exceptional flow. The Likert score for this statement is 4.47 which is substantially higher than the scores for H1 and H2. We do not receive any comment that refutes the hypothesis. The following are comments that support the hypothesis:

- *"You focus on the happy case to verify the business functionality was needed ... then [write tests] to make sure any edge cases have been properly addressed."*
- *"A test case will typically have some assertions to check for the happy cases or you could also be testing for failure. It is important to write test cases for failure path... "*

This hypothesis seems to be more or less universally supported, at least among the practitioners whom we interview and survey. Among the five hypotheses in the content dimension, this hypothesis gathers the most support.

Perform Boundary Value Analysis (H4).

Boundary value analysis refers to testing at the boundaries between partitions of the input space, which include both valid and invalid values. This hypothesis receives the second highest support among the five hypotheses with an average Likert score of 4.24. The comments that we receive include:

- *"Test cases should be considered as a whole. Some must address nominal input with intermediate values well within the application domain, some must address nominal input with special values (zero, input at boundaries, null size), some must address invalid input in order to check errors are correctly detected."*
- *"You have always want to test corner cases because that is where things tend to go wrong. You will test for a general case and then go after specific corner cases, which can cause problems."*

- *“Too much effort for too little benefit most of the time.”*
- *“Not every situation requires boundary value analysis. Boundary value analysis should only be performed on some circumstances.”*

From the comments, we learn that many practitioners perceive that there is a higher probability of finding bugs at the boundaries of input partitions (i.e., corner cases). Thus, many of them agree with the hypothesis. However, a few respondents describe that boundary value analysis requires much effort and may not pay off and thus, they suggest to only perform it for some specific circumstances.

Serve as a Reference Documentation (H5).

Well commented, named and designed test cases may serve as a good reference documentation. Most of our survey respondents agree that test cases should be designed as such – its average Likert score is 3.93. This hypothesis however receives the lowest support among the hypotheses in this dimension. The following are some comments that we receive:

- *“I am a big fan of using tests as reference documentation. Writing readable tests so that you don’t have, if want, to document the details of an API... If you can stay at the overview level in the documentation and details in the tests, it is very efficient.”*
- *“Test cases are often written before documentation examples and should provide example use cases for functionality.”*
- *“Documentation is written for humans possibly unfamiliar with the product. Test cases are written for 1 compilers & runtimes, and 2 for people likely *intimately familiar* with the product. These are not the same group of people. Test cases can be *used* in documentation, but documentation cannot consist *solely* of test cases.”*
- *“Writing easy to understand tests is hard and is not worth. It’s better to have separate reference code, which can be runnable as tests.”*

From the comments, practitioners view test cases as a good complement to traditional documentation (e.g., API’s textual documentation). High-level overview

can be given in the documentation, while details are pushed to test cases. Also, test cases can serve as early documentation since they are often written before textual documentation. However, some practitioners push back on the idea because writing easy-to-understand test cases is hard, and they view the benefit is not worth the effort.

9.3.2 Size and Complexity

Size and complexity of test cases are important attributes to consider. The size and complexity of a piece of code have often been associated to its quality [115]. Unfortunately, no or little study has focused on test code. In this dimension, we consider five hypotheses that describe characteristics of good test cases in terms of their size and complexity, and investigate developer support, or lack of, to them.

Small in Size (H6).

A large number of respondents agree that test cases should be small whereas some are neutral or even disagree with this hypothesis (average Likert score = 3.83). Some of the comments we receive are:

- 👍 *"A good test should be short, should fit on to 10 lines or less of code, is self-contained, has a clear intent and its scope is obvious..."*
- 👍 *"I am more a fan of many small tests than few big ones."*
- 👍 *"Each test method should test one feature."*
- 👎 *"Some codes need complicated test logic to cover logics of it."*
- 👎 *"I am careful to keep the simplicity, but not care the number of lines."*

Practitioners mention that as test cases should be clear and test only one functionality, they should be small in size. However, some practitioners care for simplicity without worrying about size, and sometimes test cases can be long to cover complex logic.

Understandability and Maintainability of Large Test Cases (H7).

In general, practitioners confirm that large test cases are hard to understand and

maintain (average Likert score = 3.71):

- 👍 *"Large cases attempting to do everything at once are difficult to understand and more importantly difficult to maintain. When code changes, the tests must be rewritten, which is bad."*
- 👍 *"Test cases which are large in size and doing a lot, test cases which introduce or have any kind of synchronization are difficult to maintain in the long term."*
- 👍 *"For me it is a bad sign if test becomes long and complicated. It can also be a sign of bad design."*
- 👎 *"Large test cases are often necessary, especially in testing cases that require bootstrapping."*
- 👎 *"I think there is a case for them so long as you are clear that you are doing a random walk through the system. It is Ok to have some for smoke testing... It is the exception not the rule and it is for a particular purpose."*

From the comments, large test cases are often viewed as harder to understand and maintain since they often do several things at the same time and thus are more susceptible to changes when the SUT changes. Large test cases can also be an indication of bad design (either in the test code or in the system under test (SUT)). However, they might be required for specialized testing or for cases that require bootstrapping – these should be exceptions and not the rule though.

Large, Complex Test Cases and Difficult Bugs (H8).

Previously, practitioners express that large test cases are hard to understand and maintain (H7). In this hypothesis, we would like to confirm whether practitioners agree that large test cases can be useful to detect difficult-to-find bugs. We find that 118 respondents strongly agree or agree with this hypothesis. A substantial number of respondents choose to be neutral or disagree (70 neutral respondents, and 32 respondents who disagree or strongly disagree). The average Likert score is 3.60 which is the lowest among hypotheses in this dimension. Some of the comments that we receive are:

- “*Complex test cases will cover integration environment and they can lead to some very good bugs being discovered.*”
- “*Sometimes the most awkward bugs appear when a series of steps are happening in the code.*”
- “*...will detect less bugs ultimately because they would be harder for us to understand and maintain. It goes together with the readability factor.* ”
- “*...strategy matters, not the size of test case.*”

From the comments, many practitioners agree that long and complex test cases can detect some hard-to-find bugs since they can cover long series of steps that cannot be simulated by simple test cases. However, some practitioners disagree by stating that poor understandability will make such test cases less able to find bugs *in the long run*. Others argue that what matters is the strategy practitioners apply for testing – with a good strategy, small and simple test cases can be sufficient to find many hard-to-find bugs.

