SONG, YOONKI. More Usable Recommendation Systems for Improving Software Quality. (Under the direction of Emerson Murphy-Hill.)

Recommendation systems have a well-established history in both the research literature and industrial practice. Recommendation systems for software engineering (RSSEs) are emerging to assist developers in completing various software engineering tasks. However, most RSSEs have focused on finding more relevant recommendation items and improving ranking algorithms, without improving the usability of RSSEs.

In this dissertation, I present three more usable RSSEs of software testing (called UnitPlus and Viscovery) and bug fixing (called FixBugs) for improving software quality. First, UnitPlus is designed to assist developers in writing unit test code more efficiently by recommending code. I have conducted a feasibility study for UnitPlus with four open source projects in Java to demonstrate its potential utility. The results indicate the test code can be improved by the code recommended by UnitPlus. Second, Viscovery is a structural coverage visualization tool that aims to help developers understand problems that Dynamic Symbolic Execution (DSE) tools face and to provide recommendations for developers to deal with those problems. In an experiment with 12 developers, I found Viscovery helps developers use DSE tools 50% more correctly, 29% more quickly, and substantially more enjoyably than existing DSE tools and DSE tools with textual problem-analysis results. Third, FixBugs is an interactive code recommendation tool that is designed to recommend code to fix software defects. The results of an experiment with 12 developers shows that the approach is as effective as non-interactive approaches, faster in some cases, and leads to greater user satisfaction. Finally, I present guidelines learned from the three tools to improve the usability of RSSEs.
This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License.
More Usable Recommendation Systems for Improving Software Quality

by

Yoonki Song

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2014

APPROVED BY:

Alexander Dean

Christopher Healey

Laurie Williams

Emerson Murphy-Hill

Chair of Advisory Committee
DEDICATION

To my family.
BIOGRAPHY

Yoonki Song is a member of the Developer Liberation Front (DLF) research group in Computer Science at North Carolina State University. He completed his Master of Science degree in 2002 from the Korea Advanced Institute of Science and Technology (KAIST) in Information and Communications Engineering and his Bachelor of Engineering from Chonbuk National University in Computer Engineering in 2000. His research interest is software engineering with a primary goal to develop techniques and tools that can assist developers in improving both software quality and productivity. Before joining the PhD program, he had three years of professional experience in the design, development, and testing of software systems at the Electronics and Telecommunications Research Institute (ETRI). During his PhD program, his internship experiences include International Business Machines (IBM), Research Triangle Park, NC, USA (Summer 2008 - Summer 2009) and Google Summer of Code, Mountain View, CA, USA (Summer 2010). He is a recipient of the Best Student Paper Award at Eclipse Technology Exchange (ETX) in 2007.
ACKNOWLEDGEMENTS

Many people have supported and encouraged me throughout this journey.

First of all, I would like to sincerely thank my advisor, Dr. Emerson Murphy-Hill, for his guidance during my doctoral studies. My research has benefited from his encouragement, patience, and depth and breadth of knowledge. Second, I would like to thank my dissertation committee members including Dr. Alexander Dean, Dr. Christopher Healey, and Dr. Laurie Williams for their comments and suggestions in improving my dissertation work. In addition, I would like to thank Dr. Tao Xie for providing guidance early on my Ph.D. research.

I am also grateful to my research collaborators: Nikolai Tillmann and Peli de Halleux at Microsoft Research for their support for Pex and Robert Bowdidge at Google for his support. I want to thank David Chadwick and Mark Dunn at IBM for providing opportunities to involve a software team as an intern and develop modules for a software system.

My research work was supported in part by an IBM Jazz Innovation Award, NSF grants CCF-0725190 and CCF-0845272, ARO grants W911NF-08-1-0443 and W911NF-08-1-0105, and a Google Research Award. I am grateful to the Department of Computer Science for teaching assistantship opportunity in the Spring 2010 semester and financial support during my PhD.

I am very much thankful to all my peers at the Developer Liberation Front (DLF) research group for their constant help, insightful discussion, and valuable feedback on my research: Titus Barik, Michael Bazik, Samuel Christie, Daniel Franks, Xi Ge, Brittany Johnson, Kevin Lubick, John Majikes, Feifei Wang, and Jim Witschey. Especially I acknowledge Titus for his significant assistance in the writing of Chapter 4.

I would like to thank all my dear friends and colleagues at North Carolina State University for their support when I had hard time in the course of my PhD work. Special thanks to Jeehyun Hwang, Sang Yeol Kang, Donghoon Kim, Jaewook Kwak, Da Young Lee, Madhuri Marri, Patrick Morrison, Young Hyun Oh, Rahul Pandita, Kiran Shakya, John Slankas, Will Snipes, Kunal Taneja, Suresh Thummalapenta, Shundan Xiao, Xusheng Xiao, and many others.
In addition, I would like to thank all participants in my user studies.

Lastly, and most importantly, I am very thankful to my family for having faith in me and encouraging me for all these years. My family Gwansoon Ban (late grandmother), Jeongil Song (late father), Jeongsim Kim (mother), Hyunsun Song (elder sister), Hyuncheol Song (younger brother), and Jeonghyun Song (younger sister) were my constant source of love, care, and support throughout my PhD.
# TABLE OF CONTENTS

**LIST OF TABLES**  ............................................. ix

**LIST OF FIGURES**  ............................................ x

Chapter 1 Introduction  ........................................... 1
  1.1 The Problem: Lack of Usability of RSSEs for Improving Software Quality  .... 1
  1.2 Approach: Providing More Usable Recommendation Systems  .................. 3
    1.2.1 UnitPlus: Recommending Code to Write Test Code Efficiently  .......... 3
    1.2.2 Viscovery: Recommending Solutions to Assist Structural Test Generation 4
    1.2.3 FixBugs: Recommending Code to Fix Software Defects  .................... 4
  1.3 Thesis Statement and Contributions  ...................................... 4
  1.4 The Structure of The Dissertation  ....................................... 5

Chapter 2 Recommending Code to Assist Developers in Writing Test Code  ...... 7
  2.1 Introduction  .................................................. 7
  2.2 Motivating Example  ........................................... 10
  2.3 Approach  ................................................................ 13
    2.3.1 Driver Generator  ........................................... 16
    2.3.2 Side-Effect Analyzer  ....................................... 17
    2.3.3 Observer Method Recommender  .................................. 19
    2.3.4 Test Code Miner  ............................................ 19
    2.3.5 MUT Learner  ................................................ 21
    2.3.6 Implementation  .............................................. 23
  2.4 Feasibility Study  ............................................... 25
    2.4.1 Study Setup  ................................................. 25
    2.4.2 Data Collection and Analysis  ................................... 26
    2.4.3 Results: Identification of State-Modifying and Observer Methods  ....... 26
  2.5 Discussion  ....................................................... 29
  2.6 Related Work  .................................................... 30
    2.6.1 Test Code Augmentation  ...................................... 30
    2.6.2 Side-effect Analysis  ......................................... 30
  2.7 Conclusion  ....................................................... 31

Chapter 3 Recommending Solutions to Assist Structural Test Generation  ... 32
  3.1 Introduction  ..................................................... 32
  3.2 Background  ...................................................... 36
  3.3 Motivating Example  ............................................. 38
  3.4 Approach  ......................................................... 42
    3.4.1 Overall View  ................................................ 43
    3.4.2 Dynamic Coverage View  ..................................... 44
    3.4.3 Contextual View  ............................................. 44
    3.4.4 Problem View  ................................................ 47
3.4.5 Branch List View .................................................. 48
3.4.6 Code View ......................................................... 49
3.4.7 Implementation ................................................. 51
3.5 Application of Viscovery .......................................... 51
3.5.1 Residual Coverage of Branches Caused by BPs ............ 51
3.5.2 Visualization of Search Strategies in DSE .................. 52
3.6 Experiment ........................................................... 55
3.6.1 Design ............................................................. 56
3.6.2 Procedure ......................................................... 58
3.7 Results ............................................................... 61
3.7.1 RQ1: Effectiveness .............................................. 61
3.7.2 RQ2: Efficiency ................................................ 64
3.7.3 RQ3: User Satisfaction ........................................ 66
3.7.4 Threats to Validity .............................................. 66
3.8 Discussion ........................................................... 67
3.9 Related Work ....................................................... 68
3.10 Conclusion .......................................................... 69

Chapter 4 Recommending Code to Fix Bugs ....................... 70
4.1 Introduction .......................................................... 70
4.2 Motivating Example ............................................... 72
4.3 Approach ............................................................ 73
4.3.1 Catch Exception ................................................ 74
4.3.2 String Concatenation ......................................... 76
4.3.3 Open Stream Exception Path ................................. 77
4.3.4 Missing Argument ............................................. 80
4.3.5 Workflow ........................................................ 83
4.3.6 Implementation ............................................... 85
4.4 Experiment .......................................................... 86
4.4.1 Study Design .................................................... 87
4.4.2 Procedure ........................................................ 90
4.5 Results ............................................................... 91
4.5.1 RQ1: Effectiveness and Efficiency ......................... 91
4.5.2 RQ2: Explanation of Performance ......................... 92
4.5.3 RQ3: User Satisfaction ....................................... 95
4.5.4 Threats to Validity ............................................ 99
4.6 Conclusion .......................................................... 99

Chapter 5 Guidelines .................................................. 101

Chapter 6 Related Work .............................................. 108
6.1 Studies on RSSEs .................................................. 108
6.2 Code Completion .................................................. 109
6.3 API Reuse ........................................................... 110
LIST OF TABLES

Table 2.1 Set of state-modifying methods and set of observer methods of Management, Person and Address .............................................. 14
Table 2.2 Subjects Used for Evaluating UnitPlus ........................................ 26
Table 2.3 Evaluation Results Showing the Identified State-modifying and Observer Methods 26

Table 3.1 Descriptive statistics of the participants (Viscovery) .......................... 57
Table 4.1 Descriptive statistics of the participants (FixBugs) ............................ 88
Table 4.2 User satisfaction, based on post-questionnaires. p-values indicate the likelihood that responses for one of the tools were significantly different than for the others. Overall, the participants preferred FixBugs to both Quick Fix and manual fixing. (5-point Likert Scale from “Strongly disagree (1)” to “Strongly agree (5)”)... 96

Table B.1 Mining results for String Concatenation ........................................ 148
Table B.2 Mining results for Catch Exceptions ............................................ 149
Table B.3 Mining results for String Concatenation ........................................ 150
Table B.4 Mining results for Open Stream Exception Path ................................ 151
Table B.5 Mining results for Missing Argument. (LV: Local Variable, PV: Parameter Variable, FV: Field Variable Variable, Others: Array Access, Cast Expression, Infix/Prefix/Postfix, ... ) ......................................................... 152
Table B.6 Mining results for Exception Usage Patterns. Exceptions in bold are more frequently thrown than caught. ......................................................... 153
## LIST OF FIGURES

| Figure 2.1 | Example Code: (A) Person, (B) Management, and (C) Address. | 11 |
| Figure 2.2 | (A) Sample JUnit test case for Management (B) Sample JUnit test case augmented with recommended test code in the test oracle | 12 |
| Figure 2.3 | Overview of UnitPlus | 15 |
| Figure 2.4 | The generated driver for Management and Person | 17 |
| Figure 2.5 | An example test class ManagementTest | 20 |
| Figure 2.6 | The snapshot of UnitPlus Eclipse plug-in showing side-effect information and the list of observer methods | 24 |
| Figure 2.7 | Recommended and Existing Observer methods for State-modifying methods | 27 |
| Figure 3.1 | Sketch of the class NullPointerException in xUnit | 39 |
| Figure 3.2 | Visualization of the class NullPointerException using Viscovery | 41 |
| Figure 3.3 | Overview of Viscovery | 42 |
| Figure 3.4 | The Overall View showing structural coverage information for elements such as classes and methods | 43 |
| Figure 3.5 | The Dynamic Coverage View showing dynamic code coverage achieved by the tests in Figure 3.4 | 45 |
| Figure 3.6 | The Contextual View (black squares in the gutter bar represent 0 flip count, green squares represent relatively less flip counts, and red squares represent more flip counts) | 46 |
| Figure 3.7 | The Problem View showing the problems analyzed by Covana | 48 |
| Figure 3.8 | The Branch List View. The flip count of the first branch #79 is 159 | 48 |
| Figure 3.9 | The Code View. When developers move the mouse over an “BP,” the problem annotation (C) is opened to show the analysis result of the problem and recommendations. A problematic branch is annotated with Covered Times (A), Flip Counts (B), and Problem | 50 |
| Figure 3.10 | A visual pattern of residual coverage of not-covered branches caused by BPs | 52 |
| Figure 3.11 | Visualization of three search strategies of DSE using Viscovery | 54 |
| Figure 3.12 | Experiment with a participant | 59 |
| Figure 3.13 | Effectiveness of Tools | 63 |
| Figure 4.1 | Code containing the Catch Exception defect | 72 |
| Figure 4.2 | The original Quick Fix feature provided by Eclipse | 74 |
| Figure 4.3 | Suggested fix for the Catch Exception defect shown in Figure 4.1. FixBUGS allows developers to throw or to catch exceptions by supporting the drag-and-drop action. Currently, a user is dragging SecurityException. FixBUGS displays drop zones in light grey and highlights the closest drop zone with a thicker border | 75 |
| Figure 4.4 | Code containing the String Concatenation defect. The getAddress method has a for loop and a variable of ipAddr that is concatenated in the loop | 77 |
Figure 4.5  Suggested fix for the String Concatenation defect shown in Figure 4.4. In top right is the suggestion popup. Color annotations help developers determine the impact of the change on their code. .......................... 78
Figure 4.6  Code containing the Open Stream Exception Path defect. ......................... 78
Figure 4.7  Suggested fix for the Open Stream Exception Path defect shown in Figure 4.6. 79
Figure 4.8  Another suggested fix for the Open Stream Exception Path defect shown in Figure 4.6. ............................................................. 79
Figure 4.9  Code containing the Missing Argument defect. ................................. 81
Figure 4.10  Suggested fix for the Missing Argument defect shown in Figure 4.9. The variables in green box are candidates for $%s$. The dotted borders of the variables denote that the variables can be drag-and-dropped. Currently, $aS1$ is being dragged and about to be dropped to replace the already suggested variable $s2$. ............................................................. 82
Figure 4.11  State diagram of FixBugs for follow-through bug fixing. Compared to Quick Fix, it supports the Custom Fix mode and Suspend Fix mode. ......................... 84
Figure 4.12  Percentage of Tasks Completed (Top) and Time taken (Bottom) by participants to complete given tasks manually, with Quick Fix, and with FixBugs. Overall, participants fixing defects manually took longer than participants using Quick Fix and FixBugs. .................................................... 93
Figure 4.13  Decision tree of factors, associated levels, and number of tasks (rectangle nodes) that contribute to the completion time for successful tasks, pruned at experience. The numbered circles indicate the order of the splits. ..................... 94
Chapter 1

Introduction

Recommender systems [40, 94] have a well-established history in both the research literature and industrial practice. For example, Amazon.com\(^1\) recommends items for customers to buy based on theirs and others’ purchase behaviours. Due to the broad scope of recommender systems, related research can be found in many different research fields including e-commerce, information retrieval, social networking, and software engineering.

Recommendation systems for software engineering (RSSEs) are emerging to assist developers in various tasks. Robillard and colleagues defined an RSSE [95] as:

“...a software application that provides information items estimated to be valuable for a software engineering task in a given context.”

1.1 The Problem: Lack of Usability of RSSEs for Improving Software Quality

There are many RSSEs of software artifacts\(^2\) to assist software engineering tasks. For example, RSSEs have been used for recommending project documents (Hipikat [26]), recommending API

---

\(^1\)www.amazon.com

\(^2\)Software artifacts are components as part of software systems such as source code [135, 64], test cases [48, 60], and bug reports [5, 116].
calls (CodeBroker [136], Rascal [71], Strathcona [45], XSnippet [103]), recommending method call sequences to use (ParseWeb [123]), synthesizing example code from API specifications (Prospector [68]), code-completion (FrUiT [14], RandomWalks [104], Calcite [78], Intelligent Code Completion Systems [13]), debugging recommendations (DebugAdvisor [7], HelpMeOut [42]), recommending expertise (Expertise recommender [72], ExpertiseBrowser [77], [76, 107]), improving developer awareness (YooHoo [46]), recommending web pages (Reverb [105]), recommending changes for software investigation (Suade [96]), and recommending commands [18, 70, 82].

The recommendation process of software artifacts for assisting software engineering tasks consist of three steps: invoking an RSSE explicitly by a developer or implicitly, finding relevant items by an RSSE, and presenting them by an RSSE. In the case of code recommendation, two additional steps are needed: selecting one of recommendations by a developer, and applying it to the developer’s current context\(^3\) by either an RSSE or a developer. For example, in the case of code recommendation for the additional steps, a developer needs to synthesize recommended code with her current code. These approaches have focused on finding more relevant recommendations. However, little is known about how to make these systems more usable. Murphy-Hill and Murphy point out that it is important to improve developers’ awareness of available recommendations and that hard to use RSSEs can lead the developers to misuse, mistrust, and even ignore recommendations [83].

In the software development life-cycle, software testing is an important phase that helps early detection of faults and ensures the overall quality of the developed software. In many situations, developers still manually write test code with test inputs, because the developers’ domain or application knowledge can be incorporated in manually written test input. However, the task of writing effective test oracles is non-trivial and time-consuming. On the other hand, automatically generating test cases using dynamic symbolic execution (DSE) can improve software quality. DSE is a promising approach to automatically generating high quality tests. Unfortunately, to achieve high structural coverage, DSE tools are not fully automatic, requiring sophisticated

---

\(^3\)“the situation in which something happens : the group of conditions that exist where and when something happens [139]”
interactions with developers. For example, DSE tools cannot deal with external method calls, and developers can provide mock objects to help DSE tools address the problems caused by external method calls.

Fixing software defects is also as important to improving software quality as finding defects. Defect finding tools have focused on finding more defects or reducing false positives; however, these tools often do not include code suggestions to fix the defects. As a result, developers may spend a significant amount of time fixing the defects in their code and even introducing new defects as they try to fix the old defects.

1.2 Approach: Providing More Usable Recommendation Systems

In this dissertation, I present more usable RSSEs for improving software quality particularly manual testing (UnitPlus), automated testing (Viscovery), and bug fixing (FixBugs). I briefly describe them in the following sections.

1.2.1 UnitPlus: Recommending Code to Write Test Code Efficiently

UnitPlus aim to assist developers in writing unit test cases. A unit test case consists of two parts: a test input (which is often a sequence of method calls) and a test oracle (which is often in the form of assertions). The effectiveness of a unit test case depends on its test input as well as its test oracle because the test oracle helps in exposing faults after the execution of the test input. The task of writing effective test oracles is non-trivial as this task requires domain or application knowledge and also needs knowledge of the intricate details of the class under test. In addition, when developers write new unit test cases, there is typically a lot of duplicated code, both in inputs and oracles. To assist developers in writing test code in unit test cases more efficiently, I have developed an Eclipse plug-in for JUnit test cases, called UnitPlus [115]. UnitPlus runs in the background and recommends test-code pieces for developers to choose
(and revise when needed) to put in test oracles or test inputs. Recommendations are based on static analysis of the class under test and already written unit test cases. I have conducted a feasibility study for UnitPlus with four open source Java applications to demonstrate its potential utility. In addition, UnitPlus shows the capability to improve the quality of test code in terms of test coverage.

1.2.2 Viscovery: Recommending Solutions to Assist Structural Test Generation

To help developers understand problems faced by DSE tools and improve the usability of DSE tools, I propose a visualization approach called Viscovery [117]. Viscovery visualizes the coverage achieved by DSE tools and the problem-analysis results for the not-covered areas, allowing developers to effectively and efficiently investigate the problems related to the not-covered areas. Viscovery then recommends solutions to assist DSE tools cover the not-covered areas. An empirical study with 12 developers suggests that Viscovery helps developers use DSE tools 50% more correctly, 29% more quickly, and substantially more enjoyably than existing DSE tools and DSE tools with textual problem-analysis results. These results provide a first step towards making complex testing tools more accessible to a broad scope of developers.

1.2.3 FixBugs: Recommending Code to Fix Software Defects

FixBugs is an interactive defect fixing approach with the follow-through principle that is designed to assist developers with fixing defects by recommending code [114]. To assess the usability of FixBugs, I conducted an experiment comparing FixBugs to Eclipse Quick Fix and manual fixes with 12 participants. The results show that FixBugs is as effective as non-interactive approaches, faster in some cases, and leads to greater user satisfaction.

1.3 Thesis Statement and Contributions

This dissertation is summarized with the following thesis statement:
Recommendation systems for software engineering (RSSEs) can be more effective, efficient, and satisfying to use by following a set of design guidelines: Encourage design exploration, Provide direct feedback, Allow direct manipulation, Conform to the developers’ view of the task, Automate unwanted workload, Draw users’ attention to what the users need to know about to complete their primary tasks without distraction, and Provide context-sensitive recommendations.

The primary contributions of this dissertation are as follows:

- An approach to recommend code for developers to write test code efficiently. I designed, implemented, and evaluated the approach.

- An visualization approach to assist developers understand problems that automated test generation tools face and recommend solutions to fix the problems.

- An interactive approach, based on the follow-through principle, to fix bugs by recommending context-sensitive code.

- Guidelines for helping RSSE designers create more usable tools, based on the above approaches.

1.4 The Structure of The Dissertation

The rest of this dissertation is organized as follows.

- Chapter 2 presents an approach for assisting developers in writing test code by recommending mined code. The recommended test code in test oracles can increase the effectiveness of the test case and thereby can help in finding more bugs.

- Chapter 3 describes an approach for helping developers effectively and efficiently investigate problems that DSE tools face and recommending the solutions.
• Chapter 4 presents an interactive way to recommend code for fixing software defects.

• Chapter 5 presents design guidelines to improve usability of RSSEs.

• Chapter 6 describes research work regarding recommendation systems for software engineering (RSSEs).

• Chapter 7 summarizes this work, reiterates the contributions.
Chapter 2

Recommending Code to Assist Developers in Writing Test Code

In this chapter, I present an approach, called UnitPlus\(^1\), that is designed to recommend code for developers to write test code efficiently. The remainder of this chapter is organized as follows. Section 2.1 introduces the approach. Section 2.2 gives a motivating example. Section 2.3 presents the approach and the implementation details of UnitPlus. Section 2.4 describes the experiments that I performed to evaluate UnitPlus. Section 2.5 discuss the limitations of UnitPlus. Section 2.6 presents related work. Finally, Section 2.7 concludes.

2.1 Introduction

In software development life-cycle, unit testing (also known as developer testing) is an important phase that helps early detection of faults and ensures the overall quality of the developed software. In general, a unit test case consists of two parts: a test input and a test oracle. Developer testing involves generating sufficient test inputs and checking behavior of the program under test during the execution of the test inputs. A test input includes method-call sequences and

\(^1\)This work was done in collaboration with Suresh Thummalapenta and Tao Xie. Parts of this chapter appeared in the Proceedings of the 2007 Eclipse Technology Exchange [115].
method argument values that affect fields of the class under test. A test oracle often verifies the affected fields through the class’ methods whose return type is non-void. Test inputs can either be written manually or generated automatically. Although existing automated approaches are effective in automatically generating test inputs, they often suffer from the problem of insufficient test oracles [28], especially in the absence of specification.

In many situations, developers still manually write test code for test inputs, because the developers’ domain or application knowledge can be incorporated there. Even for test oracles, developers still need to manually write test code in assertions [99], e.g., invocations of methods whose return types are non-void. On the other hand, writing effective test oracles manually is often not a trivial task as developers need to refer to the intricate details of the class under test. Often manually written test code in test inputs or oracles is the same as or similar to some previously written test code. It is tedious for developers to repeatedly type in these pieces of the same or similar test code.

To assist developers in writing test code more efficiently, I have developed UnitPlus that runs in the background and recommends test-code pieces for developers to choose (and revise when needed) to put in test oracles or test inputs. In particular, UnitPlus accepts a class and related existing test suites (including the test suite that developers are working on) as inputs. UnitPlus identifies all public methods of the given class and classifies them into two categories: state-modifying methods and observer methods. A method is classified as a state-modifying method, if the method affects (i.e., writes) the value of at least one field (or its transitively reachable field) of the given class. A method is classified as an observer, if the method’s return type is non-void. Sometimes, a method can be both a state-modifying and observer method, if the method affects some field of the given class and its return type is not void. However, UnitPlus treats this method as a state-modifying method.

