Improving Readability in Automatic Unit Test Generation

Ermira Daka

Supervisors: Gordon Fraser & Siobhán North
Department of Computer Science
The University of Sheffield

This dissertation is submitted for the degree of
Doctor of Philosophy

August 2018

I would like to dedicate this thesis to my family . . .

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work which has been carried out since the official starting date of this PhD program.

Ermira Daka
August 2018

Acknowledgements

I would like to thank my supervisor Gordon Fraser, for his support and advice throughout. It was a great opportunity to work with such professional and experienced person. I also would like to thank my second supervisor Siobhan North for her advice during the writing phase of this thesis, and Lucia Specia for her advice in machine learning.

Special thanks go to my collaborators José Miguel Rojas, José Campos, Westley Weimer, and Jonathan Dorn. Also, I want to thank my professor from University of Lugano (Universita della Svizzera Italiana) Michele Lanza, who together with Gordon Fraser gave me the possibility of starting a funded PhD.

I am grateful to my mother, father, sister and brother who supported me all the time. Last but not least, I would like to thank my husband Bleron for his support, understanding, and assistance during my PhD. Finally, I am thankful to my son Mat who came just in the middle of my research time and gave me a great pleasant and motivation.

Abstract

In object-oriented programming, quality assurance is commonly provided through writing unit tests, to exercise the operations of each class. If unit tests are created and maintained manually, this can be a time-consuming and laborious task. For this reason, automatic methods are often used to generate tests that seek to cover all paths of the tested code. Search may be guided by criteria that are opaque to the programmer, resulting in test sequences that are long and confusing.

This has a negative impact on test maintenance. Once tests have been created, the job is not done: programmers need to reason about the tests throughout the lifecycle, as the tested software units evolve. Maintenance includes diagnosing failing tests (whether due to a software fault or an invalid test) and preserving test oracles (ensuring that checked assertions are still relevant). Programmers also need to understand the tests created for code that they did not write themselves, in order to understand the intent of that code. If generated tests cannot be easily understood, then they will be extremely difficult to maintain.

The overall objective of this thesis is to reaffirm the importance of unit test maintenance; and to offer novel techniques to improve the readability of automatically generated tests.

The first contribution is an empirical survey of 225 developers from different parts of the world, who were asked to give their opinions about unit testing practices and problems. The survey responses confirm that unit testing is considered important; and that there is an appetite for higher-quality automated test generation, with a view to test maintenance.

The second contribution is a domain-specific model of unit test readability, based on human judgements. The model is used to augment automated unit test generation to produce test suites with both high coverage and improved readability. In evaluations, 30 programmers preferred our improved tests and were able to answer maintenance questions 14level of accuracy.

The third contribution is a novel algorithm for generating descriptive test names that summarise API- level coverage goals. Test optimisation ensures that each test is short, bears a clear relation to the covered code, and can be readily identified by programmers. In evaluations, 47 programmers agreed with the choice of synthesised names and that these were as descriptive as manually chosen names. Participants were also more accurate and faster at

matching generated tests against the tested code, compared to matching with manually-chosen test names.

Table of contents

List of figures xvii

List of tables xxi

1 Introduction 1

1.1 Challenges Addressed by the Thesis 3
1.2 Thesis Aims and Objectives 4
1.3 Thesis Contribution 5
1.4 Thesis Roadmap 6
1.5 Published Work 7

2 Literature Review 9

2.1 Introduction 9
2.2 Software Testing 9
2.2.1 Functional Testing 11
2.2.2 Structural Testing 11
2.3 Automation of Software Testing 12
2.3.1 Random Test Generation 13
2.3.2 Symbolic Execution 14
2.3.3 Dynamic Symbolic Execution 16
2.3.4 Search Based Test Data Generation 17
2.3.4.1 Meta-Heuristic Algorithms 18
2.3.4.1.1 Hill Climbing 18
2.3.4.1.2 Simulated Annealing 20
2.3.4.1.3 Genetic Algorithm 20
2.3.5 Hybrid Test Generation 23
2.3.5.1 Lazy Systematic Test Generation 24
2.4 Unit Test Generation for Object-Oriented Software 25

2.4.1 The State of the Art and Practice of Unit Testing 25
2.4.2 JUnit 26
2.4.3 Random Test Generation for Java 28
2.4.3.1 JCrasher 28
2.4.3.2 Randoop 29
2.4.3.3 GRT 30
2.4.3.4 JTExpert 31
2.4.3.5 T3i 32
2.4.4 Symbolic Execution for Java 32
2.4.4.1 JPF-SE 33
2.4.4.2 Symstra 33
2.4.4.3 Hex 34
2.4.5 Dynamic Symbolic Execution for Java 35
2.4.5.1 MSeqGen 36
2.4.5.2 Pex 37
2.4.5.3 Seeker 38
2.4.6 Search-Based Test Generation for Java 39
2.4.6.1 eToc 39
2.4.6.2 TestFul 40
2.4.6.3 Nighthawk 41
2.4.6.4 EvoSuite 41
2.4.7 Lazy Systematic Unit Test Generation for Java 43
2.4.7.1 JWalk 43
2.5 Software Evolution 45
2.5.1 Software Maintenance 46
2.5.2 Test Case Evolution and Maintenance 48
2.5.2.1 Regression Testing 49
2.5.2.1.1 Test Case Minimization 49
2.5.2.1.2 Test Case Selection 50
2.5.2.1.3 Test Case Prioritization 50
2.5.2.2 Test Case Repair 51
2.6 Readability in Natural Language Processing 52
2.7 Source Code Readability 54
2.7.0.1 Buse and Weimer Readability Model 55
2.7.0.2 Posnett Readability Model 57
2.8 Test Code Readability 57

2.8.1 Test Smell Detection 57
2.8.2 Test Case Minimization 58
2.8.3 Test Readability Improvements 61
2.8.4 Test Naming Improvements 62
2.8.5 Test Case Documentation 63
2.9 Conclusion 63

3 A Survey of Unit Testing Practices and Problems 65
3.1 Introduction 65
3.2 Survey Design 66
3.2.1 The Questionnaire 67
3.2.2 Selection of Participants 71
3.2.3 Threats to Validity 73
3.3 Survey Results 73
3.3.1 Demographics 73
3.3.2 RQ3.1: Can Online Marketing Research Platforms Be Used for Software Engineering Surveys? 74
3.3.3 RQ3.2: What Motivates Developers to Write Unit Tests? 76
3.3.4 RQ3.3: How Is Unit Testing Integrated into Software Development? 77
3.3.4.1 Time Spent on Writing Unit Tests 77
3.3.4.2 Handling of Failing Unit Tests 78
3.3.5 RQ3.4: How Do Developers Write Unit Tests? 81
3.3.5.1 What Do Developers Optimise Their Tests For? 82
3.3.5.2 Which Techniques Do Developers Apply When Writing Tests? 82
3.3.6 RQ3.5: How Do Developers Use Automated Unit Test Generation? 84
3.3.7 RQ3.6: How Could Unit Testing Be Improved? 85
3.3.7.1 Writing New Unit Tests 86
3.3.7.2 Treating Failing Tests 87
3.3.7.3 General Impressions 89
3.4 Conclusions 90

4 Modeling Readability to Improve Unit Tests 91
4.1 Introduction 91
4.2 Unit Test Readability Metric 92
4.2.1 Human Readability Annotation Data 93
4.2.2 Final Annotation Data Set 94

4.2.3 Features of Unit Tests 95
4.2.4 Feature Discussion 102
4.2.5 Feature Selection 103
4.3 Generating Readable Tests 104
4.4 Empirical Evaluation 107
4.4.1 Experimental Setup 107
4.4.1.1 Unit Test Generation Tool 107
4.4.1.2 Experiment Procedure 108
4.4.1.3 Threats to Validity 109
4.4.2 RQ1: Test vs. Code Readability 109
4.4.3 RQ2: Improved Test Generation 111
4.4.4 RQ3: Do Humans Prefer Readability Optimized Tests? 113
4.4.5 RQ4: Does Readability Optimization Improve Human Understanding of Tests? 116
4.5 Generating Readable Tests for Guava 126
4.5.1 Generating Readability Optimized Tests 127
4.5.1.1 User Agreement 130
4.5.1.2 Test Suite Generation 130
4.6 Conclusion 131

5 Generating Unit Tests with Descriptive Names 133
5.1 Introduction 133
5.2 Naming Tests Based on Coverage 135
5.2.1 Coverage Criteria for Test Naming 136
5.2.2 Finding Unique Method Names 138
5.2.3 Synthesizing Coverage-based Test Names 140
5.2.3.1 Identification of Unique Goals 143
5.2.3.2 Ranking Test Goals 144
5.2.3.3 Merging Test Goals 145
5.2.3.4 Resolving Ambiguities 145
5.3 Evaluation 146
5.3.1 Experimental Setup 146
5.3.1.1 Subjects 146
5.3.1.2 Treatments 146
5.3.1.3 Tasks 147
5.3.1.4 Objects 147
5.3.1.5 Procedure 149

5.3.1.6 Analysis 150
5.3.1.7 Threats to Validity 150
5.3.2 Results 151
5.3.2.1 RQ1: Agreement with Synthesized Test Names 151
5.3.2.2 RQ2: Matching Unit Tests with Synthesized Names . . . 155
5.3.2.3 RQ3: Identifying Relevant Unit Tests by Names 157
5.4 Conclusions 158

6 Conclusions and Future Work 161
6.1 Research Findings 162
6.1.1 Unit Testing Practices and Problems in Industry 162
6.1.2 Developing a Model Predictor for Test Readability 164
6.1.3 A Novel Technique for Test-Naming 166
6.2 In Support of the Thesis 167
6.3 Future Work 168
6.3.1 Extending Readability Model with a larger dataset 168
6.3.2 Understandability Model 170
6.3.3 Unit Test Name Improvement 170
6.3.3.1 Generating Unit Tests with Descriptive Variable Names . 170
6.4 Final Comments 171

7 Appendix - A 173
7.1 Survey Questions and Data 173

References 177

List of figures

1.1 Two versions of automatically generated test that exercise the same functionality but have a different appearance and readability. 3
1.2 A manually written test case. 3

2.1 V-model software development life cycle. [95] 10
2.2 Example code with a condition 14
2.3 Example method that divides two integers 14
2.4 Symbolic execution of code that subtracts two integers. Path Condition (PC) is the conjunction of all symbolic constraints along a path. In this example, symbolic execution has the first valid path when a>b (if condition in line 3), the second valid path when a<=b (else condition in line 5). 15
2.5 Fitness function calculation. 18
2.6 Hill Climbing search space 20
2.7 Genetic algorithm working procedure. 22
2.8 Crossover process. 22
2.9 Example class: Calculator with a method that returns true if the input value is multiple of three. 26
2.10 Example test suite for class Calculator for input values 0 and 3. The red assertion is where the test case fails. 27
2.11 Example class: Calculator with a method that returns true if the input value is multiple of three. 27
2.12 JUnit Eclipse interface. 28
2.13 Example test case generated with JCrasher [53] 29
2.14 Example test case generated with Randoop [146]. 30
2.15 Example test case generated with GRT [126]. 31
2.16 Example test cases generated with JTExpert. 32
2.17 A sequence of four method calls with symbolic values generated with Symstra[201]. 33

2.18 A sequence of four method calls with real values generated with Symstra [201]. 34
2.19 A sample code manipulated from Hex [36] 34
2.20 Symbolic data structure analyses of sample code `getList` [36]. 35
2.21 Example method with four integer input values, and three conditions. . . . 36
2.22 Class sample for method call sequence extraction and skeleton design from MSeqGen. 37
2.23 Example test case generated with Pex [189]. 38
2.24 Example test input generated with Seeker [187]. 38
2.25 Example test cases generated with eToc [190]. 39
2.26 TestFul GUI which list classes to be tested, and the option of creating instance for particular class. 40
2.27 Unit test cases generated with TestFul [21]. 41
2.28 `Counter` class exercised from EvoSuite. 42
2.29 Example test cases generated with EvoSuite. 43
2.30 JWalker Tester [177]. 45
2.31 Software maintenance categories, and their cost in total software maintenance process 46
2.32 Software maintenance activities [184]. 47
2.33 Coh-Metrix web tool that calculates text cohesion and coherence [86]. . . . 54
2.34 Source code readability model of Buse and Weimer. 55
2.35 An automatically generated test case. 60
2.36 Test case simplified with SimpleTest algorithm. 60

3.1 Example rating questions on AYTM 67
3.2 Example ranking questions on AYTM 68
3.3 Example of distributive questions on AYTM 68
3.4 Example of selecting questions on AYTM 69
3.5 Demographics of the survey respondents. 71
3.6 Software development background of the survey respondents. 74
3.7 Inter-rater agreement over all questions, calculated using Kendall's coefficient of concordance, ranging from 1 (complete agreement) to 0 (no agreement). 76
3.8 What motivates developers to write unit tests 77

3.9 How do developers spend their development time? The white dot in the violin plot is the median; the black box represents the interquartile range, and the pink region represents the probability density, ranging from min to max. Although writing new code is the dominating task, developers perceive to be spending more time on testing and debugging than on writing new code. 79
3.10 What do developers usually do when a test fails? In about half the cases, tests fulfil their purpose and indicate a problem in the code that needs fixing. 80
3.11 Which aspects do developers aim to optimise when writing tests? 81
3.12 Which techniques do developers apply when writing new tests? 81
3.13 Common usage scenarios of automatic test generation. 84
3.14 What is most difficult about writing unit tests? 85
3.15 What is most difficult about fixing unit tests? 85
3.16 General questions on perception of unit testing. 89

4.1 Score distribution for the human test annotation dataset. 95
4.2 Example test case with two assertions 95
4.3 Example test case throwing an exception 96
4.4 Example test case with unused identifier 96
4.5 Example test case with comments 97
4.6 Example test case 97
4.7 Example test case with different datatypes 98
4.8 Example test case with different statements 98
4.9 Example test case with class constructor and method call 99
4.10 Results of the feature selection measured in terms of Pearson's correlation with 10-fold cross validation, and agreement with user preferences for test pairs. 104
4.11 Test case generated without readability optimization feature in EvoSuite 106
4.12 Test case generated with readability optimization feature in EvoSuite . . 106
4.13 10-fold cross-validation of code and test readability models using different learners and data sets. 110
4.14 Test cases generated without and with readability optimization for class XMLElement 114
4.15 Second test pair generated without and with readability optimization for class Rational 115
4.16 Third test pair generated without and with readability optimization for class Rational 115
4.17 Example question of agreement in empirical study 116

4.18 Default and optimized test case for class `cli.Option`. 118
4.19 Default and optimized test case for class `org.joda.time.YearMonthDay`. 119
4.20 Default and optimized test case for class `nu.xom.Attribute`. 120
4.21 Default and optimized test case for `net.n3.nanoxml .StdXMLReader`. . . 121
4.22 Default and optimized test case for `comparators.FixedOrderComparator`. 122
4.23 Default and optimized test case for class `digester3.plugins.PluginRules`. 123
4.24 Default and optimized test case for class `chain.impl.ChainBase`. 124
4.25 Default and optimized test case for class `FilterListIterator`. 125
4.26 Default and optimized test case for `lang3.CharRange`. 125
4.27 Example question of understanding in empirical study 126
4.28 Readability scores of manually-written and automatically-generated test cases in 7 different configurations. 128
4.29 Percentage of users preferring the optimized test cases 129

5.1 Example class: A shopping cart that keeps track of the money spent, but has a limit for individual purchases. 134
5.2 Test suite generated with EvoSuite for class `ShoppingCart`, with original names / descriptive names. 141
5.3 Agreement results (RQ1). 152
5.4 Example of agreement question as presented to participants in the survey website. 154
5.5 Selection results (RQ2). 155
5.6 Example of selection question as presented to participants in the survey website. 156
5.7 Understanding results (RQ3). 157
5.8 Example of understanding question as presented to participants in the survey website. 158

6.1 Readability feature cluster - dendrogram 169

List of tables

2.1 Code Features of Buse and Weimer Readability Model 56

3.1 Results on the qualification question. 74
3.2 What makes it difficult to write a new unit test? 86
3.3 What makes it difficult to fix a failing test? 87

4.1 Predictive power of features based on correlation and one feature at a time analysis, and optimized regression model. 100
4.2 Model prediction agreement with user choices 111
4.3 Readability value for the 30 classes selected based on 10 runs per branch. . 112
4.4 Readability value for the 30 classes selected based on the top three pairs that maximize the difference between *default* and *optimized* configurations. . . . 113
4.5 Human understanding results of tests for the 10 classes randomly selected. . 117

5.1 Input/output coverage goals for different datatypes. 136
5.2 Coverage goal list for the `ShoppingCart` class, and their string representation. 138
5.3 Method names for overloaded methods. 138
5.4 Coverage goals for the test suite generated by EvoSuite (Figure 5.2); unique coverage goals are highlighted. 140
5.5 List of experimental objects. NUKM stands for Number of uniquely killed mutants. For each class, the test selected for the study is highlighted. The other tests names are used to be presented as options for selection and understanding tasks. 149

7.1 Survey questions and response data. 174

Chapter 1

Introduction

One of the most important aspects of software development is software testing. Considering that some software might be very critical (e.g.: programs in avionic, medical, nuclear industries), verifying and validating their correctness becomes even more important. Software testing is a method that ensures the quality of the system by exercising its functionalities. Testing takes almost 50% of the software development time [140], and with increasing program complexity it becomes more time consuming and costly. The cost increases even more if an error in the system remains undetected until later stages of the development process [153]. For example, Windows [85] and Linux [139] operating systems made a million dollar savings after finding some uncovered bugs in their code [33]. Therefore, this implies that whatever can be done to find bugs early, will have a positive impact on overall project cost and software quality.

Software testing is performed on different levels through the entire software development life-cycle, and the first test level is unit testing. Unit testing is performed during (or before) development of software modules, and exercises the smallest units in the code. In object oriented programming, unit testing is a well known technique where different frameworks have been created (e.g. JUnit, NUnit) in order to reduce human effort and increase probability of finding bugs. To further support developers on manually creating test inputs, automated techniques have been devised that generate test suites based on the program under test.

A common approach to automatically generating unit test cases is doing it randomly [13, 53, 126, 145, 146, 155]. Randomly generated tests either serve to reveal bugs such as undeclared exceptions [53] and violations of code contracts [146], or they can be enhanced with regression assertions [74, 200] which capture the current state, for example by asserting the value of observed return value. Because random test generation tends to result in test suites with low code coverage, dynamic symbolic execution has been introduced [80, 96, 188], that combines concrete and symbolic execution test generation techniques and automatically

explores program execution space. Search-based testing is another advanced technique, which uses efficient meta-heuristic search algorithms for test generation. One of the algorithms used is a Genetic Algorithm, which uses the idea of natural reproduction during test generation. However, in order to achieve better code coverage a hybrid-approach is used that combines search-based technique with dynamic symbolic execution [79, 96, 188], and overcomes the disadvantages that SBST has on exploring the search space and problematic search areas, and DSE has on constraint solving.

It has been observed that automated test generation can be cheaper than manually testing, and equally or even more effective than traditional testing techniques [22]. However, even though tests are generated automatically, any test failures require fixing either the software or the failing test, which is a manual activity that needs one to understand the behavior exercised by the test. How difficult it is to understand a unit test depends on many factors. For example, unit tests consist of objects, sequences of method calls that bring those objects to the appropriate state, and assertions that check the correctness. The difficulty of understanding such a test case is thus directly influenced by the particular choice of sequence of calls. Understanding automatically generated tests can be even more difficult than manually written tests. Tools for test generation, which in principle are intended to generate high coverage test suites and support developers, but in practice tend to generate tests that do not look as nice as manually-written ones. For example, consider the two automatically generated test cases in Figure 1.1 and one manually written test case in Figure 1.2. Although there are fundamental differences between manually written and generated tests, there is also some flexibility and some generated tests are better than the others. For example generated tests differ in terms of features like number of lines, identifiers, number of constructors, which can be optimized and make the comprehension easier.

```
ArrayIntList arrayIntList0 = new ArrayIntList();
int int0 = arrayIntList0.size();
arrayIntList0.add(0, 0);
ArrayUnsignedShortList arrayUnsignedShortList0 = new ArrayUnsignedShortList((
    IntCollection) arrayIntList0);
boolean boolean0 = arrayIntList0.addAll(0, (IntCollection) arrayUnsignedShortList0)
    ;
int int1 = arrayIntList0.removeElementAt(0);
assertEquals(1, arrayIntList0.size());
assertEquals(0, int1);
```

```
ArrayIntList arrayIntList0 = new ArrayIntList();
arrayIntList0.add(0, 0);
boolean boolean0 = arrayIntList0.add(0);
arrayIntList0.clear();
assertEquals(true, arrayIntList0.isEmpty());
```

Fig. 1.1 Two versions of automatically generated test that exercise the same functionality but have a different appearance and readability.

```
ArrayIntList list = new ArrayIntList();
list.add(42);
list.add(-3);
list.add(17);
list.add(999);
assertEquals(4, list.size());
```

Fig. 1.2 A manually written test case.

The next section will discuss in detail the main motivation of this thesis, and an overview of the relevant literature is presented in Chapter 2.

1.1 Challenges Addressed by the Thesis

Unit testing plays an important role during software developments, and consumes a considerable time in developers every day job. Indeed, test oracles need to be devised, failing tests need to be debugged, and test code needs to be maintained, posing challenges on the readability and other aspects. The difficulty of comprehending a test depends on many factors. If code is not readable, intuitively it will be more difficult to perform any tasks that require understanding it. Despite significant interest from managers and developers [40], a general understanding of software readability remains elusive. For source code Buse and Weimer

used a machine learning approach, which they applied to a dataset of code snippets with human annotated ratings of readability, allowing them to predict whether code snippets are considered readable or not [39]. Although unit tests are also just code in principle, they use a much more restricted set of language features; for example, unit tests usually do not contain conditional or looping statements. Therefore, a general code readability metric may not be well suited for unit tests.

Moreover, test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names. For example, the EvoSuite [70] and Randoop [146] tools name their tests “test0”, “test1”. These names give no hint on the content of the tests, and navigating such tests by name is impossible. Thus, even though the tests might achieve good code coverage, there is reason for concern when it comes to understanding, debugging, and maintaining these tests. The challenge, however, is that automatically generated tests tend to be non-sensical and have no clear purpose other than covering code, which makes it difficult to apply standard conventions to derive good names. Indeed, when the only purpose of a generated unit test is to cover line 8 of a class, then naively capturing this with a name like "testCoversLine27" is not helpful either. Hence, a better naming strategy that will help on test understanding would increase the effectiveness of automated test generation, too.

In summary, even that test generation tools can be used to re-generate tests whenever developers want, existing test cases need to be revised. Test revision is usually done with the tendency of understanding the intent of the test while trying to fix them or finding new important bugs on changed part of the program. Therefore, if test generation tools can produce shorter test cases, with short lines, with readable names, with no unneeded statements (e.g., unused variables, many assertions), these generated tests will more likely be accepted and adopted by developers. Indeed, this means that if developers will use existing tests more (instead of deleting them), this will decrease overall testing time and cost, and at the same time increase software quality.

1.2 Thesis Aims and Objectives

The aims of this work are as following:

- Perform an empirical study with developers working in industry about unit testing, and their needs and problems during testing.
- Improve readability of automatically generated test cases with a machine learning model which will be based on human perception of readability.

- Perform an empirical study that shows the usefulness of our test readability model.
- An algorithm that will be used to generate unit test cases with descriptive names.
- Perform an empirical study that will show how useful is our test naming approach.

1.3 Thesis Contribution

Unit testing importance and the main issues that developers have while testing their code, is investigated with an empirical study using 225 developers all over the world. The result of the study shows that unit testing is an important factor of software development and there is need for research in the area of automated testing (one of the areas of importance is test maintenance).