Large and Small Test Case Mix (H9).

Most practitioners are of the opinion that a test suite should contain a good mix of many short and a few large test cases (average Likert score = 3.96). Few of the comments that came out during interview and survey are:

- “*A combination of lots of small tests and some large tests is ideal but you cannot throw away large test by a lot of small tests.*”
- “*Small tests eg unit tests and large tests like fuzzers, integration tests, etc will find *different* bugs.*”
- “*This would cover most situations of requirements.*”
- “*For me it is better to have lots of [small] tests.*”

To summarize, practitioners have a high agreement that a combination of many small and a few large test cases is apt for most of the situations. However, some practitioners have a strong preference of keeping test cases short.

Complexity and Bugs in Test Cases (H10).

In our interviews, several practitioners state that test cases can often become long and hard to manage. This increased complexity of test cases can lead to bugs in the test code. A large number of our survey respondents agree with this hypothesis (average Likert score = 4.03). Some of the comments practitioners made to justify their support or lack of support are:

- *"If the test is really hard to read and understand and it is complex in its own right there is a good chance that... there is a bug in the test itself."*
- *'We might put ourselves at risk of not understanding the test when we come back to it later. Or a test failure during a refactoring that appeared decoupled from its requirement, might be modified. Because it is not clear why it fails."*
- *"Complex test cases make an environment less productive, but do not directly cause bugs."*

In general practitioners find that there is a higher likelihood of bugs appearing in the test code if the complexity of test cases increases. A few practitioners disagree though stating that the relationship between complex test cases and bugs is unclear. This hypothesis receives the maximum agreement in the size and complexity dimension.

9.3.3 Coverage

Code coverage, the amount of code covered by test cases, is often used as a measure of test quality. Coverage information can help practitioners in finding parts of the code which are not covered and might contain bugs.

Code Coverage, Necessary but Insufficient (H11).

A hundred and ninety four of our respondents support (agree or strongly agree with) this hypothesis – resulting in an average Likert score of 3.98. The following are some of their comments that support or refute the hypothesis:

- *"Code coverage is important but it is just one axis of the quality of the test."*
- *"The more coverage you got, generally speaking, the better."*
- *"It does not measure the combinatorial explosion of possible interactions."*
- *"I could certainly write tests that provide good code coverage but do not actually test what users are going to use from the software."*
- *"Code coverage is nice but not all that useful. Running a line isn't an indicator that you've tested it. Not running a line is an indicator that you haven't but you are better of caring about features. "*

In general, many practitioners find that code coverage is a good starting point as it gives information whether we have exercised a piece of code. However, some practitioners have strong skepticism against the usefulness of coverage as a quality metric. They argue that covering a code may not mean that it has been tested, and a test case that covers a code may not mimic what real users would do in practice.

Code Coverage and New Test Cases (H12).

Practitioners in general agree that coverage information can be leveraged to understand shortcomings of current test cases to write new tests (average Likert score = 3.96). Some practitioners provide these rationales:

- *"Use code coverage to understand what is missing in the tests and then create intelligent test based on that."*
- *"I look at my code coverage, I am not at 100% then I know I must have not got any tests for place order where there is probably some interesting business functionality"*
- *"I prefer to focus on features, rather than code coverage."*

From the practitioner comments, we find that code coverage can be helpful in writing test cases; however, for some practitioners, it is not the focus.

Higher Coverage and Detecting More Bugs (H13).

In general, practitioners agree that a higher coverage *does not mean* that a test suite can detect more bugs. This hypothesis receives an average Likert score of 4.02,

which is the highest for hypotheses in this dimension. More than 190 practitioners agree or strongly agree with this statement. Here are some of the comments:

- 👍 *"Because high coverage is useless unless you are also making the right assertions."*
- 👍 *"Because code coverage does not consider semantic of the code."*
- 👍 *"If you write good test case they will also increase code coverage, just that focussing exclusively on code coverage is not useful."*
- 👎 *"More coverage, less chance of bugs."*
- 👎 *"Hitting all code passes increases probability of finding edge test case that was not thought of."*

From the comments, many practitioners complain that code coverage does not consider the semantic of the code and is useless without good assertions. Also, they argue that achieving code coverage is not a good proxy to writing good test cases. However, eighteen practitioners whom we survey disagree with the statement stating that coverage has its place in detecting bugs.

Small Footprint (H14).

This hypothesis is a slightly controversial one; only a slight majority of our survey respondents (51.85%) agree or strongly agree that a single test case should have a small footprint (i.e., the amount of code it executes). Still, the average Likert score is 3.52, and thus the balance tips towards agreement with many practitioners (68 of them) on the fence. The following are the rationales that practitioners give to support or refute the hypothesis:

- 👍 *"Developer test should test a single responsibility but does not necessarily mean a method, i.e., a single responsibility of the thing on the test."*
- 👍 *"Simple. The larger the footprint the more bottlenecks there are in the testing process and the slower the testing process is."*
- 👎 *"...this is overrated. Tests should be maintainable. ...But worshipping this principle can often be the enemy of maintainable tests."*

☞ *"If you can write a simple test that covers a lot of code, that can make writing tests more efficient."*

The proponents of this hypothesis argue that a single test case should have a single responsibility. Also, test cases that cover a lot of code can cause a bottleneck and slow down the testing process. Others disagree stating that this hypothesis should not be followed rigidly; one argues that by covering as much code as possible with as few test cases, one can safe the cost of writing test cases.