To capture the behavior of the program after the execution of test inputs without modifying the behavior, it is necessary to identify that the method for capturing the behavior is a pure method. The reason is that only pure methods can help capture the behavior without
UNITPLUS parses the existing test suites and collects method sequences used for producing non-primitive method arguments and values used for primitive method arguments. The parsed information, referred as TestCodeDB, is loaded by UNITPLUS. Whenever developers add or change test cases in the test suite, UNITPLUS updates the TestCodeDB on the fly to reflect the changed information. The TestCodeDB is used to recommend test code for producing method arguments in test inputs or oracles. The recommended values for method arguments can be either method sequences or primitive values based on the type of the argument.

UNITPLUS runs in the background when developers write new test cases or modify existing test cases. After writing test code (e.g., a state-modifying method) in a test case, the developers can request UNITPLUS to recommend relevant observer methods as relevant test code to choose (and revise when needed) to put in test oracles. UNITPLUS identifies an observer method to be relevant for a state-modifying method if the intersection between the affected-field set of a state-modifying method and the accessed-field set of the observer method is not empty. The rationale is that the side effects (i.e., affected fields) of the state-modifying method on the receiver object state need to be observed and asserted (by the relevant observer methods) to make sure these side effects are expected.

Sometimes, a state-modifying method in the test input or a recommended observer method in the test oracle may need method argument values. UNITPLUS recommends test code of relevant method sequences or values (from TestCodeDB) for providing needed method arguments in the test input or oracle. UNITPLUS identifies a method sequence or value to be relevant for a method argument if the object produced by the method sequence or the value is of the same type as the method argument. The rationale is that the same or similar test code to be written by developers may have already been written by the developers in the past.

This chapter makes the following major contributions:

- An approach to recommend assertions with method sequences for non-primitive type or values used for primitive method arguments. The parsed information, referred as TestCodeDB, is loaded by UNITPLUS. Whenever developers add or change test cases in the test suite, UNITPLUS updates the TestCodeDB on the fly to reflect the changed information. The TestCodeDB is used to recommend test code for producing method arguments in test inputs or oracles. The recommended values for method arguments can be either method sequences or primitive values based on the type of the argument.

UNITPLUS runs in the background when developers write new test cases or modify existing test cases. After writing test code (e.g., a state-modifying method) in a test case, the developers can request UNITPLUS to recommend relevant observer methods as relevant test code to choose (and revise when needed) to put in test oracles. UNITPLUS identifies an observer method to be relevant for a state-modifying method if the intersection between the affected-field set of a state-modifying method and the accessed-field set of the observer method is not empty. The rationale is that the side effects (i.e., affected fields) of the state-modifying method on the receiver object state need to be observed and asserted (by the relevant observer methods) to make sure these side effects are expected.

Sometimes, a state-modifying method in the test input or a recommended observer method in the test oracle may need method argument values. UNITPLUS recommends test code of relevant method sequences or values (from TestCodeDB) for providing needed method arguments in the test input or oracle. UNITPLUS identifies a method sequence or value to be relevant for a method argument if the object produced by the method sequence or the value is of the same type as the method argument. The rationale is that the same or similar test code to be written by developers may have already been written by the developers in the past.

This chapter makes the following major contributions:
primitive values based on their frequency.

- A tool (as an Eclipse plug-in) for the proposed approach.
- An evaluation of the proposed approach on four real-world open source applications. The experimental results show that the approach can help developers write test code.

2.2 Motivating Example

I next explain my UnitPlus approach through an example shown in Figure 2.1 and illustrate how UnitPlus can help developers in writing unit test cases efficiently. The sample code in Figure 2.1 shows three classes Person, Management, and Address along with their fields and methods. The Person class contains two fields fName and fAge, and state-modifying, and observer methods. For example, methods setName and setAge are state-modifying methods as they affect values of fields fName and fAge, respectively, and methods getName and getAge are observer methods as their return types are not void.

Figure 2.2(A) shows a sample test case, either written manually or generated automatically, for the Management class in the form of a JUnit\(^3\) test case. In general, each JUnit test case consists of two kinds of statements: non-assertion and assertion statements. The non-assertion statements form the test input and the assertion statements form the test oracle. For example, in the testAdd test case shown in Figure 2.2(A), Lines 3 and 4 contain non-assertion statements followed by Line 5, which contains an assertion statement for verifying values of some affected field (fCount) through verifying the return of the observer method (howmany).

In the given test case, verifying the number of persons (reflected by the fCount field) after adding a Person object may not be sufficient to check the entire functionality provided by the add method. The test case can be made more effective by adding new test oracles. To recommend test code in augmenting the test oracle, UnitPlus initially classifies methods of the given classes into state-modifying and observer methods. UnitPlus also identifies the affected

\(^3\)http://www.junit.org
public class Person {
    private String fName;
    private int fAge;
    private Address fAddr;
    public Person() { fName = ""; fAge = 0; }
    public Person(String name, int age) { fName = name; fAge = age; }
    public String getName() { return fName; }
    public void setName(String name) { fName = name; }
    public int getAge() { return fAge; }
    public void setAge(int age) { fAge = age; }
    public void getAddress() { return fAddr; }
    public Address setAddress(Address addr) { fAddr = addr; }
}

public class Management {
    private Person[] fPeople;
    private int fCount;
    private static int MAX_COUNT = 10;
    public Management() { fPeople = new Person[MAX_COUNT]; fCount = 0; }
    public void add(Person p) { fPeople[fCount++] = p; }
    public int howmany() { return fCount; }
    public boolean isEmpty() { return fCount == 0 ? true : false; }
    public boolean isFull() { return fCount == MAX_COUNT ? true : false; }
    public boolean exists(Person p) {
        for (int i = 0; i < fCount; i++)
            if (fPeople[i].getName().equals(p.getName()) &&
                fPeople[i].getAge() == p.getAge())
                return true;
        return false;
    }
}

public class Address {
    private String fCity;
    private String fZipcode;
    public Address() { fCity = ""; fZipcode = ""; }
    public Address(String city, String zipcode) { fCity = city; fZipcode = zipcode; }
    public String getCity() { return fCity; }
    public void setCity(String city) { fCity = city; }
    public String getZipcode() { return fZipcode; }
    public void setZipcode(String zipcode) { fZipcode = zipcode; }
}

Figure 2.1: Example Code: (A) Person, (B) Management, and (C) Address.
public class ManagementTest extends TestCase {
    public void testAdd() throws Exception {
        Management mgmt = new Management();
        mgmt.add(new Person("Jane Doe", 20));
        assertEquals(1, mgmt.howmany());
    }
}

// (A)

public class ManagementTest extends TestCase {
    public void testAdd() throws Exception {
        Management mgmt = new Management();
        mgmt.add(new Person("Jane Doe", 20));
        assertEquals(1, mgmt.howmany());
        assertEquals(false, mgmt.isEmpty());
        assertEquals(false, mgmt.isFull());
        Person person1 = new Person();
        person1.setName("Jane Doe");
        person1.setAge(20);
        Address addr = new Address();
        addr.setCity("A");
        addr.setZipcode("12345");
        person1.setAddr(addr);
        assertEquals(true, mgmt.exists(person1));
    }
    ...
}
// (B)

Figure 2.2: (A) Sample JUnit test case for Management (B) Sample JUnit test case augmented with recommended test code in the test oracle
fields and the accessed fields for each state-modifying and observer methods, respectively.

**JUnitPlus** uses the information shown in Tables 2.1 to compute a relation between the observer and state-modifying methods. The relation is computed by calculating the intersection between the accessed-field set of the observer method and the affected-field set of the state-modifying method. The computed relation describes which observer methods are associated with a given state-modifying method. For example, **JUnitPlus** identifies that the `Management.add` method, which affects fields `fPeople` and `fCount`, is associated with the observer methods `howmany`, `isEmpty`, `isFull`, and `exists`. Whenever **JUnitPlus** identifies the `Management.add` method in a test case, **JUnitPlus** recommends the associated observer methods as test code in augmenting the test oracle. Figure 2.2(B) shows the sample test case with the augmented test oracle including the recommended test code. The augmented test oracle can verify more behaviors of the the `Management.add` method and thereby can provide better fault-detection capability.

Sometimes, the recommended observer methods may need method arguments including non-primitive arguments. To reduce developers’ effort, **JUnitPlus** learns from existing test cases and recommends method sequences or primitive values for producing the required argument type. For example, consider the test oracle shown in Line 15 of Figure 2.2(B). The observer method `exists` requires an argument of the `Person` class type. **JUnitPlus** learns the method sequence that produces an object of `Person` class from existing test cases and recommends the method sequence to the developers. In the current example, the recommended method sequence for producing the object of the `Person` class is shown between Lines 8 and 14. The suggested method sequence includes method calls for creating an object of the `Address` class as the method `setAddr` of the `Person` class has a non-primitive argument of the type `Address`.

### 2.3 Approach

The **JUnitPlus** approach consists of five major components: Driver Generator, Side-effect analyzer, Observer-Method Recommender (OMR), Test Code Miner (TCM), and Method Under
Table 2.1: Set of state-modifying methods and set of observer methods of Management, Person and Address

<table>
<thead>
<tr>
<th>State-modifying Methods</th>
<th>Affected Fields</th>
<th>Observer Methods</th>
<th>Accessed Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management.&lt;init&gt;()</td>
<td>fCount</td>
<td>Management.exists(Person)</td>
<td>fCount</td>
</tr>
<tr>
<td></td>
<td>fPeople</td>
<td></td>
<td>fPeople</td>
</tr>
<tr>
<td>Management.add(Person)</td>
<td>fCount</td>
<td>Management.exists(Person)</td>
<td>fCount</td>
</tr>
<tr>
<td></td>
<td>fPeople</td>
<td></td>
<td>fPeople</td>
</tr>
<tr>
<td>Management.add(String,int)</td>
<td>fCount</td>
<td>Management.howmany()</td>
<td>fCount</td>
</tr>
<tr>
<td></td>
<td>fPeople</td>
<td></td>
<td>fPeople</td>
</tr>
<tr>
<td>Person.&lt;init&gt;()</td>
<td>fAge, fName</td>
<td>Management.isEmpty()</td>
<td>fCount</td>
</tr>
<tr>
<td>Person.&lt;init&gt;(String,int)</td>
<td>fAge, fName</td>
<td>Management.isFull()</td>
<td>fCount</td>
</tr>
<tr>
<td></td>
<td>Person.getAge()</td>
<td></td>
<td>fAge</td>
</tr>
<tr>
<td>Person.setName(String)</td>
<td>fName</td>
<td>Person.getName()</td>
<td>fName</td>
</tr>
<tr>
<td>Address.&lt;init&gt;()</td>
<td>fCity</td>
<td>Address.getCity()</td>
<td>fCity</td>
</tr>
<tr>
<td></td>
<td>fZipCode</td>
<td></td>
<td>fZipCode</td>
</tr>
<tr>
<td>Address.&lt;init&gt;(String,int)</td>
<td>fCity</td>
<td>Address.getZipCode()</td>
<td>fZipCode</td>
</tr>
<tr>
<td></td>
<td>fZipCode</td>
<td></td>
<td>fZipCode</td>
</tr>
<tr>
<td>Address.setCity(String)</td>
<td>fCity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address.setZipCode(String)</td>
<td>fZipCode</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Test (MUT) Learner. Figure 2.3 shows an overview of the major components of my approach. My approach requires the code under test to include a main method to serve as a entry point for the analysis. If the given code under test does not contain a main method, I use the Driver Generator to generate a main method. The Driver Generator collects all public methods of each concrete class and generates a driver code for invoking all the methods. If a main class that has a main method in the production code exists, then Side-effect analyzer takes the main class as the entry point and analyzes the code under test. At the end of the analysis, Side-effect analyzer produces state-modifying and observer methods. The side-effect information is used by Observer-Method Recommender to rank and to suggest observer methods. In the background mode, Test Code Miner mines or extracts the values for primitive types and method sequences for non-primitive types. The last component, MUT Learner, is used in identifying methods under test in test cases. Test Code Miner and MUT Learner interact with each other in order to get evaluation results. The augmented TestCodeDB is used by OMR to suggest observer methods containing primitive/non-primitive arguments.

UNITPLUS runs in the background when developers write new test cases or modify existing
test cases. After writing test code (e.g., a state-modifying method) in a test input of a test case, developers can request UnitPlus to recommend relevant observer methods as relevant test code (and revise when needed) to put in test oracles. UnitPlus identifies an observer method to be relevant for a state-modifying method if the intersection between the affected-field set of a state-modifying method and the accessed-field set of the observer method is not empty. The rationale is that the side effects (i.e., affected fields) of the state-modifying method on the receiver object state need to be observed and asserted (by the relevant observer methods) to make sure that these side effects are expected. These recommended test oracles can help increase the effectiveness of the test case and thereby can increase the overall quality of the test suite. Sometimes, a state-modifying method in the test input or a recommended observer

\footnote{The accessed-field set includes read fields (as well as written fields if the observer method is a state-modifying method).}
method in the test oracle may need method argument values. UnitPlus recommends test code of related to relevant method sequences or primitive values (from TestCodeDB) for providing needed method arguments in the test input or oracle. UnitPlus identifies a method sequence or primitive value to be relevant for a method argument if the object produced by the method sequence or the primitive value is of the same type as the method argument. The rationale is that new test code to be written by developers may be the same or similar to test code that have already been written by the developers in the past.

2.3.1 Driver Generator

The side-effect analyzer takes a main method as input and analyzes all the methods and its transitive method calls in the method. The analysis client should provide the main method that serves as an entry point for the analysis. However, some programs like libraries may not have a main method. In order for the side-effect analyzer to get side-effect information, it is necessary to invoke all public methods in a generated main method. The generator has two main tasks: collect all public classes in the production code and filter out all the interfaces and abstract classes among these classes. The generator next invokes all public methods of each concrete class that can be instantiated in test code. In each driver method, Driver Generator first invokes the constructor of the corresponding concrete class to instantiate the receiver object of the concrete class. If there are more than one constructor methods in the concrete class, Driver Generator invokes all the constructor methods to instantiate different receiver objects. The last receiver object is used to invoke other public methods in the concrete class. However, the order among constructors or (normal) public methods are not important, because I want to identify whether each method is a state-modifying or observer method and the order of method invocation does not affect the identification of state-modifying and observer methods. In addition, when I invoke the methods, since some methods can have the same name with different types of arguments, it is required that the type conversion is explicitly written before the arguments. Otherwise, there can be ambiguous type problems. The driver generator uses JDT in Eclipse for scanning and
public class Main {
    public static void methods_Of_org_Management() throws Exception {
        Management management1 = new Management();
        management1.add((org.Person) null);
        management1.howmany();
        management1.isEmpty();
        ...
    }
    public static void methods_Of_org_Person() throws Exception {
        Person person1 = new Person();
        Person person2 = new Person((String) null, 0);
        person2.getName();
        person2.setName((String) null);
    ...
}
    public static void main(String[] args) throws Exception {
        methods_Of_org_Management();
        methods_Of_org_Person();
    }
}

Figure 2.4: The generated driver for Management and Person

analyzing the production code and for generating driver code. Figure 2.4 shows the generated driver for Management and Person in Section 2.2.

2.3.2 Side-Effect Analyzer

To recommend observer methods, UNITPLUS first analyzes given code under test. Given a program, Side-effect analyzer accepts a class with main method as input, loads the target modules in it, builds class hierarchy and call graph construction, does pointer analysis, and finally gets side-effect information. Base on the information, the analyzer classifies the public methods of the given class into two categories: state-modifying and observer methods. A method is classified as a state-modifying if the method affects at least one field (or its transitively reachable field) of the given class. A method is classified as an observer method if the method does not change the state of the given class and the return type of the method is non-void. A method can be both a
state-modifying and an observer method; however, if a method has at least one affected field, UnitPlus consider the method as a state-modifying method. The reason is that if UnitPlus takes the method as both state-modifying and observer method, using this observer method in test oracle may introduce another side-effect that is unexpected. For example, assume that a method $m_1$ changes value of a field $f_1$, accesses a field $f_2$, and returns some value. Another method $m_2$ accesses $f_1$, changes $f_3$, and also returns some value. If UnitPlus consider the two methods as state-modifying and observer methods, then method $m_2$ can be an observer method for method $m_1$ because method $m_2$ accesses $f_1$. However, if $m_2$ is recommended as an observer method, the method affects $f_3$ when used in test oracle. Such a change can be unexpected. Therefore, to be conservative, I consider such methods as state-modifying methods rather than I expect developers to decide whether to use a recommended observer method as there can be side-effects when the recommended observer method is also a state-modifying method.

Along with the classification of the methods, the side-effect analyzer also identifies the affected and accessed fields for each state-modifying and observer methods, respectively.

```java
public class C {
    public void m1() {
        m11();
        m12();
    }

    public void m2() {
        m1();
    }
}
```

For example, consider that a method, say $m_1$, calls $m_{11}$ and $m_{12}$. If $m_1$, $m_{11}$ and $m_{12}$ are in the same type hierarchy of the given class, the accessed and affected fields of $m_1$ include those of $m_{11}$ and $m_{12}$ along with its ($m_1$) side-effect information. In addition, consider that $m_2$ calls $m_1$. If the side-effect analyzer already analyzed $m_1$, the information of $m_2$ can be cumulated faster, because it memorizes the side-effect information of $m_1$ in order to save analysis time.
2.3.3 Observer Method Recommender

The Observer Method Recommender (OMR) component assists developers by recommending test code in augmenting test oracles in the form of assert statements. This component accepts the sets of state-modifying and observer methods as input and computes relationships among state-modifying and observer methods. An observer method is identified as relevant to a state-modifying method if intersection between the accessed-field set of the observer method and the affected-field set of the state-modifying method is not empty. For example, consider that an observer method, say $om_1$, accesses fields $f_1$ and $f_2$ of the class ExampleClass, and the state-modifying method, say $smm_1$, affects fields $f_2$ and $f_3$. The OMR component calculates the intersection between sets \{$f_1, f_2$\} of $om_1$ and \{$f_2, f_3$\} of $smm_1$. As the intersection results in set \{$f_2$\}, which is not empty, the OMR component identifies that $om_1$ is relevant to $smm_1$. The component uses the computed relationships while recommending test code in augmenting the test oracle. For example, if developers add a state-modifying method to the test case, the OMR component identifies the relevant observer methods and recommends them to the developers as test code in augmenting the test oracle.

2.3.4 Test Code Miner

The Test Code Miner (TCM) component helps developers in writing test code by suggesting method sequences or argument values for the recommended observer methods. Initially, TCM gathers method sequences or primitive values used by the existing test cases along with the locations where these method sequences or primitive values are written. The gathered information is stored in memory, and is referred as TestCodeDB. Whenever developers add or change test cases in the test suite, UnitPlus updates the TestCodeDB on the fly to reflect the changed information. The TestCodeDB is used to recommend test code for producing method arguments in test inputs or oracles. The recommended values for method arguments can be either method sequences or primitive values based on the type of the argument. When OMR requests TCM for the values of a method argument, TCM checks for the available method sequences or primitive
public class ManagementTest extends TestCase {

    public void testConstructor() {
        Management m1 = new Management();
        assertEquals(false, m1.isFull());
        assertEquals(true, m1.isEmpty());
        assertEquals(0, m1.howmany());
    }

    public void testAdd1() {
        Management m1 = new Management();
        Person p1 = new Person("John Doe", 30);
        Person p2 = new Person("Jane Doe", 25);
        Address a1 = new Address("Raleigh", "27606");
        p1.setAddr(a1);
        m1.add(p1);
        m1.add(p2);
        assertEquals(1, m1.howmany());
    }

    public void testAdd2() {
        Management m1 = new Management();
        ...
    }
}

Figure 2.5: An example test class ManagementTest.

values, and recommends those method sequences as a list ranked by their frequency. In general, there can be more than one method sequence available in the existing test cases for instantiating a required non-primitive type. UnitPlus uses a ranking criterion to sort identified method sequences based on the frequency of each identified method sequence.

For example, in Figure 2.5, consider that the ManagementTest class has three test methods testConstructor, testAdd1, and testAdd2. Assume that a developer tries to add some method sequence (Line 20) for validating the behavior of the add(Person) method with a different method sequence written in testAdd1. To achieve this, the developer needs to write test input to bring the receiver object to the desirable state of the class under test. The test input that needs to be written in testAdd2 can be similar to the test in the testAdd1. For the constructor of Management, OMR suggests methods exists(Person), getAge(), howMany(), isEmpty(), and isFull(). As exists(Person) has a non-primitive type Person, the Person should be
instantiated before invoking the `exists(Person)` method. In this scenario, TCM provides OMR with method sequences for the argument `Person`. Given the test class in Figure 2.5, TCM mines two method sequences with the following frequencies below:

```java
1 Person p1 = new Person("John Doe", 30);
2 assertEquals("Freq.:2", false. m1.exists(p1));
```

or

```java
1 Address a1 = new Address("Raleigh", "27606");
2 Person p1 = new Person("John Doe", 30);
3 p1.serAddr(a1);
4 assertEquals("Freq.:1", false, m1.exists(p1));
```

The suggested method sequences are sorted by their frequency. As shown above, the first sequence has frequency two, and the next sequence has frequency one. If their frequencies are the same, then the sequences are sorted by their length.

### 2.3.5 MUT Learner

Given a test method, the Method Under Test (MUT) Learner component identifies which method is the method under test in the given test method. I use following criteria to identify the MUT.

- **Name similarity**: If test class name and the test method name are the same, I select that method as MUT.

- **Field intersection**: The last method before the last assertion statement in the end of test method. This last method should be one of the methods declared in the production code.

- If the intersection between the affected fields of the last method and the accessed fields of the methods invoked in assertions is empty, the previous method will be a candidate for MUT. The MUT learner module will iterate until this criteria is satisfied or there is no other likely MUT.

---

21
public class ManagementTest extends TestCase {
    public void testAdd() throws Exception {
        Management mgmt = new Management();
        mgmt.add(new Person("John Doe", 30));
        assertEquals(1, mgmt.howmany());
    }
    public void testSomething() throws Exception {
        Management mgmt = new Management();
        mgmt.add(new Person("John Doe", 33));
        assertEquals(false, mgmt.isEmpty());
    }
}

The first criterion is to check with the name of the test method. Usually, in JUnit3, a test method has its name with “test” prefix. In JUnit4, the test method has @Test annotation. If the name of the test method without “test” prefix or the whole name of the test method is not matched partially, then I try to identify the last method before the last assertion statement at the end of test method as a MUT, because developers usually write test inputs before assertions.

For example, consider the test class ManagementTest with two test methods testAdd and testSomething. I can extract Management and the name of test methods without ‘test’ prefix. The extracted name of testAdd is add and the test method contains add that is invoked in Line 4. Therefore, I can identify that the MUT for the test method is add according to the name similarity. Similarly, the extracted name of testSomething is Something. However, there is no exact or partial matched method call for this test method. In that case, I compute the field intersection between the method, isEmpty, that is invoked in the last assertion block of the test method and the add method that is called before the last assertion block. If the intersection is empty, I identify the method as a MUT. In this current example add and isEmpty have a common field fCount. Therefore, I identify add as a MUT of this test method. If there is no field intersection between the current method and the methods invoked in the last assertion block, I continue to compute the field intersection of the previously invoked methods (i.e. Management.<init>()). I iterate the process until all the method calls in the test methods are compared. If none of all the methods has no field intersection, I consider the last method as
the MUT of the test method.

2.3.6 Implementation

In this section, I describe how each component is implemented. UnitPlus is fully integrated with the Eclipse platform. Eclipse is a commonly used integrated development environment that supports the addition of components, called plug-ins, to add to the IDE functionality. UnitPlus uses Java Development Tooling (JDT [2]) in Eclipse and Harpoon\textsuperscript{5} (A side-effect analyzer).

Side-Effect View

Side-effect view provides the side-effect information for the given project. If the user triggers execution of the side-effect analyzer, the view produces a tree that shows a set of state-modifying methods (with affected and accessed fields), and a set of observer methods (with accessed fields). The tree view allows the user to browse the method by selecting the method. Figure 2.6 shows the side-effect view with the side-effect information of Management, Person, and Address classes.

Mining Test Code and Recommending Observer Methods

In order to build a method-invocation-sequence graph, I used the JUNG framework\textsuperscript{6} that provides a representation module as a graph. In my current implementation, the OMR and TCM components use the Eclipse JDT for parsing Java source files.

The recommender is implemented by extending the QuickAssistProcessor provided by Eclipse JDT that allows an editor to provide context-based assistance to the user. Assistance usually involves several tasks such as automatically generating source code.