As software testing has a significant cost in software development, and test cases still have to be maintained manually, this thesis continues to investigate the unit test readability issue, and proposes a domain specific model of readability based on human judgements that applies to object oriented unit test cases. To support developers in deriving readable unit tests, we use this model in an automated approach to improve the readability of unit tests, and integrate this into an automated unit test generation tool. This model is evaluated with an empirical study, and analysis of 116 syntactic features of unit tests shows that readability is not simply a matter of the overall test length; several features related to individual lines, identifiers, or entropy have a stronger influence. Our optimized model correlates strongly with user judgments in surveys about code readability, and improves on other code readability models. Our technique to improve tests succeeds in increasing readability in more than half of all automatically generated unit tests, and validation with humans confirms that the optimized tests are preferred.

To further improve unit test readability, the thesis describes a novel technique to generate descriptive names for automatically generated unit tests. This technique is based on the insight that, while an individual generated test might not have a clearly discernible purpose on its own, the context of the test suite it is embedded in provides sufficient information to derive names which (a) describe the test's code, (b) uniquely identify the test within its test suite, and (c) provide a direct link from source code to test name. This approach is also evaluated, and participants agreed with the names synthesized with our technique. The synthesized names are as descriptive as manually written names, participants were slightly better at matching tests with synthesized names than they were at matching tests with manually written names. Finally, participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.

1.4 Thesis Roadmap

Chapter 2 – Literature Review This chapter explores the literature in automatic unit testing. It starts with software testing and its importance on software engineering, and continues with levels of testing by giving a special attention to unit testing and automation of unit testing. In more details it covers the literature in techniques of testing like Random testing, Symbolic Execution, Dynamic Symbolic Execution, Search-based testing, and Hybrid approach. Furthermore, for each technique it specially discuss the unit testing, and existing tools for test generation. The chapter then proceeds with literature in software evolution and maintenance, its observations and main categories. Then it continues with readability in general and more specifically in software and test code.

Chapter 3 – Survey on Unit Testing Practices and Problems

To gain insight into common practice and needs in unit testing, this chapter described the conducted online survey using the global online marketing research platform *AYTM*. Responses were sought from a global pool of 20 million respondents in several iterations, and results are presented for the final refinement of the survey and the qualified responses. This data is used as a basis for investigating the motivation of developers when writing unit test cases, how is unit testing integrated into software development, how do developers write unit tests, how do developers use automated unit test generation, and how could unit testing be improved.

Chapter 4 – Modeling Readability to Improve Unit Tests

This chapter addresses the problem or test readability by designing a domain-specific model of readability based on human judgements that applies to object oriented unit test cases. To support developers in deriving readable unit tests, this model is used in an automated approach to improve the readability of unit tests, and integrated into an automated unit test generation tool. Also, an analysis is presented of the syntactic features of unit tests and their importance based on human judgement, a regression model based on an optimized set of features to predict the readability of unit tests, a technique to automatically generate more readable tests, and an empirical evaluation of the model. Furthermore, the model of test readability is used for test generation for Guava library. The generated test cases are further evaluated and compared with manually written ones in terms of readability.

Chapter 5 – Generating Unit Tests with Descriptive Names This chapter proposes a novel technique to generate descriptive names for automatically generated unit tests. Firstly, it identifies all possible descriptive elements that are identifiable at the level of the test code, then selects a minimal set of these elements for each test in a test set, and finally uses this minimal set to synthesize a descriptive, unique name for each test. With an empirical study, the technique as an extension to the open source EVOSUITE test generation tool, is compared

to manually derived names in terms of developer agreement, ability to identify tests by their names, and ability to identify tests based on the code under test.

Chapter 6 – Conclusion and Future Work The final chapter of this thesis concludes the main content of this work, summarizes the main contributions, and outlines possible future work that can be done.

1.5 Published Work

- "A survey on unit testing practices and problems." In Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, IEEE, 2014. Daka, Ermira, and Gordon Fraser.

- "Modeling readability to improve unit tests." In Proceedings of the 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
  Daka, Ermira, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer.

  This paper won the distinguished paper award on the European Software Engineering Conference.

  Jonathan Dorn, and Westley Weimer helped with setting up empirical studies on Mechanical Turk and in consolidating crowdsourced results and data.
  José Campos helped to prepare the web-based experiment infrastructure, and conducting the experiments. Also, he helped preparing the data sets (including test generation).

- “Generating Readable Unit Tests for Guava.” Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering
  Daka, Ermira, José Campos, Jonathan Dorn, Gordon Fraser and Westley Weimer.

  Jonathan Dorn and Westley Weimer helped with setting up empirical studies on Mechanical Turk and in consolidating crowdsourced results and data.
  José Campos helped to prepare the web-based experiment infrastructure, conducting the experiments, and analysing the results. Also, he helped preparing the data sets (including test generation).

- "Improving Readability of Automatically Generated Unit Tests." PPIG (2015) Daka, Ermira.

- "Generating unit tests with descriptive names or: would you name your children thing1 and thing2?" International Symposium on Software Testing and Analysis - ISSTA (2017). Daka, Ermira, José Miguel Rojas and Gordon Fraser.

  José Miguel Rojas helped to prepare the web-based experiment infrastructure, conducting the experiments, and analysing the results. Also, he helped preparing the data sets for experiments.

Chapter 2

Literature Review

2.1 Introduction

Development of high-quality software is supported with *software testing*, a part of the software development life-cycle. This chapter reviews the literature in the field of software testing, and software evolution and maintenance. The first section briefly introduces software testing and its main levels and techniques. Next, it reviews the literature in the area of automatic software testing and continues with test generation techniques as part of it . We survey a number of automatic test generation techniques and report on a number of studies which have evaluated these. Particular attention is paid to *search based test generation*, which is used for test generation to achieve the objective of the thesis. Furthermore, unit testing as the focus of this theses is investigated with particular focus on Java programming language. The last part of this chapter covers software evolution and maintenance, and test maintenance issues. Test case maintenance is divided into three main subsections, *test evolution, regression testing* and *test case repair*, and in the end readability as one of the factors of software maintenance cost is reviewed.

2.2 Software Testing

Software testing is a process of program execution with the intent of finding bugs [141]. It tests the behavior of the program with a finite subset of a (typically) infinite set of possible tests [34]. A test case is an input that executes the system under test (SUT) [210]. In order to ensure software quality, it is very important to establish a rigorous and systematic testing strategy.

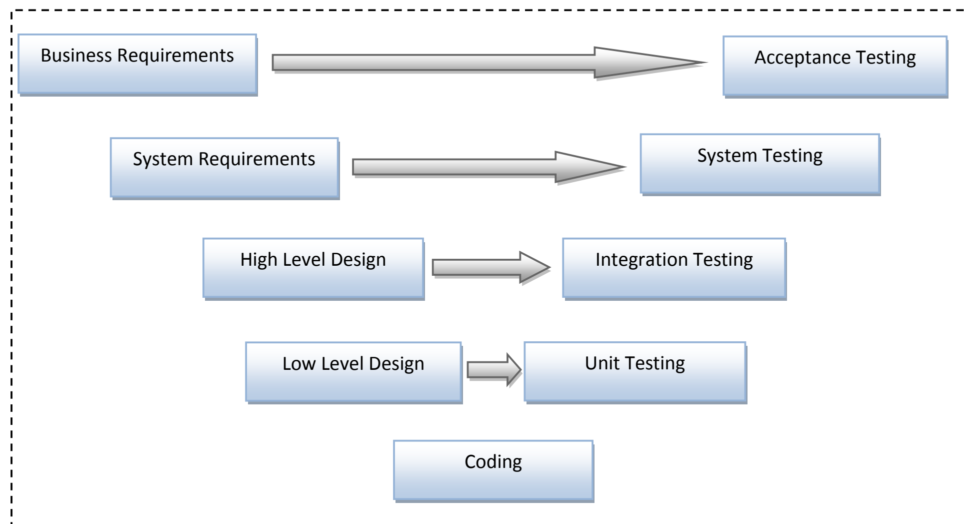


Fig. 2.1 V-model software development life cycle. [95]

Testing is a broad process applied starting from exercising a small piece of code, to verifying and validating the entire software system. Therefore, it has been categorized into four main levels of abstraction. Figure 2.1 shows the V-model that depicts four levels of software testing which are organized bottom up starting from *unit testing* to *user acceptance testing*. The left hand-side of the figure shows the different stages of software development, and the right hand-side matches the testing activities done for each of the stages. Every test level has its own objective, and together they ensure the quality of the whole software that is deployed to the end-user.

Unit Testing Unit testing is the first testing level which is based on application source code. In particular, in this level the program is tested against the component level specification (low level design). As the finest level in the hierarchy, *unit testing* is the starting point of test execution, and the input of the next test level (integration testing). **Integration Testing** Integration testing exercises the collection of program units and interactions between them. It is based on system high-level design and occurs just after unit testing. Moreover, it ensures that combination of units works together correctly. **System Testing** As the name of the level implies, this process exercises the application as whole, and ensures that it is conform to the user requirements. **Acceptance Testing** The acceptability of the software from its end-user is the final step of software testing. This process ensures that the implemented application conforms to business requirements, and is ready to be delivered to its end-user.

In addition to categorization based on the software development phase, testing techniques are further categorized based on the testing strategy applied called structural and functional

testing. Both of these categories cover different testing levels and have a different working procedure.

2.2.1 Functional Testing

Functional or black-box testing is a testing method where the internal implementation of the system is not known, or the tester sees the program as a black box. Indeed, there exists an independent specification, or model, of the system's desired behaviour, which is used to motivate testing. Test cases of this technique may be constructed based on a formal functional specification (e.g. Z, VDM), fine-state machines or extended finite-state machines or formal state-based specification (e.g. B), or (only) English statements

Black-box testing is very efficient for large and complex programs where tester and developer are independent. However, a part of the program may remain untested due to a limited number of test scenarios, and due to the missing clear specifications [103].

There are different techniques to select tests when following a black-box strategy [102], and some popular examples are as following:

- Equivalence Partitioning is a testing process that divides test conditions into groups (e.g., the system should handle them equivalently) and derive test cases. Indeed, if functional testing is based on a formal functional specification (e.g. Z, VDM) then this can be used to develop equivalence-partition tests.

  Boundary Value Analysis is a process that derives test cases by choosing boundary values such as minimum, maximum, typical or error values.

- State Transition Testing is a technique that during testing uses finite-state machines or extended finite-state machines or formal state-based specification.

2.2.2 Structural Testing

In contrast to functional testing, white-box requires only code (either source code, or some form of instrumented code) and exercises specific parts of the system, for example statements, branches, or paths, and it is evaluated with the percentage of code covered [143]. This testing technique can be applied to testing levels, such as, to unit, integration and system testing. However, due to the complexity of the program white-box testing cannot cover all possible paths of the program, and a part of the code will always remain not tested [104].

White-box testing is divided into techniques such as [104], [103]:

- Control Flow Testing is a testing technique that aims to cover aspects of control flow graph (branch condition, all-path coverage, multiple-condition decision coverage, nodes, edges, conditions, subpaths, etc.).
- Data Flow Testing is a technique that focus on variable/data paths and test the program from the point where variable is defined (assigns a value), to the point that it is used (define-use-kill). Hence, it basically ensures that all variables are defined (initialized to valid values) before they are used in the program.

White-box testing is used to exercise specific parts of the program (e.g.: branch, loop), and ensure that system code is working correctly. The specific problem addressed in this thesis is white-box testing, thus, the rest of this section surveys work on automation of software testing, with focus on structural testing techniques.

2.3 Automation of Software Testing

Software testing consumes almost 50% of the total development cost [125]. Until recently, most of the testing activities were performed manually by developers [113, 29], which is an error-prone and time-consuming process. Manual software testing cost increases more if the software complexity increases, which means that time to market increases, too [158]. Therefore, automated testing intuitively will increase efficiency, and decrease total testing costs (number of developers, time to market) [66], and serve as a systematic approach to continuous testing (retest application every time a new feature is added) throughout the software life-cycle.

Software testing contains four main activities, known as *test input generation*, *expected output generation*, *test execution*, and *test output verification*. From the mentioned activities, test execution is the process that can easier be automated, and there exist efficient automation frameworks such as JUnit, NUnit that allow unit tests to be defined and executed conveniently. However, the problem of test input generation and output verification (test oracle) still remain two important test issues.

In the development of this research thesis, we primarily focus on test input generation for unit test cases and improvement in generated test inputs.

Automatic test input generation is one of the current and future challenges in software engineering [30]. In industry test data generation usually is performed manually [30], which requires effort and is costly. Therefore, the software industry considers automated testing support to be highly desirable [142, 113, 191].

Throughout the years a number of methods for generating test data have been presented [110, 66], and a common approach is to do it *randomly*. However, as random test generation tends to result in the large sets of potentially long test cases, other techniques such as *symbolic execution, dynamic symbolic execution, and heuristic approaches* are further investigated.

This section carries out a broad literature review on automated test data generation. The strengths and weaknesses of individual approaches are evaluated in order to asses the current state-of-the-art and draw out the research questions.

2.3.1 Random Test Generation

Random testing is one of the most 'simplest' (simple to implement) and popular testing techniques. It generates test inputs randomly and independently from program specification, until a predefined given time is reached. Random test generation is conceptually easy to understand and cheap to implement, however, it tends to generate illegal test inputs (inputs that are in contradiction with input constraints) and has a low chance of finding randomly a specific input value. For example, considering the code fragment shown in Figure 2.2, the probability of finding two equal randomly generated inputs (variable a is equal to b) from a considerably large input pool, is very low, and for more complex branches it is almost zero.

Although random testing is frequently used, it has known limitations[14], and therefore Pacheco et al. [146, 147] proposed an improved version of it called feedback-directed random testing. This new method using feedback from previous executions, generates legal test inputs that helps with code coverage.

Since random testing is unpredictable and sometimes inefficient in exploring all paths of the input space, Chen et al. [49] proposed an advanced technique called Adaptive Random Testing (ART) that takes into consideration patterns of failure-causing inputs, and positively affects testing performance. In another way, it improves test generation strategy by starting with a randomly generated test input and continues by selecting inputs that are not too close to previously generated tests. This time, test inputs that are used are not close to each other and technique achieves higher code coverage. Adaptive random testing and the coverage that it can achieve is further investigated from Arcuri and Briand [16]. Using a large empirical analysis on ART, they showed that computational cost for calculating distances between test data are very high, therefore, it is highly inefficient technique even on trivial problems for distance calculation among test cases. Indeed, for test generation, ART in practice will be very expensive in terms of time, and the rate of finding failures is slightly better than with classic random testing.

```
if(a==b){
 TARGET
}
```

Fig. 2.2 Example code with a condition

Although random testing seems to be inefficient in terms of coverage, when software specification is incomplete and software source code is unavailable [12], it is a practical choice for experimenting the code under test. However, taking into consideration the complexity of software systems, this technique has a low probability to generate quality test data, and covering a suitable amount of code. Thus, in the next section, we will summarize symbolic execution, a different test generation technique, which is found to be a more advanced technique.

2.3.2 Symbolic Execution

Symbolic execution is a test generation method that executes the program based on symbolic values (real input values are replaced with symbols, for example 5 is replaced with γ) and creates the symbolic representation of the output variables. The key idea behind this method is to derive path constraints for given program paths. Therefore, during symbolic execution a program will contain symbolic values of variables at that state, path constraints in order to reach to that state, and a program counter [12]. A path constraint (PC) is a condition (boolean formula) that input should satisfy in order to execute a particular part of the program. Any PC that is not satisfiable represents an infeasible path, and any solution of the PC is a program input that executes the corresponding path.

```
1 public void sample(){
2  int a,b;
3  if(a>b){
4   result = a-b;
5  }else{
6   result = b-a;
7  }
8 }
```

Fig. 2.3 Example method that divides two integers

To illustrate the process of test generation with symbolic execution, consider the code fragment in Figure 2.3 that subtracts integer *a* from *b* if the value of the first integer *a* is greater than the value of the second integer *b*, or it subtracts integer *b* from *a* if the value of the first integer *a* is less than or equal to the value of the second integer b. Figure 2.4 shows

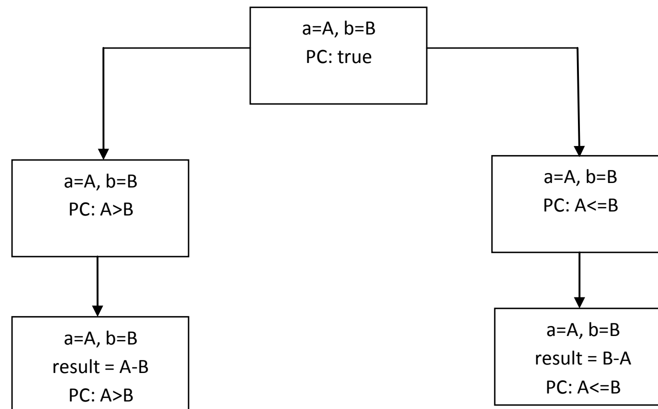


Fig. 2.4 Symbolic execution of code that subtracts two integers. Path Condition (PC) is the conjunction of all symbolic constraints along a path. In this example, symbolic execution has the first valid path when a>b (if condition in line 3), the second valid path when a<=b (else condition in line 5).

the symbolic execution tree, or the execution paths for the mentioned code fragment. As a starting point, each input variable will be assigned to a symbolic value (in this case A for a and B for b), and the *program condition-PC* is initialized to *true*. Next, there are two possible execution paths in the program, and the first one is the if-condition in line 3 that in order to be true the symbolic variable A needs to be larger than the symbolic variable B. Thus, the PC for this path is set to A>B. Then, if the if-condition is false, the symbolic variable A needs to be less than or equal to the symbolic variable B, and the PC for this path is set to A<=B. Therefore, as result there will be two feasible paths where each is possible for the given values of input *a*.

Symbolic execution was introduced in the middle of the 70s, by King [106, 107], Boyer [35], and Howden [94] . Although it was used on bug finding, improve code coverage [42, 84, 105], and fault localization [157, 42], it still suffers from limitations like path explosion or complex path constraints.

Symbolic execution may not cover all program paths, if there is a large number of them, or if the program under test is complex. In particular, the number of paths in a program grows with the complexity of the program, thus given a time limit covering all of them is not realistic. For example loops in a program are very common, a single loop can generate an exponentially large number of feasible paths which cannot be handled by symbolic execution and raises path explosion issue. Another limitation of this technique would be complex object handling (e.g., arrays). As every complex object in symbolic execution is considered as an array of bytes, and each byte should have a separated symbol, the number of symbolic variables becomes very large which may be a poblem that cannot be handled [19]. Furthermore, tools

using symbolic execution approach [83, 11] are limited in terms of the complexity of the program they can be applied to.

However, there have been attempts to address these limitations with more advanced techniques. For example compositional symbolic execution is a method that automatically computes program summaries or pre and post-conditions. This approach instead of analyzing each time the functions, it reuses the summary when computing higher-level functions [83]. However, in order to further reduce the number of paths, another technique has been proposed, called demand-driven compositional symbolic execution, which explores as few paths as possible and avoid to explore paths that guarantee not to cover the target [11]. However, the most successful and popular technique for addressing these limitations is Dynamic Symbolic Execution (DSE) covered in the next section.

2.3.3 Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is an improved version of symbolic execution. As a technique, it combines dynamic and symbolic execution, such that, during execution of the program using the concrete values, the symbolic path condition for the executed path is determined [202, 41, 84, 174]. Therefore, with DSE first there is chosen a random input value and executed. During this concrete execution the symbolic path condition is determined. For this path condition, one of the constituent conditions is negated. Solving the resulting path condition represents an input that follows a different path. The program is executed with the solution values, and the symbolic path condition for this execution is determined. Then, iteratively, another condition is negated, solved, and so on. DSE intertwines the strength of random testing and symbolic execution, and executes the program simultaneously both concretely and symbolically when facing complicated pieces of code. (For example, for non-linear, or constraints depending on external calls that cannot symbolically execute, it can simply use the concrete values.) Also, the path selection problem in DSE is predefined by the initial random inputs and the consecutive systematic exploration.

Path exploration in dynamic-symbolic execution is costly, therefore, there are heuristics which are applied in order to guide the search toward the target, or explore only a small relevant number of paths. For example, Xie et al. [202] applied a strategy called fitness-guided path exploration, which prioritizes the search by minimizing fitness values and indicating how close is a path to cover the target code. Burnim and Sen [37] in their work proposed a strategy that is guided by the control-flow graph of the program under test, and makes able the exploration of larger programs.

Although recent implementation of DSE successfully generate test cases, it still suffers from some limitations. For example, real-world applications with a large number of program

paths using databases, file systems, networks, large frameworks, and advanced programming language features, make the technique complex with huge amount of effort to build test inputs [189].

2.3.4 Search Based Test Data Generation

Search-based software engineering has been proposed for a wide variety of software engineering fields, however, one of the most active areas was search-based test generation. The first publication on search-based testing dates back to 1976 when Miller and Spooner [136] applied numerical maximization technique and generate floating point numbers for program paths. Their technique replaces all conditions containing floating point numbers with constraints, which then are used to measure how close the input comes to triggering the condition that allows the software to take the desired path. Another publication after this came from Korel [110] on 1990. Korels' work was based on the first search-based test generation approach, that executed the program with an arbitrary input vector, and if the path is traversed test is recorded, otherwise the branch distance is measured. Shortly after Korel, Xanthakis et al. [199] applied another path strategy. They tried to improve random testing, by satisfying all branch predicates, in order to execute target branch and increase path coverage.

Search based software testing (SBST) continues to remain an active area of research [133]. Other kinds of improvement have been reported for this strategy, focusing on branch coverage as as an important search criterion [132, 135, 111]. More specifically, search-based testing is a process of test data generation using search-based algorithms guided by an adequacy criterion called fitness function. The fitness function is a fundamental part of search algorithms that guides the search to find the optimal solution. In Evolutionary Algorithms, the fitness function is an important factor that leads the searching process [150]. Figure 2.5 shows a code sample and the target part that needs to be exercised. As it is seen, the target code depends on three conditions, and if the wrong choices are made at either of these *if* statements, the target will be missed. Otherwise, the target is reached, or the fitness function will drive the search through the true branches of the conditions and cover the target code. Each individual during the search is expressed with a fitness value (numerical value), which can be used to compare individuals and the best can be chosen as the next offspring (new solution).

In the context of software testing with search-based approach, the fitness function is used to evaluate generated test inputs. Therefore, based on the fitness value type the test objective will be to generate test inputs with high code coverage, or test inputs covering certain control constructs.

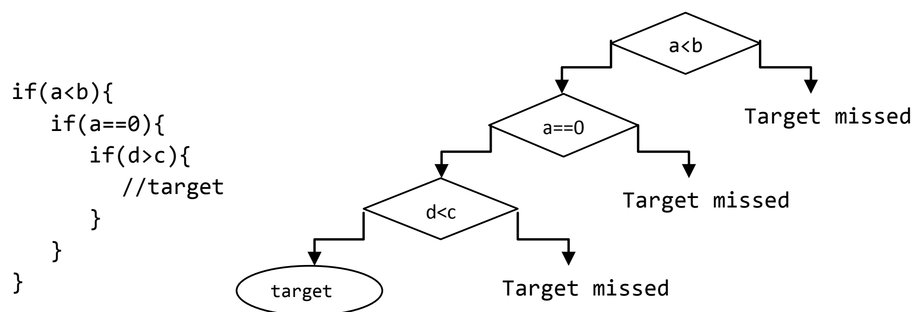


Fig. 2.5 Fitness function calculation.

2.3.4.1 Meta-Heuristic Algorithms

Meta-heuristic search techniques refer to problem independent approaches that apply various heuristic techniques in order to find a solution to difficult problems. In particular, a meta-heuristic search technique is an algorithm that can be applied to many different problems but needs to be configured/parameterized with a problem-specific fitness function. Fitness function determines how close is the solution to the optimum solution. Indeed, a search using meta-heuristics will start in random and proceed depending on the evaluation of candidate solution. Furthermore, with the feedback from objective function, the search will explore a better solution based on knowledge and feedback of the previous solution.

Search using meta-heuristic techniques can be local or global. Local search that considers only immediate neighborhood of candidate solutions during the search, and global search that uses search operators to move around freely across the search space, and is typically population based. The next section summarizes meta-heuristic search techniques used in software testing.