Maximizing Code Coverage, and Long, Not Understandable, and Brittle Test Cases (H15).

This hypothesis is also a slightly controversial one; only 54.66% of the respondents agree or highly agree that a test case that is designed to maximize coverage is often long, not understandable and brittle (i.e., breaks easily). Although this hypothesis receives the lowest agreement among others in this dimension, the balance again tips towards agreement with an average Likert score of 3.51. The following are some comments given by practitioners:

☝ *"If you try to over-focus on code coverage people will try to go through all sorts of loops... it is quite difficult. You have to go through a lot of effort to trigger that to occur and it is not just worth the effort."*

☝ *"Because the desire for coverage often makes people lose sight of the true goal of a given test case."*

☞ *"Optimizing for coverage doesn't mean complicated tests unless the code being tested is complicated."*

☞ *"The more code I can test with a maintainable test the better."*

Most practitioners agree that focussing solely on coverage can create problems since people can often lose sight on the true goal of testing, and start doing "all sorts of loops" which can be harmful. However, 43 respondents disagree and 6 strongly disagree with the hypothesis stating that one can often optimize coverage without causing issues mentioned in the hypothesis.

Code Versus Requirement Coverage (H16).

Most practitioners agree that requirement coverage is more important than code coverage resulting in the average Likert score of 4.00. We only receive positive comments supporting this hypothesis which include:

- *"The core goal for me would not be to maximize code coverage. It will be to maximize testing basic case and corner cases for a feature."*
- *"I don't believe it is an effective use of time to test the most basic of code (getter/setter, etc...)"*
- *"Code coverage is a technical measure that isn't directly related to user-facing features. User-facing features are the actual thing that an application should care about."*

From the comments, we find that practitioners prefer requirement coverage, since some code is of little value and is less likely to be buggy (e.g., getter or setter methods). Moreover, test cases that achieve requirement coverage often mimic well how clients would use a piece of SUT.

9.3.4 Maintainability

Software system evolves and so should its test cases. Maintainability of code (including test code) is an important aspect as it helps to ensure that a software system continues to serve its intended purpose.

Well-Modularized (H17).

Most respondents agree or strongly agree that test cases should be well modularized and the average Likert score is 4.27. Only 4 respondents disagree or strongly disagree with this hypothesis. Following are some of the comments that support or refute the hypothesis:

- *"Test code is code. If the test is simple for people to understand, it should be short and simple in the code."*

- *"...It might mean that you are trying to test too much stuff at once and maybe you should break that down into smaller modules or units."*
- *"It is easier to maintain it if it is. ..."*
- *"Tests that are too modularized, tend to make debugging of regressions more complex."*

From the comments, most practitioners agree that if a test case is large, it should be broken down into smaller modules or units, since it would then be simpler to read and easier to change as a software system evolves. One drawback that a respondent mentions is too modularized test may make debugging more complex.

Readable (H18).

More than 96% of the respondents agree that test cases should be readable and understandable. Among the hypotheses in this dimension, this one receives the highest Likert score of 4.59. Practitioners give a number of supportive comments, including the following:

- *"Like any code, if you have to maintain it you better be able to understand it."*
- *"Tests reflect intent. Tests should tell a story of how the code is supposed to work. Tests are one of our best tools for understanding the way code is meant to work. Tests communicate across time to future developers about the code."*

From the comments, we find that practitioners highly value readable and understandable code. A few respondents mention that this is hard to achieve though. One of them mentions: "It is challenging to keep the unit test looking nice." We do not receive any comments that refute this hypothesis.

Simpler than Tested Code (H19).

This is yet another slightly controversial hypothesis with the lowest Likert score in this dimension (i.e., 3.67). Only 57.6% of the respondents agree or strongly agree that test code should be simpler than the tested code, while 16.3% indicate

their disagreement or strong disagreement. We receive the following rationales:

- 👍 *"If the test is complicated it is harder to understand what is the actual failure. Code could get complicated but tests never should."*
- 👍 *"If the test is more complicated than the code being tested then the API being tested is too complicated."*
- 👎 *"Sometimes a fairly simple algorithm, say A*, can have a fair number of corner cases that warrant complicated test cases."*
- 👎 *"Because sometimes test cases have a more elaborate setup and teardown requirements than the code under test."*

From the comments, although many practitioners support the hypothesis, some express their reservations. The earlier group of respondents argues that simple tests are essential, for example, for effective debugging, while the latter group argues that some functionalities have many corner cases requiring complicated tests, and others require elaborate setup and teardown requirements. The findings suggest that this hypothesis can be used as a guiding principle, barring some exceptions.

Designed with Maintainability in Mind (H20).

Most practitioners agree or strongly agree that test code should be designed with maintainability in mind (average Likert score = 4.15). Some comments which support or refute this hypothesis are:

- 👍 *"Strongly Agree, it can be less fast, but should be designed with maintainability"*
- 👍 *"...if your tests aren't maintainable the code they test isn't."*
- 👎 *"Personally I will rewrite my tests instead of changing them a lot."*
- 👎 *"Spending too much time making tests clean and maintainable is a waste of time when the requirements change and the test case is no longer applicable."*

From the comments, proponents express that maintainability is a very important property of good test cases (even more important than efficiency), and if test cases are not maintainable, the code they test is also often hard to maintain. On the other

hand, we note some reservations from a few respondents who find that designing test cases with maintainability in mind may not pay off. This is true especially for software projects for which requirement changes often; for such cases, rewriting test cases from scratch may require less effort than changing test cases many times.

Traceability Links (H21).