\textsuperscript{5}http://flex.cscott.net/Harpoon/. FLEX compiler infrastructure for Java.
\textsuperscript{6}http://http://jung.sourceforge.net/, Java Universal Network/Graph Framework
Figure 2.6: The snapshot of UnitPlus Eclipse plug-in showing side-effect information and the list of observer methods
The recommendations produced by UnitPlus are presented to the user in the Eclipse IDE by selecting the desired state-modifying method and by pressing Ctrl+1. The recommendations take the form of a list of assertions with method sequences. A snapshot of the UnitPlus plug-in is shown in Figure 2.6. The snapshot shows a set of recommended observer methods for the state-modifying method add. The developers can choose any of the recommended observer methods and UnitPlus automatically adds the selected observer method to the test case.

2.4 Feasibility Study

I next describe a feasibility study conducted with four different code subjects. In sum, the study results show that the existing test suites of these subjects can be augmented with recommended test code in additional test oracles to make these test suites more effective.

2.4.1 Study Setup

I used four subjects that are open source libraries with existing test suites. JSort\(^7\) library provides sorting algorithms. JBell\(^8\) is a Java library that enables developers to perform operations such as collection filtering and/or sorting. JAutomata\(^9\) is a library used for creating, manipulating, and displaying finite-state automata within the Java platform. StringTree\(^10\) is a library for text transformation and processing package.

Table 2.2 shows characteristics of all these subjects used for in the study. In particular, the columns show the subject name, number of classes, number of test suites, and the total number of test cases in all test suites. Column “SMM” gives the number of state-modifying methods in each subject library. For each SMM, I present the total number, and the number of SMMs invoked and not invoked by the test code. Column “OM” gives the number of observer methods in each subject library.

\(^7\)http://sourceforge.net/projects/jsort
\(^8\)http://sourceforge.net/projects/jbel
\(^9\)http://sourceforge.net/projects/jautomata
\(^10\)http://sourceforge.net/projects/stringtree
Table 2.2: Subjects Used for Evaluating UnitPlus.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Description</th>
<th>Classes</th>
<th>Test Classes</th>
<th>Test Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSort</td>
<td>Sorting algorithms</td>
<td>11</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>JBel</td>
<td>Java boolean expression library</td>
<td>55</td>
<td>26</td>
<td>80</td>
</tr>
<tr>
<td>JAutomata</td>
<td>Java automata library</td>
<td>84</td>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>StringTree</td>
<td>Text transformation and processing</td>
<td>58</td>
<td>30</td>
<td>169</td>
</tr>
</tbody>
</table>

Table 2.3: Evaluation Results Showing the Identified State-modifying and Observer Methods

<table>
<thead>
<tr>
<th>Subject</th>
<th>#State-Modifying Method</th>
<th>#Observer Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSort</td>
<td>Total 6, Invoked 6, Not Invoked 0</td>
<td>3</td>
</tr>
<tr>
<td>JBel</td>
<td>Total 46, Invoked 41, Not Invoked 5</td>
<td>30</td>
</tr>
<tr>
<td>JAutomata</td>
<td>Total 93, Invoked 36, Not Invoked 57</td>
<td>96</td>
</tr>
<tr>
<td>StringTree</td>
<td>Total 169, Invoked 104, Not Invoked 65</td>
<td>131</td>
</tr>
</tbody>
</table>

2.4.2 Data Collection and Analysis

I next explain how I conducted the experiment. First, I scan through test cases that are already written in the subjects. Then, before assertion statements, I simulate what observer methods UnitPlus would suggest and then compare the set of actually written observer methods.

2.4.3 Results: Identification of State-Modifying and Observer Methods

Table 2.3 shows state-modifying and observer methods identified by UnitPlus. Column “SMM” gives the number of state-modifying methods in each subject. For each SMM, I present the total number, and the number of SMMs invoked and not invoked by the test code. Column “OM” gives the number of observer methods in each subject library. I also found some interesting results on the test suites of each subject. These results are shown in the column “Not Invoked” of Table 2.3. For example, JBel has 41 invoked and 5 uninvoked state-modifying methods, whereas JSort has 6 invoked and 0 uninvoked state-modifying methods.
Figure 2.7: Recommended and Existing Observer methods for State-modifying methods.
Figure 2.7 shows existing and recommendable observer methods for state-modifying methods of four subjects. The $x$ axis represents state modifying methods of the subject (labeled with ID numbers) and the $y$ axis represents the number of observer methods associated with each state-modifying method. For each state-modifying method, I show both the number of observer methods that are actually invoked (white bar) and the number of recommendable observer methods (black bar) in the test suites of each subject library. I observed that some libraries have a few state-modifying methods for which there are neither recommended observer methods nor invoked observer methods in the existing test suite. I ignored such state-modifying methods from the figure.

For subject libraries JAutomata, JBel, and StringTree, the study results show that the test oracles of the existing test suites can be further augmented, because in the existing test cases not all relevant observer methods are used to verify behavior for many state-modifying methods. For example, for the first state-modifying method (denoted by 1 in the $x$ axis) of the JAutomata library, the number of recommended observer methods is 17. However, none of these observer methods are verified after the state-modifying method in the existing test suite of the JAutomata library. My results indicate that for the JSort library, the developers invoked all recommended observer methods for each state-modifying method. In summary, the study results show that the existing test suites of these libraries can be augmented with the recommended test code in test oracles to make these test suites more effective.

**Threats to Validity** The threats to external validity primarily include the degree to which the subject programs, faults or program changes, and test cases are representative of true practice. The current subjects are small-scale applications. These threats could be reduced by more experiments on wider types of subjects in future work.

The threats to internal validity are instrumentation effects that can bias the results. Faults in my prototype and Harpoon might cause such effects while identifying observer and state-modifying methods. To reduce these threats, I have manually inspected the results.
2.5 Discussion

UnitPlus classifies methods of the class under test into state-modifying methods and observer methods. Based on the criteria used for classification, a method can be both a state-modifying and observer one. When this type of observer methods is used as test oracles, the recommended relevant observer methods for a state-modifying method can modify the state of the underlying object. Therefore, UnitPlus expects developers to decide whether to use a recommended observer method as there can be side-effects when the recommended observer method is also a state-modifying method.

UnitPlus currently needs developers to manually write expected return values for the recommended observer methods. In future work, I plan to automatically capture the actual return values of the recommended observer methods and then the developers need only confirm the captured return values. I expect that automatic capturing of the actual values can help to further reduce the efforts of developers while writing unit test cases.

So far I have conducted a feasibility study to show that UnitPlus can recommend additional test oracles that can be added to make the existing test cases more effective. Although the benefits of test oracle augmentation was demonstrated by an experiment conducted previously in assessing the Orstra approach [132], in future work, I plan to conduct a more comprehensive evaluation to validate whether the augmented test oracles indeed improve the fault-detection capability of those test cases.

In general, there can be more than one method sequence available in the existing test cases for instantiating a required non-primitive type. The current implementation sorts the available method sequences based on the distance from the working location to locations of method sequences to be recommended. In future work, I plan to investigate and include several other ranking heuristics for prioritizing those identified method sequences. One such ranking criterion is to sort the identified method sequences based on the frequency of each identified method sequence.
2.6 Related Work

The ideas investigated in this chapter intersect with code recommendation (see Section 6.3), test code augmentation, and side-effect analysis.

2.6.1 Test Code Augmentation

Orstra [132] is an approach to increase the effectiveness of the existing test cases by augmenting an automatically generated unit-test suite with regression oracle checking. But as the approach is non-interactive, it is not possible for developers to incorporate their domain knowledge while generating test oracles. Brett and colleagues suggested an approach, called ReAssert [29], for recommending repairs to failing test cases by changing or creating test assertions.

Rompaey and colleagues proposed a metric-based heuristic approach for ranking test cases to identify poorly designed test cases [97, 98]. Their approach suggests to refactor those test cases that violate unit test criteria. Their approach helps to minimize the maintenance cost for test code but may not be effective in increasing the effectiveness of the existing test cases.

2.6.2 Side-effect Analysis

One of the most relevant work to my approach is the approach by Sălcianu and Rinard [120, 121] and Ananian’s approach [4]. My current work is based on Sălcianu’s approach [121]. UnitPlus used Harpoon, a side-effect analyzer developed by Ananian.

UnitPlus is also related to the combined static and dynamic analysis approach proposed by Artzi and colleagues [6]. They proposed a combined static side-effect analysis and dynamic detection of parameter mutability. Rountev proposed an analysis technique for identifying side-effect-free methods in Java [100]. More recently, there are side-effect analyzing schemes based on dynamic analysis. Xu and colleagues presented a dynamic purity analysis for Java [134]. UnitPlus is different from their approach as UnitPlus uses static approach. Static analysis is more preferable than dynamic instrumentation in identifying state-modifying methods and observer methods because the static approach does not require any instrumentation, whereas
dynamic approach requires instrumentation and is restricted to only those methods that are executed during run-time. Dallmeier and Zeller developed a tool, called JPure [141], for dynamic side-effect analyzer but provided no description of their approach.

2.7 Conclusion

Manual test case writing in creating test inputs and oracles is a tedious process. I have developed a code recommendations system, called UnitPlus, that can assist developers in writing unit test cases more efficiently. UnitPlus runs in the background and recommends relevant test code in test oracles whenever the developers enter a test input in the test case. The recommended test code in test oracles can increase the effectiveness of the test case and thereby can help in finding more bugs. UnitPlus also tries to reduce developers’ effort while they are writing test cases by automatically recommending method sequences or values that can instantiate a given method argument type. I conducted a feasibility study on UnitPlus with four different subjects and showed that existing test suites can be augmented with the recommended test code in additional test oracles to make these test suites more effective.
Chapter 3

Recommending Solutions to Assist Structural Test Generation

This chapter describes a visualization approach, called Viscovery\(^1\), that aims to assist developers understand problems that automated test generation tools face and recommend solutions to fix the problems. The rest of the chapter is as follows. Section 3.1 introduces the approach. Section 3.2 presents background techniques of Viscovery. Section 3.3 explains a motivating example. Section 3.4 describes main idea of the approach. Section 3.5 shows how Viscovery could be applied to real-world programs. Section 3.6 describes the design, procedure and results of the experiment. Section 3.8 discusses about limitations and issues of my approach currently cannot address. Section 3.9 presents related work. Finally, Section 3.10 concludes.

3.1 Introduction

In software testing, structural coverage, such as statement coverage [3] and branch coverage [86], is used to identify insufficient tests for a program under test. Achieving high structural coverage (e.g., block coverage and branch coverage) of a program under test is an important goal of testing.

\(^1\)This work was done in collaboration with my advisor Emerson Murphy-Hill, Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Peli de Halleux.
However, manually producing tests with high structural coverage is laborious. To address the issue, developers can employ automated test-generation approaches, such as Dynamic Symbolic Execution (DSE) [21, 38, 108, 125], a variant of symbolic execution [55, 20], to automatically and systematically generate test inputs that achieve high structural coverage. DSE generates such test inputs by running the program with symbolic inputs. First, DSE collects the constraints on the inputs from the branch statements to form the path condition of the execution path. Then, from that path condition, DSE selects a branch node (a runtime instance of a conditional branch), flips (i.e., negates) the constraints of the selected branch node, and generates test inputs by solving the flipped path condition for exploring new paths. This process continues until all paths have been explored or the given test-generation resources (such as time) are used up.

Although DSE can generate test inputs to achieve high structural coverage for simple programs, when applied to complex programs in practice, existing DSE tools face various challenges in achieving high structural coverage. Based on recent studies [130], the top two major problems that prevent these tools from achieving high structural coverage of object-oriented programs are (1) the Object-Creation Problems (OCP), where DSE cannot produce sequences of method-calls to construct desirable object states for covering certain branches, and (2) the External-Method-Call Problems (EMCP), where DSE cannot deal with method calls to external libraries. Besides these problems, recent studies [129, 39] identify Boundary Problems (BPs) as another important type of problems where tools exceed pre-defined boundaries on resources before achieving high coverage, often caused by loops. Since tools are not perfect in dealing with various challenges in achieving high structural coverage, cooperative testing identifies problems faced by tools during test generation, and developers provide guidance to help tools solve these problems. For example, developers can provide factory methods [31, 125] that include sequences of method calls to solve OCPs, instruct existing DSE tools to instrument external-method calls or provide mock objects [67, 122, 102] to solve EMCPs, and limit loop iterations or increase resource limits to solve BPs.
Although existing tools, such as Covana [130], identify problems faced by test-generation tools, the problems are shown to developers in the format of textual output. Such textual output is not readily understandable, and provides limited assistance to developers in investigating problems. Given an identified problem in textual format, developers still need to locate the problem in the program and investigate the program to decide how to provide guidance. In addition, after the developers provide guidance to DSE, they still need to reapply DSE tools to verify whether the guidance helps DSE solve the problem and improve the coverage. If not, they need to iteratively repeat the process until the problem is solved. This process is difficult because developers must look at the textual output of the identified problems and investigate the program separately; such process is a time-consuming task.

In addition to EMCPs and OCPs, DSE faces another major type of problem, the Boundary Problem (BP), which prevents DSE from achieving high structural coverage: deciding which branch node in the path condition to flip. DSE collects path conditions and flip one of the branch nodes in the path condition to obtain a path condition that leads to a new branch, achieving new coverage. However, due to loops in the program, DSE may keep negating the constraints to increase the iterations of the loop and exceed pre-defined boundaries of resources before achieving high coverage [129]. To guide DSE to achieve higher coverage within fixed boundaries of resources, different kinds of search strategies [125, 133] can be used to provide guidance on how to choose branch nodes to flip, but these strategies are not powerful enough to deal with many situations in complex programs. Instead, as with EMCPs and OCPs, human guidance can help DSE solve BPs. For example, developers can supply DSE with assumptions to fix the number of iteration of loops or to constrain the value range of the program input. To provide such guidance, developers need information about how DSE tools spend the efforts in which branches and how such effort distribution affects the overall coverage, but existing DSE tools do not provide such information.

To help developers investigate these problems and provide guidance, developers can focus on not-covered areas of the program under test and investigate the problems related to the
not-covered areas using existing visualization tools. Existing coverage visualization tools, such as EclEmma\(^2\) and NCover\(^3\), visualize covered and not-covered code using different colors and a hierarchical style, which can help the developers focus on the not-covered branches to achieve higher structural coverage. However, when developers apply DSE tools to generate test inputs for the program under test, the coverage information provided by these visualization tools is not sufficient, since these tools do not provide any specific visualization of the problems faced by DSE tools.

In this chapter, I propose a novel coverage visualization approach, called Viscovery (Visualization of structural coverage), which visualizes the problem-analysis results and the coverage information for DSE tools. Viscovery aims to help developers understand the problems faced by DSE tools, assisting developers in providing guidance to help DSE improve coverage. Viscovery lets developers navigate between different views by clicking elements in the views or selecting items from a context menu triggered by mouse right-clicking. Developers find useful information by navigating to the most appropriate view.

This chapter makes the following main contributions:

- A novel visualization approach, Viscovery [117], designed to effectively and efficiently assist developers in understanding problems that prevent DSE tools from achieving high structural coverage.

- An implementation of the proposed approach\(^4\) based on Pex [125], a state-of-the-art DSE tool from Microsoft Research, and Covana [130], an analyzer for identifying and reporting problems that prevent the DSE tool from achieving high structural coverage.

- An empirical study demonstrating that Viscovery is 50% more correctly, 29% more quickly, and substantially more enjoyably than existing DSE tools and DSE tools with textual output of problem-analysis results.


\(^4\)My implementation and empirical study results are available at http://www4.ncsu.edu/~ysong2/viscovery/.
3.2 Background

Dynamic Symbolic Execution (DSE) systematically explores feasible paths of the program under test by running the program with different test inputs. Along the execution, DSE collects the symbolic constraints on inputs obtained from predicates in branch statements to form an expression, called a path condition. DSE then negates part of the constraints in the path condition to obtain a new path condition, and uses a constraint solver to compute new test inputs that satisfy this new path condition. These generated test inputs are fed into the program under test to explore new paths, which may follow new branches. Through iterations of input generation and path exploration, feasible paths in the program under test can be exercised eventually. However, when applied on complex programs in practice, DSE faces various challenges that prevent DSE from achieving high structural coverage.

When a path condition is fed into a constraint solver for computing new test inputs, various problems can occur to cause the new test inputs not to be computed: (1) Infeasible Path: the path condition obtained by flipping a branch node from the previously collected path condition can be infeasible; (2) Limitation of constraint solver: due to the limitation of the used constraint solver, e.g., floating-point arithmetics [12, 57], DSE may not compute test inputs to satisfy some path conditions; (3) Timeout: some path conditions are too complex and require too much time to solve; (4) Object-Creation-Problem (OCP): DSE may not be able to generate method-call sequences [124] to produce the desirable object states required by some path conditions.

DSE instruments the program under test and executes the program with the generated test inputs. However, DSE does not instrument all the libraries used by the program under test. By default, DSE does not instrument all the third-party libraries due to the explosion of the exploration space, and DSE cannot instrument native system libraries that interact with environments, such as network sockets and file systems. Hence, symbolic values may be passed as arguments to methods that are not instrumented. Xusheng and colleagues showed that DSE faces problems of external-method calls that receive symbolic values as arguments in two main situations: (1) the return values of external-method calls are used in the subsequent branches
and cause some branches not to be covered; (2) exceptions are thrown during the execution of the external-method calls to abort the whole program executions \[130\].

During program executions, DSE may encounter boundary problems where DSE keeps exploring the paths in a portion of the program under test and is prevented from exploring new paths in the remaining parts of the program. These problems are mainly caused by loops or too many paths in the program under test. The program under test may have loops whose bounds depend on the inputs (symbolic values) or whose bodies include branches executed for many times or infinitely. By negating the constraints that limit the number of iterations the loops should take, DSE keeps exploring the paths inside loops, preventing DSE from exploring new paths in the remaining parts of the program. To prevent such problems in keeping DSE only exploring the paths inside the loops, default boundary values are provided for imposing bounds for DSE, e.g., the maximum number of exploration iterations and the maximum number of negated branch nodes in a path.

To address these problems, DSE provides several configuration options for the users to assist DSE: (1) assumptions: the users can provide assumptions to constrain the inputs that can be generated by DSE. For example, the users can provide assumptions on inputs to constrain DSE not to generate null objects for the inputs or specify the range of values for inputs; (2) factory methods: the users can provide factory methods that include necessary method-call sequences of a specific object type to assist DSE to explore object states; (3) instrumentation properties: the users can specify instrumentation properties to tell DSE which class or library to instrument; (4) mock objects \[122\]: the users can provide mock objects to replace the method calls that interact with environments. To better assist the users in understanding problems that prevent DSE from achieving high structural coverage and figure out how to provide assistance, I propose a novel approach to visualize the structural coverage information and the analysis results of problems for the residual structural coverage.
3.3 Motivating Example

I next illustrate how Viscovery can help the developers understand the problems that prevent DSE from achieving high structural coverage. Consider the example shown in Figure 3.1, where DSE is applied to generate test inputs for the constructor `NullException`. Figure 3.2 shows a screen shot of Viscovery, which visualizes the structural coverage information and problem-analysis results for the class `NullException` in six specially designed views.

From the overall view ($V_1$), developers can gain knowledge of the achieved system-wide structural coverage and select one of the methods under test, `NullException`, that does not achieve high structural coverage for further analysis (42/79, 53.16% block coverage ($A$)). The dynamic coverage view ($V_2$) shows the files explored by DSE in the overall view. The contextual view ($V_3$) visualizes all the files explored by DSE. The background color of each statement shows coverage: green indicates that the statement is covered at least once and orange indicates that the statement is not covered. Through this view, developers can understand which areas of the program are covered and easily identify the two not-covered areas, $(C_1)$ and $(C_2)$, in the file $(F)$. To view the not-covered areas more clearly, the developers can use the zoom bar $(Z)$ to enlarge the file and the outline $(V_7)$ to move the areas of interest to the center. The code areas $(C_1)$ and $(C_2)$ are not covered and the loop at $(C_3)$ causes a Boundary Problem (BP).

To understand what problems cause these two areas not to be covered, developers can browse the problem view ($V_4$) to see the details of problem-analysis results produced by Covana. In this example, an BP, which occurs at Line 91, is identified by Covana. When the developer clicks the problem item, the corresponding line in the contextual view is highlighted with a light red color. By browsing the branch view ($V_5$), developers can see that the flip count of the branch in red at Line 91 is 159 ($B$), which is much larger than the flip count of the other branches. Navigating back to the source code shown in the code view ($V_6$), the developers can see that the statement at Line 91 inside a method `ConvertToString` is a `foreach` loop, which is the annotation “BP” to indicate that an BP occurs here.

As I discussed in Section 3.1, it is relatively easy to identify the loop that causes a BP but it...
public class NullException : AssertActual...Exception {
    // Method under test, only argument actual is symbolic
    public NullException(object actual)
        : base(null, actual, "Assert.Null() Failure") { }
}

public class AssertActualExpectedException {
    // constructor 1 calling constructor 2
    public AssertActualExpectedException(object expected,
                                         object actual, string userMessage)
        : this(expected, actual, userMessage, false) { }

    // constructor 2
    public AssertActualExpectedException(object expected,
                                          object actual, string userMessage, bool skip) {
        if (!skip) {
            IEnumerable e1 = actual as IEnumerable;
            IEnumerable e2 = expected as IEnumerable;
            if (e1 != null && e2 != null) {
                // not-covered area A
            }
            this.actual = ConvertToString(actual);
            this.expected = ConvertToString(expected);
            this.actual = ConvertToString(actual);
            if (actual != null && expected != null && ...) {
                // not-covered area B
            }
        }
    }

    static string ConvertToString(object value) {
        Array valueArray = value as Array;
        ...
        foreach (object valueObject in valueArray) {
            // do something inside loop
            ...
        }
    }
}

Figure 3.1: Sketch of the class NullException in xUnit.
is difficult to identify the not-covered branches caused by the BP. Since the not-covered branches caused by the BP must have flip count 0, the developers can switch back to the contextual view to search for the potential not-covered branches that are caused by the BP. In the contextual view, the gutter on the left hand side of each file indicates the flip count of each branch, with red color squares for the maximum flip count and black squares for flip count 0. Above the not-covered area ($C_2$), the developers can identify some black squares on the gutter. By clicking the black squares, the code view again shows the corresponding source code of Line 72 in the center. At Line 69, the developers can see that the method `ConvertToString`, where the BP occurs, is invoked just before the not-covered branches at Line 72, causing DSE to stop the exploration before DSE has a chance to flip the branches at Line 72. Thus, developers can determine that the residual coverage of the branch at Line 72 is caused by the BP.

After identifying the problems that cause the residual coverage of not-covered Area ($C_2$), developers still need to figure out what problems cause the residual coverage of not-covered Area ($C_1$). Moreover, since Covana cannot detect all types of problems, developers still need to check whether there are other problems that cause DSE not to cover certain branches. By browsing the branch coverage shown in the branch view ($V_5$), developers can figure out how many not-covered branches they still need to deal with. For example, one of these not-covered branches is “Branch #4,” which indicates the false branch at Line 40. To find out the cause, developers can navigate back to the code view and analyze the corresponding branch. Or the developers can navigate back to the contextual view, and use the visual patterns (illustrated in Section 3.5) to identify the problems that cannot be automatically identified by Covana.
Figure 3.2: Visualization of the class `NullException` using `Viscovery`.
By repeating this process, developers can eventually identify all the problems that cause certain branches not to be covered by DSE. If developers do not want to solve all the problems, they can use the overall view and the contextual view to identify the critical areas of the program that they want to cover, and just focus on solving the problems in these areas.

The different types of information presented by the six views and the navigation between the views provide the flexibility to allow developers to find out useful information using the most appropriate view or combination of views.

### 3.4 Approach

Using a code coverage tool is an iterative process. A developer writes tests manually or generates them automatically, runs his coverage tool, and checks coverage. This requires the user to understand what the results of the coverage tool. Unfortunately, output of coverage visualization tools for DSE is very complicated.
My coverage visualization approach shown in Figure 3.3 aims at helping developers identify problems that prevent DSE from covering certain branches, conveniently navigate to the related areas of code to help developers investigate the problems, and provide solutions to the problems. Viscovery takes as inputs the structural coverage information produced by DSE and the problem-analysis results produced by Covana and presents the information in six views.