2.3.4.1.1 Hill Climbing Hill Climbing is a well known local search algorithm that starts from a random point in the search space and attempts to improve a single candidate solution. Same as any other local search algorithm, that considers only immediate neighborhood of candidate solution, hill climbing moves from one point to the other, and makes local changes until an optimal solution is found. In particular the solutions are driven by the fitness function or the objective function of the search, and whether a better objective value is higher or

lower value, depends on whether the search is trying to maximize or minimize the objective function.

Algorithm 1 shows the flow of the process that starts from a random solution *s*. The search will continue, and if a better solution *s'* is found the current solution will be replaced. Indeed, this means that the objective value of the new solution ΔE(s') is better than the objective value of the current solution ΔE(s). Next, the new search will start and neighbors of the new solution are investigated by replacing the starting point until no better solution is found.

The neighbors of the starting point in hill climbing approach are selected based on "steepest ascent" or "random" strategies [133]. If the steepest ascent approach is used, then all neighbors are evaluated and the best is chosen. However, if the random approach is used than algorithm selects randomly a neighbor and if that is better the search continues, otherwise the search is stuck, and an optimum has been reached.

Hill Climbing algorithm gives fast results, however, since algorithm selects the best candidate near to the randomly chosen starting point [133], it may accept local optimum which is not the best solution in the search space. For example Figure 2.6 shows a sample search space with several local optima (in red) and one global optimum (in green) points. Hence, the best solution can be achieved only if the search starts around global optimum point, otherwise it will get stuck in any local optimum point and the search will stop.

Algorithm 1 Hill Climbing Algorithm

1: **procedure** HILLCLIMBINGSEARCH(s,s')
2: $s \leftarrow$ *Current solution*
3: **while** *Termination is not reached* **do**
4: $s' \leftarrow$ *New point in space*
5: **if** $\Delta E(s') > \Delta E(s)$ **then**
6: $s \leftarrow s'$
7: **end if**
8: **end while**
9: **end procedure**

To overcome the problem of local optima in Hill Climbing, one of the solutions would be to restart the search from randomly generated initial states and collect results until no better result is found. This approach that helps to find better solution came as meta-algorithm built on top of Hill Climbing and it is called *Random-restart Hill Climbing*.

2.3.4.1.2 Simulated Annealing The Simulated Annealing algorithm works similar to Hill Climbing, with the difference that it accepts movements to worse solutions (compared to

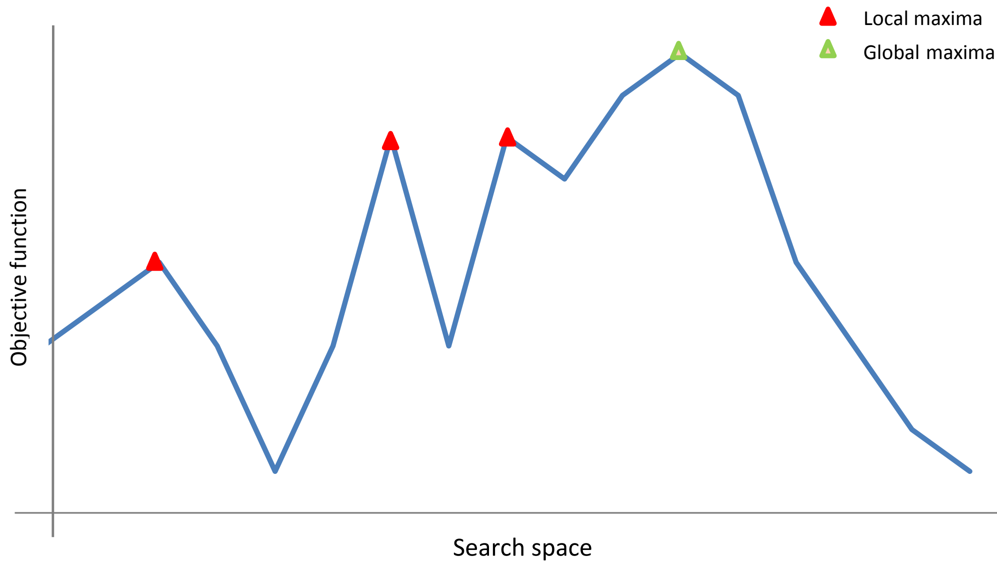


Fig. 2.6 Hill Climbing search space

the current solution). The advantage of this technique is to avoid accepting local maxima and search further for better solutions.

Algorithm 2 illustrates the working process of Simulated Annealing. The search starts with a random solution s in the search space and a temperature t. Search will continue to find a new solution s', and accepts if its objective function is higher than the current solutions objective function $\Delta E(s') > \Delta E(s)$, or if the probability of acceptance ($p = e^{-\Delta E(s')/t}$) is greater than the implemented random number R. In particular, the probability of accepting a neighbour with a worse fitness value as solution is dependent on the temperature-t (cooling schedule). When the search starts the temperature t is high and as the search progresses the temperature decreases, thus allowing the choice of local optima [133].

2.3.4.1.3 Genetic Algorithm Genetic algorithm (GA) is one of the well known forms of the global search algorithm. It represents an adaptive search technique, which uses natural evolution analogy. In terms of software testing GA searches the domain for inputs that satisfy the specific objective function.

In the 1960s Rechenberg, a scientist from Germany, was the first to introduce "evolution strategy" in searching problems, whose work was further followed by Schwefel on 1975. The genetic algorithm was independent of evolution algorithm, and it was intro-

Algorithm 2 Simulated Annealing Algorithm

1: **procedure** SIMULATEDANNEALINGSEARCH(s,s',t,R)
2: $s \leftarrow Current\ Solution$
3: $t \leftarrow Initial\ Temperature$
4: $R \leftarrow Implemented\ random\ number$
5: **while** $t = 0$ **do**
6: $s' \leftarrow NeighbourSolution$
7: **if** $\Delta E(s') > \Delta E(s)$ **then**
8: $s \leftarrow s'$
9: **else**
10: **if** $R < probability\ of\ accepting\ new\ solution\ depending\ on\ temperature\ t$ **then**
11: $s \leftarrow s'$
12: **end if**
13: **end if**
14: $decrease\ temperature\ t$
15: **end while**
16: **end procedure**

duced from John Holland almost at the same time when Rechenberg introduced evolution strategy [133]. In these early years, there were presented recombination strategies such as mutation and crossover which was further the basis of almost all theoretical work on genetic algorithms [138].

The main idea of a *genetic algorithm* is to combine individuals of a population and produce offspring that are better than their parents. Therefore, the search will start with an initial population, and in each iteration, the fitness of each individual (chromosome/genome) is evaluated, and best individuals are selected and used for reproduction. Fitness evaluation means calculating the objective function and assigning a value based on which chromosomes are compared.

At a high level, the whole process goes through *selection, crossover* and *mutation* which work as follows.

- *Selection* is the first step of the search process, where a set of individuals are selected in order to produce new offspring. Individual selection is done based on the fitness value assigned to them, where fitter solutions are more likely to be selected.

- *Crossover* processes once the individuals are selected. Therefore, a crossover point is chosen and new offspring are created by combining parents at that point. Figure 2.8 represent the crossover process where the parents are combines and two offsprings are generated with the aim of having higher fitness value. However, as crossover may

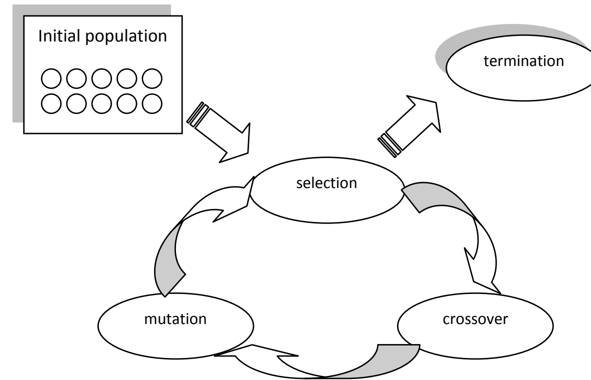


Fig. 2.7 Genetic algorithm working procedure.

not suffice in finding solutions with enough diversity from the initial population, the mutation process is applied.

- *Mutation* includes a walk through the search space to maintain diversity within the population. Every offspring goes through mutation and their variables are flipped by a small perturbation to prevent the population from being too similar.

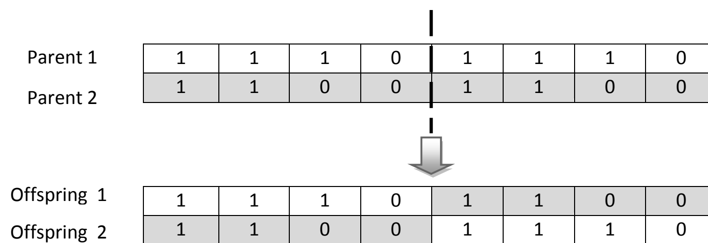


Fig. 2.8 Crossover process.

Figure 2.7 presents the search process which starts with initial population and continues until the fitness function is achieved.

Search-based technique is successfully used on test data generation. In order to produce test input this techniques requires defining some sets of properties and using an appropriate meta-heuristic technique. These properties are specification of a testing criteria (e.g., branch coverage), representation of candidate solution which can be encoded (e.g., input vector), outlining the program input domain, and defining the fitness function in order to calculate

the fitness values for candidate solutions. Even though search-based testing is found to be an effective approach for test generation, the next section describes a combined approach that will cover drawbacks of this approach during test generation.

2.3.5 Hybrid Test Generation

Automatic test generation techniques are found to be good contributors in software quality assurance. For example, random testing is cheap and easy to run, symbolic and dynamic symbolic tend to achieve high code coverage, search based technique brings good testing solutions for potentially infinite search space within reasonable amount of time. However, to improve software quality by generating test cases with high code coverage, so-called *hybrid* approaches have been proposed that combine strength of search-based testing and dynamic symbolic execution.

One of the earliest hybrid test generation techniques came from Inkumsah et al. [96]. They created a novel framework called Evacon that integrates DSE and evolutionary technique and generates tests with higher code coverage that any of these techniques can achieve alone. In particular, Evacon uses evolutionary testing to search for best method sequences and then symbolic execution to generate desirable method arguments by exploring alternate paths within the methods under test. Furthermore, a hybrid approach is applied in Pex and CORAL [112] test generation tools and addresses the issue of floating point computations in DSE that most constraint solvers have. Therefore, the proposed technique uses SBST to solve floating point problems in the code on which DSE is limited by the power of constraint solver. This work was evaluated and it shows that in order to achieve an improvement it needs adequate resources.

DSE and SBST are combined together by Malburg and Fraser [127] to produce better test inputs. With this approach, authors tend to overcome the disadvantages that SBST has on exploring the search space and problematic search areas, and DSE has on constraint solving, and achieve high code coverage with generated test cases. It starts as a search based technique where a genetic algorithm evolves the population of candidate solution and adds fitness value to each one. However, in order to avoid the slow search and being stuck into specific places, mutation operation considers the path conditions that represent the execution path of a candidate solution and negates one of them like dynamic symbolic execution does. This hybrid testing approach was found to result with branch coverage 28% over search-based, and 13% over constraint-based technique. Combination of DSE and SBST is further explored by Baars et al. [18]. They proposed a new search technique that uses information collected from DSE to complete the search. The new approach replaces the existing branch distance and approach level measures with path distance and approximation

level. Also, they introduced a metric or a technique that accounts for uncertainty such as in the presence of loops. This technique was evaluated and found to be more effective than local and global search-based testing techniques. CBST is another combination of constraint-based testing and SBST proposed from Sakti et al. [167]. In this work, they investigated a new method that instead of generating test inputs, it models a relaxed version of the unit under test or candidate test inputs based on which SBST creates actual inputs. This new technique was compared with search-based and constraint-based testing, and show that the combination of both outperforms them.

Galeotti et al. [80] introduced a technique that during test generation uses GA and take feedback whether the problem is suitable to be solved from DSE. Therefore, using GA and DSE the whole test suite is optimized and test cases with high code coverage percentage are produced. DSE was implemented in EVOSUITE test generator that uses SBST, and applied for evaluation of the approach. The new hybrid configuration increases the branch coverage, even though there are still open questions compared to state-of-the-art tools like Pex, and it is beneficial in practice.

Combining techniques is a useful approach that absorbs the strength of each technique and improves test generation. However, combining different approaches in one place is not trivial [124], therefore, yet there remain several open issues that can be improved and make hybrid approach usable.

2.3.5.1 Lazy Systematic Test Generation

Lazy systematic testing is a combination of search-based code exploration, followed by exhaustive model-based testing [177]. The issue of foreseeing unexpected interleavings of methods that affect the state of the test object, is usually hard for programmers during testing [175, 176].

Lazy systematic testing is a different testing method based on two ideas of *lazy specification* and *systematic testing*. Lazy specification (evaluation) means that a stable program specification is not necessary to be committed until the programmer wants to close it (the program design is completed). The main idea behind this is that the program specification can change or evolve in parallel with changing source code. Systematic testing, in contrast to random testing, tests the program unit completely up to some criterion, or it tests the unit exhaustively until it conforms to some specification. Therefore, lazy systematic testing is a technique that can provide exhaustive testing from a specification that was acquired incrementally.

Despite the fact that lazy systematic testing can fully exercise a class under test, it still suffers from the fact that it requires tester to supply a custom value-generator (to replace generated values), to ensure full coverage of one example. [180].

2.4 Unit Test Generation for Object-Oriented Software

A unit test is a piece of code written by developer for testing the any unit in the application, in isolation from the other parts of the program. In an object-oriented program, a unit test is a sequence of method calls and assertions, that can exercise units of the application. For example Figure 2.9 presents a Java code, or a method that `returns true` if a number is multiple of three. The test suite in Figure 2.10 contains three test cases with method calls and assertions, or statements that exercise `multipleOfThree`. If the test suite is run, the first test case will fail (2.10 assertion in red), and developers can easily find the bug in the code (the corrected version is in Figure 2.11).

2.4.1 The State of the Art and Practice of Unit Testing

The growth of software systems has affected the way developers test their programs. For example, in Extreme Programming (XP) one development approach recommends designing the (initially non-executable) tests in some test-harness, then developing the code to suit, which is called test-first development [25]. With this approach first test cases are designed, and then developers do programming of the units that satisfy the tests.

Unit testing and its importance are highly investigated in the research area [24, 25]. Today, almost every programming language has its own unit test framework (e.g., JUnit for Java, NUnit for C#) [2, 3] that contribute in almost every stage of software development [91].

Indeed, unit testing has found application in practice, too, and there are a number of surveys that have shown this ([166, 191, 82, 81, 142, 113, 46]). For example, Runeson conducted a survey on unit testing practices [166]. This survey contained 50 questions such as 1) what is unit testing? and 2) strengths and weaknesses of unit testing. After surveying 12 different companies with 15 participants, the author concluded that companies with a clear understanding of unit testing will likely understand and accept their responsibility. Furthermore, the results show that unit testing and test maintenance are difficult activities, however, it does not survey the potential that automated unit test generation has on test improvement. Torkar and Mankefors [191] conducted a survey on software reuse and testing. This survey presented the insight about boundary value analysis which is commonly applied by developers when writing unit tests. While developers state that they reuse code to a fairly

high extent, unfortunately, they do not test the reused code. Therefore, the survey revealed a strong desire of developers for automated testing tools, and tools that tell them how well their tests are. Geras et al. [82] conducted a survey on testing practices in Alberta, and found that unit testing was the most popular testing approach, yet was only applied by 30% of testers. A larger survey in all of Canada [81] confirmed the relative popularity of unit testing, and interestingly also identified growing interest in mutation analysis. Causevic et al. [46] found that unit testing was more popular for web applications than for other domains such as embedded systems.

Although there are still gaps between research and industry practices [99], conducted surveys confirmed the relative popularity of unit testing in the industry and revealed a strong desire of developers for better tools when writing unit tests.

2.4.2 JUnit

The main objective of this thesis is realized using Java programing language. Therefore, this section will illustrate the unit testing process with Java, and the example presented is a simple Java class that is tested with a JUnit test platform [2].

JUnit is a unit testing framework for classes written in Java language. Any Java class that imports `org.junit` library is detected as *JUnit* test suite (a set of test cases), and every method with *@Test* signature is a test case that can exercise methods in the class. A JUnit test case is a set of method sequences and assertions. Method calls in a test case represent the test scenario [194] which put the SUT into the desired state, while assertions or predicate functions check test expected value after the execution of the methods in the test. During unit testing, each test cases exercise a specific method in the class.

```
public class Calculator {
 boolean multipleOfThree(int x){
  if (x % 3 ==0)
   return true;
  return false;
 }
}
```

Fig. 2.9 Example class: Calculator with a method that returns true if the input value is multiple of three.

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void testZeroMultipleOfThree() {
    Calculator calc = new Calculator();
    assertFalse(calc.multipleOFThree(0));
  }
  @Test
  public void testThreeMultipleOfThree() {
    Calculator calc = new Calculator();
    assertTrue(calc.multipleOFThree(3));
  }
  @Test
  public void testFiveMultipleOfThree() {
    Calculator calc = new Calculator();
    assertFalse(calc.multipleOFThree(5));
  }
}
```

Fig. 2.10 Example test suite for class Calculator for input values 0 and 3. The red assertion is where the test case fails.

```java
public class Calculator {
  boolean multipleOFThree(int x){
    if (x != 0 && x % 3 == 0)
      return true;
    return false;
}
```

Fig. 2.11 Example class: Calculator with a method that returns true if the input value is multiple of three.

Figure 2.10 shows a JUnit test suite. The test cases are written for class `Calculator` in Figure 2.9 which contains only one method called `multipleOfThree`. Developers can write as many test cases as they like, and JUnit platform will run them showing a result as in the Figure 2.12. Figure 2.12 is JUnit GUI interface which shows the number of test methods, number of failures, and a color bar that is red if a test fails, or green otherwise.

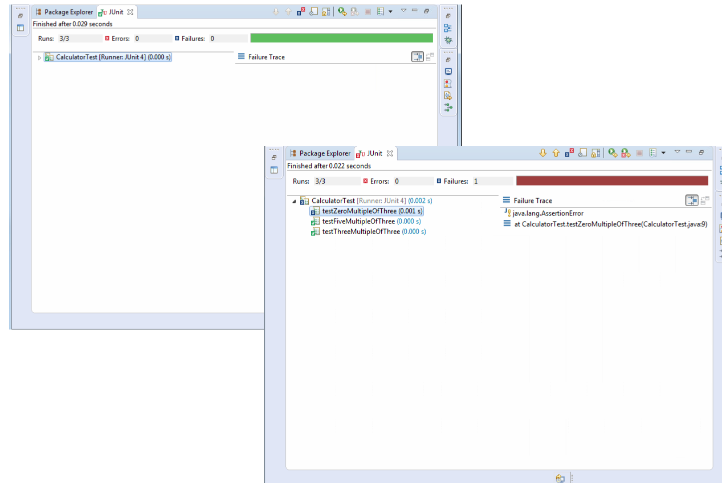


Fig. 2.12 JUnit Eclipse interface.

The next section describes Java testing tools for each each of the testing techniques covered in the previous section (Section 2.3).

2.4.3 Random Test Generation for Java

A generated unit test case with a random approach is a set of method calls that instantiate the unit under test and change the state of the object, and assertions that check the final result of that unit. At a high level, from the code under test, the random approach will randomly select arguments and create a test case. For example to test a function $f(a,b)$ it will randomly select arguments for a and b and then apply them to f.

Randomly generating unit tests is found to be cheap and fast, therefore a number of tools have been developed forpurpose [146, 53, 4, 98, 144]. The aim of the listed tools is to explore the input space by selecting random inputs that do not cause the software to raise exceptions.

2.4.3.1 JCrasher

JCrasher[53] is a random test generation tool for Java code. It generates random test sequences for the class-under-test and monitors when the program crashes as a consequence of exceptions raised by the code causes. As input, JCrasher takes the class under test and a time limit and constructs a random series of method invocations, whose supplied random argument values respect their declared types.

```
public void test93() throws Throwable {
 try {
  Color c4 = new Color(-1.0f, -1.0f, -1.0f);
  Canvas c5 = new Canvas(-1, -1);
  c5.Write(-1.0f, -1, 0, c4);
 } //[..]
}
```

Fig. 2.13 Example test case generated with JCrasher [53]

JCrasher provides filtering heuristics that seek to determine whether a raised exception was due to an underlying program fault, or whether the test sequence simply violated a method precondition. The tool only notifies programmers about the exceptions that correspond to underlying faults. The heuristics are based on classifying types of Java exception: a fatal Error is always considered a fault; a declared Exception of the class-under-test is always considered a precondition-violation; and an undeclared RuntimeException must be further analysed. If this is a low-level exception raised by the JVM, it is considered a fault in the class-under-test; but if it describes a bad argument or state value, then it is considered a precondition-violation, unless it was raised in a public method called transitively by the method-under test.

Figure 2.13 shows a test case generated with JCrasher for method `write` of class `Canvas`. Generated statements in the test bring the program in the state that it should throw an exception, and any failure is called robustness exception which does not represent a failure as the input violates routine precondition.

2.4.3.2 Randoop

Randoop [146] is a tool that automatically generates unit test inputs for Java classes. Randoop is based on a random approach, that takes as input a set of classes under test, a time limit, optionally a set of contracts, and generates two test suites where the first one contains contract-violating (tests that exercise scenarios where the code under test leads to the violation), and the second one contains regression test cases (tests that exercise code current behavior). In particular, from the set of target classes and their methods, Randoop randomly picks an existing sequence from its pool (one that does not end in an exception) and tries to append a new call by satisfying the parameters with existing objects. Randoop repeats this process until a predefined time limit is reached.

```java
public static void test1() {
  LinkedList l1 = new LinkedList();
  Object o1 = new Object();
  l1.addFirst(o1);
  TreeSet t1 = new TreeSet(l1);
  Set s1 = Collections.unmodifiableSet(t1);
  Assert.assertTrue(s1.equals(s1)); // Fails
}
```

Fig. 2.14 Example test case generated with Randoop [146].

Randoop improves random test generation, by implementing a new method called feedback-directed random testing, that executes tests as they are created and obtains feedback from them, in order to determine whether the test sequence may be used as the prefix for further test sequences. In particular, generated test cases are sequences of method calls as in Figure 2.14, and for each call, argument values may be selected from the results of previous calls in the sequence. Randoop also expects the user to supply contract-checking classes that verify properties of the Java language specification, for example that the *equals()* method is reflexive. Each contract-checker provides a method boolean *check(Object)* which returns false, iff the contract is violated. Thus, when a test is created it is executed and checked against a set of contracts. If an exception is raised, the sequence is considered illegal and is discarded. If any contracts are violated, the sequence is saved as a contract-violating test. Otherwise, the sequence executes normally, so is saved as a regression test. Only regression sequences may be extended in the next test cycle, since extending a sequence that already terminates with an exception is redundant.

Randoop can successfully generate test cases for Java programs, however, depending on a time limit and program size, it generates large and complex test suites that are too complicated to understand [118].

2.4.3.3 GRT

Guided random testing GRT [126], is an extension of Randoop that applies static analyses to the system under test, and together with the information collected during run-time it guides further the test input generation. The original idea of GRT is to improve feedback-random testing by generating test with higher code coverage.

Indeed, GRT is an automatic approach which as input has the code under test, and for the given time limit outputs the generated test cases. In general, it contains two main phases called static and runtime. In the static phase the tool performs static analysis on the method under test and favor methods that can change object states, and in the dynamic phase which

is based on feedback-directed random testing, the loop starts with the selection of the method for testing whose inputs are generated and method sequences are created. Next, the created sequences are executed and in the end evaluated and stored in the main object pool. Figure 2.15 shows a test case generated with GRT, or a set of method sequences which tend to improve code coverage.