More than 175 practitioners agree or strongly agree that traceability links should be maintained between test cases, code and requirements (average Likert score = 3.97). Only seven respondents disagree or strongly disagree with the hypothesis. The following are some of the rationales that our respondents give to support the hypothesis:

- 👍 *"Can reduce other workload and help improve the efficiency of the team."*
- 👍 *"You can quickly locate the part needs to be updated, to make quick updates and to update documentation."*
- 👎 *"It sounds like a lot of project management overhead, which would lead to slower development velocity. "*

From the comments, we find that practitioners value traceability links as these can be used to help practitioners to quickly identify parts requiring changes when a software system evolves. However, some think maintaining such links creates significant project management overhead.

9.3.5 Bug Detection

Bug detection is one of the main reasons of writing test cases. When practitioners write a new functionality or add a piece of code, they need to test whether that code is working fine or not.

Attempt to Break Functionality (H22).

A total of 207 respondents agree or strongly agree that a test case should attempt to break a functionality. The hypothesis receives an average Likert score of 4.12. The following are some comments that we receive:

- 👍 *"We will never be able to predict the full range of crazy things users do with our product. The more ways we can think of to try to break our code, the less it will break when users actually go do crazy things."*
- 👍 *"Many bugs can be found more easily by testing edge cases that developers didn't think about. Often these bugs can have impacts on real-world workloads as well."*
- 👍 *"In a distributed system, it is common that some component can't perform the designed function well either due to network issues or machine hang etc."*
- 👎 *"First and foremost, tests should ensure the code works as expected, in the environment its expected to run. Having other negative tests is less important."*

Overall, practitioners agree that test cases should try hard to break functionalities. This can be done by testing corner cases, simulating network issues or other environment problems, or performing "crazy" things that users may do with a system. By testing for such cases, practitioners can have a stronger assurance that a system would work well in practice under diverse environments and usage patterns. Seven respondents disagree or highly disagree though and a rationale that one of them provides is testing positive cases is more important than negative ones. When practitioners are hard pressed for time, testing positive cases may matter more.

Test Even the Simplest Things (H23).

The majority of respondents agree that testing even the simplest things is valuable (average Likert score = 3.91). However, a minority of respondents (i.e., 9.80%) disagree or strongly disagree with 16.73% respondents on the fence. The following are their rationales:

- *"Even the simplest of things tested can give you useful information in the sense that it will make sure when someone makes a change in the future, even to the simplest of things like hashcode for example or an equality check, people do not actually break that in the future."*
- *"Whenever you think something cannot go wrong, it probably will."*
- *"Even write the stupid test because sometimes it is the one that will find the very stupid bugs."*
- *"It is wasting time and codes."*
- *There's a line where test cases become more of a burden to carry than the value they provide. Adding a unit test for a tautology or for something incredibly simple is simply duplication."*

The proponents argue that "the simplest things" (e.g., equals() and hashcode()) may also break sometime in the future, and people make "stupid" mistakes. The opponents on the hand argue that testing simplest things may not add much value and adding them is a waste of time and code.

Add New Test Cases For Fixed Bugs (H24).

We receive a high agreement for this hypothesis (average Likert score = 4.40), which is the highest for this dimension. We only receive positive comments, which include the following:

- *"If there is a bug in the code, then writing the test helps to clarify what the error is."*
- *"The test should be written *before* fixing the bug, to ensure you actually understand the bug. Then, once the bug is fixed, you *have* the test, so keep it."*
- *"If a bug happened once, it can happen again."*

Practitioners support this hypothesis since writing test cases helps one to understand a bug. Moreover, the generated test case can help to ensure that the bug will not happen again without being detected.

Use Assertions to Detect Subtle Errors (H25).

A test assertion contains an expression which describes a property that should be (or should never be) observed for a system under test. Most of our respondents agree that assertions are a crucial part of test code and can be helpful in detecting subtle errors (average Likert score = 4.02). We present some practitioner comments below:

- *"They can do that if you push the envelope a bit in the testing. If you don't just stick to the normal case."*
- *"Yes because something you might be taking for granted to be true could very well be false."*
- *"You need something to fail, you need to have assertions in a test otherwise you are just exercising the system and not making any statements about what it should be doing."*

Practitioners argue that assertions are essential and one cannot only rely on the appearance of exceptions alone to detect failures. However, expressions used in the assertions need to be designed well so that they can detect bad cases effectively.

Commenting Test Code with Common Errors and Possible Causes (H26).

A large number of our respondents (i.e., 193) agree or strongly agree that commenting test code with common errors and possible causes is a good idea (average Likert score = 3.99). A few disagree though. Following are some of the comments:

- *"The name often will say the scenario I am trying to test out or there will be; this is especially true for complicated test, where if I write a test today and go back a month later, I think it is going to be difficult to understand what I am trying to test."*
- *"...comments is usually a convenient way to document those things..."*
- *"New people don't know your code/history."*
- *"Comments are not executable, and thus, not self-validating. This applies to comments in test code as much as comments in SUT code."*

“*Comments that are outdated can do more harm than good. If the comments are misleading then they can cause people to waste time exploring dead ends.*”

Proponents argue that this is a good practice to help one understands test code when he/she needs to revisit it again; it is also helpful for others who are not the original writer of the test code. On the other hand, others argue that the comments may get outdated and cause more harm than good.

9.3.6 Others

Deterministic (H27).

Most practitioners we survey agree that a good test case should be deterministic and produce the same output every time it is run (average Likert score = 4.05). However, again a few disagree. The following are some of their explanations:

“If a test involves some aspect of randomness, it can be very hard if not impossible to reproduce a failure”

“If tests pass or fail due to random factors then they get ignored and become useless.”

“Sometimes it is good to see transient failures to detect a race condition, for example.”

From the comments, we can infer that non-determinism in a test case can make it harder to debug when the test case fails, and the test case may then be rendered useless; however, in some cases, such tests may be useful to detect concurrency issues such as race condition.