3.4.1 Overall View

The overall view displays block coverage information produced by DSE at the level of namespaces, classes, and methods. Figure 3.4 shows the overall view for the NullException example explained in Section 3.1. The view contains six columns: Element, Duration, Exploration, Tests, Block, and Methods. Column Element displays the program in the form of a hierarchical tree structure grouped by namespaces, classes, and methods. Column Duration shows exploration time of DSE. Column Exploration presents the number of explored test methods. Column Tests shows the number of generated tests, the number of failed tests, the number of tests that throw exceptions and the number of duplicate tests. Column Block represents block coverage of the method, both textually and with a progress bar. Column Methods displays method coverage.

I adapt the visualization techniques used in the existing test coverage tools such as EclEmma and NCover to present coverage information in a tree hierarchy of the program structure. Using this tree hierarchy, developers are shown which parts of a program are not covered in the
namespace level and can then narrow their search by displaying coverage information at the class or method level.

### 3.4.2 Dynamic Coverage View

The coverage view shows dynamic code coverage information achieved by generated tests. Figure 3.5 displays the dynamic coverage information achieved by all test methods of `NullExceptionTest` shown in Figure 3.4. The view consists of four columns: `Element`, `Block (User Code Under Test)`, `Block (User or Test Code)`, and `#Methods`. Column `Element` displays the structure of source code such as class or method. Column `#Methods` shows the number of executed methods in the source file. Column `Block (User Code Under Test)` shows the number of blocks of the code under test, while Column `Block (User or Test Code)` shows the number of blocks of the code that are part of code under test and are not tested by the selected element in Figure 3.4.

Developers can sort the data by block (user code under test) to see which how many blocks of each element are covered. For example, in this figure, the coverage information is ordered by the percentage of block coverage (user code under test) in ascending order. `AssertActualExpectedException` has the lowest block coverage and one of its constructors has fairly low block coverage (14/49 (28.57%)) in red. This view is linked with the code view (explained in Section 3.4.6). Hence, the coverage view helps developers indicate which elements are mostly not-covered and guides the developers to see which actual part of code is not-covered by using the contextual view or the code view.

### 3.4.3 Contextual View

The contextual view shows all the source code files explored by DSE to help developers understand which areas of the program are covered and which are not. To better interpret the achieved coverage by DSE, VISCOVERY uses four colors: **Green** means that the code under test is covered; **Orange** means that the code under test is not covered; **light blue** means that the test code is
Figure 3.5: The Dynamic Coverage View showing dynamic code coverage achieved by the tests in Figure 3.4

covered; yellow means that the test code is not covered. The visualization of all the source code files explored by DSE gives developers another visual way to view the system-wide structural coverage.

To integrate information about flip counts, this view uses colored squares in the gutter around the edge of each file to represent the flip counts of branches. The color of each square represents the value of flip counts, ranging from green to red. The $RGB$ value of this color is computed as:

$$
R = \left( \frac{FC_{cur}}{FC_{total}} \right) \times 255, G = 255 - \left( \frac{FC_{cur}}{FC_{total}} \right) \times 255, B = 0
$$

where

- $FC_{cur}$ is the flip count of the corresponding branch
- $FC_{total}$ is total flip counts of all branches

Using this color scheme, if a branch has larger flip count than other branches, the corresponding square for the branch will have a redder color. A problem usually occurs when there is a zero flip count of a branch; I emphasize this using a black square. The reason why I compute the color in this way is that if DSE explores the program under test normally and achieves
Figure 3.6: The Contextual View (black squares in the gutter bar represent 0 flip count, green squares represent relatively less flip counts, and red squares represent more flip counts).
high structural coverage, the values of the flip count for every branch in the code are usually similar, and thus the colored squares for each branch have similar colors. However, when BPs occur, there must be some branches with much larger value of flip counts than others, and some branches with zero flip count. In this case, the branches where BPs occur would have red squares in the gutter and the branches with zero flip counts would have black squares in the gutter. Hence, these squares are designed to catch a developer’s attention and assist the developer in figuring out the causes. For example, there are several black squares, green squares, and one red square in Figure 3.6. In designing and using Viscovery, I have identified several visual patterns that can be used to address the challenges of BPs, infeasible branches, and identifying dependencies between not-covered branches.

The contextual view integrates three types of information: structural coverage, flip count, and problem-analysis information. To let developers choose which type of information should be presented, this view provides several filtering options. For example, the view allows developers to choose whether to display only the code under test, only the test code, or both. In addition, by zooming in the code areas of interest, developers can view details of the code.

### 3.4.4 Problem View

The problem view lists details of the identified problems. Figure 3.7 shows how the problem view presents problem-analysis results. Column *Element* shows the identified problems with the branches whose residual coverages are caused by the problems. Column *Type* presents the type of the problem, such as Boundary Problem (BP), Object-Creation Problem (OCP), or External-Method-Call Problem (EMCP) gives a short description of the problem, which is different for different types of problems. For example, the description of an OCP is the object type for which DSE cannot generate a desirable method-call sequence, while the description of a BP is the method where the BP occurs. Column *Analysis Result* shows the detailed information of the problem-analysis results produced by Covana.

Through this view, developers can see the detailed information of the analysis results for
each problem, including where problems occur, and which branches whose residual coverage is caused by the problems. Detailed problem-analysis information can help developers figure out how to provide guidance to DSE for problem solving.

### 3.4.5 Branch List View

The branch list view shows the details of the branches in the program under test. Figure 3.8 shows the branch list view for an example program under test. Column *Branch* shows the branch in the program under test. Column *Document* presents the class name containing the branch. Column *Line* shows the line number of the branch. Column *Covered Times* shows how many times the branch is covered by the executed tests. Column *Flip Count* shows the flip count of the branch. Column *Type* presents the type of the branch, which can be *implicit* and *explicit*. Implicit branches are branches that are added when a field of an object is accessed or a member method of an object is invoked, while explicit branches are the branches of conditional
statements in the source code.

This view is designed to help developers identify not-covered branches by checking branch coverage. Given a loop whose iteration count depends on a program input, DSE may keep flipping the branch that terminates the loop, increasing the total iteration of the loop. In this case, the flip count of the branch becomes much larger than the other branches’, and the subsequent branches may suffer from starving, having zero flip counts. Thus, by checking the flip count of each branch, developers can identify BPs and the branches whose residual coverages are caused by the BPs. This view highlights in red the branches that are related to problems identified by Covana. When a user selects a certain branch in the view, the code view automatically shows the code.

3.4.6 Code View

The code view shows the source code with different background colors for presenting statement coverage information. For consistency, the color scheme of the background colors are the same as the contextual view. To integrate the problem-analysis result, branches whose residual coverage is caused by the problems are annotated with labels to show the type of the problems. Figure 3.9 illustrates how the code view shows source code with three annotations: structural coverage information, flip count information, and problem-analysis results with recommendations.

The Covered Times annotation in light blue displays which test classes and test methods executed the branch as well as the number of times that the branch is covered. As you can see, the branch is covered by the BubbleSortTest1R class and the Sort2 method, and the BubbleSortTest1D class and the Sort method. The second annotation Flip Counts, indicated in green presents the number of times that this branch flipped by Pex. The third Problem annotation indicated in red shows the detail information and recommendations to fix the problem. The colors used here may help users identify and search the annotations [19, 17]. As we can see, the solution to cover the block is to modify existing Pex attributes, to change the search strategy, or to refer to external links.
Figure 3.9: The Code View. When developers move the mouse over an “BP,” the problem annotation (C) is opened to show the analysis result of the problem and recommendations. A problematic branch is annotated with Covered Times (A), Flip Counts (B), and Problem.
3.4.7 Implementation

To apply the coverage visualization approach in practice, I designed and implemented the prototype as a Rich Client Platform (RCP). I chose to develop an RCP due to the reusability and extendability benefits that Eclipse frameworks provide. The implementation lets users arrange the views freely on the screen and work with them in parallel. The navigation between different views enables the users to look for useful information using the most appropriate view or view combination.

I used Pex [125], a white-box testing tool that generates test inputs by performing DSE, to explore real-world open source projects, and Covana [130] to analyze the problems that DSE tools face. The structural coverage information produced by Pex and problem-analysis results produced by Covana are in XML files, which are fed into Viscovery as inputs. However, Viscovery is not limited to any particular DSE tool or problem-analysis tool. The prototype can be extended easily to use other DSE and problem-analysis tools that provide the required information.

3.5 Application of Viscovery

In this section, I first describe a visual pattern (Section 3.5.1) and the visualization of three commonly-used search strategies of DSE tools (Section 3.5.2). The visualization provides developers with visual feedback on how search strategies affect the overall coverage and help the users decide what guidance should be provided.

3.5.1 Residual Coverage of Branches Caused by BPs

Viscovery can visualize structural coverage information produced by DSE and problem-analysis results produced by Covana for real-world projects, such as xUnit. xUnit is a unit testing framework for .NET program development. xUnit includes 223 classes and interfaces, totaling about 13 thousand lines of code.
Figure 3.10 shows a screenshot of a visualization of the class `NullException` shown in Figure 3.1 using the contextual view. The red square in the gutter indicates that the branch at (A) has a larger flip count value than the other branches. As discussed in Section 3.3, the statement at (A) is a `foreach` loop that causes a BP. Since branches whose residual coverage is caused by BPs must have zero flip counts, the developers can identify these branches by looking for the branches that have black squares in the gutter. In Figure 3.10, the branch that has black squares in the gutter is at (B). By examining the source code, the developers can confirm that the residual coverage of the branch is indeed caused by the BP. This visual pattern, “the first red square then a black square,” guides developers in searching for branches whose residual coverage is caused by BPs using Viscovery.

### 3.5.2 Visualization of Search Strategies in DSE

To achieve high structural coverage, DSE uses different search strategies to decide which branch node should be flipped before the others. Since the contextual view and the code view of
Viscovery both show the information of the flip counts for each branch, Viscovery can be used to provide visual feedback to help the developers understand the effectiveness of different strategies. To demonstrate how Viscovery visualizes the search strategies of DSE, I applied DSE and Covana on the illustrative example of Fitnex [133] using three commonly-used search strategies:

- Depth-First: a search strategy where the last not-flipped branch node in an explored path is chosen.
- Random: a search strategy where branch nodes to flip are chosen randomly in the already explored paths (but no branch node is selected twice).
- Fitnex: a search strategy where branch nodes to flip are chosen based on the fitness values to achieve higher structural coverage.

Each search strategy lets DSE explore the example program until BPs occur to stop the exploration. Figure 3.11 shows the visualization of these three search strategies of DSE on the example. The first row shows the contextual views for the Depth-First (A), Random (B), and Fitnex (C) search strategies. The second row shows the corresponding code views. From the visualization, I can easily see that Depth-First (A) achieves the worst coverage, Random (B) performs better than Depth-First, and Fitnex (C) achieves full coverage. By looking at the colored squares in the gutters, I can see that using Depth-First, DSE keeps flipping the branch nodes in the for statement to unroll the loop. As a result, subsequent branches are not flipped by DSE.
Figure 3.11: Visualization of three search strategies of DSE using Viscovery.
In (B), the colored squares for the two branches are similar, indicating that their flip count values are almost the same. By checking the source code shown on the bottom of (B), I can see that the values of flip count for these two branches are 56 and 47, respectively. This result shows that the Random strategy evenly chooses a branch node to flip and leads to better coverage than the Depth-First strategy. However, the probability of flipping the branch nodes in a specific order to cover the **true** branch `if(x == 110)` is relatively low, and thus the Random strategy cannot achieve full branch coverage before a BP occurs. Instead of randomly choosing a branch node to flip, the Fitnex strategy computes fitness values for branches and chooses the branch node that has the best fitness value to flip. As a result, the Fitnex strategy guides DSE to achieve high branch coverage in fewer explorations. In (C), I can see that the Fitnex strategy achieves full branch coverage before the BP occurs.

This example demonstrates that Viscovery can be employed to visualize the effectiveness of search strategies of DSE. By looking at the colored squares in the gutter, developers can easily determine how DSE chooses different branch nodes to flip. This visual feedback can help the developers understand the effectiveness of the search strategy under investigation and improve the search strategy if necessary.

### 3.6 Experiment

To evaluate the usability of Viscovery in helping users investigate problems and provide guidance, I conducted an empirical study. In particular, I investigated the following research questions:

- **RQ1**: [Effectiveness] How effective is Viscovery in helping developers understand problems and provide guidance?

- **RQ2**: [Efficiency] How efficient is Viscovery in helping users understand problems and provide guidance?
• RQ3: [User Satisfaction] How are the participants satisfied with VISCOVERY in terms of problem understanding and providing guidance?

3.6.1 Design

Participants. The target population of VISCOVERY was software developers who had experience of object-oriented programming languages, such as C# or Java, and basic knowledge of software testing. Table 3.1 shows the demographics of the participants. Through mailing lists or personal contacts, I recruited 12 participants (2 female)\(^5\): 2 undergraduate and 9 graduate students in computer science department at North Carolina State University, and 1 software engineer from industry. On average, each participant had 9.3 years of software development experience and 4.9 years of object-oriented programming experience. They also had 3.0 years of software testing experience. No participant had Pex experience. All participants were not aware of VISCOVERY before the study.

Tools. I provided participants with three code coverage visualization tools: Pex reports (\(P\)), Pex reports + textual problem results (\(P+T\)), and VISCOVERY. The comparison of \(P\) and \(P+T\) gives us insights on whether textual problem-analysis results can effectively help developers understand the problems faced by DSE tools. The comparison of \(P+T\) and VISCOVERY gives us insights on whether visualization of problem-analysis results can further improve developers’ problem understanding. Pex reports show the code coverage result information in HTML. Textual problem results contains the problems identified by a problem analyzer \([130]\) and all the branches executed by the test code. I converted the textual results into Excel format for participants to navigate. VISCOVERY visualizes the textual results. During the study, I called these three tools Tool A, Tool B, and Tool C, respectively. I used a within-subjects design. The order of tools and tasks was completely counterbalanced.

\(^5\)The consent form and pre-questionnaires are presented in Appendix A.
Table 3.1: Descriptive statistics of the participants (Viscovery).

<table>
<thead>
<tr>
<th>Participant</th>
<th>Development (yrs)</th>
<th>Testing (yrs)</th>
<th>OOP (yrs)</th>
<th>IDEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>Eclipse, Visual Studio</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>Eclipse, Xcode, NetBeans</td>
</tr>
<tr>
<td>P3</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>Eclipse, Visual Studio</td>
</tr>
<tr>
<td>P4</td>
<td>4</td>
<td>2</td>
<td>2.5</td>
<td>Eclipse, Visual Studio, NetBeans</td>
</tr>
<tr>
<td>P5</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>Eclipse, Visual Studio, NetBeans</td>
</tr>
<tr>
<td>P6</td>
<td>20</td>
<td>1</td>
<td>5</td>
<td>Eclipse, Visual Studio</td>
</tr>
<tr>
<td>P7</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>Eclipse, Visual Studio, NetBeans</td>
</tr>
<tr>
<td>P8</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>NetBeans</td>
</tr>
<tr>
<td>P9</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>Eclipse</td>
</tr>
<tr>
<td>P10</td>
<td>5</td>
<td>0</td>
<td>2</td>
<td>Visual Studio</td>
</tr>
<tr>
<td>P11</td>
<td>2.5</td>
<td>1.5</td>
<td>2.5</td>
<td>Eclipse</td>
</tr>
<tr>
<td>P12</td>
<td>30</td>
<td>2</td>
<td>3</td>
<td>Eclipse</td>
</tr>
</tbody>
</table>

Mean 9.3 3.0 4.9 -
**Code Subjects.** I chose subject programs from various open source projects in C#: DotNetZip\(^6\), Data Structures and Algorithms (DSA)\(^7\), Json.NET\(^8\), Stuff.NET\(^9\), and xUnit.NET\(^10\). I first applied Pex to generate parameterized unit tests (PUTs) \([126]\) for the methods of the projects under test. I filtered the methods that contain C# specific features (such as LINQ), so that participants who use other object-oriented programming languages (such as Java) can understand the code easily. I then applied Pex to generate test inputs for one or multiple PUTs of each project, and collected the test results, the coverage, and the problems (BPs, OCPs, and EMCPs) identified by the problem analyzer. The collected information is then used in the study.

**Equipment.** In my experiment, participants used a laptop with the screen having a 1400 × 900 resolution and an external monitor having a 1600 × 900 resolution. Participants used the main monitor for answering questions and the external monitor for using the tools.

### 3.6.2 Procedure

**Pilot Study.** I recruited seven participants for my pilot study. Based on the feedback from the pilot study, I adjusted the difficulty and time limit of each task, and revised the training materials. In addition, I decided to transform the textual reports in XML format into Excel format, so that participants can have a structured view of the textual reports and more easily locate information based on columns.

**Consent Form and Pre-questionnaire.** Through the consent-form, I informed the participants that their tasks were being recorded during the experiment. I collected the demographics through the pre-questionnaire.

**Training.** I provided a 15 minute training session to each participant. First, I briefly explained manual testing, automated testing, and their differences. I then showed how Pex \([125]\) works and introduced how to use Tool A, Tool B, and Tool C. To help participants get familiarized

\(^6\)[http://dotnetzip.codeplex.com.]
\(^7\)[http://dsa.codeplex.com/].
\(^8\)[http://json.codeplex.com].
\(^9\)[http://stuffnet.codeplex.com].
\(^10\)[http://xunit.codeplex.com/].
with the tools, I gave them 10 minutes to practice the tools. I also provided the print outs of the training material for participants’ references. I allowed participants to ask questions if they had difficulties in using the tools during the experiment.

**Tasks**\(^{11}\). Next, I asked participants to use the tools to complete a set of nine tasks within 6 minutes each. Each task consisted of five questions:

- **Q1**: (a) Identify the class name with the lowest block coverage first (.5 point), then (b) Identify the method name with the lowest block coverage ratio (.5 point).

- **Q2**: Identify the line number or the offset number of the branch in the method (1 point).

\(^{11}\)The detailed questions of each task are presented in Appendix A.
• Q3: (a) Identify the test class name that executed the method (.5 point), then (b) Identify the test method name that executed the method (.5 point).

• Q4: (a) Select the primary problem why that part of the code is not covered (.5 point), then (b) Provide the reason you selected the problem (.5 point).

• Q5: Provide your solution for Pex to cover that part of the code (1 point).

I listed these questions based on the testing tasks for improving residual structural coverage. The first and second questions are prerequisites for other questions because software testers usually want to find out first which part of code under test has the lowest coverage. The third question is necessary for testers to decide which part of the code is supposed to be covered by executing a test method (i.e., a PUT) with the test inputs generated by Pex. The fourth and fifth questions are the most important, because testers could not provide guidance to Pex without knowing reasons and solutions to them. I did not ask them to change test code, because it is difficult to create a factory method or mock objects as solutions within the given time limit. How well testers implement the solutions is also beyond the scope of this experiment.

**Post-questionnaire**

After the experiment, I asked participants to answer the post-questionnaire. The participants were asked to compare the three tools and provide ratings for the six views of Viscovery. The whole experiment for each participant took 70-90 minutes including the training session and the practice.

**Data Collection and Analysis.**

*Correctness data.* To measure the effectiveness, My colleague and I graded participants’ answers. I used Cohen’ kappa [22] for the inter-rater reliability. I resolved conflicted rates through discussions. I performed Friedman test [23] to analyze the ratings of three groups, since the number of participants may not have normal distribution. I used \( \alpha = .05 \) as the significance level. I ran post-hoc analysis using a Wilcoxon Signed-Rank test to evaluate all pairwise differences between the three tools with the Bonferroni correction.

---

12The detailed post-questionnaires are presented in Appendix A.
Timing data. I measured the time taken for each question of each task. If the given time (6 minutes) for each task is over before finishing the given questions, I treated as missing data. I performed a Friedman test for analyzing the timing data of the three tools.

Feedback from Participants. Through post-questionnaires, I collected how the participants are satisfied with the tools, using Likert scale [63] ranging from “Strongly disagree (1)” to “Strongly agree (5)” for the level of agreement. I ran a Friedman test for analyzing the user satisfaction and Wilcoxon Signed-Rank test for the post-hoc analysis.

3.7 Results

3.7.1 RQ1: Effectiveness

Figure 3.13(A) shows the mean grade score per task by tools with almost perfect agreement between the raters (Cohen’ $k = .972$). Participants with Viscovery obtained a mean score of 4.5 (Standard Deviation = .6). In contrast, $P$ users obtained 2.7 ($sd = 1.1$) and $P+T$ users obtained 3.0 ($sd = .9$), respectively. I found a significant difference on the mean grade obtained ($p < .001$). Post hoc analysis revealed that Viscovery users outperformed $P$ users ($p < .001$) and $P+T$ users ($p < .001$).

To get a more fine-grained analysis at task completion, I investigated how participants solved each question. Figure 3.13(B) shows the mean grade per question of tasks. For Q1 and Q2, there was no significant difference among tools ($p > .05$).

Viscovery users obtained .94 for Q3, while $P$ users got .71 and $P+T$ users got .77. There was a significant difference among tools for Q3 ($p = .002$). Viscovery users obtained higher mean score than $P$ users ($p = .001$) and $P+T$ users ($p = .005$).

For Q4 (problem investigation), Viscovery users obtained .82, whereas $P$ users got .29 and $P+T$ users got .53. As I expected, there was a significant difference among tools ($p = .0001$). Pairwise comparisons indicated that Viscovery users performed significantly better than $P+T$ users ($p = .0001$) and $P+T$ users also performed significantly better than $P$ users ($p = .0001$).
Viscovery users mostly referred to the problem annotation in the Code view, while $P+T$ users investigated the given Excel files or browsed the uncovered part of code. The participants with $P+T$ answered less correctly than the participants with Viscovery. For example, a participant with $P+T$ answered “Calling WriteLine method” or “examining source code. Log didn’t make any sense to me,” while a participant with Viscovery answered “create appropriate mock object to pass into the method from the test code.”
Figure 3.13: Effectiveness of Tools.
For Q5 (solution providing), VISCOVERY users obtained .95, whereas P users got .46 and 
$P+T$ users got .33. I found that there was a significant difference among the tools ($p < .001$). 
Post-hoc analysis revealed that VISCOVERY had a significant difference from $P+T$ ($p = .0001$) 
and $P$ ($p < .001$), respectively. Like Q4, $P$ and $P+T$ users answered less correctly. For example, 
a participant with $P+T$ answered “try a test that exercises code to generate the exception” for 
an EMCP, whereas the participant with VISCOVERY answered “provide mock objects for the 
Path...” for the same kind of problem.

Overall, I found VISCOVERY users outperformed $P$ users and $P+T$ users for identifying test 
classes and test methods, identifying the reasons why the part of code was not covered, and 
figuring out the solutions to the problems.

3.7.2 RQ2: Efficiency

Figure 3.13(C) shows the task completion times per tool. I excluded tasks that participants 
could not complete. The mean completion time for each task of was 223.9 seconds (SD = 86.4) 
for VISCOVERY users, 291.8 seconds (SD = 59.3) for $P+T$ users, and 313.8 seconds (SD = 64.20) 
for $P$ users. I found that there was a significant difference among the tools ($p = .016$). Pairwise 
comparisons revealed that VISCOVERY users were significantly faster than $P$ users ($p = .01$); 
however, there was no significant difference from VISCOVERY users from $P+T$ users ($p = .19$).

Figure 3.13(D) shows task completion times per question. I found a significant main effect of 
VISCOVERY on time taken for Q1 ($p < .001$). Pairwise comparisons showed that VISCOVERY users 
answered significantly faster than $P$ users ($p = .0003$) and $P+T$ users ($p < .001$), respectively.

However, there was no significant difference between $P+T$ users and $P$ users ($p = .394$). 
During the experiment, I observed most VISCOVERY users browsed the Coverage view and 
quickly found the class having the lowest coverage and the method having the lowest coverage; 
however, $P$ and $P+T$ users spent more time to find the class and the method.

There was no significant difference among tools for Q2 ($p = .323$). To identify the uncovered 
code, VISCOVERY users needed to browse the Code view, to zoom in the code using the
Contextual view, or to click the class in a tree of the Coverage view. However, VISCOVERY users mostly tried to click the method with the lowest coverage that they thought.

There was a significant difference ($p = .009$) for Q3. Pairwise comparisons revealed that there was a significant difference between VISCOVERY users and $P+T$ users ($p < .001$). However, there were no significant differences between $P$ users and $P+T$ users ($p = .05$), and between $P$ users and VISCOVERY users ($p = .104$), respectively. Most VISCOVERY users quickly found the test class and the test method that executed the method that participants answered for Q2 by referring to the covered times annotation in the Code view. However $P$ and VISCOVERY users tried to browse the Pex report, manually investigated the test code, and searched for the associated method call in the test code.