```
public void test() throws Throwable {
  ChangeSet var0 = new ChangeSet();
  SevenZArchiveEntry var1 = new SevenZArchiveEntry();
  byte[] var3 = new byte[] {(byte)10, (byte)10};
  java.io.ByteArrayInputStream var5 = new java.io.ByteArrayInputStream(var3, 1, 1);

  TarArchiveInputStream var6 = new TarArchiveInputStream((java.io.InputStream)var5)
      ;
  var0.add((ArchiveEntry)var1, (java.io.InputStream)var6);
  var0.deleteDir("0x7875 Zip Extra Field: UID=1000 GID=1002");
}
```

Fig. 2.15 Example test case generated with GRT [126].

2.4.3.4 JTExpert

JTExpert [168] is a guided random test generation tool, that achieves to generate tests with high code-coverage. It analyses the internal structure of the class under test and minimizes the search space for test generation. In general, JTExpert divides the problem of test generation into three main categories called D1 or instance generation for CUT and other required objects, D2 or finding sequences of method calls to put the CUT into the desired state, and D3 or finding an adequate method to reach the test target.

JTExpert during test generation targets all uncovered branches at the same time, and it does not waste time with unreachable branches. Also, it benefits from a significant number of accidentally covered branches. Indeed, after test generation, JTExpert produces JUnit test cases for every class under test. For example, Figure 2.16 shows a test cases generated for commons.ArrayUtils.java class.

```
@Test
public void TestCase24() throws Throwable {
 ArrayUtils clsUTArrayUtils=null;
 clsUTArrayUtils=new ArrayUtils();
 int clsUTArrayUtilsP2P2=721;
 int clsUTArrayUtilsP2P3=-2;
 double[] clsUTArrayUtilsP2R=null;
 clsUTArrayUtilsP2R=ArrayUtils.subarray((double[])null,clsUTArrayUtilsP2P2,
     clsUTArrayUtilsP2P3);
 assertNull(clsUTArrayUtilsP2R);
 String clsUTArrayUtilsP3R=null;
 clsUTArrayUtilsP3R=clsUTArrayUtils.toString();
}
```

Fig. 2.16 Example test cases generated with JTExpert.

2.4.3.5 T3i

T3i [156] is an automatic test generation tool controllable by user. With T3i user can configure and generate various kind of test suites, and can combine different operations (e.g., Hoare triples, LTD formulas, algebraic operations) and construct test suites with specific properties. T3i treats test suites as first class objects, and uses random technique at the back-end.

For example queries
$query(S).with(\phi).sat()$
$S' = query(S).with(\phi).collet()$
are as result of T3i test generation. The first query checks for specific property ϕ on test suite S, while the second query collects all those sequences on the new test suite S'.

2.4.4 Symbolic Execution for Java

Symbolic execution is applied in automated input generation for unit testing. At a high level, symbolic execution for a method under test will generate symbolic values and continue to assign those values to any assignments. In case of branches or loops, symbolic execution will generate path constraints for every branch. Therefore, whenever a symbolic execution along a path terminates, a specific condition is solved, which generates concrete values or test inputs.

The next section presents a group of tools that use symbolic execution, and solve generated input constraints and translate them to actual values that can be executed.

2.4.4.1 JPF-SE

JPF-SE (Java path finder symbolic execution) is an extension of PathFinder (JPF) state model checker [10]. It enables symbolic execution of Java programs to be performed during model checking. More precisely, JPF-SE: (1) checks the behavior of the code under test using symbolic values, (2) using JPF build-in capability, it effectively search through the program state space (e.g., systematic analysis of different thread interleavings, heuristic search, state abstraction, symmetry and partial order reductions) (3) do modular analyses on isolation, (4) do automated test input generation, (5) prove light-weight properties of Java programs, and (6) manipulate symbolic numeric constraints.

JPF-SE has been used for test generation of Java classes and checking concurrent Java programs, showing positive results. However, as it still can work only with numerical values, JPF-SE is being extended in order to handle complex data structures, too.

2.4.4.2 Symstra

Symstra [201] is a framework for generating unit test cases using symbolic execution approach. For code under test, Symstra generates method call sequences with symbolic values, and by exploring the symbolic object state (all possible solution of symbolic argument) it can generate tests faster.

```
public void test() {
 BST t = new BST ();
 t.insert(x1)
 t.insert(x2)
 t.insert(x3)
 t.insert(x4)
}
```

Fig. 2.17 A sequence of four method calls with symbolic values generated with Symstra[201].

For example, for given sequence with four method calls in Figure 2.17 for class `BST`, Symstra will create an execution tree, and for every feasible path will create a test case. If we consider the path constraint $x_1 > x_2$ and $x_1 = x_2$, two tests that can be generated are as in Figure 2.18.

```
public void test1() {
  BST t1 = new BST ();
  t1.insert(0);
  t1.insert(-1);
  t1.insert(0)
}
public void test2() {
  BST t2 = new BST ();
  t2.insert(2147483647);
  t2.insert(2147483647);
  t2.insert(-2147483648)
}
```

Fig. 2.18 A sequence of four method calls with real values generated with Symstra [201].

2.4.4.3 Hex

The limitation that symbolic execution has on dealing with complex data types, has been investigated by Braione, Denore, and Pezze. They proposed a novel approach called HEap eXploration Logic (HEX) [36], which symbolically executes programs by taking as input both numeric values and heap data structures, and need complex representation constraints over these inputs. Hex symbolically executes the target program, focusing on valid data structures, avoiding invalid data structures, and preventing many consequent false alarms.

Figure 2.19 presents a code snippet containing List data-type tested from Hex, and Figure 2.20 shows the symbolic data structure analyze of Hex for the given code.

```
List getList (List list, int foo){
if (foo < 0) {
  List first = list;
  List second = first.next ;
  return second.next ;
}
else {
  if (foo>10) return null;
    if (foo>5) return new List();
  }
}
```

Fig. 2.19 A sample code manipulated from Hex [36]

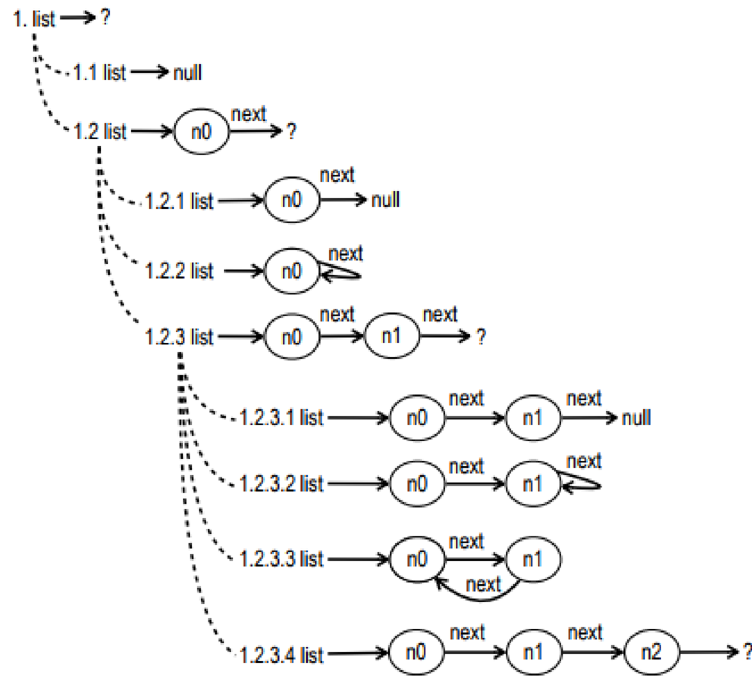


Fig. 2.20 Symbolic data structure analyses of sample code `getList` [36].

2.4.5 Dynamic Symbolic Execution for Java

DSE as an improved technique that is used for unit test generation, too. In order to explore a higher number of paths within a limited cost, heuristics-based approaches are adopted, which in general are successful. For example, if Example 2.21 is going to be tested with dynamic symbolic execution with values 0,1,2,3 for a,b,c,d values respectively, the first path condition is $< a \neq b >$. After the first execution, the symbolic predicate is negated to $< a = b >$ which creates a constraint to be solved. Next, the constraint solver can return 0 for a and b values, and the new path condition will be $< a = b \wedge b \neq c >$. Therefore, after a while the constraint solver will return equal values for a,b,c and create new symbolic predicates until all paths of the program are tested.

```
void testMe(int a, int b, int c, int d){
if(a==b)
if(b==c)
if(c==d)
System.out.println("OK");
}
```

Fig. 2.21 Example method with four integer input values, and three conditions.

Below, we list some of the tools used for unit test generation with DSE.

2.4.5.1 MSeqGen

MSeqGen [186] is a technique that mines Java code bases and extracts sequences related to receiver or argument object types of a method under test. The proposed approach uses random testing and dynamic symbolic execution for test generation in order to achieve higher structural coverage.

MSeqGen for test generation accepts the code under test for which it identifies declared or used classes and interfaces. Next, it searches for possible methods of target class and constructs their control-flow graphs (based on which it extracts sequences of calls). In the end, MSeqGen generalizes extracted calls into skeletons by replacing constant values with symbolic values.

Figure 2.22 illustrates an example of method call sequence extraction for class `MyClass`, and the skeleton with symbolic values created from method calls. First, the tool will create method sequences using the class under test, and with the control-flow graph for that specific method it extracts sequences of the class and method. Furthermore, using dynamic symbolic execution approach, MSeqGen will create the skeleton with symbolic values.

```
class MyClass {
 private int testMe;
 private String ipAddr;

 public void Mut1(MyClass mc, String IPAddress) {
  if(mc.getTestMe() > 100) {
   if(IsAValidIPAddress(IPAddress)) { ... }
  }
 }
}

 Method-call sequence (MCS)
MyClass mcObj = new MyClass();
mcObj.SetTestMe(10);
mcObj.SetIpAddr("127.0.0.1");

 Skeleton
int symvar = *, string ipaddr = *;
MyClass mcObj = new MyClass();
mcObj.SetTestMe(symvar);
mcObj.SetIpAddr(ipaddr);
```

Fig. 2.22 Class sample for method call sequence extraction and skeleton design from MSeq-Gen.

2.4.5.2 Pex

Pex [189] is a non-Java tool which uses dynamic symbolic execution to automatically generate unit test cases (for *.NET*). As a well known DSE testing tool Pex can be used for any other language that can be converted to the language compatible for Pex. In particular, it is used for OO programs where the approach identifies constructors and method calls, and use them to generate method call sequences with symbolic values. Pex using collected constraints generates concrete test values and performs a systematic program analysis using dynamic symbolic execution to determine test inputs.

In particular, a unit test generated with Pex is a parameterized unit test case (a method that takes parameters) containing sequences of method calls that bring the CUT to the desire state, and assertion statements that exercise code expected behavior. Figure 2.23 presents a unit test case generated with Pex. This generated test case has a custom attribute named *[PexMethod]* which is used to distinguish generated tests that can run automatically.

```
[PexMethod]
public void AddSpec(
 // data
 int capacity, object element) {
 // assumptions
 PexAssume.IsTrue(capacity >= 0);
 // method sequence
 ArrayList a = new ArrayList (capacity);
 a.Add(element );
 // assertions
 Assert.IsTrue(a[0] == element );
}
```

Fig. 2.23 Example test case generated with Pex [189].

2.4.5.3 Seeker

Seeker [187] is a test generation tool that combines static and dynamic analyzes and covers a large percentage of code under test. *Seeker* using (1) dynamic analyses generates method sequences and then using (2) static analyses searches for uncovered branches by statically generating sequences. However, as all statically generated sequences are not important, with (3) dynamic analyses it explores sequences that do not help produce object states and eliminates them.

```
00: BidirectionalGraph bidGraph;
01: Random random;
02: VertexAndEdgeProvider s0 = new VertexAndEdgeProvider();
03: Vertex s1 = new Vertex();
04: bidGraph = new BidirectionalGraph ((IVEProvider)s0, PexSafeHelpers.
    ByteToBoolean((byte)16));
05: bidirectionalGraph.AddVertex((IVertex)s1);
06: random = new Random();
07: RandomGraph.Graph((IEdgeMutableGraph)bidGraph, 0, 1, random, false);
```

Fig. 2.24 Example test input generated with Seeker [187].

In particular, *Seeker* achieve to generate tests with better code coverage compared to state-of-the-art tools of DSE and Random approach. Figure 2.24 presents a test sequence generated with Seeker that detects an infinite loop in class `QuickGraph`. However, in practice the number of paths in a program may be extremely large, specially if it contains loops and/or recursions, which creates the path explosion issue.

2.4.6 Search-Based Test Generation for Java

Usage of the genetic algorithm for unit test generation is more powerful than existing techniques (e.g, random testing, symbolic execution). Produced sequences of objects and method calls with respective input values, and assertions that check expected versus actual result exercise the CUT in the best possible way [190]. At the high level, generation of unit test cases using search-based technique starts by specifying a testing criterion (e.g., branch, statement, line coverage), candidate solutions, search space and the fitness function based on which candidates are evaluated. Therefore, starting from an initial candidate solution, the algorithm will search for the best candidate based on the fitness value assigned to each one.

Using this technique, a number of research and commercial tools have been developed that use SBST for unit test generation in different programming languages. Some of the research tools used for unit test generation in Java are:

2.4.6.1 eToc

eToc [190] is one of the first tools that use a genetic algorithm for unit test generation. The tool is divided in four main categories such as *branch instrumentation, chromosome forming* and *test generation and execution*. In particular, every class under test first is analyzed and transformed using a *branch instrumentor* that identifies the method calls and control dependencies, which then continues with chromosome former who generates chromosomes (inputs) based on method signature and transforms them using mutation. Next a test generator creates a test case for each chromosome, and in the end each test case (for each chromosome) is executed as JUnit test class.

In general, eToc test generation stats by generating a set of test cases for class under test while maximizing the branch coverage. However, assertions in the test are added manually each time a method call returns a value, or each time it throws an exception, or at the end of each test case, the final state of the object under test is examined. Figure 2.25 presents an eTOC unit test case. *m* is the method under test that belongs to class *A* (class under test) and contains two parameters where the second one is class *B*.

```
A a = new A();
B b = new B();
b.f(2);
a.m(5, b);
```

Fig. 2.25 Example test cases generated with eToc [190].

2.4.6.2 TestFul

TestFul [21] is a search-based unit test generation approach for Java classes. It uses branch and statement coverage as objective functions to put objects in useful states, that later are used from algorithms such as hill climbing and evolutionary algorithm to cover the uncovered parts of the class under test. Therefore, TestFul generated unit test cases tend to have high code coverage.

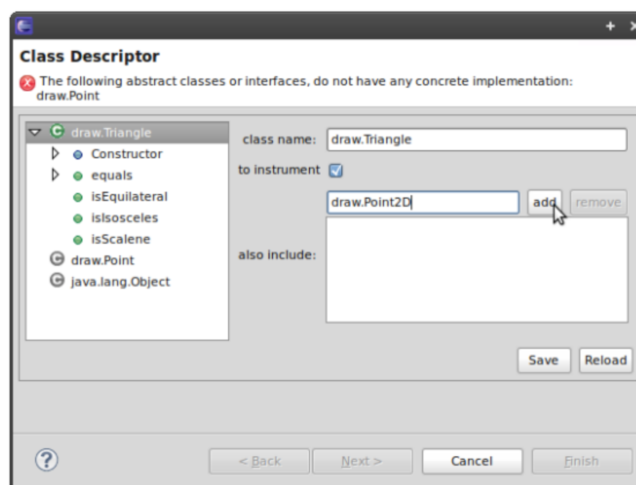


Fig. 2.26 TestFul GUI which list classes to be tested, and the option of creating instance for particular class.

TestFul contains an instrument that statistically analyses the class under test and measures the code coverage, a test generator which uses an evolutionary algorithm to generate instrumented code under test, and an Eclipse user interface to run test generation. Figure 2.26 shows the GUI of TestFul which lists the possible classes to be tested, and has the possibility that checks whether it can create an instance for each class under test. If all specified classes have their concrete implementations available then it gathers information about their behavior, and in the end, based on users given time constraint, tests are generated. Figure 2.27 illustrates automatically generated unit test cases with TestFul. Generated test cases can be reviewed manually by developer processing the test, or executed automatically with JUnit testing framework.

```
i_1 = 6;
i_3 = -1356361227;
p_3 = new draw.Point3D(i_1, i_1, i_3);
p3d_2 = new draw.Point3D();
assertEquals(1356361227.0, p3d_2.getDistance(p_3));
```

```
p_2 = null;
try {
 t_0 = new Triangle(p_1, p_2, p_3);
 fail("Expecting a java.lang.Exception");
}
catch (java.lang.Exception e) {
 assertEquals("The point cannot be null", e.getMessage());
}
```

Fig. 2.27 Unit test cases generated with TestFul [21].

2.4.6.3 Nighthawk

Nighthawk [14] is a test generation tool that applies genetic algorithm to derive method parameters for random test generation. With *Nighthawk* a unit test is a set of method calls that search for best method parameters that maximize code-coverage. In particular, the tool is made up of two levels, the lower level that initialize and maintain a pool of method parameter values according to a policy specified in the chromosome, and the upper level that search for best parameter values.

Nighthawk random test generation process starts the search over target method M (target method) chromosomes (inputs) and finds the fittest ones, which are further specified as test inputs. These inputs are the ones with higher code coverage satisfying the objective of the tool.

2.4.6.4 EvoSuite

EvoSuite [70] is another test generation tool for classes written in Java. It uses search-based approach to automatically generate test suites that achieve high code coverage, containing test cases with assertions. Furthermore, it uses several novel techniques that instead of covering individual coverage criteria, it evolves whole test suite with respect to entire coverage criteria [71], and minimize test cases [76]. Therefore, the test generation process of EvoSuite starts by producing a set of test cases, next it evolve using evolutionary search, and ends by producing the test suite containing best result and minimizing it by giving an optimal test suite.

The test suite generated with EVOSUITE is a set of test cases $T = \{t_1, t_2, ..., t_n\}$, and each test case is represented with a set of statements $t = \langle s_1, s_2, ..., s_n \rangle$ that execute the class under test and has its own coverage percentage. For example, for the given class in Figure 2.28, EVOSUITE will generate a test suite as shown in Figure 2.29. As it can be seen, statements in test cases can be of the different type such as primitive, constructor, method, or field statements. Therefore, for a given byte-code of a class under test, EVOSUITE generates test suite containing test cases with statements of different length, and each of generated unit tests have a unique covered branch, which in total achieves high code coverage.

EvoSuite as a mature research prototype was used to generate test cases for different Java project in different domains [69, 71], and in testing tool competitions it ranked as the most successful tool [73].

```
public class Counter {
 public Counter() {}
 private static int counter = 0;
 public static boolean setCounter(int value) throws IllegalStateException {
  if (counter > 0)
   throw new IllegalStateException("Only one call to the method is allowed");
  counter++;
  if (value < 0)
   return true;
  else
   return false;
 }
 public static int getCounter() {
  return counter;
 }
}
```

Fig. 2.28 `Counter` class exercised from EVOSUITE.

```
public void test0() throws Throwable {
 Counter.getCounter();
 Counter.setCounter('0');
 try {
  Counter.setCounter('0');
  fail("Expecting exception: IllegalStateException");

 } catch(IllegalStateException e) '{'
  //
  // Only one call to bar is allowed
  //
  assertThrownBy("com.examples.with.different.packagename.staticfield.Counter", e
      );
 }
}
```

```
public void test1() throws Throwable {
 Counter counter = new Counter();
 assertEquals(0, counter.getCounter());
}
```