Side Effect Free (H28).

Almost all our respondents agree that test cases should be side effect free. In this dimension, we receive the highest agreement for this hypothesis with 203 respondents agreeing or strongly agreeing with it. We present some of the comments below:

☝ *"If tests impact each other, it becomes extremely hard to reproduce and interpret test failures."*

☟ *"Test cases can not guarantee the absence of side effects, but it can be reduced."*

Side effect free test cases make debugging easier when failure happens. However, at times it is hard to guarantee complete absence of side effect.

Tag Test Cases as Slow or Fast (H29).

Several common testing frameworks like JUnit provide the functionality of adding tags to test cases. Most of our respondents agree or strongly agree that the use of such tags to indicate, for example, fast or slow tests, is helpful (average Likert score = 3.93). There are many who are on the fence though (i.e., 61 respondents). We only receive positive comments and the following are some of them:

☝ *"It is very important to have fast tests and if you have slow tests, maybe define tags or categories. It can be the fast ones and slow ones are activated by a different switch."*

☝ *"For practitioners' convenience when debugging suite-wise problems or regressions."*

☝ *"I usually use BDD develop my project. And it's important to me that it is running fast test when I am developing and more detailed but slower test before I commit my code."*

From the comments, we find that tags can be helpful as they support running of selective tests which can make practitioners complete their tasks faster. Practitioners can run fast test cases for quick verification and slow test cases can be run occasionally.

9.4 Implications

For Researchers: Our research suggests new directions for empirical software engineering researchers. Developer perception matters [56, 131, 27, 59] but they may

not always be correct [26]. Moreover, some of the hypotheses are slightly controversial with two sizable camps for and against them. For example, hypothesis H14 (each test case should have a small footprint, i.e., the amount of code it covers) only receives an average score of 3.52 and is only supported by 51.85% of our respondents. One way to nicely augment our study is to mine software repositories and analyse history of projects to get a deeper understanding of such slightly controversial hypotheses. For example, one can correlate test case footprint with its effectiveness to find bugs based on historical data. Another way, is to perform controlled experiments or field studies, and investigate the correlation between test case footprint and the time it takes for debugging test case failures and/or maintaining test cases. Clearly, it is not possible to perform all such studies and describe them in one paper. Thus, we encourage others to perform such future studies to provide further empirical evidence to further support or refute our hypotheses.

Our results also highlight opportunities for automated software engineering researchers to build tools that can help practitioners create better test cases:

- One can envision a tool that can detect smells in test code by looking for violations of some of the 29 hypotheses, especially those that receive high average Likert scores.
- From the ratings and comments that we receive for H17 and H18, many practitioners value well-modularized, well-written and well-commented test code which follows a consistent coding style. However, creating such test cases is a challenging task. Automated tools can potentially be built to suggest suitable test code refactoring or renaming to improve the modularity, readability, and understandability of test cases. To the best of our knowledge, no such tool currently exists.
- From ratings and comments that we receive for H16, we find that practitioners value requirement coverage more than code coverage. Unfortunately, to the best of our knowledge, there is no tool that can take a requirement document expressed in natural language and generate a set of maintainable test cases

from it. Existing work on automated test case generation [91, 32, 2] mainly focus on generating test cases that can cover more code. Recently, Jensen et al. propose a domain specific language for practitioners to express business rules which can then be converted to tests [48]. However, most requirements are in the form of natural language and converting them to domain specific rules may take much time and effort.

- From ratings and comments that we receive for H21, practitioners value traceability links between test cases, source code, and requirements. However, for many projects, these links may not have been made explicit and kept up-to-date. Past studies have looked into recovering traceability links between source code and requirements by employing information retrieval [22] and future tools can extend these existing works by incorporating static analysis to infer and maintain 3-way links between test code, source code, and requirements.

For Practitioners: Novices are often unsure on characteristics of good test cases and what factors they need to consider to write such test cases. Our findings provide a list of characteristics that matter to experienced practitioners. The average Likert score of all the hypotheses are above 3.5 (somewhat/close to "agree") and 12 hypotheses are above 4.0 (between "agree" and "strongly agree"). The top 5 hypotheses agreed by most respondents are: H3, H17, H18, H24 and H28. We encourage novices to consider these important factors when designing test cases. For example, following H3, they should check for both normal and exception flow, and following H28, test cases should not have side effects.

Our survey respondents consist of experienced practitioners, and they disagree on a number of hypotheses. Our results present different practitioner perspectives which often highlight tradeoffs and special circumstances. For example, based on practitioner ratings and comments for H23, we find that testing "simplest things" may detect future problems or "stupid" mistakes, but these "simplest things" may be large in number and testing them (e.g., hashcode(), equals() methods) may consume

much time and resources. For H26, we find that commenting test code with common errors and possible causes may be helpful to aid understanding, but these comments may also be a source of problems if they get outdated. For H1, most respondents agree that a test case that tests one aspect of a requirement is good since the test case would be easier to understand; however, for test cases that require long time to run, putting many things in one test may have its place. Our findings bring up such tradeoffs and special considerations which may not be obvious to even experienced practitioners (and thus the difference in opinions).

Chapter 10

Conclusion and Future Work

Testing and debugging are two important activites during software development lifecycle. While testing deals with writing and running test cases to prevent bugs, debugging deals with finding location of a bug when an issue is reported in the issue tracking system. With increasing size and complexity of software, there is an increased need to find issues with the current testing and debugging techniques and at the same time undestand practitioners' view points to bring forward the gap between practitioners' expectations and the research ouput.

This dissertation sheds light on various aspects of testing and degugging: adoption and adequacy of testing, testing culture, researchers' bias and practitioners' expectations of bug localization, designing good test cases and helping developers make testing decisions. I provide a quick recap of the empirical studies I have conducted and recommendations for researchers and practitioners.