For Q4, VISCOVERY users took 71.9 seconds, whereas $P$ users took 141.7 seconds and $P+T$ users took 86 seconds. I found that there was a significant difference among tools ($p = .009$). Multiple comparisons revealed that $P$ users took significantly more time than $P+T$ users ($p = .001$) and VISCOVERY users ($p < .001$). $P$ users mostly spent time to navigate Pex reports manually and browse around the uncovered code to find the reason why the part of code is not covered. $P+T$ users browsed textual coverage results to find the reason and manually map the location (such as the line number) in the textual information to the uncovered code.

For Q5, VISCOVERY users took 29.7 seconds to answer, while $P$ users took 54.8 seconds and $P+T$ users took 53.4 seconds. However, there are no significant differences among tools for Q5 ($p = 0.395$). VISCOVERY users spent the most time to type the answer after finding the solution in the problem annotation of the Code view, whereas $P$ and $P+T$ users tried to figure out the solution just by navigating the code. A participant with $P$ took only 33 seconds with no answer for Q5, because he spent most of the given time to answer previous questions, while the participant with VISCOVERY took 24 seconds with a correct answer.
3.7.3 RQ3: User Satisfaction

The mean ratings of $P$, $P+T$, and Viscovery were 3, 2, and 4, respectively. Overall, participants preferred Viscovery to $P$ and $P+T$ ($p = .0031$). There was a significant difference between Viscovery and $P$ ($p = .0022$) and between Viscovery and $P+T$ ($p = .0053$), whereas there was no significant difference between $P$ and $P+T$ ($p = 1.0$).

3.7.4 Threats to Validity

The problems used in each testing task may affect the results. To mitigate this threat, I randomized the sequence of tools ($P$, $P+T$, and Viscovery). Moreover, I used similar length of code and difficulty for the tasks. To measure completion time of each question, when the participants stopped entering the question, I considered it as the ending time of the question. When I compared completion time of each question, I included all questions of each task that participants took time to answer whether their answers are correct or not, because it would not be enough to compare the correctness per question. Another threat comes from the measurement of time taken. To measure correctness of each task, I used Cohen’s kappa. One of the raters already knew which tools the participants used, but the other was blinded. The threats to external validity primarily include the degree to which the participants, the subject programs, faults or program changes, and test cases are representative of true practice. This threat could be reduced by more experiments on wider types of subjects in future work. However, 5 graduate students participated in this experiment had previous industrial experience and more than 10 years of software development experience. Faults in Viscovery and Covana might cause such effects while identifying problems that DSE encountered. To reduce these threats, I have manually inspected the results.
3.8 Discussion

Although the research questions are answered, to improve Viscovery’s capability in assisting the developers to identify problems and provide better usability, I plan to integrate more features into Viscovery.

Integrating Viscovery with Pex on Visual Studio. The current implementation of Viscovery is loosely coupled with Pex in Visual Studio. That is to say, Viscovery reads in the XML-format outputs of Pex and Covana. If reports of other DSE tools are in the same format, Viscovery can visualize their testing results too. As mentioned in Section 3.7.1, I observed that some of the participants using Viscovery got confused when they navigated the code. A participant said, “Having the windows be more in sync with each other on Tool C [Viscovery] would be helpful. I expect when I click on a method that I’m taken immediately to that part of the source code.” A participant said, “Navigation from method to code view would also be helpful.” If Viscovery is integrated with Visual Studio and uses the infrastructure of the tool, it can provide more fine-grained navigation.

Improving View Layout. During the experiment, I observed that the participants using Viscovery did not frequently use the Contextual view compared to the Code view to answer questions. The Contextual view lays in the center and occupies the same space as the Code view. Most participants even made the Code view larger and made the Contextual view smaller. It is useful to present the whole code in graphical mode with test coverage results; however, most participants just skimmed the view briefly and started browsing the Code view.

Missing Problems. Viscovery mainly depends on the problem-analysis results of Covana. Viscovery visualizes problems analyzed by Covana to indicate why a certain part of code is not covered; however, it does not mean that Viscovery provides all the reasons for each not-covered code under test. Some part of code may not be covered for other reasons.
3.9 Related Work

Besides recommendation systems (see Section 6), Viscovery is also related to code coverage and software visualization.

Berner and colleagues reported study results of the use of unit testing [10]. They recommended that it needs to keep the feedback cycle among coding and testing through visualization and annotations. Lawrance and colleagues conducted an empirical study to investigate the effects of structural coverage visualizations on developers’ testing behaviors [59]. Persheid and colleagues proposed an approach for helping developers prioritize their testing effect by providing feedback [91].

Pavlopoulou and Young proposed a technique for monitoring residual structural coverage [89]. In the deployed environment, they tried to provide richer feedback from actual use of deployed software to developers. Their approach mainly focuses on how to reduce the performance overhead for gathering structural coverage from deployed software, while my approach focuses on how to effectively visualize residual structural coverage and problem-analysis results, helping the developers to understand the problems, and recommending solutions to the problems. Xusheng and colleagues proposed Covana, an approach to effectively identify problems by analyzing the runtime information of DSE and filtering out irrelevant problems using the residual structural coverage. By adapting the problem-analysis results from Covana [130], my new approach uses visualization techniques to address the challenges faced by Covana by providing visual patterns.

Jones and colleagues developed the Tarantula approach [54] to visualize test-case coverage information for fault localization. In the Tarantula approach, the greener a statement is, the more times it is executed by passing test cases, while the redder a statement is, the more times it is executed by failing test cases. Similar to their approach, the Contextual view of Viscovery assigns redder colored square to a branch whose flip count value is larger, and greener square for a branch whose flip count value is smaller. I also use background colors to indicate whether the statement belongs to the code under test or test code and whether it is covered or not. Their approach identifies possible faulty code based on the visual patterns of test coverage, whereas
Viscovery illustrates the visual pattern of the colored squares and the background colors to help the developers identify problems and uses other views to help figure out solutions for these problems.

Storey and colleagues developed the SHriMP Views system [118], which provides visualization based on zooming to display hierarchical views of software. Their approach is mainly concerned with exploring source code and its hierarchical structure, whereas the code view of Viscovery presents the source code with the information of structural coverage and problem-analysis results, and aims to help developers investigate source code to understand problems and figure out solutions to those problems.

3.10 Conclusion

High structural coverage can give developers high confidence of the quality of the program under test. Dynamic Symbolic Execution (DSE) tools can be employed to automate the process of test generation. When DSE tools are applied to complex program in practice, these tools face various problems that prevent the tools from achieving high structural coverage. In this chapter, I proposed a solution recommendation system with a coverage visualization approach that is designed to assist developers in understanding problems that DSE tools face and providing guidance to address the problems. I implemented my approach and evaluated it with 12 participants. The results show that Viscovery helps developers use DSE tools effectively, efficiently, and more enjoyably, compared to DSE tools and DSE tools with textual output of problem-analysis results. I believe these results provide a first step towards making complex testing tools more accessible to a broad scope of developers.
Chapter 4

Recommending Code to Fix Bugs

In this chapter, I present an interactive approach\(^1\), based on the follow-through principle, to fix bugs by recommending context-sensitive code. The rest of this chapter is organized as follows. Section 4.1 introduces the approach. Section 4.2 presents an example of the motivation behind my work. Section 4.3 describes my approach for assisting developers interactively with fixing defects. Section 4.4 presents an experiment to evaluate the usability of FixBugs. Section 4.5 presents quantitative and qualitative results of the experiment. Finally, Section 4.6 concludes this chapter.

4.1 Introduction

Defect finding tools like Lint [53], FindBugs [47], and Clang [138] help developers improve software code quality. Research on these tools has primarily focused on detecting new types of defects such as coding errors, confusing code, style violations [137, 143] and security vulnerabilities [65]. Though useful, in my previous study developers indicated that fixing software defects is as important to improving software quality as finding them [51].

On the other hand, follow-through is a principle that is used in a variety of disciplines to successfully complete tasks [9, 35, 43, 92]. The idea behind this principle is a simple one: during

\(^1\)This work was done in collaboration with Titus Barik, Brittany Johnson, and my advisor Emerson Murphy-Hill.
a task, what happens after an action is just as important as the action itself.

The same principle should apply to software engineering, but software development tools today aren’t following through. The problem is that operationalizing follow-through for tools is a challenging one, because it requires tool designers to consider the usage of their tools holistically, rather than in isolation. A follow-through principle not only requires that developers have a rich toolbox of available tools, but also requires that these tools are provided (and withdrawn) at the right place and time, that the *interactions* between different tools work cohesively, and that the tools appropriately leverage their capabilities while simultaneously respecting developers’ intentions.

The typical ways tools support fixing defects, then, if they support automatic fixes at all, is by offering a one-shot, *non-interactive* resolution. A one-shot approach is one in which the tool requires no developer intervention to resolve the defect after initial selection. As one example, the Eclipse [140] programming environment implements such a strategy through the use of *Quick Fixes*. A Quick Fix provides a list of suggestions along with a code preview or description of the resolution. For instance, the “Add final-modifier to field” Quick Fix detects occurrences of fields that are potentially missing the *final* property. The resolution of this fix is straight-forward and even self-describing: add the *final* keyword to the corresponding field.

From the perspective of follow-through, however, solely removing the compiler error is only a single component of successful task completion. It’s also important that the developer understands the impact of the change introduced by the fix, has the opportunity to evaluate the design space of alternative fixes, and is given the ability to influence the fix in the event that the fix only partially supports their intentions. Worse, not all errors have straight-forward fixes like the aforementioned *final* fix. Concretely, I randomly sampled 131 of the 548 defects from FindBugs and manually classified their fix as either *non-interactive* or *interactive*. The results indicate that 80 of the 131 defects (61%) would require interactive resolution, with some developer intervention necessary to resolve one or more ambiguities.

I think tools that implement the principle of follow-through can significantly help developers.
public static Object invoke(Object obj, String mthName, Class argType, Object arg) {
    try {
        Method mth;
        mth = obj.getClass().getMethod(mthName,
            new Class[] { argType });
        return mth.invoke(obj, new Object[] { arg });
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

Figure 4.1: Code containing the Catch Exception defect.

The contributions of this work are:

- A follow-through approach, called FixBugs [114], to constructing software development tools, which I apply to the domain of defect finding tools.
- Evaluations that demonstrate the benefits and trade-offs of FixBugs.
- Design guidelines for implementing the follow-through approach in other software development tools.

### 4.2 Motivating Example

In this section, I illustrate some of the limitations of existing one-shot Quick Fix approaches when the tool does not embody follow-through principles. Consider a hypothetical developer, Kevin, who applies a defect finding tool on source code from Apache Ant\(^2\) shown in Figure 4.1. FindBugs detects a Catch Exception defect and places a bug marker (Line 8) to indicate the defect location to him, shown in Figure 4.2. Specifically, this catch block is considered “dodgy code” because the construct accidentally catches RuntimeException as well, masking potential defects.

\(^2\)http://ant.apache.org/
He then clicks on the marker to invoke Quick Fix for the suggested solutions. However, Eclipse only suggests “Add finally block.” This suggestion is not useful to fix the defect, because it is not the correct fix for this particular problem. In fact, FindBugs did not offer a suggestion at all, and the suggested fix is one that is provided by default in Eclipse.

Because the tool has failed to follow-through, Kevin must fix the defect manually. Since the tool is no longer supporting his task, this is a time consuming process [58]. He investigates the code in the editor to determine which lines can generate exceptions, investigates which exceptions are actually being thrown, and then must decide, for each exception, whether it should be thrown, caught, or caught explicitly and rethrown.

In contrast to a one-shot tool, a follow-through tool would provide Kevin with support to correct the defect, even if the tool is unable to identify a fix that is immediately satisfactory to Kevin. In the next section, I discuss my approach to follow-through for program analysis defects in different scenarios, including the Catch Exception defect that Kevin encountered.

4.3 Approach

In this section, I describe my approach to implementing the follow-through principle in a tool I designed and built for the Eclipse developer environment, called FixBugs. FixBugs is an extension of FindBugs, a static analysis tool for Java and detects defects across several categories. I choose to use FindBugs defects instead of those from the Eclipse Java compiler because FindBugs provides a rationale (through a “Bug Info” view) for why each defect is actually a problem. From this corpus, I filtered out one-shot, easily solvable defects having a single trivial solution. Of the remaining defects, I highlight four defects that serve as a means to demonstrate the benefits of follow-through: Catch Exception (Section 4.3.1), String Concatenation (Section 4.3.2), Open Stream Exception Path (Section 4.3.3), and Missing Argument (Section 4.3.4).
4.3.1 Catch Exception

Consider the code example illustrated in Figure 4.1, containing the Catch Exception defect. As shown in Figure 4.3, FixBugs first suggests multi-catch\(^3\) as a default suggestion that catches multiple exceptions of different types in a catch clause and handles the exceptions in the same way. If the development environment is based on Java SE 6 or earlier, a list of uni-catch\(^4\) clauses are suggested as a default suggestion. For Java SE 7 and onwards, canonical multi-catch clauses are used, initially grouping the exceptions by the responsible method.

FixBugs uses colors to map method invocations to their respective thrown exceptions. Using the mapping, developers can easily notice that the invoke method in the try block with a light blue border will throw an IllegalAccess-, IllegalArgument-, and InvocationTarget-

\(^3\)http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20

\(^4\)Uni-catch is a clause that catches only one type of exception (Figure 4.2), whereas multi-catch does multiple types of exceptions (Figure 4.3).
Figure 4.3: Suggested fix for the Catch Exception defect shown in Figure 4.1. FixBugs allows developers to throw or to catch exceptions by supporting the drag-and-drop action. Currently, a user is dragging `SecurityException`. FixBugs displays drop zones in light grey and highlights the closest drop zone with a thicker border.

exception. Although not demonstrated in Figure 4.3, if two or more method invocations were to throw the same exception, then FixBugs presents the exception in a separate catch clause with a dark gray highlight.

**Suggestion Popups.** As shown in Figure 4.3, when the user invokes FixBugs, a movable suggestion popup opens that presents prepackaged solutions to use while fixing the defect. The “(Original)” option is for reverting to the original buggy code, especially useful for comparing the original code to the suggested code. FixBugs enables “My Last Change” when the user performs drag-and-drop at least once. This option helps the user revert to their changes made while using FixBugs if they want to compare them to, say, a prepackaged fix.

To provide a default suggestion for the Catch Exception defect, and other defects in this chapter, I mined 24 open source projects. The data mining revealed that developers, when dealing with exceptions, append a throws clause to the method signature 45.1% (4,462) of the time and catch the exception the remaining 54.9% (5,424) of the time. In a one-shot tool, developers would have to be satisfied with these prepackaged suggestions, and resort to manual
Drag-and-drop. Implementing the follow-through principle, FixBugs continues to support the developer if they are not satisfied with the prepackaged fixes. The tool allows them to use drag-and-drop to visually manipulate where the exceptions should be handled as demonstrated in Figure 4.3. FixBugs displays a dotted border on each exception to indicate that exceptions are draggable and to differentiate between non-draggable items, such as method invocations. When the user begins to drag an exception, FixBugs displays allowable drop zones with light grey boxes, and dynamically highlights the closest drop zone to the current location of the mouse cursor. After the user drops the exception on a preferred drop zone, the drop zones disappear and FixBugs automatically generates the code in the editor to reflect the change. The exceptions maintain their colors until the developer indicates they have finished resolving the defect.

4.3.2 String Concatenation

Figure 4.4 shows an example from the log4j source code consisting of `getAddress` which returns an Internet Protocol (IP) address of a local host. The method contains a `for` statement (Lines 8 through 13) and a `String` object (Line 7) that is concatenated repeatedly in the loop. This code may produce a potential performance defect, because string concatenations in a loop create many temporary objects and increase garbage collection [11]. To avoid this performance defect, it is better to use a `StringBuffer` or `StringBuilder` to build the `String`.

As shown in Figure 4.5, FixBugs suggests the code for fixing the defect. By default, FixBugs provides a code suggestion using `StringBuffer`. Mining the 24 open source corpus, I found that `StringBuffer` is used more often (70.1%) than `StringBuilder` (29.9%) and use `StringBuffer` as the default suggestion.

FixBugs first finds the `String` variable being concatenated in a loop where the marker is located and creates a `StringBuffer` variable right before the loop. Then, FixBugs replaces all

---

5 Mining results for default suggestions are presented in Appendix B.3.
6 The primary difference between `StringBuilder` and `StringBuffer` is that `StringBuilder` is not synchronized.
public String getAddress() {
    String result = null;
    try {
        InetAddress addr = InetAddress.getLocalHost();
        byte[] ip = addr.getAddress();
        int i = 4;
        String ipAddr = "";
        for (byte b : ip) {
            ipAddr += (b & 0xFF);
            if (--i > 0) {
                ipAddr += ".";
            }
        }
        result = ipAddr;
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return result;
}

Figure 4.4: Code containing the String Concatenation defect. The `getAddress` method has a `for` loop and a variable of `ipAddr` that is concatenated in the loop.

Concatenations in the loop with the `append` method before copying the value of the `StringBuffer` to the original `String`.

**Color Annotations.** FixBugs implements the follow-through principle and supports the developer after selecting a resolution by showing them the impact of the change. Specifically, FixBugs applies color annotations directly on the editor to denote the parts of code that have been added or changed. Code in **light green** was newly added and code in **orange** was modified. In addition, the gutter contains a green plus sign icon to indicate added code, and a pencil icon to indicate a section of changed.

### 4.3.3 Open Stream Exception Path

Figure 4.6 shows an example that contains a potential defect labeled by FindBugs as Open Stream Exception Path. The `read` method creates an `InputStream` object (Line 3) in a `try` statement. The `Preferences.importPreferences` method (Line 4) takes the stream object as an argument.
Figure 4.5: Suggested fix for the String Concatenation defect shown in Figure 4.4. In top right is the suggestion popup. Color annotations help developers determine the impact of the change on their code.

```
public void read() {
    try {
        InputStream in = new FileInputStream("f.txt");
        Preferences.importPreferences(in);
        in.close();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InvalidPreferencesFormatException e) {
        e.printStackTrace();
    }
    return result;
}
```

Figure 4.6: Code containing the Open Stream Exception Path defect.
public void read() {
    InputStream in = null;
    try {
        in = new FileInputStream("f.txt");
        Preferences.importPreferences(in);
    } catch (IOException e) {
        e.printStackTrace();
    }
    catch (InvalidPreferencesFormatException e) {
        e.printStackTrace();
    }
    finally {
        if (in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

Figure 4.7: Suggested fix for the Open Stream Exception Path defect shown in Figure 4.6.

public void read() {
    try (InputStream in = new FileInputStream("f.
        Preferences.importPreferences(in);
    } catch (IOException e) {
        e.printStackTrace();
    }
    catch (InvalidPreferencesFormatException e) {
        e.printStackTrace();
    }
}

Figure 4.8: Another suggested fix for the Open Stream Exception Path defect shown in Figure 4.6.
Then, the method closes the stream object (Line 5). However, the `importPreferences` method may throw `InvalidPreferencesFormatException` before executing the `close` method. In this case, I cannot be sure whether the stream object will be closed safely. In other words, this may result in a file descriptor leak. To fix the defect, it is better to create a `finally` block and call the `close` method within that block to ensure that the stream object is closed. Java SE 7 has an additional approach to fix this defect, known as “try-with-resources,” for cleaner resource management. The two solutions are illustrated in Figures 4.7 and 4.8, respectively.

I found that none of the 24 open source projects used try-with-resources, probably due to the newness of this feature. As a result, I chose to use the “Split Variable/Move close() to finally block” as the default suggestion. To implement the “Split Variable/Move close() to finally block” suggestion, FixBugs first finds the stream variable, and splits the variable declaration and construction if they are performed in the same statement. Next, it moves the stream variable before the `try` statement to be in the scope of the `finally` block that is created next. Lastly, the `close` method is moved to the `finally` block with a `null` condition check and a `try` statement is added to envelop the `close` method. To implement the “try-with-resources” suggestion, FixBugs simply finds the stream variable and moves it to the resource statements of `try`, as shown Figure 4.8.

As with the String Concatenation defect, the tool supports follow-through using suggestion popups and color annotations, which can also be applied to Open Stream Exception Path defect. Note that the tool does not provide String Concatenation and Open Stream Exception Path with drag-and-drop, because the feature is not relevant to solving either of these defects.

### 4.3.4 Missing Argument

Figure 4.9 illustrates a code example containing the missing argument defect. The problem is that the argument for `%s` is missing (Line 8). When the method is executed, a runtime exception will occur. The solution to the defect is to provide an expression of type `String` for `%s`.

```java
public class MissingArgument {
    private String fS1 = "test";

    public void foo(String aS1) {
        float f1 = 0.0f;
        String s1 = "Hello";
        String s2 = "World";
        System.out.printf("%f %s", f1);
        String s3 = "FixBugs";
        System.out.printf("%s", s3);
    }

    public String bar() {
        return "bar";
    }
}
```

Figure 4.9: Code containing the Missing Argument defect.

FixBugs first notices the incorrect format string (i.e., "%f %s" in Line 8) and draws a green squiggly annotation under "%s" indicating that an actual argument has been missed. Then, it finds all candidate variables of that type (String) in the scope of the MissingArgument defect to be used as suggestions. In this example, candidate suggestions will be fS1, aS1, s1, s2, or other expressions evaluated as a String such as a method call returning String type (bar in Line 13). When data mining the open source corpora, I found that the closest variable to the printf is used 24.8% of the time, which makes for a good default suggestion.

Like in the Catch Exception case, I allow users to drag-and-drop one of the candidate variables or to just click a variable for the missing argument. As before, FixBugs provides users with a “My Last Change” option to easily compare their edits to the initial state and the prepackage fixes. Users may prefer other expressions to String variables for the missing argument, so FixBugs allows users to edit code to provide an expression manually. For example, a user might want to make a function call like `System.out.printf("%f %s", f1, bar());`. 
Figure 4.10: Suggested fix for the Missing Argument defect shown in Figure 4.9. The variables in green box are candidates for %s. The dotted borders of the variables denote that the variables can be drag-and-dropped. Currently, aS1 is being dragged and about to be dropped to replace the already suggested variable s2.
4.3.5 Workflow

In this section, I step away from any particular defect resolution and describe how FixBugs implements a general, follow-through workflow for defects identified by FindBugs. A state diagram for this workflow is shown in Figure 4.11, and it consists of five states: Buggy Code (Initial), Default Fix, Custom Fix, Suspend Fix, and Exit.

Assume, for purposes of illustration, that a developer runs a defect finding tool, which then finds several defects in her code. The tool places markers on the defects in the editor, and she is now in the initial, Buggy Code state.

Given buggy code, she (A) invokes FixBugs by clicking a marker in the gutter of the editor, or by pressing a shortcut key. When FixBugs is invoked, it immediately applies a default suggestion and opens a popup with suggestions, placing her in the state Default Fix. If she is satisfied with the default suggestion, she can (B) quit FixBugs by pressing the ESC key or the close button on the popup. The default suggestion will remain, and since her task is completed, she is now in the terminating Exit state.

On the other hand, from the Default Fix state, she can (C) revert to the original code if needed. After reverting, she can (D) quit FixBugs by pressing the ESC key or the close button on the popup. She can also (A) look at the default fix again.

In a one-shot approach, these are essentially the only options available to her. That is, she can pick any of the prepackaged suggestions, and either accept it or undo it and try again with a different suggestion. If none of the suggestions are satisfactory, she must provide a resolution manually.

A follow-through approach, such as in FixBugs, allows additional support from the Default Fix state. In the Default Fix state, FixBugs allows her to interactively fix the bug by (E) offering drag-and-drop or presenting other affordances. This places her in the Custom Fix state. Moreover, it allows her to (F) perform other common tasks as appropriate to a defect, such rename refactoring on the suggested code, if she wants to change the suggested variable name. In the Custom Fix state, she can continue to (G) perform the rename refactoring or (H)
Figure 4.11: State diagram of FixBugs for follow-through bug fixing. Compared to Quick Fix, it supports the Custom Fix mode and Suspend Fix mode.
drag-and-drop code. Then, she can (I) exit FixBugs, (J) revert to the original code, or (K) go back to the Default Fix.