```
public void test2() throws Throwable {
 boolean boolean0 = Counter.setCounter((-15));
 assertTrue(boolean0);
}
```

Fig. 2.29 Example test cases generated with EVOSUITE.

2.4.7 Lazy Systematic Unit Test Generation for Java

Lazy systematic unit testing tends to have an exhaustive testing process based on the structure of the test code, and information collected from developers performing the test. With lazy systematic unit testing, the specification of the class under test grows incrementally using automatic analysis and developer inputs (developers confirmation about outputs of automatically generated tests), moving the test to a bounded exhaustive test. Moreover, with lazy systematic testing generated test reports become detailed and shorter, which implies that overall test oracle cost will decrease, too.

2.4.7.1 JWalk

JWalk is a lazy-systematic unit test generation tool presented by Simons [177, 179, 181]. Using specification-based test generation algorithm, it verifies the detailed algebraic structure,

or high-level states and transitions of the code under test. In particular, JWalk takes as input the class under test and constructs incrementally a test suite oracle, by combining dynamic code analyses and interaction with user. Figure 2.30 presents JWalkTester and test cycles created during test generation. During test generation with JWalk, the human tester confirms or rejects certain key test results, which enable the process of test oracle construction and fully automatic testing process.

However, JWalk is an alternative to regression testing [175], and it is compared against JUnit [179, 178, 180] showing that JWalk is about one hundred times as fast as developing JUnit tests manually. Indeed, it automates test-creation as well as test-execution and catches method-interleaving cases that the developer usually forgets to test.

JWalkEditor is an editor developed around core JWalk, which enables a feedback-based development methodology [179]. Using JWalkEditor developers can prototype their code, and the tool offers them a growing associated specification. Hence, the tool first asks the programmer to confirm the key test cases, afterwards, it uses these confirmed results as automatic test oracles, unless a novel test outcome is observed. By interleaving methods in all possible combinations, JWalk tests corner cases that programmers often forget to test. Furthermore, if the programmer later makes changes to the tested code that invalidate the saved test oracle, JWalk will request confirmation for previously unseen behaviour in an incremental way. J-sensitive editor and compiler are part of the tool, too, and help programmers to prototype Java class design and generate test cases as they code. This possibility of JWalk (code-inferring-test) achieves to generate tests that are often missed from test-driven development approaches.

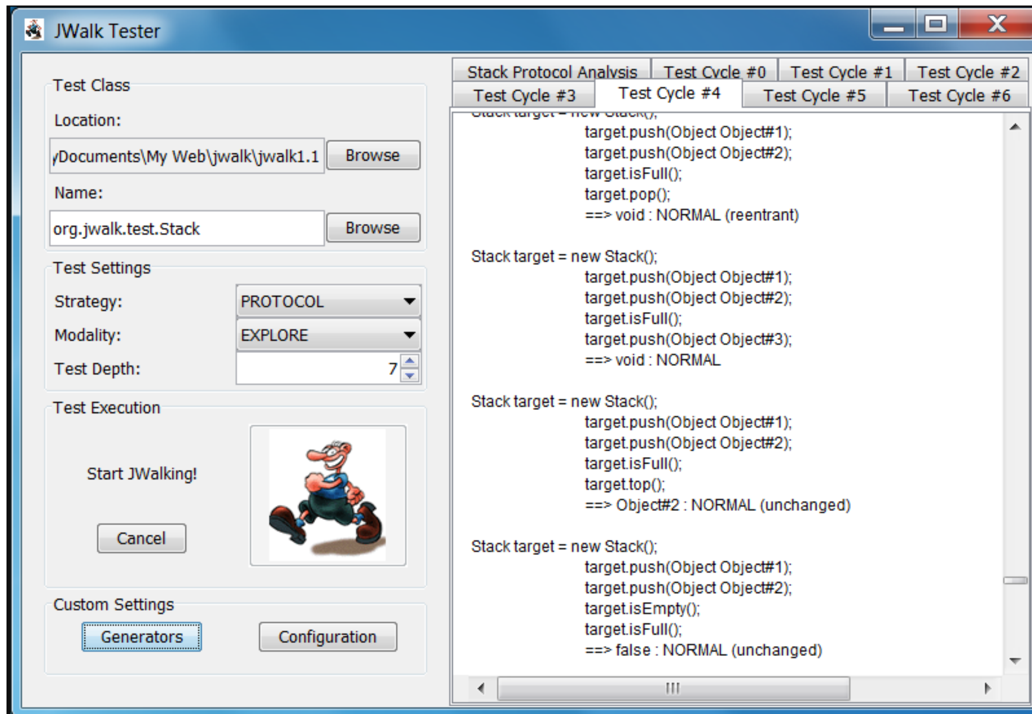


Fig. 2.30 JWalker Tester [177].

2.5 Software Evolution

Software changes are inevitable, and the main challenge of developers is to implement those changes in the code and further maintain the code. The evolution of software systems over time has been studied by a number of researchers. Eight laws of software evolution or Lehmans' Laws, identify a number of observations about how software evolves over time [114–116], such as:

- *Continuing change* means that software continuously have to adapt to changes in order to satisfy user needs.
- *Increasing complexity* means that as software changes it becomes more and more complex, and in order to reduce this complexity developers need to further work on it.
- *Self regulation* means that program evolution is self regulating process, or during software maintenance software growth rate is regulated [20].

- *Organization stability over system lifetime* means that software average growth rate in each system release is approximately constant.
- *Conservation of familiarity* means that over time the interest in a software system decreases, thus, its incremental and long-term growth rate tend to decline.
- *Continuing growth* presents the ability of system to continuously grow in order to satisfy user needs.
- *Declining quality* means that the quality of the software will always decrease if it is not adapted to changes .
- *Feedback system* means that in order to achieve significant improvements, the evolution process constitutes by a multi-loop, multi-level, multi-party feedback system.

2.5.1 Software Maintenance

Every little change made to the system after delivery is called software maintenance. In IEEE Standard 1219 [128] and later on ISO/IEC [97], software maintenance has been defined as a post-delivery activity to correct faults, to improve performance, or to adapt application to its new environment. In particular, it is a costly process and consumes from 60% to 80% of total development cost [48].

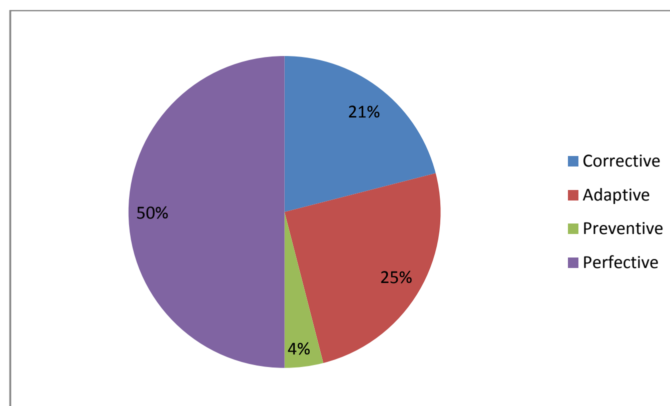


Fig. 2.31 Software maintenance categories, and their cost in total software maintenance process

In an early study of Lientz and Swanson [121], software maintenance has been divided in four categories such as:

- *adaptive*, that include modifications done in the software because of changes in running environment.
- *perfective*, that means modifications in the system because of new user requirements.
- *corrective*, which include changes initiated by the defects in the software.
- *preventive*, that covers changes in the system in order to prevent future bugs in application.

Figure 2.31 shows the distribution of each category and their cost in the total maintenance process. Indeed, perfective and adaptive changes are defined to cover the higher amount of total cost (Figure 2.31), which means that software needs changes because users require additional functionalities in the system, or because new bugs have been discovered which need corrections.

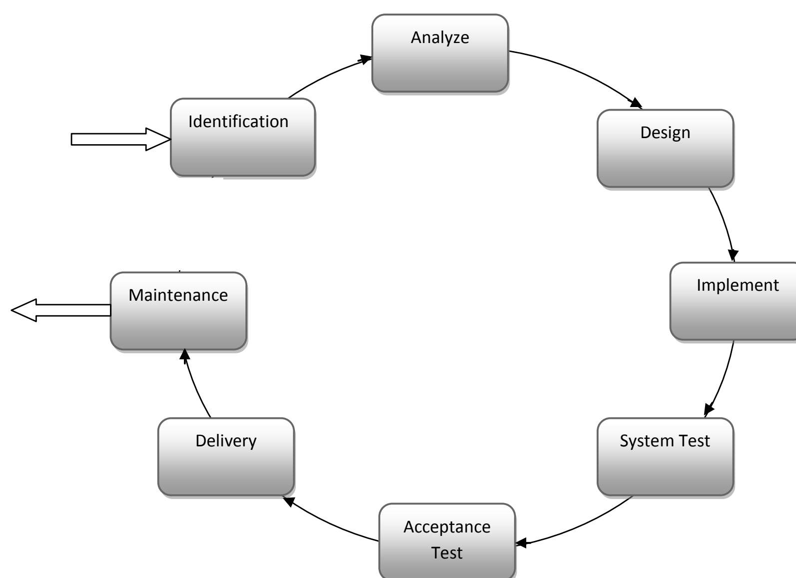


Fig. 2.32 Software maintenance activities [184].

Software maintenance is investigated by many researchers and there are different models such as Quick-Fix, Reuse-oriented model, Boehm's Model that define maintenance activities, which further are standardized on IEEE [184] that provides a framework for software maintenance activities as in Figure 2.32.

The first phase of software maintenance is the identification of requirements for modification. Ever change requested by customer needs to be identified and further analyzed

for their impact on the system. When changes are analyzed, then they are designed and implemented which further continues with testing. Testing is an important task in every stage of maintenance. In particular, every little change in the system needs to be tested in unit level, and further continue with the system and acceptance testing, and delivery to the user. The maintenance process is a continuous loop which does not stop unless the system becomes unused.

2.5.2 Test Case Evolution and Maintenance

During software evolution test cases evolve, too. Changes in the software imply the need for new tests and changes in the old ones. For new software features, new test cases are created, and for every little change, they are run periodically. Hence, test evolution means adaptation with new functionalities of evolving source code.

In order to identify the evolution of test cases and how much they affect the overall code coverage, Pinto et al. [152] presented TestEvol a tool which applies a systematic study of test evolution. TestEvol using two versions of a software and their test cases computes and classifies static and behavioral differences in a system. Thus, for each pair of a broken test case (unmodified test case when application changes), it compares the test outputs to identify the changes and computes their coverage information. TestEvol has been evaluated on several real-world applications showing useful results in how test suite changes during evolution. In particular, it was used to investigate questions such as types of changes in test suites in practice, how often tests require complex modifications, or why tests are deleted/added. Overall taken scenarios, authors categorized five types of method call modifications (call added, call deleted, parameter added, parameter deleted, and parameter modified), three type of test deletions (hard-to-fix test, obsolete test, redundant test), and three reasons why new tests are added (bug-fix test, new-features test, coverage-augmentation test). Also, they showed the usefulness of tool in practice for testers that want to understand their tests during maintenance, or researchers that deal with test repair.

Software maintenance cost is mainly spent on perfective and adaptive changes, or to the new user requirements and new bugs discovered [121]. Hence, this whole situation derives the fact that testing is a crucial part of software maintenance, as any new feature or any additional change needs testing.

Same as production code, test code needs to go through good programming practices [173], and be maintainable for future changes in the system. Indeed, Athanasiou et al. [17] showed that a maintainable test code should be of good quality. Test code quality

formula

$$TestCodeQuality = \sqrt[3]{Completeness * Effectiveness * Maintainability} \quad (2.1)$$

shows that, test code quality is related with code completeness, effectiveness, and maintainability, and for a high quality code all of three components should be with high values. Indeed, this means that a good quality test case is easy and cheaper to be maintained.

As software evolution is an expensive process in general, many researchers focus on reducing the cost of test maintenance by defining new techniques for regression testing and test case repair.

2.5.2.1 Regression Testing

As the software system evolves, developers have to perform regression testing and make sure that existing system functions still work. In particular, regression testing is done to verify that changes or new functionalities in the system did not introduce new bugs. As a process, it is estimated to take almost half of software maintenance time [26].

Yoo and Harman [204] present an extensive survey on regression testing, and cover its three main techniques: minimization, selection, and prioritization. This survey covers existing works on regression testing, the issues during regression testing, and how each of the three techniques serve as factor on decreasing the general cost. The next section describes regression testing techniques and the way they reduce total software testing cost.

2.5.2.1.1 Test Case Minimization Test minimization means eliminating redundant test cases in order to reduce the number of tests that needs to be run. Test case minimization approach was investigated and the tool TestTube [50] is one of the earlier approaches that identify which test cases need to be executed in order to verify the updated part of the program. The aim of TestTube is to eliminate test cases that cover unchanged entities, and run only those that cover modified parts of the program. The approach behind this tool is to firstly divide the program into subparts, and secondly monitor the execution of test cases by identifying which part of the program is covered by which test case. Therefore, when the program is changed, if there exist any test case that covers the modified part (entities) of the program, it is rerun while the others are ignored. This tool was evaluated using two real software programs, and it is seen that it can reduce almost 50% of overall test cases. Another work came from Binkley [31] who proposed two algorithms for reducing the cost of regression testing. Both algorithms use code semantic information to identify the difference between two versions of the program. The first algorithm, based on certified (early version)

and modified version of the program, extract differences that capture the changed part of the program. The second algorithm identifies test cases that exercise new changes in the modified version of the system. Therefore, both algorithms have been evaluated and successfully reduce cost of regression testing by reducing the number of test cases that need to be executed in order to verify the changes that have been introduced to the application.

2.5.2.1.2 Test Case Selection Test case selection attempts to reduce costs by selecting and running only a subset of test in the existing test suite that are relevant to the changes applied in the system.

A variety of test selection techniques have been described in the survey of Rothermel and Harrold [163]. Three most discussed techniques are minimization-based, data-flow based and safe selection techniques. Minimization-based (Linear Equation Techniques) regression testing is one described technique which tend to select minimal set of test cases that cover the modified part of the program. This means that, even that it might have many test cases covering the same part of program, it will select only one of them. Data-flow test case selection technique selects only test cases that exercise data that have been modified. For example, a test case that exercise a data interaction which is deleted from (added to, or modified from) the old version of the program need to be selected. Safe selection technique is another approach that in contrast to other selection techniques, selects all test cases that would have revealed a fault in the modified program. Hence, safe selection is a method that does not focus on any criteria, but select all test cases that produce different output with modified program.

2.5.2.1.3 Test Case Prioritization The last regression technique is test case prioritization. Test prioritization aims at ordering test cases with some specific criteria, such that tests with higher priority are executed first, and less important tests are left in the end. Rotheram et al. [164] proposed nine different techniques of test prioritization that improve fault detection rate and help developers on early fault correction. All of these techniques (no-prioritization, random, optimal, total branch coverage, additional branch coverage, total statement coverage, additional statement coverage, total fault-exposing-potential, additional fault-exposing-potential) either based on no/random order, prior knowledge of faults in program, branch/statement coverage, or by mutating the program and calculating mutation score for each test (ratio of mutants detected by a test), priorities the test cases and improve fault detection rate.

Test case prioritization techniques assists to regression testing in different ways. Wong et al. [198] proposed a hybrid approach that minimizes the set of test cases such that it satisfies

the objective function with respect to some coverage criteria, and second it orders tests using the modification based approach. This approach was implemented in a tool called ATAC which resulted in significant size reduction of regression testing.

Test maintenance may cause a large number of test failures [151], too. Therefore, when changes in the system cause failures in test cases, developers usually revise the application code, repair the broken test, or delete in order to make them pass.

2.5.2.2 Test Case Repair

Repairing test cases that do not compile due to changes in the system, or writing new tests for additional parts of the system is an expensive and time-consuming activity [59].

In order to assist developers on failing test cases during software maintenance, another tool called ReAssert has been proposed [59]. This novel technique suggests repairs to failing tests by fixing assertions and making them pass. It performs a combined dynamic and static analysis to find repairs that developers are likely to accept, and was the first known technique to help developers on test repair for evolving software. However, as a technique it cannot repair more than 45% of failures in open-source applications, and usually suggests naive or sub-optimal solutions [60]. ReAssert was further improved to "symbolic test repair"- a technique that uses symbolic execution to generate concrete values for repaired tests. This new technique improved the old one by increasing the percentage of fixed bugs and providing better fixes. However, same as the old version of ReAssert, this approach is focused only on repairing test assertions, without considering the possibility of new method calls that may repair the test and not change the assertions at all. Mirzaaghaei et al. [137] proposed a different test case repair approach that can automatically evolve test suites. It achieves to repair old test cases that are broken due to method signatures (when method parameters are added, deleted or modified), and generates new ones as result of changes in the software system. In particular, it seems to fix test cases only if they have changes in their method parameters, however, it does not provide any fix for new or deleted methods in the program. Thus, the proposed technique still is limited at providing appropriate test repairs.

Mentioned works so far (e.g., [59, 60, 137]) assume that test failures are due to obsolete tests. However, Hao et al. [92] use machine learning to predict whether a test failure is due to a code or a test problem. This classification model with features divided into three categories (complexity, change, and testing), was trained with data collected after analyzing software and its source code. As result, this approach was found effective at correctly classifying the causes of test failures when it is applied within the same program.

A different technique that also supports failing test cases, was proposed by Zhang et al. [208]. FailureDoc automated technique documents test (in form of comment) by indicating

the fixes that will make a test pass, and make it easier for developers to understand which part of the code is causing the test failure. It contains four phases, such as value replacement in the test from the created object pool; execution observation which executes every mutated test, prunes irrelevant statements, and selectively observes their outcomes; failure correlation which uses statical algorithm to correlate replaced values with their corresponding outcomes; and documentation generation which summarizes properties of the observed failing objects, and translates them to explanatory code comments. FailureDoc has been evaluated on five real projects and the generated comments are found relevant and useful in bug diagnoses.

Despite the fact that previously proposed techniques (e.g., [152, 137, 60, 59, 88, 208]) help developers and testers to maintain test cases, test maintenance is still a time consuming task [60], and the cost increases even more if test cases are generated automatically [158, 101, 123]. Indeed, automatically generated tests aim on high code coverage with statements (method calls and assertions) that maximize it, without considering test comprehension. Also, approaches for automatically generating test cases do not take into the consideration human effort in manually checking the inputs produced [5], and one of the features that increases more this effort is readability of the test. In the next section, we will cover readability in general, and the work is done for readability optimization in natural language and software source code.

2.6 Readability in Natural Language Processing

The main objective of this thesis is to investigate test code readability, and how it can be improved. Readability is an important principle in almost every practice, including any text that people read. In order to assign a readability level to the text, there were implemented formulas that express the relationship between difficulty that people have on reading a certain text and a measure of the linguistic characteristics of that text [130].

Early investigations into text readability can be traced back to the 1940s. More than 40 formulas that calculate text readability have been created. *"Cloze procedure"* [58] is one of the earliest tools that measure the readability of a text. *SMOG* [129] is another formula that calculates text readability and combines the number of syllables or words in the text with the number of sentences. However, the most used readability calculators are *Flesch Reading Ease* and *Flesch-Kincaid Grade Level* [86].

The Flesch Reading Ease Score

The Flesch Reading Ease Score measures the readability of a text based on the average sentence length and average word length (Formula 2.2). The results of this calculator range

from 0 to 100, where text with a score less than 30 is considered very difficult, and text with a score greater than 90 is considered very easy [32].

$$FleschReadingEase = 206.835 - 1.015 * (avgSentenceLength) - 84.6 * (avgWordLength) \tag{2.2}$$

The Flesch Reading Ease Score scale was used from many word processing programs, such as Microsoft Word, WordPerfect, WordPro. However, instead U.S. grade level used the Flesch-Kincaid Grade Level which was easier to be interpreted.

The Flesch-Kincaid Grade Level

The Flesch-Kincaid Grade Level similarly to the previous readability calculator, uses average sentence and average word length with different weightings (Formula 2.3). Although the result of both formulas correlates, Flesch-Kincaid Grade Level was developed under contract to the U.S. Navy to indicate how difficult a passage in English is to understand.

$$Flesch - KincaidGrade = 0.39 * (avgSentenceLength) + 11.8 * (avgWordLength) - 15.59 \tag{2.3}$$

Flesch's formulas of readability have been criticized by many researches, as it was not taking into account other important factors of the sentence such as cohesion and coherence, or sentence structure. The next readability assessment tool that takes into consideration cohesion and coherence is Coh-Metrix.

Coh-Metrix Coh-Metrix [1] is a program that computes cohesion and coherence of the written and spoken text [86]. It is offered as a tool that evaluates a text using lexical, syntactic and semantic features for test readability evaluation. This web-based software tool (Figure 2.33) analyses texts on over 50 types of cohesion relation and over 200 measures of languages text and readability level. Scarton [172] has further investigated on Coh-Metrix and created Coh-Merix Port a readability assessment tool for text evaluation. This tool performs on Portuguese language using 41 psycholinguistic metrics. However, as including such factors does not necessarily produce better readability prediction [27], Flesch's formulas are still being used for more than 40 years.

Another text readability assessment approach is introduced from Aluisio [8]. This approach is able to compute the text complexity level using existing and new readability assessment feature. The author presence this approach as an improved version of already

[1] http://www.cohmetrix.com

Fig. 2.33 Coh-Metrix web tool that calculates text cohesion and coherence [86].

existing readability calculators, since it uses a feature set that better explains the complexity of the text, targets new audience (people with different literacy level), and investigates different statistical models for non-linear data scales.

Readability in NLP is highly investigated, and beside text readability calculators, there are other tools that help writing readable text, too. Sato et al. [171] proposed an approach that has the possibility of revising text that is not readable and shows a possible solution with the same meaning. The tool is implemented for the Japanese language in four different levels, ranging from elementary to advanced level of the language.

2.7 Source Code Readability

Source code maintenance and testability are the main factors of software quality [67], and readability is seen as a crucial factor that needs to be improved for qualified software [185]. One of the most time-consuming activities during software maintenance is reading code [64, 159, 165]. Less time required to read a code leads to less effort spent on each phase of the software life-cycle, which means that code readability affects the overall software cost [51].

Lately, models of code readability have been proposed that can help improve the readability of source code.

2.7.0.1 Buse and Weimer Readability Model

Buse and Weimer investigated readability in the source code and published a novel technique which can measure code readability level [38, 39].

Initially, in order to learn about the readability of source code in general and what developers think about it, authors collected 100 code snippets from different open-source projects and using an empirical study they collect human ratings based on their perception of readability for each snippet. Furthermore, they collected 25 readability features (as in Table 2.1) where each represents either the average value per line, or maximum value for all lines in any code snippet. 100 code snippets rated from users were further translated in vectors of 25 readability features and the last feature that represents the average human rate.

In particular, the main idea of the work was to train a machine learning model which can predict code readability as well as a human can. Therefore, the code readability model concept is: given a set of inputs $\{x_1, x_2, x_3, ...x_n\}$ and a set of outputs $\{y_1, y_2, y_3, ...y_n\}$, machine learns how to predict a new label sequence $y = h(x)$ given an input sequence x. This means that given code readability features (input x), and average human ratings (output y), model is trained to predict readability for unknown instances (code snippets). As result, the Buse and Weimer readability model was able to predict from 0 to 1 the readability level for any code, and can be 80 % more effective than a human (Figure 2.34).

```
for (int j = 0; j<dataset.length; j++){
   if(dataset[j]!=null){
      for(int k =0; k<dataset[j].length; k++){
         if(dataset[j][k]>outlier.get(k)){
            dataset[j]=null;
            break;
         }
      }
   }
 }

###
0.6606496572494507
```

Fig. 2.34 Source code readability model of Buse and Weimer.

Given all readability features as in Table 2.1, readability model was used to investigate feature importance, and found that ‘average line length’ and ‘average number of identifiers per line’ are very important to readability and negatively impact the readability score. While ’average blank lines’, and ’average comments’ positively correlate to readability score, meaning that their presence raise readability of code snippets, average blank lines are found to be more effective than comments in general.

In addition, authors found a significant correlation between code readability and software quality feature such as code changes, automated defect reports, and defect log messages. The same experiment was repeated in the extended version of the paper [38] with a broader base of benchmark programs with over 2 million lines of code, and for a second time reported that readability has a significant level of correlation with software quality metrics such as defects, code churn, and self-reported stability. Additionally, they measured code complexity (with McCabe's Cyclomatic measure [131]) and found no strong correlation with readability, while complexity shows to be more related to code length. In the end, they added a longitudinal experiment showing that projects with unchanging readability have equivalent unchanging defect densities, while projects with a sharp drop in readability had risen in defect density, too.

Feature name	average	maximum
line length	+	+
identifiers	+	+
identifier length	+	+
indentations	+	+
keywords	+	+
numbers	+	+
comments	+	
periods	+	
commas	+	
spaces	+	
parentheses	+	
arithmetic operators	+	
comparison operators	+	
assignments	+	
branches	+	
loops	+	
blank lines	+	
occurrences of any single character		+
occurrences of any single identifier		+

Table 2.1 Code Features of Buse and Weimer Readability Model

This code readability model, was one of the first to simply measure code readability level. However, after this initial work, Posnett et al. [154] presented a simpler technique of code readability based on code size and entropy.

2.7.0.2 Posnett Readability Model

The new simpler readability model is based on three features instead. The readability formula of this calculator is:

$$Readability = 8.87 - 0.033V + 0.40Lines - 1.5Entropy \tag{2.4}$$

where V is Halstead's Volume (quantitative measure of complexity from the operators and operands in the code [90]) , or, Lines is a number of code lines, and Entropy is the distribution of tokens/characters in the snippet. As result, it can be seen that for a given entropy level, if size increases it contributes positively to code readability.

Beside these two source code readability predictors, further published work has contributed to code readability in other ways. Fry [78] in his study checks the effect of code readability in the fault localization process. He conducted an empirical study, using 65 human annotators for fault localization in Java code samples. As a result, the author created a fault localization complexity model using readability features as part of this feature set.

As an important factor of software quality [170], the work of Sampaio and Barbosa [170] [170] introduces 33 good practices of software readability. Introduced practices are surveyed with a group of teachers, and found important to be taught in software development courses.

2.8 Test Code Readability

Lack of test readability does not affect the functionality of the program at all, however, it increases the cost of software maintenance by making it difficult to understand the code under test while fixing it [75]. Test case comprehension has been investigated and a certain body of published work has been devoted to improving test case comprehension in order to support software maintenance.

2.8.1 Test Smell Detection

Martin Fowler [68] defines code smell as something that is quick to spot, and does not always indicate a problem in code. Similar to code smells, test smells represent a set of poor

design solutions in the test case [193]. Van Deursen et al. [193] introduced 11 test smells that are usually seen in existing tests and defines six refactoring types for making tests more understandable without deleting them. As mentioned in the paper, most of the test cases appear to have one or some test smells often due to the lack of time during testing, or wrong programming techniques. Bad test smells are spread in a large part of the system, and it has a strong negative impact on maintenance [23]. As they decrease the quality of the software, several approaches are proposed to remove/decrease them [193, 192, 134].

One of the techniques that can detect test fixture smells and guide test code refactoring is proposed from Greiler et al. [87]. They introduced five new test fixture smells together with General Fixture Smell (Test Maverick, Dead Fields, Lack of Cohesion of Test Methods, Obscure In-Line Setup, Vague Header Setup), and implemented a tool called TestHound that can analyze and detect any of the six smells. TestHound is evaluated with three industrial software systems showing that test fixture smells are available in existing software, and using 13 developers the tool is found to be useful for fixture management. However, as developer would benefit more from a tool that assists as system evolve and identifies test smells immediately and learns which software changes lead to test smell increases, they proposed *TestEvoHound* [88] an improved version of *TestHound*. *TestEvoHound* analyzes fixture related test smells of multiple versions of software, and tracks test evolution strategy. Based on this technique, a class containing more methods will have more test-fixture smells. After evaluating the tool in five open-source projects, the authors came up with strategies and guidance of how to avoid test fixture smells.

2.8.2 Test Case Minimization

Test comprehension and the possibilities of improving it were earlier investigated, and one of the techniques proposed was test minimization. The idea of test minimization is to reduce the size of large test cases in order to make it easier for the reader to understand while keeping the intent of the test. One of the test minimization techniques proposed is *Delta Debugging* [205]. The Delta Debugging minimization (ddmin) approach was applied to failing test cases, where statements of the test are deleted until the failure is no longer detected. The main advantage of the technique was to simplify the test for developers and isolates the failure causing part of the program. Zeller and Hildebrandt [205] proposed an automated testing technique with Delta Debugging. This means that during testing, each time that a test fails, Delta Debugging can be used to isolate and simplify the circumstances of failure. Furthermore, the author showed that Delta Debugging can be applied in a number of settings, such as finding failure causing parts in the program invocation, in the program input, or in the sequence of user interactions.

The next syntax-based test minimization technique is *program-slicing* [195]. Program slicing is based on control/data dependency information that decomposes the program into smaller parts that are related with each other. Zhang et al. [209] in his paper using empirical evaluation compared three different slicing algorithms (full, data, and relevant) for detecting faulty statements. To show this, authors implemented a tool that uses dynamic instrumentation and reduced ordered Binary Decision Diagrams. After comparing the full slices (based on dynamic data dependency, it identifies statements that influence the computation of faulty output value), data slices, (based on dynamic data and/or control dependency, it identifies statements that influence the computation of faulty output value), and relevant slices (or algorithm that additionally contains predicates to identify faulty statements), the results of the paper show that data slices are smaller than full and relevant slices, but can capture fewer faults. Full slices in the other hand, are bigger than data slices and can cover more faults. However, relevant slices have shown the best performance capturing all the faulty statements and only slightly larger than full slices.

Test minimization continued to be investigated, and Lei et al. [117] proposed a novel approach for minimizing randomly generated and failing unit test cases. In particular, in this work authors applied Delta Debugging of Zeller and Hildebrandt [205] which divides tests into separated files and deletes statements until the failure is no longer visible. The main idea of this approach is that, a failing test case with granularity level (level of details in test) *n* is minimized following three main steps; Reduce to subset, is the first step that divides the failing test into n tests (starting from n=2) and if any of the tests fails it will be the new staring point going into the same loop; Reduce to complement, is the second step which separates tests into n test cases and forms their complements. Each complemented test with a deleted statement will be run and if any fails than it will be the new starting point; Increase granularity, or the last step will multiply by 2 the granularity level n decided in the beginning, and if it is greater than the number of lines in the test than is changed to the number of lines in test. In this way, the three strategies are tried repeatedly until the test is locally minimized. The empirical evaluation done on this minimization strategy shows that it can significantly reduce test sizes allowing developers to debug much smaller test cases. Another test minimization approach for failing randomly generated test cases was proposed by Leitner et al. [118]. This technique in contrast to the work of Lei et al. [117], combines delta debugging with static slicing and minimize test cases faster. The main idea of this approach was first to use static analyzes (backward data dependency analyzes) and than minimize the test. However, as *ddmin* minimized tests appear to be slightly smaller than tests minimized with static slicing, authors added *ddmin* to their basic approach and gain another

1% minimization. Indeed, the new technique reduces failing test size is found to be 11 times faster than the previous one, while showing similar results.

As both of the mentioned techniques (*ddmin and slicing*) are based on program syntactical information, Zhang [207] proposed a novel approach called Simple test that, based on program semantic information, minimizes the test cases and reduces developers comprehension effort. SimpleTest starts from an initial test and following the dependency list replaces the referred expressions with alternative ones, and each time constructs a simpler test as long as the same code is still covered. Figure 2.35 shows a test case which goes through SimpleTest simplification technique, while Figure 2.36 shows the minimized test. Indeed, the simplification starts following the dependency list, and in the end, lines 2-6 are deleted resulting in a minimized and simpler test case. Compared to existing approaches, SimpleTest results are usually shorter test cases within a moderate time (slower than code slicing and faster than Delta Debugging).