10.1 Summary

Adoption and Adequacy of Testing

In chapters 3 and 4, I presented large-scale studies on adoption and adequacy of testing in open-source projects. Using a dataset of over 20,000 projects, I tried to understand popularity of test cases, correlations between test cases and various met-

rics such as developers, bug count, bug reporters and programming languages. I find that over 60% of the projects contain test cases, projects with test cases have higher LOC than those without test cases and projects with more number of developers have more test cases. Furthermore, projects written in popular languages such as C++, ANSI C and PHP have higher mean numbers of test cases.

In another study on over 300 projects, I studied the adequacy of testing and found that average coverage is only 41.96% and median coverage is only 40.30%. Furthermore, I analysed correlations between coverage and various metrics such as LOC, cyclomatic complexity, number of developers and CK metrics (DIT, CBO, LCOM, NOC and RFC) at the project and file level. At the project level, there is a weak correlation between coverage and all other metrics except number of developers. On the contrary, at the file level, there is a weak positive correlation between coverage and LOC, complexity, CBO, NOC, RFC and no correlation between coverage and metrics LCOM and number of developers.

Understanding the Testing Culture

In chapter 5, I presented a study to understand the testing culture of app developers in open-source and industry. First, I measured the current state of testing in Android apps by collecting 627 apps and find that the mean and median values of line and block coverage are very low. Then, I surveyed Android developers to understand current testing tools used and challenges faced by them. Popular tools such as JUnit, MonkeyRunner, Robotium etc. are widely used. Challenges often faced by these developers are time constraint, compatibility issues, lack of exposure to tools, cumbersome usage, lack of support, unclear benefits and poor documentation among others. A survey on Microsoft developers show that they commonly use automated testing tools for executing test cases, finding potential bugs, analysing code coverage, performing load testing, generating test cases etc. and manjority of them prefer using internal tools. Furthemore, challenges faced by Microsoft developers overlap with the challenges faced by Android developers.

Bug Localization: Researchers' Bias and Practitioners' Expectations

To bring forward the gap between practitioners' expectations and current research output, I presented two studies in chapter 6 and 7. First, I studied three different biases that can have a potential impact on bug localization. These biases are wrongly classified bug reports, already localized reports and incorrect ground truth files. Through an analysis of over 5,000 bug reports, I found that out of the three biases, already localized reports have a significant impact on bug localization. The files that are already localized, i.e., the bug report contains name of one or all of the buggy files, must be removed before running bug localization technique. Second, I surveyed over 300 practitioners spread in more than 30 countries to understand their expectations and thresholds for adoption of bug localization technqiues. We find that although practitioners are enthusiastic about research in fault localization, they have high thresholds for adoption. Practitioners expect a fault localization technique to satisfy some criteria in terms of debugging data availability, granularity level, trustworthiness (reliability), scalability, efficiency, ability to provide rationale, and IDE integration. Furthermore, from a literature review of research papers published in the last 5 years on fault localization, I find that there is a need to make state-of-the-art fault localization techniques more trustworthy, scalable, able to provide insightful rationales, and integrated to popular IDEs.

Learning to Test

In chapter 8, I proposed a technique that takes as input several features: impact, functionality, process, ownership, code and semantic to provide recommendations to developers on program elements to test. I also proposed an enhanced cross-project model learning to deal with cold start problem and use data from a different project. Using this technique on over 100 projects, I proved that the technique performs better than several baselines and state-of-the-art defect prediction techniques.

Good Test Cases

In chapeter 9, I presented a work where I conducted interviews and surveys with practitioners to understand what makes good test cases. Initially, I conducted interviews with industrial and open-source practitioners to understand the characteristics of good test cases. These results were validated using a set of hypotheses and a survey responded by 254 practitioners. In the end, several categories were identified to check test cases: test case contents, size and complexity, coverage, maintainability, bug detection, and others.

10.2 Future Direction

Empirical Validation of Good Test Cases

Test cases are a central piece in testing and practitioners put in a significant amount of time writing and maintaining them. I interviewed and surveyed many practitioners to understand what make good test cases across different dimensions: content, size and complexity, coverage, maintainability, bug detection and other. The results provide practitioners' belief and insights on how to design better test cases. Next step in this direction would be to collect a large dataset and validate claims of practitioners. This will help us understand the gap between practitioners' expectations of good test cases and the current state-of-practice. Furthermore, these insights can be used by tool builders to generate automated test cases that are in line with practitioners' expectations.

Longitudinal Study on Testing

Testing is a continuous process overlapped with software development. In this thesis, I presented several studies on testing, coverage and test suite effectiveness considering a particular snapshot or point in time. These studies can be complemented by performing a longitudinal analysis to understand the impact of testing and cover-

age on number of bugs *over time*. Furthermore, in addition to correlation as considered in this thesis, studying causation can shed more light on circumstances when testing has a significant impact.

Bibliography

[1] Survey form. https://github.com/smusis/automated-debugging. Accessed: 2016-11-20.

[2] Agitar One, http://www.agitar.com/solutions/products/automated _junit_generation.html, (Last Accessed on March 9, 2016).

[3] H. Abdi. Bonferroni and sidok correlations for multiple comparisons. *Encyclopedia of Measurement and Statistics*, 2007.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[5] S. Al-Gahtani and M. King. Attitudes, satisfaction and usage: Factors contributing to each in the acceptance of information technology. *Behaviour and Information Technology*, 18:277–297, 4 1999.

[6] N. Ali, A. Sabané, Y. Gueheneuc, and G. Antoniol. Improving bug location using binary class relationships. In *12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 174–183, 2012.

[7] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[8] Apache. Apache commons io. In *https://github.com/apache/commons-io*, Last accessed on February 25, 2017.

[9] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Fault localization for dynamic web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):314–335, 2012.

[10] G. K. Baah, A. Podgurski, and M. J. Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 146–156. ACM, 2011.