From both the Default Fix state (M) and the Custom Fix state (L), when she starts typing her code to provide her solution manually, the Suspend Fix mode is activated. In this mode, FixBugs hides existing annotations and linked nodes so that she can focus on (N) typing her code, and displays a Resume button on the suggestion popup. She can go back to the Custom Fix mode (O) by clicking the Resume button or (P) exit FixBugs. If she clicks on the Resume button, it restores the color annotations and linked nodes for her to continue using FixBugs, placing her in the Custom Fix state. In my implementation, resumption is best effort. If she has changed the code significantly, resumption is not always possible. In this case, the developer may return to any of the previous known-good states, using the suggestion popup.

In short, by implementing the follow-through principle within FixBugs, the developer is supported not only for the initial fix, but until the task is successfully completed. In addition, the tool enables her to explore multiple options, and understand the impact of the fix through color annotations before accepting the change. Unlike one-shot approaches, the tool continues to support the developer, such as through drag-and-drop, even when the developer has deviated from one of the prepackaged fixes.

4.3.6 Implementation

I developed my approach as an Eclipse plug-in that facilitates follow-through tool design. The prototype currently consists of two plug-ins: ft.base and ft.fixbugs. The ft.base plug-in is a common, tool-agnostic framework that provides components to support follow-through functionality. A follow-through implementation for a task, such as refactoring, browsing version control history, source code navigation, or program analysis defect resolution, would all derive from ft.base. For this chapter, I extended ft.base to the domain of program analysis defect resolution, using the ft.fixbugs plug-in, e.g., FixBugs.

The ft.base plug-in offers several components that follow-through tools can leverage in
their own implementation. Currently, these components include drop zones, suggestion popups, color annotations, and relevant event listeners, in order to assist the developer with completing the task. The \texttt{ft.base} plug-in also provides hooks for managing dependencies and interactions between components, and some of these components can be injected automatically when the tool observes certain developer interactions.

Each FindBugs notification is mapped to a specific bug type \textit{resolver}, implemented using the \texttt{ft.fixbugs} plug-in. When a FindBugs notification occurs, a pattern matcher activates the appropriate resolver. Then, the particular resolver requests components from \texttt{ft.base} that are needed to support the developer for the particular bug type.

For example, consider when a fix for a String Concatenation defect introduces a new variable, such as \texttt{StringBuffer stringBuffer}, and the developer clicks on the variable. In this situation, an in-place rename refactoring component becomes available because my expectation is that the developer may want to change the default name.

The implementation in this chapter is a prototype and has been principally designed in service to a user study to show that a follow-through approach for tool design can be both useful and usable for developers. As such, it does not solve every defect presented by FindBugs. In Future Work (Section 7.2), I suggest some of the challenges to making follow-through capabilities more intelligent and automated.

### 4.4 Experiment

After I designed and implemented a prototype of the follow-through fixing approach, I performed an experiment to evaluate the usability of the prototype. I investigated the following research questions:

- RQ1: How effective and efficient is \texttt{FixBugs} in fixing defects over Quick Fix and manual approaches?

- RQ2: When tasks are successfully completed, what explains the differences in effectiveness
and efficiency between tools?

- RQ3: How satisfied are developers with the prototype when with fixing defects, and what are some of the trade-offs in using FixBugs against other approaches?

4.4.1 Study Design

Participants. I recruited 12 graduate students (henceforth, participants P1-P12) for my study through a flyer and a mailing list in the Department of Computer Science at North Carolina State University. Table 4.1 shows the demographics of the participants. On average, they had 6.2 years of development experience, 4.2 years of experience with Java, and 3.5 years of experience with Eclipse. All participants were familiar with Eclipse and 7 of them had used static analysis tools like FindBugs. Only P11 had no prior experience with the Quick Fix feature. No participant had seen FixBugs or used it before the study. Four of the participants were female.

Tools. I chose three approaches to fixing defects: manual fixing (MF), the Eclipse Quick Fix tool (QF), and FixBugs (FB). During the study, I called Eclipse Quick Fix “Quick Fix version 1” and FixBugs “Quick Fix version 2” so as not to bias participants by letting them know which version I implemented. I evaluated these approaches through the three defects: the Catch Exception (CE), String Concatenation (SC), and Open Stream Exception (OSEP), as described in Section 4.3.

FindBugs does not integrate with Quick Fix. Thus, to compare the usability of manual fixing, Quick Fix, and FixBugs, I extended Quick Fix to provide suggestions for these three defects. These extensions to Quick Fix allowed us to compare the usability of FixBugs and Quick Fixes for the same defect.

Counterbalancing. I used a within-subjects design for the participants to be exposed to all tools. To control for order effects, I asked participants to use the tools in six different orders, because I had three tools.
Table 4.1: Descriptive statistics of the participants (FixBugs).

<table>
<thead>
<tr>
<th>Participant</th>
<th>Development (yrs)</th>
<th>Java (yrs)</th>
<th>Eclipse (yrs)</th>
<th>IDEs</th>
<th>Static analysis tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>20</td>
<td>17</td>
<td>10</td>
<td>Eclipse$^8$</td>
<td>FindBugs, Fortify, Jtest$^9$</td>
</tr>
<tr>
<td>P2</td>
<td>9</td>
<td>6</td>
<td>2</td>
<td>Eclipse, IntelliJ IDEA$^{10}$, NetBeans$^{11}$</td>
<td>(None)</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>6.5</td>
<td>6</td>
<td>Eclipse, NetBeans</td>
<td>FindBugs</td>
</tr>
<tr>
<td>P4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>Eclipse</td>
<td>FindBugs</td>
</tr>
<tr>
<td>P5</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>Eclipse</td>
<td>(None)</td>
</tr>
<tr>
<td>P6</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Eclipse, Rational App. Dev.$^{12}$</td>
<td>FindBugs</td>
</tr>
<tr>
<td>P7</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Eclipse, NetBeans</td>
<td>Checkstyle$^{13}$, FindBugs</td>
</tr>
<tr>
<td>P8</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>BlueJ$^{14}$, Eclipse, Xcode$^{16}$</td>
<td>Clang$^{15}$</td>
</tr>
<tr>
<td>P9</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>Eclipse</td>
<td>Checkstyle</td>
</tr>
<tr>
<td>P10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Eclipse, NetBeans</td>
<td>(None)</td>
</tr>
<tr>
<td>P11</td>
<td>3.5</td>
<td>1</td>
<td>1</td>
<td>Eclipse, NetBeans</td>
<td>(None)</td>
</tr>
<tr>
<td>P12</td>
<td>1.5</td>
<td>1</td>
<td>1</td>
<td>Eclipse, NetBeans</td>
<td>(None)</td>
</tr>
<tr>
<td>Mean</td>
<td>6.2</td>
<td>4.7</td>
<td>3.5</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
**Tasks.** I gave participants 18 tasks (henceforth, T1-T18) to complete. I gave participants four minutes for each of the first three tasks, intended as a warm-up tasks that minimize the problem of practice effects [109], and then three minutes for remaining tasks.

Once time expired, if the defect was not fixed, I marked the task as incomplete. Each task contained one of the three defects mentioned previously. When participants encountered the Catch Exception defect, I explicitly asked them to throw the following exceptions: `CertificateException`, `ClassNotFoundException`, `IOException`, `SAXNotRecognizedException`, and `SAXNotSupportedException`.17

I chose example code from FindBugs, from a paper on static analysis tools [101], and from well-known, real-world, open source projects such as Apache Ant, Apache Lucene18, ArgoUML19, HtmlUnit20, iText21, JHotDraw, and log4j. I ordered the tasks using a metric for readability [15]; the readability of the code generally decreased over the course of each participant’s session. To allow participants to focus on fixing the defects, I modified the code to isolate the defects; I removed extraneous code until all the files referenced for a given task contained a total of no more than 107 lines of code.

**Pilot Study.** I conducted a pilot study with 12 participants: 2 professors, 9 graduate students, and 1 undergraduate student. From the pilot study, I obtained parameters such as the selection of training examples, selection of tasks, and time limits for each task so that the study could be completed within an hour. I do not include the results of the pilot study in the analysis.

---

17I mined the 24 open source projects and found these exceptions are more often thrown than caught. For example, `IOException` was caught 1,878 times, but thrown by 5,881 methods. For more detailed information, see Appendix B.6.
18http://lucene.apache.org/core/
19http://argouml.tigris.org/
20http://htmlunit.sourceforge.net/
21http://itextpdf.com/
4.4.2 Procedure

**Pre-questionnaire**\(^{22}\). Using a pre-study consent form and questionnaire, I obtained consent to record screencasts (video of their screen and voice audio), and I determined the participants’ development experience.

**Training.** I provided the training session\(^{23}\), and conducted the actual experiment. This was done to reduce participant response bias by confounding the creator of the tool \([33]\).

I used three examples for the training session. These examples demonstrated the drag-and-drop, suggestion popups, and color annotation features of the tool to the participant. I also explained how to navigate between defects using FindBugs’ “Bug Explorer” and how to browse detailed information in FindBugs’ “Bug Info” view. The training examples were contextualized through some of the new features of Java 7, such as `try-with-resources` and `multi-catch exception`. A previous pilot study showed that participants did not have familiarity with these features.

This training took about 10-15 minutes. All participants had training regardless of prior experience.

**Experiment.** During the study, I allowed participants to use any information from FindBugs and developer community sites such as StackOverflow\(^{24}\). I also allowed them to use any Eclipse Quick Fixes, such as “add throws declaration” or “surround with try/catch,” because these could have been part of participants’ existing workflows. I also answered any questions participants had, as long as doing so would not influence how they chose to fix the defects.

To better understand participants’ behavior, I also asked them to think aloud \([62]\) during the study. I did not allow them to go back to previous tasks to prevent them from correcting earlier manual fixes using information learned from later Quick Fixes. Including the training tasks, each session took about 45-60 minutes. Afterwards, I asked each participant to fill

---

\(^{22}\) The detailed pre-questionnaires used for the study can be found at Appendix B.2.

\(^{23}\) The detailed training script is presented in Appendix B.1.

\(^{24}\) [http://www.stackoverflow.com/](http://www.stackoverflow.com/)
out a post-questionnaire\textsuperscript{25} regarding user satisfaction. I did not provide compensation for participation.

\textbf{Data Collection and Analysis.} To compare the effectiveness and efficiency (RQ1) of the three tools, I counted how many tasks participants completed successfully using each tool. To check if each participant completed the given task by removing the defect, I looked to see if the bug marker disappeared in the time allotted. To compare efficiency, I calculated the time required to complete each task. I started measuring when participants opened the file containing the defect and stopped measurement when they explicitly said they were done. To analyze the results of the experiment, I used a Wilcoxon signed rank test ($\alpha = .05$).

To provide explanation of the performance differences among the three approaches for tasks that were successfully completed, I computed a decision tree model (RQ2) to identify the contributing factors that best explain the variations in task completion times. The inputs to the decision tree model use both task data as well as pre-questionnaire responses.

To measure user satisfaction (RQ3), I asked questions in the post-questionnaire to see how satisfied participants were with the tools. For each question on the post-questionnaire, I used a 5-point Likert scale \textsuperscript{[63]} ranging from “Strongly disagree (1)” to “Strongly agree (5)” for levels of agreement. For the questionnaire, I used a Wilcoxon signed rank test to compare FixBugs to Quick Fix and manual fixing. To qualitatively determine the trade-offs when using FixBugs for the drag-and-drop, suggestion popups, and color annotation features of the tool, I processed and annotated participant screencasts.

\section*{4.5 Results}

\subsection*{4.5.1 RQ1: Effectiveness and Efficiency}

Figure 4.12 shows the proportion of participants who successfully completed each task and the time taken (in seconds) to complete the task. Participants manually fixing defects successfully

\textsuperscript{25} The post-questionnaire can be found at Appendix B.2.
completed 38 out of the 72 tasks (52.8%), whereas participants using Quick Fix completed 69 out of 69 tasks\(^{26}\) (100.0%) and participants using FixBugs completed all 72 tasks (100.0%). Overall, the mean time taken for tasks completed manually was 111.9 seconds (sd = 38.3). In comparison, the mean times taken using Quick Fix and FixBugs were 65.8 (sd = 45.9) and 54.1 (sd = 35.0) seconds, respectively.

For the Catch Exception defect, I found that FixBugs outperformed Quick Fix (p = .001) and manual fixing (p = .001). For the String Concatenation defect, I found manual fixing took significantly longer than the other tools (p = .001); FixBugs and Quick Fix are not significantly different from one another (p = .954). Similarly, for the Open Stream Exception Path defect, I were unable to identify any significant differences in task completion time between FixBugs and Quick Fix (p = .954). I found that manual fixing took significantly more time than either of the other tools (p = .001).

4.5.2 RQ2: Explanation of Performance

I am interested the factors that explain the variations in time taken to successfully complete tasks. While experience is important, I am interested in factors beyond experience that would help explain participant efficiency when using the tool.

To address this research question, I constructed a decision tree using the task data and pre-questionnaire responses as inputs to the model. I considered the following experimental input factors: Tool (MF, QF, FB), Task (T1-T18), Bug Type (CE, SC, OSEP). I also considered the following pre-questionnaire input factors (from Table 4.1): Experience (Developer), Experience (Java), Experience (Eclipse), and Prior Use of Quick Fix (Y, N). The output, or response variable, is the predicted time to complete the task given these factors.

I chose decision trees over more sophisticated models because their results are human interpretable. I applied a recursive binary partitioning algorithm to successfully completed tasks [36]. The tree in Figure 4.13 starts with the 180 completed tasks at the root. A best factor

\(^{26}\) Three tasks had to be discarded. For details, see Section 4.5.4
Figure 4.12: Percentage of Tasks Completed (Top) and Time taken (Bottom) by participants to complete given tasks manually, with Quick Fix, and with FIXBUGS. Overall, participants fixing defects manually took longer than participants using Quick Fix and FIXBUGS.
at a given depth is selected using a LogWorth statistic [36]. This continues, building the tree one factor at a time. By default, the recursive algorithm has no stop rule. I terminated splitting when the algorithm selected any experience factor as the next splitting point for the leaf. In short, for each leaf in Figure 4.13, the algorithm would have split it using an experience factor.

The overall coefficient of determination is $R^2 = 0.414$, where 0.3-0.4 is generally acceptable for human subject studies. In Figure 4.13, at split 1, the most predictive factor for explaining the task time for the participants in this study is whether or not a tool was used. This is consistent with the results in Figure 4.12, since tasks in which a tool is used has significantly better times than tasks performed manually.

At split 2, the bug type being resolved is the best predictive factor for timing. This is likely because SC and OSEP use suggestion popups with color annotations and have few interactions, whereas CE requires participants to drag-and-drop various exceptions, which requires more time. At split 3, the best predictor of the time difference for CE is the use of FixBugs against Quick Fix, because participants using Quick Fix had to manually edit the source. In contrast, participants using FixBugs could complete the task using drag-and-drop. At split 4, SC and OSEP, the prediction is reflecting the extra time of the warm-up tasks.
In addition, it is notable that for SC and OSEP, the choice of tool is not a good predictor at this depth. The explanation, from Section 4.5.3, is that participants infrequently explored alternative suggestions, and in most cases assumed that the tool applies the fix correctly. In doing so, the interaction for these bug types between FixBugs and Quick Fix is unintentionally similar, and I am unable to quantitatively demonstrate the benefits of suggestions popups and color annotations. However, I qualitatively show support for these features in Section 4.5.3.

4.5.3 RQ3: User Satisfaction

Post-Questionnaire

Table 4.2 shows the results from the post-questionnaires. All questions had Likert responses; I assigned 1 to strong disagreement and 5 to strong agreement. Columns 2 through 4 represent median rating scores of manual fixing, Quick Fix, and FixBugs, respectively. Columns 5 through 7 contain $p$-values indicating the statistical significance of differences among manual fixing, Quick Fix, and FixBugs.

Participants thought FixBugs helped them to quickly and effectively understand what will happen to their code once the fix is applied (Q1, QF-FB: $p = .006$, FB-MF: $p = .005$). They felt FixBugs minimized the work required to fix defects (Q2, QF-FB: $p = .014$, FB-MF: $p = .002$). Participants also said that with FixBugs makes it easy to differentiate between changes in code, presumably due to the color annotation feature (Q3, QF-FB: $p = .003$, FB-MF: $p = .003$). Both FixBugs and Quick Fix allowed participants to quickly make changes to their code (Q4, MF-QF: $p = .002$, FB-MF: $p = .003$). Participants felt that all three tools allowed them to solve the defect how they wanted (Q5, $p = .083$). Finally, although participants would use either tool if available, participants indicated a significant preference for FixBugs over Quick Fix (Q6, FB-QF: $p = .014$).
Table 4.2: User satisfaction, based on post-questionnaires. *p*-values indicate the likelihood that responses for one of the tools were significantly different than for the others. Overall, the participants preferred FixBugs to both Quick Fix and manual fixing. (5-point Likert Scale from “Strongly disagree (1)” to “Strongly agree (5)”).

<table>
<thead>
<tr>
<th>Question</th>
<th>MF</th>
<th>QF</th>
<th>FB</th>
<th>MF-FB</th>
<th>QF-FB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1 {MF, QF, FB} helps me to quickly and effectively understand what will happen to my code once the fix is applied</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>.005</td>
<td>.006</td>
</tr>
<tr>
<td>Q2 {MF, QF, FB} reduces the amount of work required to fix a bug.</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>.002</td>
<td>.014</td>
</tr>
<tr>
<td>Q3 {MF, QF, FB} makes it easy to differentiate between original new and modified code.</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>.003</td>
<td>.003</td>
</tr>
<tr>
<td>Q4 With {MF, QF, FB}, I could quickly make the changes to my code.</td>
<td>2.5</td>
<td>4</td>
<td>4.5</td>
<td>.003</td>
<td>.655</td>
</tr>
<tr>
<td>Q5 {QF, FB} fixed my code the way I wanted it fixed (i.e. made the changes I wanted made)</td>
<td>-</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>.083</td>
</tr>
<tr>
<td>Q6 If {QF, FB} was available in my favorite development environment, I would use it when I programmed</td>
<td>-</td>
<td>4</td>
<td>5</td>
<td>-</td>
<td>.014</td>
</tr>
</tbody>
</table>

**Qualitative Observations**

The results thus far do not offer explanation for the specific features of my implementation that participants found useful or difficult. Thus, I annotated the video recordings and noted remarks regarding tools used and the ways in which participants interacted with drag-and-drop, suggestion popups, and color annotations in FixBugs. Each reference to a participant interaction includes the task number, the version of the tool, and the bug type in parentheses.

**Drag-and-drop.** During training, P8 indicated, “I like [drop zones] from a usability standpoint. That makes me know what is going to happen.” When P8 (T4, CE-FB) uses the tool again, he stated, “That’s pretty painless. My instinct was to delete it and copy and paste, but with the drag and drop, it’s okay.”

Some participants tried to apply the drag-and-drop to QF, which of course does not have highlighted drop sources or support drop zones. Here, P3 (T7, CE-QF) exclaimed, “ClassNotFoundException should be thrown. Oops. It’s not movable!”. She attempted to
perform this behavior again in T7 (CE-QF) and T16 (CF-QF). P6 (T1, QF-CE) tried to
drag-and-drop as well. It’s possible that some of this is attributable to task priming. However,
nearly all of the participants activated the suggestions popups feature using the mouse, rather
than through keyboard shortcuts. In this case, continuing to use the mouse to complete the fix
is perhaps an intuitive and natural extension of this process.

A limitation of drag-and-drop emerges in situations where a significant amount of scrolling
is required. P8 (T13, CE-FB) suggested, “If I have to always scroll to the top, it’s not the most
usable. If I could either do something on the keyboard or right-click and say ‘moves to throws’,
that might make it a little more usable.” I observed similar difficulties with P4 (T13, CE-FB), P9
(T10, CE-FB), and P10 (T13, CE-FB), with instances of long methods that required extensive
scrolling because of the distance between the catch and throw drop zones.

**Suggestion Popups.** Participants liked suggestion popups because it allowed them to
explore alternatives. P8 (T9, OSEP-FB) stated, “I can go back and see which one to use.
That’s a little bit cleaner [than QF] so I like that. I like that I could toggle between the two of
them to see which one looks better.” Without this feature, participants had several difficulties
understanding the impact of the change after applying the fix. For example, P3 (T12, OSEP-QF)
applied one of the fixes, and stated, “It looks like I fixed the bug, but I wanted to compare the
original with the fixed code, but it isn’t there.” She proceeded to undo, and reapplied the entire
Quick Fix procedure to get back to the fixed state. P7 (T2, SC-QF) had the same issue. The
next time P7 (T5, CE-FB) used FixBugs, he commented: “It’s better than [QF] because you
can see the original.”

I did identify some usability issues with the suggestion popup. To apply the fix, the
participant had to click the ‘close’ button (an × icon), or hit escape, which implied canceling,
rather than accepting the fix. P2 (T2, SC-FB) asked, “How do you say okay [accept]? Oh yeah,
you just have to close it. That is confusing because ‘close’ means ‘cancel’ to me.”

Another oversight on the part was that the option to activate FixBugs did not always
give the same description of the problem as the FindBugs tooltip. For example, for an Open
Stream Exception, activating FixBugs requires the participant to click “[Quickfix ver. 2] Fix this Open Stream Exception Path Bug.”, but the FindBugs description is “[method] may fail to close stream on exemption.” P1 (T12, OSEP-FB) stated, “The popup message wasn’t what I was expecting.” P11 (T6, OSEP-FB) and P12 (T3, OSEP-FB) were unsure of activating the suggestion popup because the error description appeared unrelated to their problem. P11’s time of 143 seconds and P12’s time of 215 seconds for this task account for the high errors bars in Figure 3.12.

**Color Annotations** Feedback on color annotations was mixed. P8 stated, “The color mapping is handy. That makes it nice and clear what is getting thrown and where.” P12 remarked during training, “The tool will identify all the strings that are there. That’s great.”

On the other hand, participants fixing the String Concatenation defect were confused about the icons in the gutter and the meaning of the colors. P2 (T2, SC-FB), pointed at the icons in the gutter and asked: “What does that do? [Hovers with mouse.] It doesn’t do anything.” P8 (T11, SC-FB) states, “I have these icons over here. It wasn’t intuitive to begin with.” When verifying the solution, P12 (T3, CE-FB) asks, “What does the orange highlight mean?” Confusion concerning color also occurred in the Catch Exception defect. P8 (T4, CE-FB) thought aloud: “So we got the colors. `getMethod` would throw `ClassNotFoundException`. So blue corresponds to gray? Is that...? That’s not necessarily intuitive.”

Despite these issues, color annotations can still be useful. In some cases, I was able to infer that participants found the color annotations useful by observing the position of their cursor during verification of the fix. P9 (T8, SC-FB) moved his cursor over each of the highlighted lines and delayed briefly at each point, but did not delay on non-highlighted lines. P6, in (T5, SC-FB) and (T14, SC-FB) had a similar cursor navigation pattern. Without color annotations, P3 (T3, OSEP-QF) scrolled back and forth repeatedly after applying the Quick Fix to figure out the impact: “Am I done? I think I’m done, there’s no bug marker?” With color annotations, P3 (T1, CE-FB) immediately pointed to the highlighted exception in green, and asserted, “It added the catch block here.”
4.5.4 Threats to Validity

In this section, I briefly discuss threats to the validity of the study. I inadvertently included warm-up tasks during the counterbalancing procedure and could not discard them from the analysis as intended. This is most apparent in Figure 4.13, at the fourth split. A bug in my implementation required us to discard two trials: P8 (T1) and P12 (T1). A third trial, P12 (T15) had to be discarded because the participant was asked to use Quick Fix, but the system was accidentally left in manual-only mode. In total, three tasks were discarded for Section 4.5.1.

The Wilcoxon signed rank test does not allow for missing data. For the discarded tasks, I performed mean substitution as an approximation. Overall, this affected 3 out of 216 tasks. For incomplete manual tasks, I made the assumption that the task would have been completed just after the time limit. For T1-T3, this is 240s (3m), and 180s (4m) for others. Note this procedure only gives an advantage to manual fixing, and not to my tool.

I used Mac OS X on a MacBook Pro during the user study, which may have confused or slowed down those participants who were more familiar with Windows. The small number of participants may prevent us from generalizing to the larger developer population. Additionally, the participants in the study were graduate students rather than software engineers in industry; however, some of them had significant industry experience. Finally, though I made efforts to reduce participant response bias, it’s possible that participants responded to the post-questionnaire (Section 4.5.3) more favorably than they would otherwise. I provide qualitative observations to mitigate post-questionnaire threats.