```
public void test1() {
 Object var1 = new Object();
 Integer var2 = 0;
 Integer var3 = 1;
 Object[] var4 = new Object [] {var1, var2, var3};
 List var5 = Arrays.asList(var4);
 List var6 = var5.subList(var2, var3);
 TreeSet var7 = new TreeSet();
 boolean var8 = var7.add(var6);
 Set var9 = Collections.synchronizedSet(var7);
 //This assertion fails
 assertTrue(var9.equals(var9));
}
```

Fig. 2.35 An automatically generated test case.

```
public void test1() {
 Object var1 = new Object();
 TreeSet var7 = new TreeSet();
 boolean var8 = var7.add(var1);
 Set var9 = Collections.synchronizedSet(var7);
 //This assertion fails
 assertTrue(var9.equals(var9));
}
```

Fig. 2.36 Test case simplified with SimpleTest algorithm.

2.8.3 Test Readability Improvements

Different approaches have been proposed for optimizing test cases in terms of readability. For example, Robinson et al. [160] presented six enhancements (Remove Non-deterministic Observations, Prevent Changes to the Underlying System, Modify Inappropriate Calls, Observe Pure Methods, Filter Lexically Redundant Tests, Use Source Code Literals as Arguments) applied to Randoop and feedback directed test generation, in order to create effective and maintainable regression tests. This new approach was evaluated with large industrial software programs resulting in maintainable test cases with code coverage and mutation score better than manually written tests. Another optimization technique came from Fraser and Zeller [77]. Their approach that generates unit test cases based on mutation analyses (rather than coverage criteria), can reduce assertions in the test while affecting the overall test length. In particular, the technique presented by comparing executions of a test case on a program and its mutants was able to reduce assertions that are not able to distinguish between a program and its mutants. This approach is evaluated with well-tested open source programs, and their manual test cases are taken into consideration for comparison. Thus, even though manual tests are shorter, the average number of assertions is less in automatically generated tests. Similarly, in order to reduce the test length and improve fault localization, Xuan et al. [203] proposed a methodology called test case purification. This technique contains three main stages, such as test case atomization, test case slicing, and rank refinement. In the first phase, for each failing test a set of single-assertion test cases is generated. Next, test case slicing will remove statements that are unrelated to that assertion for each test case. And in the end, rank refinement will combine single assertion test cases with an existing fault localization technique and sorts their statements. This test case purification technique was evaluated on six open-source projects with 1800 seeded bugs and achieves better fault localization on 18 to 43% of faults, and worse on 1.3 to 2.4% of faults. However, in terms of fault localization (test purification on Tarantula), it obtains the best results among the evaluated techniques.

Recently, the work of Palomba et al. [148] considers code quality as one of the objectives during automatic test generation. As poorly designed tests have a negative impact on test maintenance, this new approach considers code cohesion and test coupling and created a metric that can measure how much maintainable are test cases. They try to maximize cohesion based on the insight that test should be simple and not verify too many functionalities, and minimize coupling by looking at the textual similarity between tests, such that only a few tests are affected by any future change. Furthermore, the technique is evaluated using open-source projects, the EVOSUITE test generation tool, and the MOSA multi-objective technique,

concluding in positive results with shorter test cases and higher code coverage compared to tools default configuration.

Another approach for improving the readability of string test inputs was developed by Afshan, McMinn and Stevenson [5]. This work includes a natural language model into a search-based test generation process, in order to improve string inputs from the perspective of human readability. Using an empirical study they evaluated their model which showed a great improvement in input readability. The developed technique improves readability of string inputs without weakening the test criteria, however, given a certain testing time, readable string inputs reduce oracle checking time and increase the number of test cases that may be considered.

All the lister papers above can minimize test cases in order to localize the failure, or make easier the maintenance of the program, and contribute to test readability by making the tests shorter and easier to read. However, in the next section, we will cover the works done on test naming improvements, that explicitly change the readability of the test by generating readable test names.

2.8.4 Test Naming Improvements

Descriptive test names help on test understanding and maintenance [28]. As finding descriptive names is difficult, there has been work on generating names automatically. Host and Ostvold [93] mine naming rules from a corpus of source code, and then suggest new names for methods that do not match these rules. More recently, Allamanis et al. [6] applied a log-bilinear neural network to learn a model that can suggest test names based on features extracted from the source code. When it comes to unit tests, however, many of the features that such approaches exploit (e.g., structure of the source code) are not available, as tests tend to be simple, short sequences of calls. In particular, for automatically generated tests other important features are also absent, such as descriptive variable names. Zhang et al. [206] proposed an approach to generate names for unit tests based on the common structure of tests and their names. In particular, for a given test they identify the action (e.g., the method under test), the scenario under test (e.g., the parameters and context of the action), and the expected outcome (e.g., the assertion). These three parts are then converted to text using a template-based approach, which includes only the action, the action and the expected outcome, or the action, the expected outcome and a summary of the scenario under test. Even if the question of which of the three pattern types should be used in practice is answered, automatically generated tests are problematic again: The description of the scenario under test relies on descriptive variable names, which generated tests generally do not have. The expected outcome is assumed to be a single test assertion, but generated tests often have

many assertions. Finally, since automatically generated tests are often generated for coverage, they may target more than one method under test, which also limits the technique above.

2.8.5 Test Case Documentation

Automatically generated test cases may be too long and not readable. Despite this, researchers invest in the problem of test summarization or documentation. Liu [123], proposed a platform-independent and tool-neutral comprehensive testing language called TestTalk. It can be automatically executed and derive test descriptions for automated test cases. TestTalk, which can be executed by different testing techniques, aims at reducing the overall cost of test maintenance. An alternative approach that helps to achieve the same came from Panichella et al. [149]. They present a tool called TestScribe, a novel approach that can generate summaries for test snippets. This tool that in particular describes the function of the test case is found to highly increase bug finding and comprehensibility of tests. Another similar idea was proposed from Li et al. [119]. Their new tool UnitTestScribe which can automatically generate documentation in unit test level, helps with test maintenance and detection of outdated tests. The proposed approach was better than the existing one, as by combining static analyses, natural language processing, backward slicing, and code summaries, achieves to generate the descriptions (using method under test, assertions, and data dependencies in unit test methods). UnitTestScribe is evaluated using human surveys with industrial developers and graduated students, who indicated that the proposed technique can generate complete, concise, and easy to read test summaries. Even though the mentioned techniques ([149], [119], [123]) have positive effect on test comprehension, one commonality that they have is that they can increase significantly the total maintenance cost if tests are unreadable.

Even though test summaries simplify test comprehension, they depend on the code that is generated, and automatically generated test case might be too long and unreadable. With this work, we aim to add the readability as an additional feature in the unit test case generation tool, and help developers to better understand the tests cases.

2.9 Conclusion

This Chapter explored the literature in the area of software testing, with a specific focus on automated software testing techniques like *Random Testing, Symbolic Execution, Dynamic Symbolic Execution, Meta-heuristics Technique, Search-based Software Testing*, and *Hybrid Test Approach*.

The chapter then investigates the research done on software maintenance. It summarizes the maintenance cost and the main factors that affect it. Furthermore, by dividing the process in *test evolution*, and *test maintenance*, it separately discusses the main drawbacks that test maintenance has in software life-cycle, and so far done research.

As the main focus of the thesis, the last part of this chapter discusses the literature on software readability and especially on test code readability. It summarizes the main issues that this area has, and possible actions are taken to improve it.

Chapter 3

A Survey of Unit Testing Practices and Problems

The content of this chapter is based on the conference paper previously published in the Software Reliability Engineering (ISSRE) conference in 2014 [54].

3.1 Introduction

The previous chapter discussed the importance of unit testing and surveyed several approaches for improving unit testing. Furthermore, there exist many published surveys that investigate how developers in the industry do unit testing [166, 191, 82, 81, 142, 113, 46]. However, while this body of work demonstrates that unit testing has found general acceptance in principle, in this chapter we describe a survey that we conducted online into how unit testing is practiced for real by testers, in order to focus on what they consider to be the main issues and what could be done to make unit testing easier.

Almost every programming language has its own unit testing framework (e.g., JUnit for Java, NUnit for C#), and unit testing has become an accepted practice, often even mandated by development processes (e.g., test-driven development). Nevertheless, software quality remains an issue. Software engineering researchers therefore argue that there is a need to push automation in testing further — not just to support the automated execution of saved suites of programmer-written tests, but also to generate these tests automatically.

Research has produced many different variations of automated unit test generation. A common assumption in software testing research is that simply providing the developer with a small set of efficient test cases will make the task of testing easier (e.g., [71]). However, it is unclear whether the need perceived by *researchers* meets the actual demands of *practitioners*.

This chapter summarizes the results of an online survey that was conducted to gain insight into common practice and needs in unit testing. The survey was performed using the global online marketing research platform *AYTM* (Ask Your Target Market, http://www.aytm.org). As we are not aware of a previous use of this source for surveys in software engineering research, we tried to establish the quality of responses, and how to best use this source of data for research.

RQ3.1: Can online marketing research platforms be used for software engineering surveys?

We queried responses from AYTM's global pool of 20 million respondents in several iterations, and in this chapter present the results of the final refinement of the survey and the qualified responses to it. Based on this data, we investigated the following research questions on unit testing (research questions of this chapter):

RQ3.2: What motivates developers to write unit tests?

RQ3.3: How is unit testing integrated into software development?

RQ3.4: How do developers write unit tests?

RQ3.5: How do developers use automated unit test generation?

RQ3.6: How could unit testing be improved?

Our experiences show that using a global online marketing research platform is easy in principle, but the specific requirements of a research-oriented software engineering survey show the limits of what is possible given the respondent pool and interface (though everything is possible if one is willing/able to pay the costs, which quickly explode). The results of our survey confirm that unit testing is commonly applied, and provide evidence that suggests there is potential for automated tools to improve unit testing. The survey responses point out areas of importance in unit testing, providing guidance for future research in the area, and on automated unit test generation in particular.

3.2 Survey Design

Before conducting the final survey, the underlying questionnaire and sampling approach went through several iterations. In this section we summarise the approach in detail.

3.2.1 The Questionnaire

We designed the questionnaire in four iterations: Initially, we set up an online survey system (LimeSurvey[1]) on our web site and created a survey related to RQ3–RQ6. We advertised this survey in social media (Google+, Facebook, Twitter, LinkedIn), Usenet news groups, developer forums, and mailing lists of industrial contacts, and promised participants a chance to win an Amazon voucher of GBP 20. Within a time frame of two months we received no more than 30 completed responses. From this response rate it became quite clear that with the channels we were exploring we would not achieve a satisfying response rate for any revised follow-up surveys.

Nevertheless, the 30 initial responses showed us deficiencies in our survey design, and allowed us to redesign the survey accordingly. To have a hope of a satisfying response rate, we considered different commercial marketing research platforms, and decided on AYTM[2] as their overall set of features and prices was most appealing. On November 30, 2013 we launched a revised survey on AYTM, and this time had 100 responses within 48 hours.

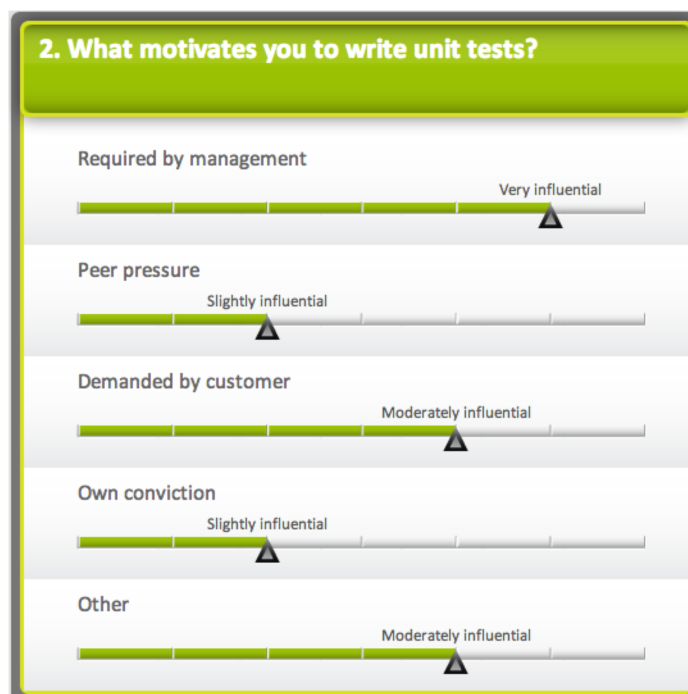


Fig. 3.1 Example rating questions on AYTM

[1] http://www.limesurvey.org/
[2] http://www.aytm.com

8. Please rank the following aspects of writing a new unit test according to their difficulty.

Drag up or down to reorder:

1. Identifying which code/scenario to test
2. Isolating the unit under test (e.g. handling databases, filesystem, dependencies)
3. Finding a sequence of calls to bring the unit under test into the target state
4. Determining what to check (i.e., finding good assertions)
5. Finding and creating relevant input values

Fig. 3.2 Example ranking questions on AYTM

3. How do you spend your software development time (in percentages)?

Please distribute remaining 1 between the items below

Writing new tests 58

Debugging/fixing 23

Writing new code 18

Refactoring 0

Other 0

Fig. 3.3 Example of distributive questions on AYTM

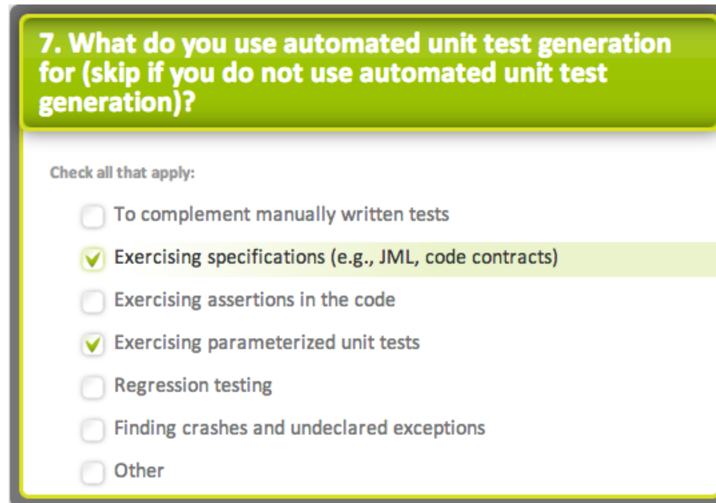


Fig. 3.4 Example of selecting questions on AYTM

Though satisfied with the response rate, some of the data was rather difficult to interpret or surprising. For example, when asked about which techniques developers applied, almost 50% of participants claimed to be using automated test generation. This is more than we expected, therefore we decided to drill down further into this aspect, revised the survey questions again and added an open text question asking respondents to list the automated test generation tools they used. We ran a revised survey including this question in two batches: The first 50 responses were obtained in the time from December 19 to December 20. After analysing the data in detail, we launched a second batch on January 9, 2014, where participants from the first batch were excluded. The second batch completed 150 responses on January 13, 2014. We received 153 responses to the open text question, and many of the tools listed were not actually test generation tools. There are 33 tools that we could not identify as testing tools, 42 tools that are related to testing (e.g., coverage analysis tools) but not to test generation. 23 tools are test generation tools; however, only 8 out of these are actually unit test generation tools. Interestingly, these are all tools from academia (Randoop, TestGen4J, GenUTest, Pathcrawler, PEX, Crest, JCrasher, Klover).

The quality of these responses put a question mark over all data retrieved via AYTM, and we thus designed a final iteration of the survey. In this final iteration, besides revising the questions, we added a qualification question (Q13) asking respondents to select all true statements out of a list of three questions about software testing, such that we can determine the quality of responses (two answers are expected to be true, the other one false). We intentionally asked this question last in the questionnaire, as we would expect some participants to be put off by an exam-like question at the beginning of the survey. The data from the revised version is the one on which this chapter is based.

Appendix 7.1 lists the survey questions used (together with the raw response data). This survey had four different types of questions: For rating questions, a 7-point Likert scale was used [122] with answer choices adapted to the questions (the questionnaire originally started with 5-point scale questions, but we increased the number of points to 7 in later revisions of the survey to increase discriminating power), for which AYTM provides an interface based on sliders (see Figure 3.1). Ranking questions are provided using a drag and drop interface in AYTM (see Figure 3.2). In the final revision of the survey we restricted the number of choices in ranking questions to five to increase chances of getting truthful responses. Distribution questions asked participants to distribute 100% on several available options. AYTM offers a very intuitive interface for this type of question which makes it easy and quick to answer (see Figure 3.3). There is one choice question, where respondents had to tick boxes of which options apply (see Figure 3.4).

AYTM obtains its responses from a large pool of participants (>20 million), and demographics for this pool are already known such that we did not have to ask standard demographic questions. However, we added questions about the years of professional programming experience (Q10), size of the typical software development team (Q11), and programming languages typically used (Q12). Thus, in total respondents had to answer 13 questions, which should take less than an estimated 15 minutes to complete.

We configured AYTM to ask the survey questions in the same order to all respondents, but the order of answer choices was randomised for each respondent (except for options such as "Other").

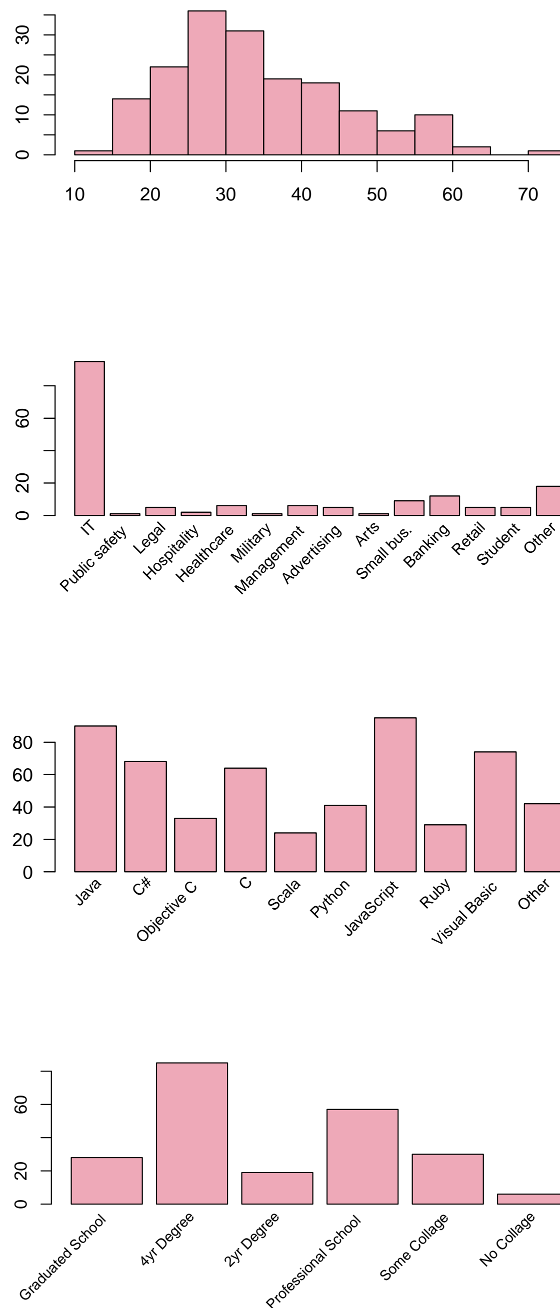


Fig. 3.5 Demographics of the survey respondents.

3.2.2 Selection of Participants

The AYTM platform claims to have a pool of over 20 million respondents globally. Once a survey is launched, it is made available to this pool and members can choose to participate

until the target response rate has been achieved. Respondents are paid a fee that depends on the number and type of questions. The fee to pay to AYTM also depends on the restrictions on the sample of the population, i.e., the more difficult it is to obtain the target sample the more expensive the survey is. For example, global responses are cheaper than US-only responses, and the more restrictions one puts on the sample, the further the price per response increases.

As AYTM has demographic information about its pool members, the target respondents can be filtered according to the available demographic information (e.g., gender, age, income, children, race, education, employment status, career, relationship status), and respondents can also be chosen from different pools geographically (World Wide, UK, US, etc.) We decided not to filter respondents by any of these criteria, as unit testing should be of interest to any software developer, regardless of these aspects.