[11] A. Begel and T. Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *36th International Conference on Software Engineering (ICSE)*, pages 12–13, 2014.

[12] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, (ESEC/FSE)*, pages 179–190, 2015.

[13] Y. Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1):1–127, 2009.

[14] S. Berner, R. Weber, and R. K. Keller. Enhancing software testing by judicious use of code coverage information. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 612–620, 2007.

[15] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 133–143, 2013.

[16] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[17] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *14th International Systems and Software Product Line Conference (SPLC)*, pages 241–255, 2010.

[18] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 378–381, 2012.

[19] C. Casalnuovo, P. Devanbu, A. Oliveira, V. Filkov, and B. Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 755–766, 2015.

[20] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 452–462, 2012.

[21] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16(1):321–357, 2002.

[22] J. Cleland-Huang and J. Guo. Towards more intelligent trace retrieval algorithms. In *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pages 1–6, 2014.

[23] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Hillsdale: Lawrence Erlbaum, 1988.

[24] D. R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):215–242, 1958.

[25] T. Debatty. Java string similarity. In *https://github.com/tdebatty/java-string-similarity*, Last accessed on February 25, 2017.

[26] P. Devanbu, T. Zimmermann, and C. Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 108–119, 2016.

[27] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 50–60, 2015.

[28] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering (TSE)*, pages 797–814, 2000.

[29] R. A. Fisher. On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85:87–94, 1922.

[30] E. Foundation. Eclipse Java Development Tools (JDT). In *http://www.eclipse.org/jdt/*, Last accessed on February 25, 2017.

[31] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[32] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, 39(2):276–291, 2013.

[33] Y. Freund and L. Mason. The alternating decision tree learning algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML)*, pages 124–133, 1999.

[34] N. Friedman, D. Geiger, and M. Goldszmidt. Bayesian network classifiers. *Machine Learning*, 29(2-3), 1997.

[35] Gartner. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. In *http://www.gartner.com/newsroom/id/2665715?fnl=search*. Last accessed on October 22, 2014.

[36] G. Gill and C. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering (TSE)*, 17(12):1284–1288, 1991.

[37] W. H. Gomaa and A. A. Fahmy. A survey of text similarity approaches. *International Journal of Computer Applications (IJCA)*, 68(13):13–18, 2013.

[38] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. RERAN: timing-and touch-sensitive record and replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 72–81, 2013.

[39] D. Gopinath, R. N. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 40–49, 2012.

[40] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 72–82, 2014.

[41] G. Gousios. Java callgraph. In *https://github.com/gousiosg/java-callgraph*, Last accessed on February 25, 2017.

[42] M. Greiler, A. van Deursen, and M.-A. D. Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[43] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155. IEEE Press, 2012.

[44] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 392–401, 2013.

[45] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, July 2006.

[46] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 191–200, 1994.

[47] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 435–445, 2014.

[48] S. H. Jensen, S. Thummalapenta, S. Sinha, and S. Chandra. Test generation from business rules. In *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[49] W. Jin and A. Orso. BugRedux: reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 474–484, 2012.

[50] W. Jin and A. Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–223, 2013.

[51] W. Jin and A. Orso. Automated support for reproducing and debugging field failures. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(4):26:1–26:35, 2015.

[52] Joda.org. Joda-time. In *https://github.com/JodaOrg/joda-time*, Last accessed on February 25, 2017.

[53] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[54] Jung. Jung library,. In *http://jung.sourceforge.net/*, Last accessed on February 25, 2017.

[55] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 351–360, 2011.

[56] R. Khadka, B. V. Batlajery, A. M. Saeidi, S. Jansen, and J. Hage. How do professionals perceive legacy systems and software modernization? In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 36–47, 2014.

[57] S. Khoshnood, M. Kusano, and C. Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 165–176, 2015.

[58] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering (TSE)*, 39(11):1597–1610, 2013.

[59] M. Kim, T. Zimmermann, R. DeLine, and A. Begel. The emerging role of data scientists on software development teams. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 96–107, 2016.

[60] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering (TSE)*, 40(7):633–649, 2014.

[61] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 481–490, 2011.

[62] Y. W. Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 145–155, 2003.

[63] P. J. H. King. Decision tables. *The Computer Journal*, 10(2):135–142, 1967.

[64] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo. Understanding the test automation culture of app developers. In *8th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[65] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 803–814, 2014.

[66] P. S. Kochhar, D. Wijedasa, and D. Lo. A large scale study of multiple programming languages and code quality. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 563–573, 2016.

[67] M. Kropp and P. Morales. Automated GUI testing on the Android platform. *IMVS Fokus Report*, 24(4), 2010.

[68] A. Kumar Maji, K. Hao, S. Sultana, and S. Bagchi. Characterizing failures in mobile oses: A case study with Android and Symbian. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 249–258, 2010.

[69] T. B. Le, D. Lo, and F. Thung. Should I follow this fault localization tool's output? - automated prediction of fault localization effectiveness. *Empirical Software Engineering (EMSE)*, 20(5):1237–1274, 2015.

[70] T. B. Le, F. Thung, and D. Lo. Predicting effectiveness of IR-based bug localization techniques. In *25th IEEE International Symposium on Software Reliability Engineering, (ISSRE)*, pages 335–345, 2014.

[71] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, 2015.

[72] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE, 2012.

[73] W. Lewis, R. Agarwal, and V. Sambamurthy. Sources of influence on beliefs about information technology use: An empirical study of knowledge workers. *MIS Quarterly*, 27:657–678, 4 2003.

[74] D. Lo, N. Nagappan, and T. Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 415–425, 2015.

[75] F. S. Ltd. Sourcemeter. In *https://www.sourcemeter.com/*.

[76] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[77] Lucia, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 74–77, 2012.

[78] A. MacHiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 224–234, 2013.