4.6 Conclusion

In this chapter, I presented an interactive code recommendation system for fixing bugs and applied it to the domain of program analysis defect resolution, through an Eclipse plug-in called FixBugs. The experimental results show that FixBugs provides additional affordances to developers by combining the flexibility of manual fixing with the efficiency of automated fixes,
while still being as efficient and effective as one-shot Quick Fixes. In some cases, FixBugs is significantly faster. I think my approach to tool design will not only identify areas where tools are currently not meeting these demands, but also inform novel implementations that benefit developers because the tools are more intuitive, more informative, and more supportive.
Chapter 5

Guidelines

This chapter lists design guidelines for RSSEs to help tool designers adopt them in creating more usable tools. The guidelines presented here are inspired by general usability guidelines [84] [66] [112], but have been purposed to help tool designers adopt these guidelines in their own implementations.

How is this guideline set similar or different from other guideline sets? To my knowledge, there is no previous research has proposed design guidelines for improving the usability of RSSEs. However, many researchers in other domains of software engineering have proposed usability guidelines. I postulate that software tools in domains other than RSSEs for software testing and defect resolution can benefit from these guidelines. Murphy-Hill presented usability guidelines for building refactoring tools to remedy insufficient use [81]. Similarly, Mealy and colleagues proposed guideline set for building more usable refactoring tools by categorizing and distilling existing well known usability guidelines to address the refactoring process [73]. Parnin and Görg presented design guidelines for lightweight visualizations that help program management and assist development tasks [88].

By referring to guidelines that I mentioned, I created early mock-up designs and kept revising the designs to improve the usability of the RSSEs (UnitPlus, Viscovery, and FixBugs). Then, I built the three RSSEs based on the mock-up designs and tailored specific guidelines.
While I implemented Viscovery as a stand alone application, I implemented UnitPlus and FixBugs in Eclipse as plug-ins, a well-known IDE that many developers are familiar with. The integration allows me to easily recruit qualified participants. Before conducting actual user studies, I performed pilot studies with some amount of participants to revise the design of studies. Finally, I conducted user studies and observed how participants use the RSSEs to complete tasks for measuring the usability of the RSSEs.

For each usability guideline, I present examples of FixBugs, Viscovery, or UnitPlus of how the guideline can be realized. In addition, I discuss how each guideline can be applied to in general.

- **Encourage design exploration.** Developers spend a considerable amount of their programming efforts exploring source code [30] [56]. Analogously, developers should be able to explore the space of solutions and easily change one solution for another. The design of FixBugs encourages exploration through several alternative suggestions and allows developers to get back to or apply any of these suggestions without leaving the tool. FixBugs also provides a default solution as an exploration starting point to help developers fix defects, supporting further manual actions if required. This guideline is generally applicable to future bug types that have multiple solutions. For example, a switch fall-through missing multiple break statements can be a defect. A developer may want add break statements for some of cases, not for other cases, or all cases. So it is required to allow the developer to choose which cases need to have break statements.

- **Provide direct feedback.** Direct feedback is important to any interaction. Norman suggests that “the state of the system should be clearly available to the user, ideally in a form that is unambiguous and that makes the set of options readily available [85].” Similarly, Lund provides a guideline, “Every action should have a reaction [66].”

While UnitPlus presents code suggestions in preview, FixBugs provides direct feedback when a developer selects a suggestion by showing the changed code directly on the source
code rather than presenting the code in preview [127]. FixBugs differentiates draggable program elements from non-draggable elements for some bug solutions supporting drag-and-drop. I referred to Lund’s guideline, “Things that look different should act different, and Nielsen’s heuristic, “Recognition rather than recall [84].” FixBugs also provides color mappings between method invocations and exceptions that help developers relate code. In the FixBugs experiment, I found most participants with annotations in FixBugs quickly recognize what is changed. It is important for future code recommendation systems to support direct feedback in the developers’ context, because direct feedback helps developers predict its effect on their context in the editor [127].

- **Allow direct manipulation.** The developer should be able to directly manipulate the objects of interest at the appropriate representation and level of abstraction [110] [111]. FixBugs allows direction manipulation through the availability to drag and drop semantically addressable units for a particular task, such as exception handling. Developers can simply pick their exception of interest, and explicitly choose one of the candidate drop zones appropriate for that exception. Moreover, FixBugs even allows user input without leaving the tool if there is no preferred recommendation (see Section 4.3.4). This guideline is also quite related to Murphy-Hill’s guideline, “Task-centricity [81],” indicating a tool should not distract from the developers’ primary task. I believe this guideline can be applicable to all source code-based recommendation systems [74].

- **Conform to the developers’ view of the task.** Johnson suggests that software interfaces should strive for naturalness and find the correct point on the power/complexity trade-off [52]. In a software engineering context, software development environments today provide an overwhelming abundance of features, but not all features are needed for all tasks.

In FixBugs, I conform to the developers’ view of the task by making available the appropriate features at the appropriate time, and withdrawing those features when they
are no longer needed. This guideline is similar to a guideline suggested by Lund, “If it is not needed, it is not needed [66].” and more specifically related to Murphy-Hill’s guideline, “Task-specificity [81].” For example, drag-and-drop is available in the Catch Exception or the Missing Argument defects because it aligns with the task, but not in the String Concatenation or Open Stream Exception Path defects, because this affordance is not relevant for those defects. Furthermore, the rename refactoring is available in the String Concatenation defect because the recommendations for the defect introduce a new variable, whereas the rename refactoring is not available because the recommendations for the Open Stream Exception Path defect do not. Future tool designers could use the drag-and-drop feature and / or the rename refactoring for their recommendations.

- **Automate unwanted workload.** It is important to reduce the amount of work required to free cognitive resources for high-level tasks [37]. This guideline is similar to “Minimal input actions by user” suggested by Smith and Mosier [113].

UNITPLUS mines user’s code and then recommends the mined code. It can reduce workload for developers in writing test code (see Section 2.3.4). FixBugs automates unwanted workload in many ways. For example, the suggestion popup allows developers to quickly switch among alternative solutions without having to perform undo and redo actions. This is an implementation of the guidelines, “Permit easy reversal of actions” suggested by Shneiderman [112].” Without FixBugs, developers may need to manually go back and forth to get their preferred code by selecting undo and redo menu. The color annotations show developers the impact of the code change, without leaving the developer to ascertain the impact of the fix. The drop zones allow developers to focus on the high-level task of choosing drop targets, while the tool provides boilerplate code appropriate for the selected target.

- **Draw users’ attention to what the users need to know about to complete their primary tasks without distraction.** RSSEs should assist developers to complete
software engineering tasks such as testing and bug fixing. It is important for developers to draw attention to what they need to know about to complete their tasks. Annotations and popups are two well known user interfaces to draw users’ attention [69] [119]. An annotation is a special visualization presentation containing text or an image. However, multiple annotations can sometimes distract users from their tasks [83]. To reduce distraction, I referred to Murphy-Hill’s “Bias” in the identification step of refactoring tools for placing emphasis [81]. VISCOVERY places annotations (Covered Times, Flip Counts, and Problem) on only problematic branches rather than all branches (see Section 3.4.6). When a developer moves the mouse over an annotation, VISCOVERY presents the detail content of the selected annotation on a dedicated popup to make her stick [16]. In the VISCOVERY experiment, I found participants with annotations of VISCOVERY quickly figured out and precisely answered given questions than participants with other tools (see sections 3.7.1 and 3.7.2).

Like VISCOVERY, FixBugs uses annotations and a popup to draw users' attention. When a user invokes FixBugs, FixBugs opens a modeless popup containing multiple suggestions in a corner of the code editor. The popup does not distract her from editing or browsing code. Sometimes, the popup may hide code. Then, FixBugs allows her to move the popup to other corners of the editor. FixBugs keeps the popup opened until she explicitly closes the popup by pressing the close button or the Esc key. It is related to a Nielsen’s heuristic, “Clearly marked exits [84].” FixBugs places color annotations to highlight only which part of code was added in green, changed in orange, or is relevant in gray (see Section 4.3.2). When a user enters her input, the annotations disappear with the resume button, because the annotations may distract her. When she presses the resume button, the color annotations appear again. Similarly, when she drags a code element such as a variable name, FixBugs only presents available drop zones. When the user drops the element, FixBugs clears the drop zones. This guideline is a specific form of Lund’s guideline, “Color is information [66].” More specifically, this guideline is quite related to
Mealy and colleagues’ one of usability requirements, “Allow suggested changes to be viewed, and changed, or cancelled prior to transformation [73].” In the FixBugs experiment, I found participants with color annotations of FixBugs quickly figured out what were changed on their code, whereas participants without color annotations of FixBugs scrolled back and fourth to figure out the impact (see Section 4.5.3). When future tool designers add recommendations for more bug types, annotations can be generally applicable to highlight which parts of code are changed, added, or related.

This guideline could be applicable to other kinds of tools such refactoring tools or reuse-oriented recommendation systems [50]. For example, there are many reuse-oriented code recommendation systems such as Code Recommenders [13]. These systems have mainly focused on finding more relevant code snippets and prioritizing them, not synthesizing them in the users’ context. A future tool designer can apply this usability guideline, when the designer wants to make developers notice which parts of the code have been changed. Sometimes, there will be some compilation errors after applying a selected code example due to many reasons. For example, variable names between the applied code example and the developer’s current context could be different. To fix the compilation error, she needs to figure out which variable names are actually different and how to fix it. The tool may place arrow annotations overlaid on the variable names for her to change variable names.

- **Provide context-sensitive recommendations.** It is important to provide context-sensitive code into a user’s current programming context [1] [24]. If a code recommendation is not context-sensitive, the chance that introduces compilation errors and leads developers to write buggy code may be increased. In the FixBugs experiment, I observed participants using FixBugs and Eclipse Quick Fix quickly and accurately fixed given bugs than participants who tried to fix bugs manually using code examples in bug descriptions of a bug finding tool (FindBugs) (see Section 4.5.1). Manual synthesizing a code recommendation with user’s current context is time consuming and error-prone. In case of source-code based recommendation systems and reuse-oriented recommendation systems, providing
context-sensitive recommendation with users is essential to improve tool usability.
Chapter 6

Related Work

This chapter summarizes previous research work on recommendation systems for software engineering (RSSEs) related to my dissertation.

6.1 Studies on RSSEs

Robillard and colleagues introduced three RSSE design dimensions: nature of the context (input), recommendation engine, and output mode [95]. Based on the dimensions, UnitPlus is explicitly invoked and implicitly extracts fully qualified name of selected method call in developers’ programming context (Hybrid). The recommendation engine of UnitPlus uses source code only as input data and ranks recommendable method calls by frequency heuristics. Then, UnitPlus operates in pull mode and provide developers with recommendations based on their explicit requests. Viscovery is also explicitly invoked by developers. Viscovery provides problem-analysis results and solutions to address problems found and then presents the recommendations by developers’ explicit requests. Like UnitPlus, FixBugs is explicitly invoked by developers and uses developers’ programming context to find items (code). FixBugs recommends code to fix bugs in given context and ranks by frequency and then presents the recommendations by developers’ explicit requests and applies a selected recommendation to the context automatically.
Inozemtseva and colleagues categorized a recommendation system that does not use data
mining to generate its recommendations as a recommendation system in-the-small (RITS) and
a recommendation using data mining a Data Mining Recommendation System (DMRS) [49].
They mentioned RITSs tend to be lightweight and provide recommendations quickly on every
query. As they also mentioned, RITSs may not have the cold start problem [106]. Based on
Inozemtseva and colleagues’ categorization, UnitPlus is a DMRS, whereas Viscovery and
FixBugs are RITSs.

6.2 Code Completion

Mooty and colleagues proposed Calcite [78] which suggests code examples for constructing a given
class or interface. The tool is integrated into Eclipse and uses the default completion proposal
context menu to allow developers to choose their suggestions. However, the suggested code
snippets may incur compilation errors when developers apply the snippets into the underlying
code in the editor, because the code snippets are retrieved by a web search engine. Calcite
however does not provide context-sensitive code; it allows developers to fix the compilation
errors. In addition, because it extends Eclipse Quick Fix, Calcite does not display which part of
code is changed, added, or relevant after applying one of code suggestions. Furthermore, it may
spend time to understand what are changed and the changed code even introduces other code
defects while fixing the compilation errors using the suggestions.

Omar and colleagues proposed a code completion for Eclipse, similar to Calcite, called
Graphite [87]. They described new user interfaces and the design constraints for their technique.
Similarly, FixBugs is designed to assist developers with writing code, integrated into the IDE,
provides a new interactive interface, and gives quick feedback on the editor. However, FixBugs
is designed to help developers fix defects, whereas their approach helps developers in creating
objects or regular expressions. In addition, Graphite does not provide context-sensitive code in
the editor; it tries to provide extensibility by using HTML and JavaScript rather than JDT. In
contrast, FixBugs actively uses JDT to analyze the underlying context in the editor and to
provide context-sensitive code suggestions to fix defects.

6.3 API Reuse

There are several projects on code recommendation to support API reuse.

Holmes and Murphy proposed Strathcona [45] a code recommendation tool that uses structural context. When invoked, Strathcona extracts a set of contextual information from a given programming context, then searches relevant code examples in a given repository. Shahvechaphan and Claypool proposed a code assistant framework, called XSnippet [103], which supports developers to search code snippets. XSnippet retrieves relevant code snippets to recommend using context-sensitive information such as object types. ParseWeb [123] suggests relevant method call sequences. Unlike Strathcona and XSnippet, ParseWeb uses a code search engine and finds method-invocation sequences on the web. ParseWeb use query in the form “Source object type → Destination object type” as input. In contrast, UnitPlus takes as input a selected method (a state-modifying method), extracts the fully qualified method name, then retrieves relevant items (observer methods) to recommend.

These approaches including UnitPlus may help developers find code examples to improve API reuse. However, these approaches cannot help developers synthesize the recommended code examples with their programming context. FixBugs, on the other hand, supports context-sensitive code synthesis. In addition, even if these approaches try to find relevant code from repositories to APIs new to the user, the retrieved code is not closely related to the test code the user works. On the other hand, UnitPlus can recommend more relevant code to her test code, since UnitPlus analyzes side-effect information of her production code.

MSeqGen [124] captures method-invocation sequences from existing code bases and uses those method sequences for helping test-generation approaches. UnitPlus, on the other hand, does methods sequences in existing test code but recommends those sequences to developers in writing new unit tests efficiently.


6.4 Software Evolution

Dagenais and Robillard proposed a recommendation approach, SemDiff [27], that suggests relevant adaptive changes for framework evolution. Similar to their approach, UnitPlus recommends relevant observer methods with method sequences for validating object behaviors.

Fluri and colleagues proposed a recommendation approach, ChangeCommander [34], that provides suggestions for context changes on method invocations and automated code adaptation support. For these invocations, ChangeCommander highlights them using annotations to emphasize that the context related to the method invocation has been changed. ChangeCommander uses similar constructs like JDT and Quick Fix in Eclipse and interacts with developers. ChangeCommander collects context changes and suggests method invocations for fixing faults in production code, while UnitPlus mines existing test code and recommends method sequences for non-primitive types.

6.5 Bug Fixing

The design of FixBugs was inspired by previous HCI studies that resemble efforts towards supporting follow-through in bug fixing tools. There are several research works that extend the Eclipse JDT to provide context-sensitive bug fixes [44, 131]. Vakilian and colleagues proposed a tool called Keshmesh [128] for detecting and fixing concurrency defects in Java programs. These approaches provide code suggestions using Eclipse’s Quick Fix interface. Muşlu and colleagues recently proposed Quick Fix Scout [80, 79] to analyze suggested code changes before applying them and presents the number of remaining compilation errors in front of the quick fixes. Given a fix, it informs the developer of how many errors the fix will introduce or remove. It analyzes suggested code changes before applying them and presents the number of remaining compilation errors in front of the quick fixes. Then it sorts by the number of remaining errors for developers to inform them of the consequences of each code transformation.

The follow-through approach of FixBugs is complementary to static analysis approaches.
and can take advantage of technical improvements in these areas, while providing enhanced interactions to support the developer. For example, FixBugs can aid Quick Fix Scout by directly applying code suggestions to existing code in the editor (e.g., through color annotations), so that developers to easily navigate suggestions and observe the impact of applying a suggestion.

### 6.6 Studies on Tool Usability

Vakilian and colleagues studied the usability of automated refactoring tools. To improve the usability of these tools [127], they mentioned several implications such as preference to lightweight invocation of use, providing better preview to inspect code changes to avoid disuse, and providing predictability and interactivity to avoid misuse. I believe FixBugs addresses these implications.

Lee and colleagues proposed an approach, called DNDRefactoring [61], for improving the usability of refactoring tools via drag-and-drop. As they found, while performing refactoring via drag-and-drop, drag sources and drop targets are not always intuitive. In contrast to DNDRefactoring, FixBugs provides developers with drop zones so that all of the participants in my study successfully performed drag-and-drop. I assume that the reason for this is that FixBugs displays draggable code elements and possible drop zones as guidance while code elements are being dragged. The design of FixBugs addresses the two aforementioned “best practices” and the drag-and-drop concern. FixBugs also uses colors to indicate added and edited code and highlights draggable objects and droppable targets.

### 6.7 Visual Aides

Dekel and Herbsleb proposed an approach, called eMoose [32], designed to improve API documentation usability. It decorates method invocations using annotations to increase awareness of important information associated with these targets. On the other hand, FixBugs decorates code with annotations to increase awareness of what has been changed. Moreover, FixBugs provides a color mapping between method invocations and exceptions that the invocations may
throw. FixBugs also uses two types of border annotations (i.e., solid or dashed border) to differentiate between draggable and non-draggable code elements.

Cottrell and colleagues proposed an approach, called Jigsaw [25], that aims to help with code reuse tasks by comparing structural correspondences between originally copied seed code and paste seed code. Jigsaw uses colors on code based on similarity scores with confidence to draw users’ awareness, while FixBugs uses colors to differentiate which part of code is added, changed, or relevant to the original code.
Chapter 7

Conclusion

7.1 Summary of Contributions

In this dissertation, I have presented three RSSEs for improving software quality in software development life-cycle particularly manual software testing (UNITPLUS; see Chapter 2), automated software testing (VISOVERY; see Chapter 3) and bug fixing (FIXBUGS; see Chapter 4).

First, I presented UNITPLUS, an assistant tool integrated in Eclipse, that is designed to help developers in writing test code efficiently. UNITPLUS collects side-effect information from source code (production code) using a side-effect analyzer, mines the code, and recommends mined code to developers by frequency when they write test code. The recommendation is based on static analysis of the class under test and already written unit test cases. Through a feasibility study with four open source projects, I found UNITPLUS can potentially assist developers in writing test code by showing that the existing test suites of the projects can be augmented with the test code recommended by UNITPLUS in making the test suites more effective.

Second, I described VISOVERY, an approach for visualizing coverage information from tests generated by Pex, presenting problem-analysis results for the areas not-covered, and providing recommendations for solving the problems that the DSE tool faces. The main goal of VISOVERY is to help developers figure out why the Dynamic Symbolic Execution (DSE) tool was unable to
get full coverage and understand what needs to be done to achieve better coverage. I evaluated the usability of Viscovery by conducting a user study with 12 developers. The results show that Viscovery can improve the usability of test coverage visualization tools.

Third, I presented FixBugs, an interactive bug fixing approach that uses the follow-through principle to fix bugs by recommending context-sensitive code. I implemented my approach and evaluated it with 12 participants to compare it with manual fixing and Eclipse Quick Fix, a non-interactive tool. Interactive fixes helped participants fix defects just as effectively and, for some defects, faster than non-interactive fixes. Participants indicated they satisfied with FixBugs than with the non-interactive tool or with manual fixes.

Finally, I presented the guidelines learned from the three research works to improve the usability of RSSEs: encourage design exploration, provide direct feedback, allow direct manipulation, confirm to the developers’ view of the task, automate unwanted workload, draw users’ attention to what the users need to know about to complete their primary tasks without distraction., and provide context-sensitive recommendations.

7.2 Future Work

This dissertation presented three RSSE tools that are designed to assist developers in completing software development tasks correctly, efficiently, and more enjoyably. However, there are also many opportunities for future work.

- **Integrate Viscovery with the IDE.** UnitPlus and FixBugs are integrated with Eclipse [41], whereas Viscovery is not. The current implementation of Viscovery is loosely coupled with Pex in Visual Studio [142]. That is to say, Viscovery reads in the XML-format outputs of Pex and Covana. Viscovery can also visualize testing result, if reports of other DSE tools are in the same format. As mentioned in Section 3.7.1, I observed that some of the participants using Viscovery got confused when they navigated the code. One participant said, “Having the windows be more in sync with each other
on Tool C [Viscovery] would be helpful. I expect when I click on a method that I’m taken immediately to that part of the source code.” Another participant said, “Navigation from method to code view would also be helpful.” If Viscovery is integrated with Visual Studio and uses the infrastructure of the tool, it can provide more fine-grained navigation.

• **Extend FixBugs.** It can be a future direction to provide better user interfaces and code suggestions for fixing more defects. For example, when developers are trying to fix the Missing Argument defect, FixBugs provides all available candidate variables. However, some of the variables may not be displayed in the current area of the editor, e.g. a field variable declared at the end of a long class. If FixBugs presented the variables in a popup, users could quickly recognize and use the variables for the missing argument, without needing to scroll up or down to find them. I plan to extend our current work to incorporate this feature and then evaluate it. Currently, FixBugs only provides fixes for 4 out of more than 500 bug patterns that FindBugs can detect. Investigation to the remaining defects FindBugs finds will be needed to come up with interactive user interfaces for developers to fix defects.

• **Apply Broadly Follow-through Principles.** My expectation is that applying follow-through principles to tool design will be broadly beneficial to software developers. For example, Perez and Jackson argued that the Git version control system puzzles even experienced developers because of its complex, underlying conceptual model [90]. I think follow-through principles can assist developers in version control tasks by presenting the developer with a simplified interaction that provides only the relevant tools for the given task at the appropriate time, for example, when a developer encounters a merge conflict. As another example, during code reviews, Bacchelli and Bird found that although finding defects is a priority, context and change understanding is equally important [8]. A follow-through tool would not only help reviewers find defects, but assist in context and change understanding as they perform code reviews. The follow-through principle
raises technical challenges that need to be addressed. Lacking explicit cues, how can a tool automatically infer when a developer is performing a particular task, such as when manually refactoring, and provide them with appropriate follow-through tools? How do I know when the developer has finished a task? And how can I computationally decide which tools to provide? The current implementation of FixBugs has demonstrated that follow-through tools are useful, but significant research will be needed to provide developers with more intelligent and automated support to help them all the way through to the end.
REFERENCES


126


APPENDICES
Appendix A

Study Materials of Viscovery

In this appendix, I present study materials of the experiment in Chapter 3: consent form and pre-questionnaires, tasks, post-questionnaires.
Consent Form and Pre-Questionnaires (1 of 3)

This three-page form is used to sign up for a user study on coverage visualization tools
To complete this study, you must:
- Be familiar with programming in C# or Java
- Be familiar with Visual Studio
- Have basic knowledge of software testing

By the definition above, I am eligible to participate.

☐ Yes  ☐ No

Consent Form (Page 2 of 3)

May we record screen and audio for post-analysis?

☐ Yes  ☐ No

Informed Consent:

"I have read and understand the above information. I can save a copy of this webpage for my records. I agree to participate in this study with the understanding that I may choose not to participate or to stop participating at any time without penalty or loss of benefits to which I am otherwise entitled."

Your Email Address:

Pre-Questionnaires (Page 3 of 3)

Before the study, we would like to know a few things about you.

Which of the following Integrated Development Environments (IDEs) have you used?
☐ Eclipse  ☐ IntelliJ IDEA  ☐ NetBeans  ☐ Visual Studio  ☐ Xcode
☐ Others

How many years of software development experience do you have?
Please enter a number in years. (e.g., 2 or 3.5)

How many years of software testing experience do you have?
Please enter a number in years. (e.g. 2 or 3.5)

How many years of experience do you have programming in C# or Java?
Please enter "C#" or "Java" and a number in years. (e.g. C#, 2 or Java, 3.5)

Previous Pex (an Automated White Box Testing Tool by Microsoft) experience?
☐ Yes  ☐ No
Viscovery Tasks

Q1. Identify the class name with the **lowest block coverage** ratio first and the method name with the lowest block coverage ratio in the class.

   Class Name: __________________________

   Method Name: __________________________

Q2. Identify the line number or the offset number of the **first problematic branch** in the method with the lowest block coverage ratio. Answer only one of the questions below.