To target a questionnaire to specific groups (e.g., software developers), AYTM offers the possibility to ask up to three pre-qualification questions, such that respondents who do not answer the pre-qualification questions as desired will be ejected from the survey. We initially used the default choice of "software developers" offered by AYTM, which results in the pre-qualification question, "Are you a software developer?". Only respondents who answered "yes" were allowed to complete the survey. One can add up to three further custom questions, and we considered doing this in the final iteration of the survey; however, every additional pre-qualification question leads to a significant increase of the cost per question (in our case this doubled the price per response!). Thus, in our case it was cheaper to include the pre-qualification question as part of the survey, and to filter the data *after* the survey was completed; this is what we did in our final iteration.

As our learning process on AYTM required several iterations of the survey, we hit the limits of the standard AYTM interface when trying to run repeated surveys. The standard interface offers only the possibility to exclude participants of *one* past survey. Thus, as soon as we wanted to post our third iteration of the survey, we were unable to select the two previous survey for participant exclusion. The AYTM support staff is very helpful and responsive in this respect, but asks for a 25% fee to exclude further surveys.

A more severe problem we hit with AYTM was that after running several iterations of our survey, we no longer were able to achieve the response rates we desired with further iterations, while excluding all past participants. Interaction with AYTM staff revealed that the number of software developers contained in the pool of 20 million users varies frequently, and is apparently small, only in the order of a few hundred at most. Once we had finalised our survey, we were thus unable to get the sample sizes we would have liked to, and therefore had to make do with 100 global responses. In addition, we were able to retrieve 125 responses

from the US pool (with the AYTM staff excluding overlap). For the final survey, AYTM charged a fee of US$3.85 per response for completing the survey from the global pool, and US$9.25 for per response for the US pool (plus exclusion fee). These two batches were launched on March 9, 2014, and both completed within 48 hours. For the analyses in this chapter, we combine the responses of these two pools.

3.2.3 Threats to Validity

We tried to follow general guidelines as well as possible [108]. However, there are, of course, several threats to the validity of our findings. First, our sample has to be seen as a convenience sample of respondents available in the global AYTM pool. We used a simple pre-qualification question and a more thorough qualification question to filter the data. However, it may be that a targeted survey in a particular industry or domain would lead to different responses. The survey may suffer from the self-selection principle, as participants in AYTM's pool chose to participate in the survey on their own. This may bias the results towards people that are more likely to answer such surveys, for example people with more extra spare time or trying to earn extra money through AYTM. It is possible that our sample size of 225 in total (171 for the analysis) is too small; for example, some of the differences between individual sub-groups might turn out to be statistically significant if we had a larger sample. However, the number of software engineers in the global pool of AYTM is limited, and we queried as many participants as possible given the available pool and our budget. Even though AYTM has a very convenient interface and accepts only surveys that can be swiftly answered, we still had 13 questions in total each with several answer choices. This may potentially reduce the reliability of responses. The inter-rater-agreement seems to be on the low side, but as filtering based on a qualification question did not increase it significantly, this may be an artefact of running a survey globally. Although we revised the questions several times and ran several pilot studies, it is still possible that they suffer from biased phrasing.

3.3 Survey Results

3.3.1 Demographics

Figure 3.5 and 3.6 show the demographics of our sample. The age ranges from 14 (the minimum age for AYTM respondents) to 72, with 37% of all respondents between 25 to 35. The majority of respondents work in IT, although there is a mix of respondents from other domains as well. The largest share of respondents comes from the USA (necessitated by our two-part sampling), followed by India. However, in general the sample achieves a nice

global spread with responses from Europa (32), North (143) and South America (13), Africa (1), Australia (1), and Asia (35). The majority of respondents have either a 4 year degree or a professional degree.

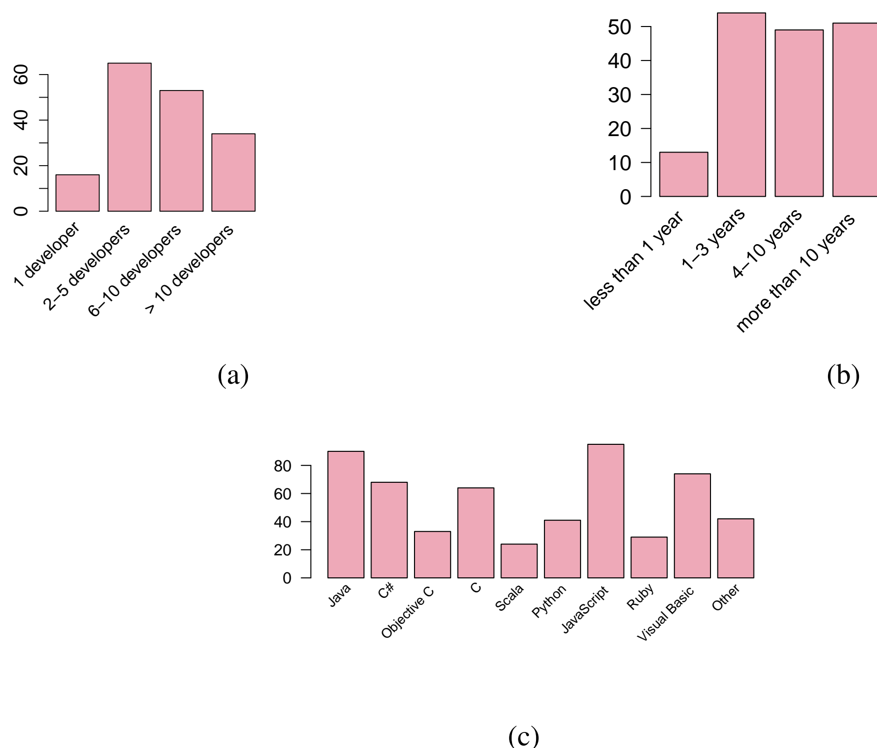


Fig. 3.6 Software development background of the survey respondents.

The programming languages most used are JavaScript and Java (Figure 3.6c), followed by C#, Objective C, Scala; thus, the majority of responses is based on programmers using object-oriented languages.

3.3.2 RQ3.1: Can Online Marketing Research Platforms Be Used for Software Engineering Surveys?

Table 3.1 Results on the qualification question.

	All respondents (%)	Education (%)						Experience (%)			
		4yr degree	Professional	2yr degree	Grad school	Some college	No college	>10 yr	4-10 yrs	1-3 yr	<1 yr
At least 1 question right	87.55	88.23	91.22	89.47	82.14	86.66	66.66	95.00	81.94	86.11	93.33
At least 2 questions right	76.00	76.47	70.17	73.68	82.14	83.33	66.66	85.00	68.06	75.00	86.66
All 3 questions right	21.77	25.88	24.56	0.00	17.85	26.66	0.00	31.66	25.00	13.88	13.33
1st question right	60.88	62.35	59.64	57.89	60.71	66.66	33.33	71.66	52.77	61.11	66.66
2nd question right	64.00	68.23	71.92	57.89	53.57	56.66	33.33	76.66	63.88	55.55	53.33
3rd question right	60.44	60.00	54.38	47.36	67.85	73.33	66.66	63.33	58.33	58.33	73.33

When conducting a survey, obtaining a good sample of responses from a representative population can be a challenging task. During our pilot studies, we did not receive many responses when publishing surveys in online discussion boards and social networking platforms, and so online marketing research platforms such as AYTM offer an attractive alternative. The large pool of respondents comes with demographic details one can use to specify the target population, and one has the ability to use qualification questions to further limit the respondents. The standard pre-qualification question when selecting software developers in AYTM is a simple question of "Are you a software developer?", and the question remains how good the quality of the resulting sample is.

We designed an additional qualification question that requires a basic level of understanding of what unit testing is (Q13). The aim of this qualification question is to target all software developers, good and bad. However, we would like to disregard potential respondents who try to cheat the system to collect payment from AYTM. For this, our assumption is that any software developer answering the questions truthfully should be able to classify at least two out of the three statements correctly.

Table 3.1 summarises the percentages of respondents that answered correctly: each question, at least one or two questions right, and respondents that correctly answered all questions. After analysing results about which questions are mostly answered, we divided respondents in two main groups, based on their education, and experience.

Less than a quarter of the respondents (22%) classified all three statements correctly, which is lower than expected. However, 76% had at least two statements correct, so we assume that more than 2/3 of the responses we received on AYTM are from actual software developers who truthfully answered the questions. Education and experience has only a slight positive influence on the number of correct answers. Consequently, it seems that AYTM may not be the best source for highly skilled responses, but one can still largely expect honest responses.

There are two options provided by AYTM to improve the data sample: First, our qualification question could be asked as a pre-qualification question, such that respondents who do not answer it correctly are ejected from the survey. Second, as we learned from interactions with AYTM after conducting the survey, when using open text questions one has the possibility to reject (a certain percentage of the) answers that are obviously of low quality.

To analyse the quality of the responses, we used Kendall's coefficient of concordance to analyse the inter-rater agreement for each of the responses. Kendall's coefficient was chosen over Spearman's correlation coefficient, because of its smaller gross error sensitivity (GES), and smaller asymptotic variance (AV) [52]. The details for each question are listed in the appendix; Figure 3.7 summarises the agreement for sub-groups based on the number of

correct responses. The agreement within all respondents who got at least one question right is very low, as is the agreement for those who got all three answers right. In comparison, the responses retrieved from the group of people who got at least two questions right is marginally higher. Based on our intuition that two correct responses would be acceptable and the higher agreement within that group, from now on we will only look at the data of this group of respondents, i.e., the data of 171 out of 225 respondents.

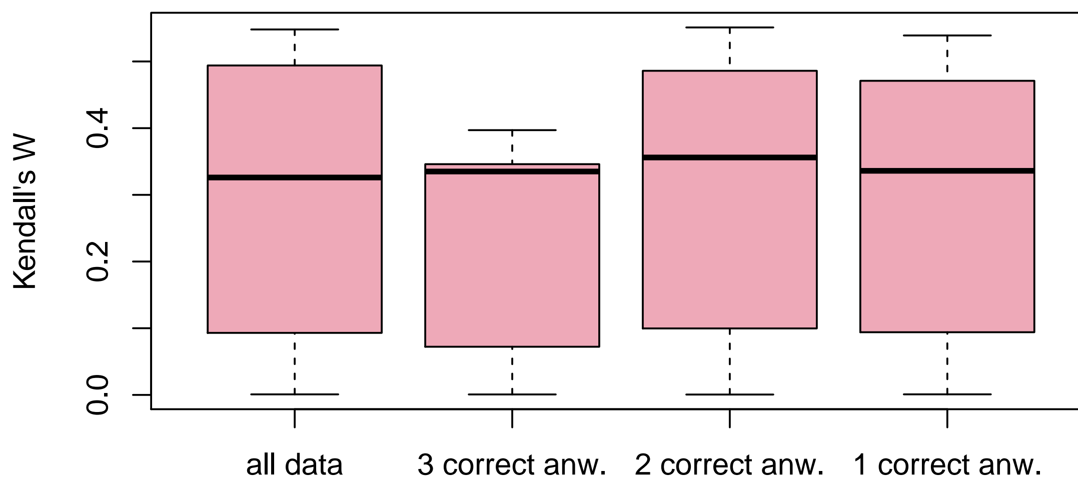


Fig. 3.7 Inter-rater agreement over all questions, calculated using Kendall's coefficient of concordance, ranging from 1 (complete agreement) to 0 (no agreement).

3.3.3 RQ3.2: What Motivates Developers to Write Unit Tests?

To better understand unit testing practice we would first like to understand *why* developers do unit testing. The first survey question (Q1) lists five options, each as a 7-point Likert scale rating question ranging from "very influential" to "not at all influential". Figure 3.8 summarises the responses to this question.

The main reason for unit testing is own conviction — developers test because they believe that testing is helpful. However, the requirement by management is listed as almost as influential as the own conviction. This is interesting as it is commonly said that management will happily cut down testing time in order for a product to be delivered on time. However, given this response it is reasonable to assume that unit testing is a standard practice. Customer

demand is listed as quite important, too, though clearly lower than the main two reasons. Likely in many cases customers will not know enough about software development to make any demands on unit testing. Peer pressure is rated comparatively low; maybe being test infected (a developer for whom creating tests, as first-class artifacts, is a priority [9]) is not infectious after all. In fact, the "other" response got higher agreement than peer pressure, suggesting that there are reasons we did not think of.

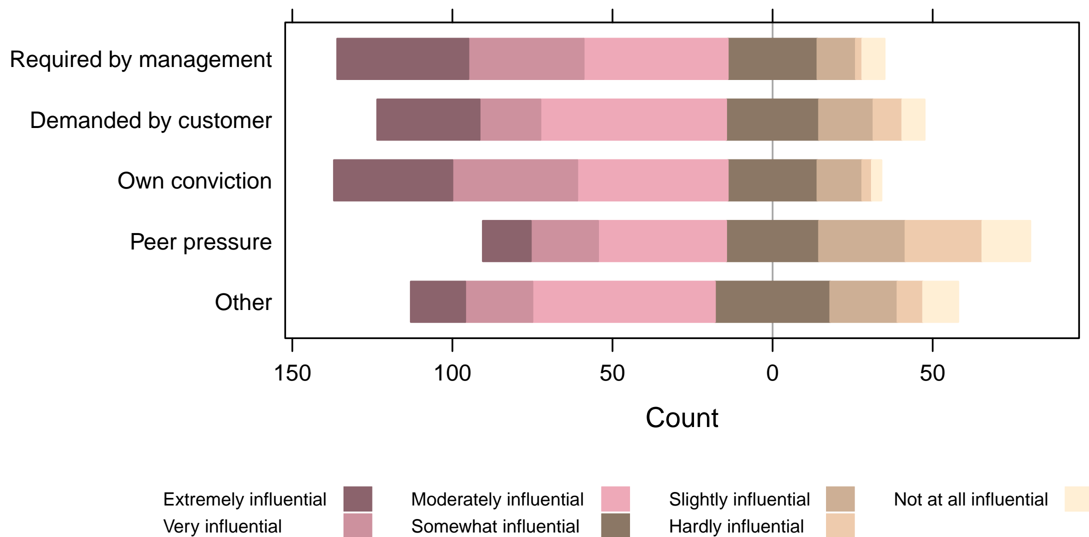


Fig. 3.8 What motivates developers to write unit tests

3.3.4 RQ3.3: How Is Unit Testing Integrated into Software Development?

How much time do developers believe to spend on unit testing, and what are the dominant activities when working with unit tests?

To answer this, we posed two questions to the survey participants.

3.3.4.1 Time Spent on Writing Unit Tests

Figure 3.9 shows the time that programmers estimate to spend on different coding activities (Q2) as a violin plot. As expected, the largest chunk of time (33.04% on average) is spent on writing new code. The second largest chunk of time is debugging (25.32%), followed by fixing (17.4% on average), and writing tests (15.8%). In total, developers estimate that they spend 66.96% of their time doing things other than coding, which seems realistic.

Although out of the named activities, the least time is perceived to be spent on writing tests, the time claimed to be spent on writing new tests is still quite substantial compared to the time spent on writing new code: almost half as much time as for writing new code is spent on writing new tests. This reminds of the often cited estimate that 50% of the software development time goes to testing. In terms of writing code and tests, this may not be true, and one may speculate that spending more time on writing tests would be necessary. On the other hand, the broader term "testing" will include the treatment of failing tests, i.e., debugging and fixing, and in that sense developers perceive to be spending almost two thirds of their time on activities related to testing. In that sense, expecting more time being spent on testing is maybe not realistic, but there clearly is potential for unit testing research to help developers produce *better* tests that make debugging and fixing easier.

Notably there are some outliers in the plot for time spent on writing new tests; for example, there are respondents who spend much more time on testing. These might include developers who are "test infected", i.e., who got hooked to the concept of writing tests. Interestingly, there are 21 respondents who declare to do *no* testing at all — 12% of all (qualified) respondents!

We investigated whether there are any trends when comparing different groups of developers, and checked statistical differences between groups by calculating the median value and the 95% confidence interval with bootstrapping. If the confidence intervals of two group do not overlap, then the difference between these groups is statistically significant. In terms of programming experience there seems to be a slight trend that with increasing experience less time is spent on writing tests, but there are no significant differences. However, it is reasonable to assume that with increasing experience developers become more efficient at writing their tests — or more sloppy. There are no differences in terms of the programming languages used; however, respondents were allowed to select more than one language. Considering the team size, there seems to be a slight trend that more time is spent on writing tests in larger teams, but again our data is not sufficient to provide significant results on this.

3.3.4.2 Handling of Failing Unit Tests

When using unit testing, then the time spent on debugging and fixing (25.32%+17.40% of the development time on average) will be heavily influenced by the tests — every failing test requires inspection and corrective actions. To investigate this, we asked respondents to estimate how often they react in different ways to a failing unit test (Q3), again in terms of percentages. Figure 3.10 summarises the responses: In almost half of the cases (47.2% on average), developers think that unit tests fulfilled their purpose and detected an error that requires code fixing. However, in 52.8% of the cases they believe that is not the case, and

either have to fix the tests, or do not treat the failure at all (either by deleting or by ignoring the test). This is plausible — during development, functionality can change, and tests need updating. This is, however, an important point for automated unit test generation: Automated test generation research is typically driven and evaluated in terms of how much code and what level of code complexity can be covered. However, depending on the usage scenario, unit testing tools are not used as fire-and-forget tools — they produce unit tests that need to be maintained. For this, different aspects such as readability and robustness against code changes are very important.

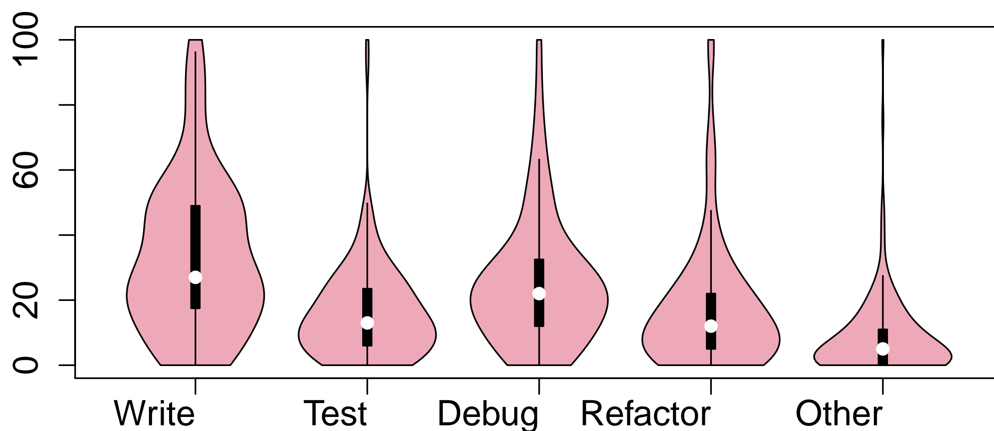


Fig. 3.9 How do developers spend their development time? The white dot in the violin plot is the median; the black box represents the interquartile range, and the pink region represents the probability density, ranging from min to max. Although writing new code is the dominating task, developers perceive to be spending more time on testing and debugging than on writing new code.

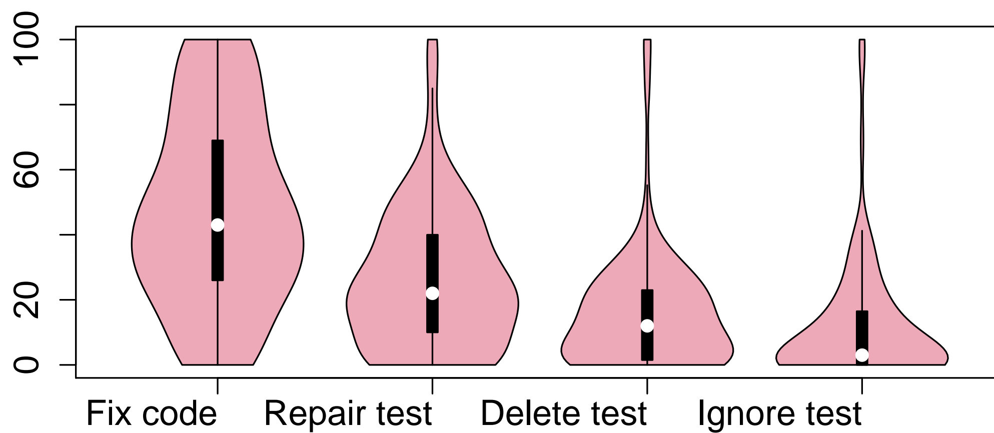


Fig. 3.10 What do developers usually do when a test fails? In about half the cases, tests fulfil their purpose and indicate a problem in the code that needs fixing.

Again the results are quite homogeneous across groups. Developers with more than 10 years experience claim to fix code more often than the others, and claim to ignore or delete failing tests less often than other groups. On the other hand, inexperienced developers claim to spend more time on repairing tests than other experience groups. However, most of these results are not significant. Notably, less experienced and developers with 1-3 years of experience delete tests significantly more often than developers with more than 10 years of experience.

3.3.5 RQ3.4: How Do Developers Write Unit Tests?

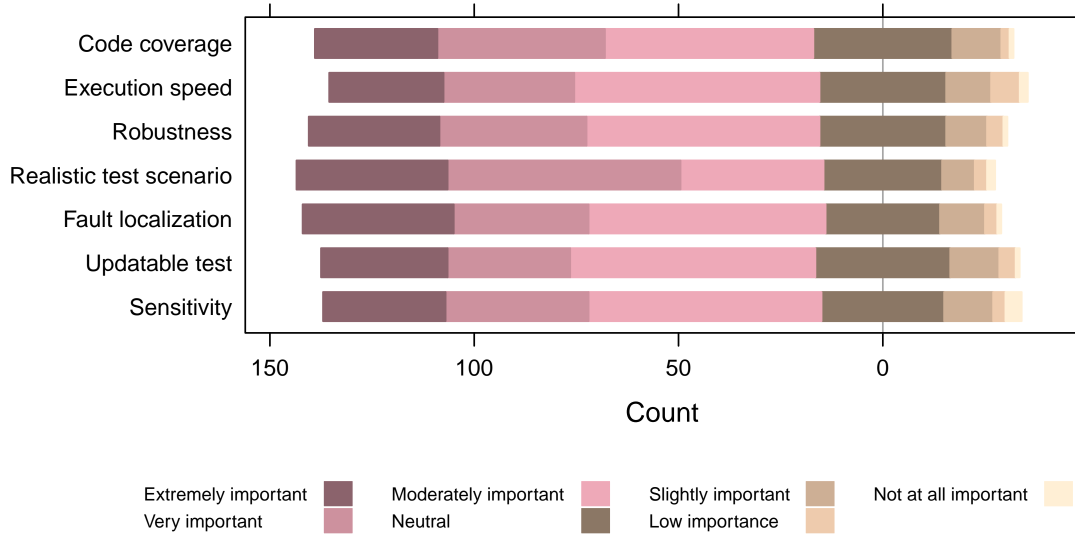


Fig. 3.11 Which aspects do developers aim to optimise when writing tests?

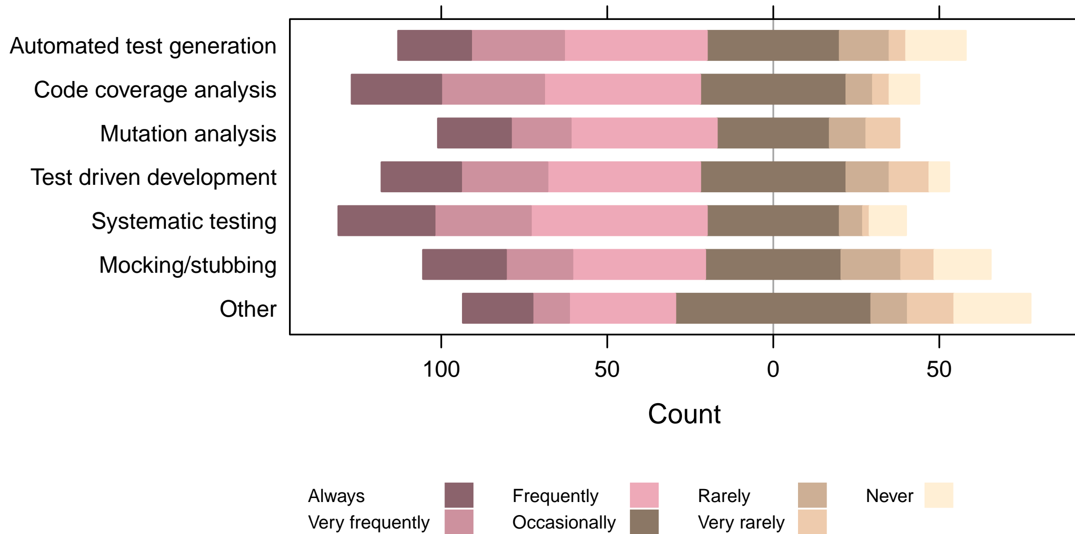


Fig. 3.12 Which techniques do developers apply when writing new tests?

The respondents claimed to be spending about 16% of their time on writing unit tests; the next research question now looks at *how* developers write these tests, by asking two questions.

3.3.5.1 What Do Developers Optimise Their Tests For?

The first question (Q4) asked respondents to rate which aspects they try to optimise when writing new unit tests, listing several options. Figure 3.11 displays the results; it turns out that this is a tricky question: The answer choices are all generally regarded as properties of good unit tests, so most respondents tend to agree with all of the choices to a good degree. The highest level of agreement can be seen for "realistic test scenario", which is of particular interest for unit test generation tools. For example, a recent study [89] found that using a random unit test generation tool all of 181 reported failures were due to unrealistic scenarios in the context of the system. However, this is clearly not a solved problem and requires further attention from testing researchers, and it is a prime concern for developers.

Execution speed and sensitivity against code changes are aspects that developers seem to regard as less important than the other properties, but on the whole there is rather strong agreement with all of the options, and it is difficult to discern real trends – developers seem to be claiming that they are optimising their tests to everything that makes a good test, at least to some degree.

While the conclusions we can draw about what specifically developers do when writing a new test are limited, this is still useful information for research on automated unit test generation: Random unit test generation tools, which arguably are very popular among testing researchers, typically optimise for *none* of these aspects. Coverage oriented tools tend to optimise for *one*: coverage, which is not a top rated concern. The other dimensions are areas where there are opportunities for automated testing research, in particular making the tests robust against changes, or easy to update.

3.3.5.2 Which Techniques Do Developers Apply When Writing Tests?

The second question in this group (Q5) asked respondents to select techniques that they apply when writing their unit tests. We listed five different techniques, again asking for a rating on a 7-point Likert scale. Figure 3.12 shows the results of this question: Most developers claim to apply some systematic approach when writing unit tests. (In the question, we listed boundary value analysis as an example of a systematic technique.) Code coverage analysis is listed as the second most frequently applied technique. These two techniques could be seen as related: If one uses code coverage, then it is likely that one systematically aims to cover code. Test-driven development also seems to be quite common practice, suggesting that many of the tests are written before the code.

Somewhat surprising, almost 54% of developers answered to use automated test generation at least frequently. This result needs to be taken with a grain of salt; as we saw from our

pilot studies the concept of automated test generation is not something that developers are particularly familiar with, and so, unlike for more common techniques like code coverage, the interpretation of what is automated test generation is not coherent across developers. Clearly, this suggests that practitioners need to be made better aware of progress in automated unit testing research. However, in discussions with developers one sometimes observes a indisposition against the idea of automated test generation, with arguments about how the act of writing the tests improves the software. The response to this question seems to suggest that this is not the prevailing opinion, and developers may be more open minded about automated test generation than often believed.