[79] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[80] L. Mariani, F. Pastore, and M. Pezze. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering (TSE)*, 37(4):486–508, 2011.

[81] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering (TSE)*, 2(4):308–320, 1976.

[82] T. McDonnell, B. Ray, and M. Kim. An empirical study of API stability and adoption in the Android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM)*, pages 70–79, 2013.

[83] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 554–564, 2013.

[84] A. Memon, A. Porter, and A. Sussman. Community-based, collaborative testing and analysis. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER)*, pages 239–244, 2010.

[85] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[86] A. Mockus, N. Nagappan, and T. T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 291–301, 2009.

[87] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: An empirical case study. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 521–530, 2008.

[88] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[89] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 263–272. IEEE, 2011.

[90] R. Opdenakker. Advantages and disadvantages of four interview techniques in qualitative research. *Forum: Qualitative Social Research, (Last accessed on March 9, 2016)*, 7(4), 2006.

[91] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA)*, pages 815–816, 2007.

[92] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[93] S. Park, B. M. M. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 35:1–35:11, 2012.

[94] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209, 2011.

[95] F. Pastore, L. Mariani, and A. Goffi. Radar: A tool for debugging regression problems in c/c++ software. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 1335–1338, 2013.

[96] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler. Dynamic analysis of upgrades in c/c++ software. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 91–100, 2012.

[97] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 2016.

[98] H. Pham and X. Zhang. NHPP software reliability and cost models with testing coverage. *European Journal of Operational Research*, 145(2):443–454, 2003.

[99] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3):19:1–19:29, 2012.

[100] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[101] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 424–434. ACM, 2014.

[102] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (ESEC/FSE)*, pages 322–331. ACM, 2011.

[103] S. Rao and A. C. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the 8th International Working Conference on Mining Software Repositories (MSR)*, pages 43–52, 2011.

[104] S. Rao, H. Medeiros, and A. Kak. An incremental update framework for efficient retrieval from software libraries for bug localization. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 62–71. IEEE, 2013.

[105] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 155–165, 2014.

[106] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering*, pages 485–495. IEEE Press, 2012.

[107] J. Roβler, G. Fraser, A. Zeller, and A. Orso. Isolating failure causes through test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 309–319, 2012.

[108] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the Android market. In *20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122, 2012.

[109] J. R. Ruthruff, M. M. Burnett, and G. Rothermel. An empirical study of fault localization for end-user programmers. In *27th International Conference on Software Engineering (ICSE)*, pages 352–361, 2005.

[110] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355, 2013.

[111] G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 303–313, 2015.

[112] B. Sisman and A. C. Kak. Assisting code search with automatic query reformulation for bug localization. In *10th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 309–318. IEEE, 2013.

[113] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[114] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems Journal*, 2002.

[115] R. Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering (TSE)*, 29(4):297–310, 2003.

[116] C. Sun and S. Khoo. Mining succinct predicated bug signatures. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 576–586, 2013.

[117] M. D. Syer, M. Nagappan, B. Adams, and A. Hassan. Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open source Android apps. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, pages 283–297, 2013.

[118] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an Android application. In *Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 377–386, 2011.

[119] M. Talukder and A. Quazi. The impact of social influence on individuals' adoption of innovation. *Journal of Organizational Computing and Electronic Commerce*, 21:111–135, 2011.

[120] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 2002.

[121] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 189–206, 2011.

[122] F. Thung, T.-D. B. Le, P. S. Kochhar, and D. Lo. Buglocalizer: Integrated tool support for bug localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 767–770, 2014.

[123] F. Thung, D. Lo, and L. Jiang. Automatic recovery of root causes from bug-fixing changes. In *20th Working Conference on Reverse Engineering (WCRE)*, pages 92–101. IEEE, 2013.

[124] M. J. ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification & Reliability*, 17:95–133, 2007.

[125] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, 2015.

[126] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 297–308, 2016.

[127] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC)*, pages 53–63, 2014.

[128] S. Wang, D. Lo, and L. Jiang. Understanding widespread changes: A taxonomic study. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 5–14, 2013.

[129] A. H. Watson, T. J. Mccabe, and D. R. Wallace. Structured testing: A software testing methodology using the cyclomatic complexity metric. In *National Institute of Standards and Technology (NIST)*, 1996.

[130] W. Wen. Software fault localization based on program slicing spectrum. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1511–1514. IEEE Press, 2012.

[131] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 260–271, 2015.

[132] W. E. Wong and V. Debroy. A survey of software fault localization. In *Technical Report UTDCS-45-09*, 2009.

[133] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–214, 2014.

[134] X. Xia, D. Lo, P. S. Kochhar, Z. Xing, X. Wang, and S. Li. Experience report: An industrial experience report on test outsourcing practices. In *26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 370–380, 2015.

[135] X. Xia, D. Lo, J. Tang, and S. Li. Customer satisfaction feedback in an IT outsourcing company: a case study on the insigma hengtian company. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 34:1–34:5, 2015.

[136] X. Xia, D. Lo, F. Zhu, X. Wang, and B. Zhou. Software internationalization and localization: An industrial experience. In *18th International Conference on Engineering of Complex Computer Systems (ICECCS))*, pages 222–231, 2013.

[137] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):31:1–31:40, 2013.

[138] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 191–200, 2014.

[139] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 52–63. ACM, 2014.

[140] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 689–699, 2014.

[141] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):19:1–19:29, 2013.

[142] M. Young and M. Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2005.

[143] A. Zaidman, B. V. Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering (EMSE)*, 16:325–364, 2011.

[144] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 312–321. IEEE Press, 2013.

[145] L. Zhao and S. Elbaum. A survey on quality related activities in open source. *ACM SIGSOFT Software Engineering Notes*, 25(3):54–57, 2000.

[146] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 14–24, 2012.

[147] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 415–425, 2015.