   (Optional) Line Number of the branch: __________________________

   (Optional) Offset of the branch: __________________________

   (Optional) Number of the branch: __________________________

Q3. Identify the test class and the test method that executed the problematic branch or problems.

   Test Class Name: __________________________

   Test Method Name: __________________________

Q4 (A). Select the primary problem why that part of the code is not covered.

   - Boundary Problem
   - External Method Call Problem (Uninstrumented Method)
   - Object Creation Problem

Q4 (B). Provide the reason for the problem?

   __________________________

Q5. Provide your solution for Pex to cover that part of the code.

   __________________________
Viscovery: Post-questionnaire

How would you rate Tool A, Tool B, and Tool C?

<table>
<thead>
<tr>
<th></th>
<th>Very Satisfied</th>
<th>Satisfied</th>
<th>Neutral</th>
<th>Dissatisfied</th>
<th>Very Dissatisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tool C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Study Materials and Mining Results of FixBugs

This appendix contains study materials and mining results of FixBugs (see Chapter 4): Training Script, Cheat sheet, Consent form and Pre-questionnaires, Post-questionnaire, and Mining results.

B.1 Training Script

Hello. We are Yoonki Song (the first investigator) and Brittany Johnson (the second investigator). We are currently conducting research on improving the usability of static analysis tools. As part of our research, we are conducting a user study on the “Quick Fix”/“Quick Assist” tool features used to fix defects found by static analysis tools. A “Quick Fix” is offered when the code is broken and a “Quick Assist” is a minor refactoring offered by the tool.

B.1.1 Part 1: Introduction to the FixBugs study

Today, we will be asking you to use and evaluate two different versions of Quick Fix. During this evaluation, you will be asked to complete a set of 3 tasks; each task is comprised of 6 subtasks.
In each subtask, you will be given a segment of “buggy code” and asked to fix it. For each set of subtasks, you will be asked to fix the defects either using one of the versions of Quick Fix and/or manually. The first set of tasks should take approximately 4 minutes. The remaining sets should take approximately 3 minutes. During the study, we encourage thinking aloud as you are making your decisions. We hope you will share with us your thought process as you are completing the tasks. You will be able to ask minor questions if needed while completing the tasks, but in order to get reliable feedback we hope to avoid this by giving a thorough explanation now and throughout.

Here, you see we have some “buggy” code; there is a missing parameter in the system.out.printf method call (point to show what method). One way to fix this would be for you to manually go in and make the changes yourself. Another option would be to use a quick fix. Now, the first thing to understand about using the quick fixes is how to invoke the tools. There are two ways to invoke a “quick fix”: 1) use the keyboard shortcut (Ctrl + 1) or 2) put your cursor on the line where the bug is and wait for the pop up.

Do both methods then ask participant to go ahead and do this.

You can see that there are two quick fix versions available: Quick Fix Version 1 and Quick Fix version 2. Each version can be applied by clicking the fix, however there are differences between the two which we will discuss in more detail as we get to each version’s set of subtasks.

You can also see that when you hover over the bug marker, a description comes up (and a more detailed explanation is available on the bottom of the screen). In this example, we are explaining what the problem is, but once I begin I will only give general context to the problem. Just imagine that you are working on some code and have gotten this bug that you need to fix. You can use the information provided by the tool to help you decide what warning means and what the best fix will be. Once you are done with each task, you can move onto the next; once you have moved on to the next task you will not be able to return to make any modifications so make sure you are done before moving on.

Now that we have explained how to invoke the two versions of quick fix, let us get started
with your tasks so you can get the chance to use them! For the first set of tasks, you will be
given buggy code and asked to fix it using Quick Fix version 1. Using quick fix version 1 is fairly
straightforward; you invoke the tool using the methods we discussed previously and then click
the fix (version 1). The fix will automatically be applied, however, if you are not satisfied with
what the quick fix did you can make manual modifications. Go to Example code to show how it
works. Any questions before you begin?

Okay, now for the next set of tasks you will be asked to manually fix the buggy code you are
given; this means neither version of quick fix can be used. We also ask that you attempt to fix
the bug without using the Internet for help. Any questions before we begin this portion?

Now, for the last set of tasks we are going to ask you to fix the buggy code using quick fix
version 2. Quick fix version 2 is invoked the same way as quick fix version 1. Once you click the
fix, it is automatically applied, however, you will be able to make certain modifications. Let’s
look at the example code again to see how this version of quick fix works.

B.1.2 Part 2: Introduction to multi-catch and try-with-resources

try-with-resources

The try-with-resources statement is a try statement that declares one or more resources. A
resource is an object that must be closed after the program is finished with it. The try-with-
resources statement ensures that each resource is closed at the end of the statement. Any
object that implements java.lang.AutoCloseable, which includes all objects which implement
java.io.Closeable, can be used as a resource.

The following example reads the first line from a file. It uses an instance of BufferedReader
to read data from the file. BufferedReader is a resource that must be closed after the program
is finished with it:

```java
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

139
In this example, the resource declared in the try-with-resources statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the try keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

**multi-catch**

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

Consider the following example, which contains duplicate code in each of the catch blocks:

```java
        catch (IOException ex) {
            logger.log(ex);
            throw ex;
        }

    } catch (SQLException ex) {
        logger.log(ex);
        throw ex;
    }

    In releases prior to Java SE 7, it is difficult to create a common method to eliminate the duplicated code because the variable `ex` has different types.

    The following example, which is valid in Java SE 7 and later, eliminates the duplicated code:
```

```java
        catch (IOException|SQLException ex) {
            logger.log(ex);
            throw ex;
        }
```
The catch clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (|).

Go back to first example again and go through invoking the tool/applying the fix – explain:

- highlighting
- default vs. original vs. last change
- drag and drop capabilities (dotted outline)

You can see here the now that Quick Fix version 2 has been applied, multiple exceptions have been caught in the catch blocks here. You can also see that there are dotted outlines around each exception. This means that you are able to drag and drop each into one of the various “drop zones” available. Drop zones look like this (begin to drag an exception to show what a drop zone looks like). You can see that you are able to re-organize the exceptions within the catch blocks as well as drag the exception to be thrown (show how this is done). Any item that contains this dotted border is drag-and-droppable into drop zones.

- movable popup box
- ways to exit/keep fix

Before we begin, please be aware that once you complete a task, you will not be able to revisit that task.

Now, do you have any questions before completing these last tasks?

### B.2 Cheat sheet

Cheat sheet for Tasks #1, #4, #7, #10, #13, and #16

If one or more of the following exceptions appear or are expected to appear in your code, please throw them rather than catch them:

- `java.io.IOException`
• java.lang.ClassNotFoundException

• java.security.cert.CertificateException

• org.xml.sax.SAXNotRecognizedException

• org.xml.sax.SAXNotSupportedException

**Specifying the exceptions thrown by a Method**

To specify that the method containing the given bug can throw the exceptions above, add a `throws` clause to the method declaration. For Example, You could just write the following.

```java
public void method() throws IOException {
    ...
}
```
FixBugs: Consent Form and Pre-Questionnaires (1 of 3)

This three-page form is used to sign up for a user study on an bug fixing feature.
To complete this study, you must:
- Be familiar with programming in Java
- Have some experience with Eclipse IDE (www.eclipse.org)

By the definition above, I am eligible to participate.

☐ Yes ☐ No

Consent Form (Page 2 of 3)

May we record screen and audio for post-analysis?

☐ Yes ☐ No

Informed Consent:

"I have read and understand the above information. I can save a copy of this webpage for my records. I agree to participate in this study with the understanding that I may choose not to participate or to stop participating at any time without penalty or loss of benefits to which I am otherwise entitled."

Your Email Address: ____________________________________________________________________

Pre-Questionnaires (Page 3 of 3)

Before the study, we would like to know a few things about you.

Which of the following Integrated Development Environments (IDEs) for Java have you used?

For more information, please visit http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Java

☐ BlueJ ☐ Eclipse ☐ Greany ☐ IntelliJ IDEA ☐ JBuilder ☐ JCreator ☐ JDeveloper ☐ KDevelop
☐ NetBeans ☐ Rational Application Developer ☐ Servoy ☐ Xcode ☐ Others

How many years of software development experience do you have? Please enter a number in years. (e.g., 3.5)

__________________________________________________________________________

How many years of experience do you have programming in Java? Please enter a number in years. (e.g., 3.5)

__________________________________________________________________________

How many years of Eclipse experience do you have? Please enter a number in years (e.g., 3.5)

__________________________________________________________________________

Which of the following static analysis tools have you used?

For more information, please visit http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis#Java

☐ CheckStyle ☐ FindBugs ☐ Hammurapi ☐ PMD
☐ Soot ☐ Squale ☐ Jtest ☐ LDRA Testbed
☐ SemmleCode ☐ SonarQ ☐ Kalistick ☐ Others

Have you ever used the "Quick fix" or "Quick Assist" feature offered by Eclipse?

For more information, please visit http://wiki.eclipse.org/Image:Quickfix1.jpg

☐ Yes ☐ No
FixBugs: Post-questionnaire

Original code

```java
public class Training2 {
    public static Object invoke(Object obj, String methodName, Class argType, Object arg) {
        try {
            Method method;
            method = obj.getClass().getMethod(methodName, new Class[] { argType });
            return method.invoke(obj, new Object[] { arg });
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

“Quick Fix” version 1

```java
public class Training2 {
    public static Object invoke(Object obj, String methodName, Class argType, Object arg) {
        try {
            Method method;
            method = obj.getClass().getMethod(methodName, new Class[] { argType });
            return method.invoke(obj, new Object[] { arg });
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

```
```

“Quick Fix” version 2

```java
public class Training2 {
    public static Object invoke(Object obj, String methodName, Class argType, Object arg) {
        try {
            Method method;
            method = obj.getClass().getMethod(methodName, new Class[] { argType });
            return method.invoke(obj, new Object[] { arg });
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

```
```
**Quick Fix** version 1 helps me to quickly and effectively understand what will happen to my code once the fix is applied.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

**Quick Fix** version 2 helps me to quickly and effectively understand what will happen to my code once the fix is applied.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

Manually fixing my code helps me to quickly and effectively understand what will happen to my code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

**Quick Fix** version 1 reduces the amount of work required to fix a bug.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

**Quick Fix** version 2 reduces the amount of work required to fix a bug.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

Manually fixing reduces the amount of work required to fix a bug.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
</table>

**Quick Fix** version 1 makes it easy to differentiate between original, new and modified code.
**Quick Fix** version 2 makes it easy to differentiate between original, new and modified code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Manually fixing makes it easy to differentiate between original, new and modified code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Quick Fix version 1, I could quickly make the changes to my code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Quick Fix version 2, I could quickly make the changes to my code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Manual Fixing, I could quickly make the changes to my code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Quick Fix* version 1 fixed my code the way I wanted it fixed (i.e. made the changes I wanted made).

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With Quick Fix version 2, I could quickly make the changes to my code.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If Quick Fix version 1 was available in my favorite development environment, I would use it when I programmed.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If Quick Fix version 2 was available in my favorite development environment, I would use it when I programmed.

<table>
<thead>
<tr>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neither Agree or Disagree</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.3 Mining Results
Table B.1: Mining results for String Concatenation.

<table>
<thead>
<tr>
<th>Project name</th>
<th>StringBuffer</th>
<th>StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr-3.4</td>
<td>11</td>
<td>49</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>126</td>
<td>2</td>
</tr>
<tr>
<td>apache-commons-collections-3.2.1</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>apache-commons-lang3-3.1</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>apache-jena-2.7.3</td>
<td>90</td>
<td>93</td>
</tr>
<tr>
<td>apache-log4j-1.2.17</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>apache-maven-3.0.4</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>apache-xerces2-j-2.11.0</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>ArgoUML-0.24</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>checkstyle-5.2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>dbunit-2.4.8</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>emma-2.0.5312</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>htmlunit-2.10</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>iText-5.0.0</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td>Jdepend-2.9.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jEdit-4.5.2</td>
<td>1</td>
<td>76</td>
</tr>
<tr>
<td>JFreeChart-1.0.13</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>JHotDraw-7.0.x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NanoXML-2.2.3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>NetCDF-4.2</td>
<td>68</td>
<td>185</td>
</tr>
<tr>
<td>Tinyuml2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Weka-3.7</td>
<td>459</td>
<td>4</td>
</tr>
<tr>
<td>xmlunit-1.3</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Xstream-1.4.3</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>Total (Percentage)</td>
<td>1173 (70.1%)</td>
<td>500 (29.9%)</td>
</tr>
</tbody>
</table>
Table B.2: Mining results for Catch Exceptions.

<table>
<thead>
<tr>
<th>Project name</th>
<th>No-catch</th>
<th>Catch</th>
<th>Multi-catch</th>
<th>Throw</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr-3.4</td>
<td>175</td>
<td>433</td>
<td>0</td>
<td>108</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>160</td>
<td>931</td>
<td>0</td>
<td>831</td>
</tr>
<tr>
<td>apache-c...-...-3.2.1</td>
<td>2</td>
<td>49</td>
<td>0</td>
<td>93</td>
</tr>
<tr>
<td>apache-c...-lang3-3.1</td>
<td>4</td>
<td>71</td>
<td>0</td>
<td>76</td>
</tr>
<tr>
<td>apache-jena-2.7.3</td>
<td>140</td>
<td>965</td>
<td>0</td>
<td>792</td>
</tr>
<tr>
<td>apache-log4j-1.2.17</td>
<td>1</td>
<td>204</td>
<td>0</td>
<td>67</td>
</tr>
<tr>
<td>apache-maven-3.0.4</td>
<td>37</td>
<td>260</td>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>apache-xerces2-j-2.11.0</td>
<td>5</td>
<td>666</td>
<td>0</td>
<td>917</td>
</tr>
<tr>
<td>ArgoUML-0.24</td>
<td>13</td>
<td>729</td>
<td>0</td>
<td>403</td>
</tr>
<tr>
<td>checkstyle-5.2</td>
<td>9</td>
<td>118</td>
<td>0</td>
<td>208</td>
</tr>
<tr>
<td>dbunit-2.4.8</td>
<td>34</td>
<td>188</td>
<td>0</td>
<td>575</td>
</tr>
<tr>
<td>emma-2.0.5312</td>
<td>21</td>
<td>167</td>
<td>0</td>
<td>158</td>
</tr>
<tr>
<td>htmlunit-2.10</td>
<td>15</td>
<td>209</td>
<td>0</td>
<td>202</td>
</tr>
<tr>
<td>iText-5.0.0</td>
<td>45</td>
<td>380</td>
<td>0</td>
<td>557</td>
</tr>
<tr>
<td>Jdepend-2.9.1</td>
<td>3</td>
<td>12</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>jEdit-4.5.2</td>
<td>122</td>
<td>600</td>
<td>0</td>
<td>301</td>
</tr>
<tr>
<td>JFreeChart-1.0.13</td>
<td>7</td>
<td>49</td>
<td>0</td>
<td>465</td>
</tr>
<tr>
<td>JHotDraw-7.0.x</td>
<td>1</td>
<td>54</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>NanoXML-2.2.3</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>86</td>
</tr>
<tr>
<td>NetCDF-4.2</td>
<td>71</td>
<td>1437</td>
<td>0</td>
<td>2829</td>
</tr>
<tr>
<td>TinyUML12</td>
<td>1</td>
<td>22</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Weka-3.7</td>
<td>1</td>
<td>1536</td>
<td>0</td>
<td>1764</td>
</tr>
<tr>
<td>xmlunit-1.3</td>
<td>1</td>
<td>34</td>
<td>0</td>
<td>196</td>
</tr>
<tr>
<td>Xstream-1.4.3</td>
<td>14</td>
<td>159</td>
<td>0</td>
<td>106</td>
</tr>
<tr>
<td><strong>Total (Percentage)</strong></td>
<td>882 (4.2%)</td>
<td>9280 (43.8%)</td>
<td>0 (0.00%)</td>
<td>11016 (52.0%)</td>
</tr>
</tbody>
</table>
Table B.3: Mining results for String Concatenation.

<table>
<thead>
<tr>
<th>Project name</th>
<th>StringBuffer</th>
<th>StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr-3.4</td>
<td>11</td>
<td>49</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>126</td>
<td>2</td>
</tr>
<tr>
<td>apache-commons-collections-3.2.1</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>apache-commons-lang3-3.1</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>apache-jena-2.7.3</td>
<td>90</td>
<td>93</td>
</tr>
<tr>
<td>apache-log4j-1.2.17</td>
<td>29</td>
<td>0</td>
</tr>
<tr>
<td>apache-maven-3.0.4</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>apache-xerces2-j-2.11.0</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>ArgoUML-0.24</td>
<td>63</td>
<td>0</td>
</tr>
<tr>
<td>checkstyle-5.2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>dbunit-2.4.8</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>emma-2.0.5312</td>
<td>38</td>
<td>0</td>
</tr>
<tr>
<td>htmlunit-2.10</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>iText-5.0.0</td>
<td>82</td>
<td>0</td>
</tr>
<tr>
<td>Jdepend-2.9.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jEdit-4.5.2</td>
<td>1</td>
<td>76</td>
</tr>
<tr>
<td>JFreeChart-1.0.13</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>JHotDraw-7.0.x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NanoXML-2.2.3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>NetCDF-4.2</td>
<td>68</td>
<td>185</td>
</tr>
<tr>
<td>Tinyuml2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Weka-3.7</td>
<td>459</td>
<td>4</td>
</tr>
<tr>
<td>xmlunit-1.3</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Xstream-1.4.3</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total (Percentage)</strong></td>
<td><strong>1173 (70.1%)</strong></td>
<td><strong>500 (29.9%)</strong></td>
</tr>
</tbody>
</table>
## Table B.4: Mining results for Open Stream Exception Path.

<table>
<thead>
<tr>
<th>Project name</th>
<th>close() in finally</th>
<th>try-with-resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr-3.4</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>36</td>
<td>0</td>
</tr>
<tr>
<td>apache-commons-collections-3.2.1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>apache-commons-lang3-3.1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>apache-jena-2.7.3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>apache-log4j-1.2.17</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>apache-maven-3.0.4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>apache-xerces2-j-2.11.0</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>ArgoUML-0.24</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>checkstyle-5.2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>dbunit-2.4.8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>emma-2.0.5312</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>htmlunit-2.10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>iText-5.0.0</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>Jdepend-2.9.1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>jEdit-4.5.2</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>JFreeChart-1.0.13</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>JHotDraw-7.0.x</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NanoXML-2.2.3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NetCDF-4.2</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Tinyuml2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Weka-3.7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>xmlunit-1.3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Xstream-1.4.3</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total (Percentage)</strong></td>
<td><strong>189 (100.0%)</strong></td>
<td><strong>0 (.0%)</strong></td>
</tr>
</tbody>
</table>
Table B.5: Mining results for Missing Argument. (LV: Local Variable, PV: Parameter Variable, FV: Field Variable Variable, Others: Array Access, Cast Expression, Infix/PREFIX/Postfix, ... )

<table>
<thead>
<tr>
<th>Project name</th>
<th>Closest</th>
<th></th>
<th></th>
<th>Non-Closest</th>
<th></th>
<th></th>
<th>FV</th>
<th>Mth</th>
<th>Call</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LV</td>
<td>PV</td>
<td>FV</td>
<td>LV</td>
<td>PV</td>
<td>FV</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>antlr-3.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-ant-1.8.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-c...-c...-3.2.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-c...-lang3-3.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-jena-2.7.3</td>
<td>22</td>
<td>6</td>
<td>0</td>
<td>31</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>30</td>
<td>18</td>
<td>-</td>
</tr>
<tr>
<td>apache-log4j-1.2.17</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-maven-3.0.4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>apache-x...2-j-2.11.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ArgoUML-0.24</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>checkstyle-5.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dbunit-2.4.8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>emma-2.0.5312</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>htmlunit-2.10</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>iText-5.0.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Jdepend-2.9.1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>jEdit-4.5.2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JFreeChart-1.0.13</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JHotDraw-7.0.x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NanoXML-2.2.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NetCDF-4.2</td>
<td>81</td>
<td>33</td>
<td>6</td>
<td>99</td>
<td>22</td>
<td>10</td>
<td>49</td>
<td>141</td>
<td>43</td>
<td>-</td>
</tr>
<tr>
<td>Tinyuml2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Weka-3.7</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>xmlunit-1.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XStream-1.4.3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>103</td>
<td>39</td>
<td>6</td>
<td>130</td>
<td>29</td>
<td>10</td>
<td>49</td>
<td>171</td>
<td>61</td>
<td>-</td>
</tr>
</tbody>
</table>

Percentage       17.2%  6.5%  1.0%  21.7%  4.9%  1.7%  52.0%  28.6%  10.2%
Table B.6: Mining results for Exception Usage Patterns. Exceptions in bold are more frequently thrown than caught.

<table>
<thead>
<tr>
<th>Exception (Fully Qualified Name)</th>
<th>#Caught</th>
<th>#Caught total</th>
<th>#Thrown</th>
<th>#Thrown total</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Exception</td>
<td>2816</td>
<td>51.3</td>
<td>2671</td>
<td>48.7</td>
</tr>
<tr>
<td>java.io.IOException</td>
<td>1878</td>
<td>24.2</td>
<td>5881</td>
<td>75.8</td>
</tr>
<tr>
<td>org.antlr.runtime.RecognitionException</td>
<td>383</td>
<td>40.2</td>
<td>571</td>
<td>59.9</td>
</tr>
<tr>
<td>javax.jmi.reflect.InvalidObjectException</td>
<td>364</td>
<td>100.0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>java.lang.Throwable</td>
<td>315</td>
<td>89.7</td>
<td>36</td>
<td>10.3</td>
</tr>
<tr>
<td>java.lang.NumberFormatException</td>
<td>310</td>
<td>86.6</td>
<td>48</td>
<td>13.4</td>
</tr>
<tr>
<td>java.sql.SQLException</td>
<td>239</td>
<td>38.3</td>
<td>385</td>
<td>61.7</td>
</tr>
<tr>
<td>java.lang.ClassNotFoundException</td>
<td>193</td>
<td>40.6</td>
<td>283</td>
<td>59.5</td>
</tr>
<tr>
<td>java.lang_INTERRUPTED_EXCEPTION</td>
<td>164</td>
<td>71.6</td>
<td>65</td>
<td>28.4</td>
</tr>
<tr>
<td>ucar.ma2.InvalidRangeException</td>
<td>154</td>
<td>32.4</td>
<td>322</td>
<td>67.7</td>
</tr>
<tr>
<td>org.apache...XMLConfigurationEx...</td>
<td>145</td>
<td>53.1</td>
<td>128</td>
<td>46.9</td>
</tr>
<tr>
<td>java.lang_IllegalArgumentException</td>
<td>132</td>
<td>21.9</td>
<td>480</td>
<td>78.1</td>
</tr>
<tr>
<td>java.net_MalformedURLException</td>
<td>123</td>
<td>67.6</td>
<td>59</td>
<td>32.4</td>
</tr>
<tr>
<td>java.lang_RuntimeException</td>
<td>122</td>
<td>86.5</td>
<td>19</td>
<td>13.5</td>
</tr>
<tr>
<td>java.lang.reflect.InvocationTargetException</td>
<td>121</td>
<td>80.1</td>
<td>30</td>
<td>19.9</td>
</tr>
<tr>
<td>java.text_ParseException</td>
<td>116</td>
<td>69.9</td>
<td>50</td>
<td>30.1</td>
</tr>
<tr>
<td>java.lang_SecurityException</td>
<td>108</td>
<td>61.0</td>
<td>69</td>
<td>39.0</td>
</tr>
<tr>
<td>java.io_FileNotFoundException</td>
<td>108</td>
<td>89.3</td>
<td>13</td>
<td>10.7</td>
</tr>
<tr>
<td>java.lang_NoSuchMethodException</td>
<td>107</td>
<td>72.3</td>
<td>41</td>
<td>27.7</td>
</tr>
<tr>
<td>java.lang_InstantiationException</td>
<td>107</td>
<td>81.7</td>
<td>24</td>
<td>18.3</td>
</tr>
<tr>
<td>java.lang.CloneNotSupportedException</td>
<td>91</td>
<td>68.4</td>
<td>42</td>
<td>31.6</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>org.xml.sax_SAXNotRecognizedEx...</td>
<td>14</td>
<td>21.5</td>
<td>51</td>
<td>78.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>org.xml.sax_SAXNotSupportedEx...</td>
<td>14</td>
<td>22.6</td>
<td>48</td>
<td>77.4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>java.security.cert.CertificateEx...</td>
<td>0</td>
<td>0.0</td>
<td>5</td>
<td>100.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>