Mutation analysis is applied less often than other techniques, but we would classify mutation analysis as an advanced technique – it is liked and endorsed by researchers, but would not be expected to be common in practice. Thus, the result that 69% of the respondents claim to use mutation analysis at least occasionally is surprising: For example, recent workshops of the mutation testing community featured many discussions on why mutation analysis is not being picked up by practitioners. Although the survey results confirm that mutation analysis is not as common as code coverage, it does seem that mutation analysis is more common in practice than believed. Indeed we have subjectively perceived a recent increase of availability of new mutation analysis tools for various languages, which would confirm this trend.

At the lower end of the scale, stubbing and mocking are less common than the other techniques. This is of relevance to automated test generation, as high coverage could easily be achieved by automatically generating stubs of all dependencies and tailoring their behaviour to what is needed to cover code. However, adding stubs may cause maintenance problems and unrealistic behaviour – which are aspects developers would like to optimise in their tests.

3.3.6 RQ3.5: How Do Developers Use Automated Unit Test Generation?

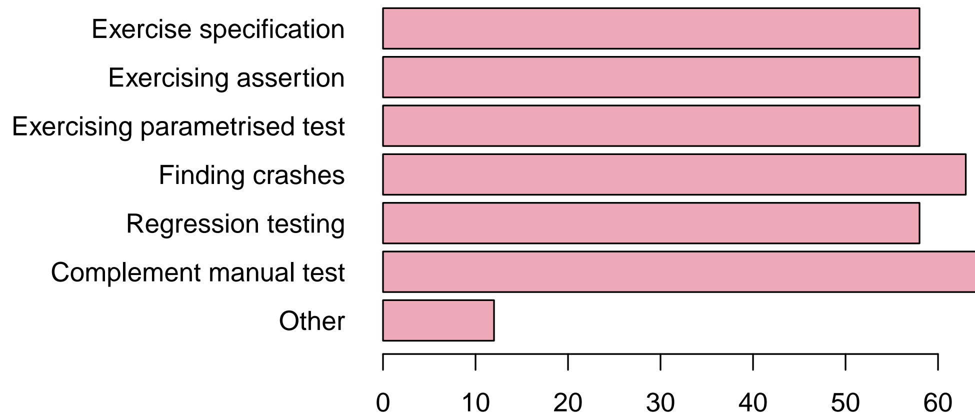


Fig. 3.13 Common usage scenarios of automatic test generation.

The previous question suggested that automated unit test generation is applied quite frequently in practice. The question now is, how is it used?

Figure 3.13 summarises which options respondents selected in Q6: The main application area for automated test generation tools in unit testing is to complement manual tests. From a test generation research point of view, this is relevant as it suggests that such automatically generated tests are elevated to the same importance as manually written tests, and will therefore suffer from the same problems: Test oracles need to be devised, failing tests need to be debugged, and test code needs to be maintained, posing challenges on the readability and other aspects.

The second most frequently given answer is that automatically generated tests are used to find crashes. Indeed, this is what automated test generation tools are particularly good at, as this can be done fully automatically without any specification or test oracles.

Exercising specifications, assertions, and parameterised unit tests are less frequent, even that they would allow a fully automated test generation process. However, it is conceivable that this would change with availability of more advanced (and usable) automated test generation tools: Automated test generation could serve as better incentive to write assertions or code contracts. Regression testing is also ranked less often, and again that seems like a missed opportunity, as in regression testing the previous program version can typically take the place of the specification, thus allowing full automation in principle.

3.3.7 RQ3.6: How Could Unit Testing Be Improved?

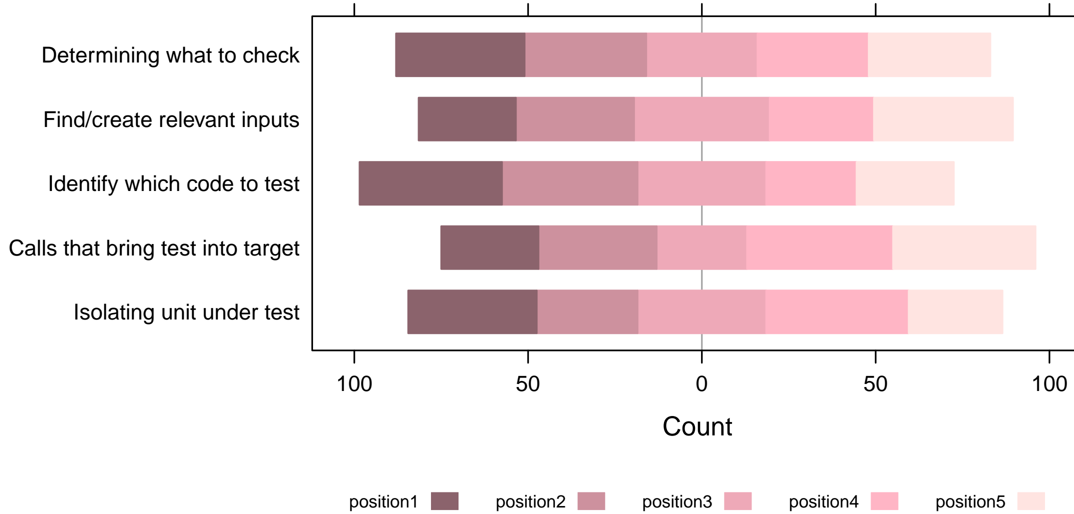


Fig. 3.14 What is most difficult about writing unit tests?

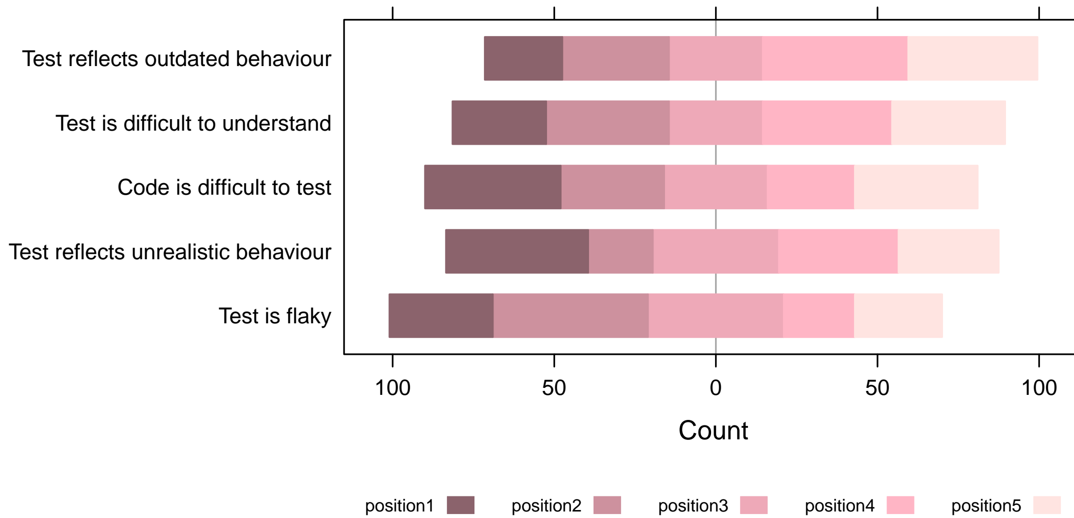


Fig. 3.15 What is most difficult about fixing unit tests?

The final part of our questionnaire aims at understanding where developers perceive difficulties and potential for improvement. What do developers believe to be the most difficult aspects of unit testing? To answer this, our questionnaire posed two ranking questions (Q7, Q8) to find out what developers consider to be most difficult about writing and fixing tests,

respectively, and one question of agreement to general statements about unit testing (Q9). For the ranking questions, respondents were given five answer choices and were requested to rank them by their difficulty. In earlier iterations of the survey we offered more choices, and reduced them to five in order to increase chances of obtaining useable data. We analysed the data for these questions using a Borda count [1] in order to rank the responses cumulatively across all respondents: For each answer choice a score is calculated such that every time the answer is ranked last 0 points are added to the score, up to 4 points for each first rank.

3.3.7.1 Writing New Unit Tests

Figure 3.14 shows the results on what makes it difficult to write a new unit test (Q7), and Table 3.2 summarises the Borda ranking. In contrast to the previous questions, there is less agreement on this question; the inter-rater agreement is generally very low, limiting the conclusions we can draw from this data: Kendall's coefficient of concordance is close to 0.

Table 3.2 What makes it difficult to write a new unit test?

Rank	Aspect	Borda count
1	Identifying which code to test	522
2	Isolating unit under test	521
3	Determining what to check	520
4	Finding relevant input values	493
5	Finding a sequence of calls	479

As far as we can trust the data, developers believe that the most difficult aspect is to identify which code to test. In automated unit test generation, this is generally addressed simply by using coverage criteria (e.g., search-based testing), ignored completely (random testing), or irrelevant because all paths are explored (DSE). Considering that 149 out of 171 respondents answered to use code coverage tools at least occasionally, one would expect that this should be an easy problem–a code coverage tool would point out exactly which code to test. However, as also indicated in the lower agreement on the priority of code coverage when writing new tests (Section 3.5), the problem of deciding what to test seems to be more difficult than that, and developers are not simply striving to cover code. Under this light, it is questionable whether the use of code coverage to drive automated test generation will make developers happy in the long run. Indeed, considering the responses to Q4, they rather seem to be trying to find realistic scenarios.

The aspect ranked second is to isolate the unit under test. Indeed this is also a major challenge for automated unit test generation [69], but there is little tool support one could get in this task (e.g., [62]). The test generation literature often ignores this aspect, and it seems there is opportunity to improve software testing in this respect. However, often this problem is also simply due to low testability in the code written by developers, and whether improving automated tools to cope with bad code is the right approach is debatable.

The aspect ranked third is the question of what to check, e.g., which assertions to add to a test. Intuitively this is less of a problem when a test is written manually, as the developer will have a specific scenario in mind, which implies also what aspects to check. For automatically generated tests this becomes more of a problem – if a test does not represent a realistic scenario, what is it testing, and what needs to be checked? This is the well known *oracle problem*. Automated tools can only either rely on external input in terms of specifications, or relay this question to the tester. Consequently, this ranking can be seen as positive reinforcement for research on automated test generation, for example on mutation-based approaches to support developers in making this decision (e.g., [183, 76]).

The last two positions in the ranking are on finding the right input values and the right sequences of calls. Interestingly, this is precisely what automated test generation tools are being optimised for and evaluated against.

We looked at the ranking for different sub-groups, but largely found agreement with the global ranking. Notably, inexperienced programmers rank the problem of what to check higher, whereas there is little agreement on how difficult it is to determine relevant input values. With increasing experience the problem of deciding which code to test also seems to become more important.

Table 3.3 What makes it difficult to fix a failing test?

Rank	Aspect	Borda count
1	The test is flaky	549
2	The code under test is difficult to understand	526
3	The test reflects unrealistic behaviour	522
4	The test is difficult to understand	499
5	The test reflects outdated behaviour	469

3.3.7.2 Treating Failing Tests

Figure 3.15 shows the results on what makes it difficult to fix a failing unit test (Q8), and Table 3.3 summarises the Borda ranking.

Again the inter-respondent agreement is low. The top ranked response of what makes a test difficult to fix is if the test is flaky, i.e., it fails sometimes, but not always. For example, tests that depend on non-deterministic code, environmental properties, or multi-threaded code are inherently difficult to debug and thus to fix. To some degree, the problem with a flaky test is in the code, not the test, and thus the second ranked choice is related: Tests are difficult to fix if the code under test is difficult to understand. This is not unexpected, but from a test generation point of view the code usually has to be taken as a given, and improvements can only be achieved in the generated tests. However, ensuring that generated tests are not flaky is an important question, and we are not aware of any work in that area.

The last three options reflect properties of the tests, rather than the code. Out of the three, the top ranked option is the problem of unrealistic tests, and this mirrors the concern of developers to derive realistic scenarios when writing tests (Q4). This is cause for concern for automated test generation tools — how should an automatic test generation tool distinguish between realistic and unrealistic behaviour? Specifications or other artefacts may provide hints; a recent idea is also to use the system context of a unit to determine realistic behaviour [89], but this may not always be an option in unit test generation, leaving an open research problem.

This is followed by the test being difficult to understand, and considering that understandability of tests is ranked less difficult than understandability of code, there may be an opportunity for automated test generation: If good can tests be generated, then these may help in understanding the code. Finally, the last ranked option is if the test represents outdated behaviour — which one would actually expect to be a common case when software evolves.

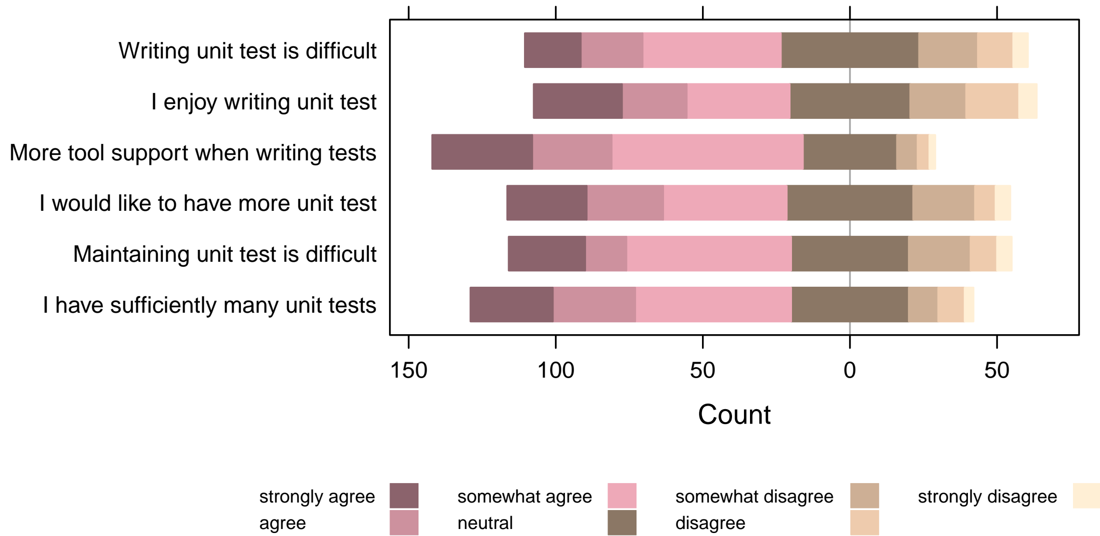


Fig. 3.16 General questions on perception of unit testing.

3.3.7.3 General Impressions

The final question (Q9, followed by the demographic questions in the questionnaire) of the survey asked respondents to specify their level of agreement with different statements about unit testing. Figure 3.16 shows these statements and the responses. A clear response is given to the question of whether more tool support is needed when writing tests — developers seem to be very open to this idea, and eager to see more tools.

Interestingly, there is quite strong agreement that developers already have sufficiently many tests, although there is also quite strong (though slightly weaker) agreement that developers would like to have more tests. Thus, even though developers believe to have enough tests, they would be open to further tests — this could be seen as a call to arms for unit test generation researchers: Developers are willing to use tools, and they would like to see more tests, so an obvious choice would be to provide tools that generate these additional tests automatically.

On the other hand, there is more agreement by developers that maintaining unit tests is difficult compared to writing unit tests. This poses a challenge to automated test generation tools, where research is still very much focused on solving the latter problem, with few exceptions (e.g., [160]). Finally, the lowest agreement can be seen for the question of whether developers enjoy writing unit tests. There is still slight agreement that writing unit tests is enjoyable (87/171 respondents answered that they at least "slightly" enjoy writing unit tests), but this question also received the largest share of negative responses. With only about half

of the developers attributing a positive feeling to unit testing, there seems to be a clear need to act; providing advanced tools that help developers may be one way to make testing more enjoyable.

3.4 Conclusions

The need to improve software quality is generally accepted, but the question remains how to achieve this. Improving unit testing is one possibility, and software engineering research is providing practitioners with new techniques that can improve the quality and productivity of their testing activities. Our aim was to better understand common practice in unit testing, in order to identify possibilities for improvement, in particular automated unit test generation.

In general, the responses point out areas of unit testing where automated unit test generation could support developers, and where further research is necessary:

- Developers need help to decide *what to test*, rather than which specific input value to select. Most research on automated unit testing side-steps this problem (e.g., by making the choice random or driven by a coverage criterion).

- Unit tests need to be *realistic*: A prime concern of developers when writing new tests, but also when treating failing tests, is whether they are realistic. An unrealistic scenario will make it more difficult to fix the test, and developers may not be convinced to fix their code based on unrealistic tests.

- Unit tests need to be *maintainable*: Even when automatically generated, a unit test may be integrated into the regular code base, where it needs to be manually maintained like any other code. If the test no longer reflects updated behaviour, it needs to be easy to detect this, and it needs to be easy to fix it.

- Unit test generation is most efficient at determining input values, but the unsolved challenge is to determine *what to check*. Ultimately, this means the long standing test oracle problem bugs unit testing like any other test generation approach, and advances will make test generation tools more valuable to practitioners.

The online survey presented in this chapter confirmed that unit testing plays an important role during software development, and test maintenance is an important aspect to be covered. Therefore the following chapter describes a novel test readability model that will help to generate more readable test cases and decrease test maintenance effort.

Chapter 4

Modeling Readability to Improve Unit Tests

The content of this chapter is based on two conference papers that were previously published in the 2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering [56, 55].

4.1 Introduction

As described in Chapter 2, readability is an important aspect in almost every area. Just as linguistic models have been applied to assess the readability of natural language texts, so have models for improving the readability of code also been developed for software practice [39]. In the previous Chapter we mentioned the importance of unit testing and software maintenance, which implies that developers need maintainable test cases.

Producing good tests is a tedious and error prone task, and over their lifetime, these tests often need to be read and understood by different people. Developers use their own tests to guide their implementation activities, receive tests from automated unit test generation tools to improve their test suites, and rely on the tests written by developers of other code. Any test failures require fixing either the software or the faulty test, and any passing test may be consulted by developers as documentation and usage example for the code under test. Test comprehension is a manual activity that requires one to understand the behavior represented by a test—a task that may not be easy if the test was written a week ago, difficult if it was written by a different person, and challenging if the test was generated automatically.

Test understanding depends on many factors. Unit tests for object-oriented languages typically consist of sequences of calls to instantiate various objects, bring them to appropriate states, and create interactions between them. The particular choice of sequence of calls and

values can have a large impact on the readability of the test, which is related to the visual appearance of the code. For source code, Buse and Weimer [39] applied machine learning on a dataset of code snippets with human annotated ratings of readability, allowing them to predict whether code snippets are considered readable or not. Although unit tests are also just code in principle, they use a much more restricted set of language features; for example, unit tests usually do not contain conditional or looping statements. Therefore, a general code readability metric may not be well suited for unit tests.

In this chapter, we address the test readability problem by designing a domain-specific model of readability based on human judgements that applies to object oriented unit test cases. To support developers in deriving readable unit tests, we use this model in an automated approach to improve the readability of unit tests, and integrate this into an automated unit test generation tool. Furthermore, this chapter presents:

- An analysis of the syntactic features of unit tests and their importance based on human judgement (Section 4.2).
- A regression model based on an optimized set of features to predict the readability of unit tests (Section 4.2).
- A technique to automatically generate more readable tests (Section 4.3).
- An empirical comparison between code and test readability models (Section 4.4).
- An empirical evaluation of the test improvement technique (Section 4.4).
- An empirical evaluation of whether humans prefer the tests optimized by our technique to the non-optimized versions (Section 4.4).
- An empirical study into the effects of readability on test understanding (Section 4.4).
- An empirical study of generating readable tests for Guava Library (Section 4.5).

4.2 Unit Test Readability Metric

The source code readability metric by Buse and Weimer [38] is built on a dataset of human annotator ratings, where each code snippet received readability ratings in the range of 1 to 5. Our aim is to create a predictive model that tells us how readable a given unit test is. Whereas Buse and Weimer trained a classifier to distinguish between readable and less readable code, our aim goes beyond this: We would like to use the model to guide test generation in producing more readable tests. Therefore, we desire a regression model that predicts relative readability scores for unit tests.

Our overall approach begins with producing a dataset of tests annotated with numeric human ratings of readability. We then identify a range of syntactic features that may be predictive of readability (e.g., identifier length, token entropy, etc.). We then use supervised

machine learning to construct a predictive model of test case readability from those features. To predict the readability of a new test case we calculate its feature values and apply the learned model. In this chapter, we use a simple linear regression learner [197], although in principle other regression learners are also applicable (e.g., multilayer perceptron [197]). However, in linear regression the resulting model (which consists of weightings for individual features) can easily be interpreted, and the learning is quick, which facilitates the selection of suitable subsets of features.

In this section, we describe how we collected the data to learn this model, the features of unit tests we considered, and the machine learning we applied to create the final model.

4.2.1 Human Readability Annotation Data

Both the test cases considered and the human annotators chosen influence the quality of our readability model. While it is relatively easy to assemble a diverse group of unit tests, particular attention must be paid to participant selection and quality in this sort of human study. For scalability we used crowdsourcing to obtain participants, but found that the use of a qualification test is critical for such crowdsourced participants.

Supervised learning requires a training set. In this work we used a number of diverse and indicative open-source Java projects: Apache Commons, Apache POI, JFreeChart, JDOM, JText and Guava. Each of these projects comes with an extensive test suite of developer-written unit tests. In addition, we applied the EvoSuite [70] unit test generation tool in its default configuration to produce a branch coverage test suite for each of the projects. Training tests were then selected manually with the aim to achieve a high degree of diversity (e.g., short and long tests, tests with exceptions, if-conditions, etc.)

To collect the human annotator data, we used Amazon Mechanical Turk,[1] and asked crowd workers to rate unit tests on a scale of 1 to 5 using the presentation setup of Buse and Weimer [38]. As in previous work, annotators were not given a formal definition of what to consider readable, and were instead instructed to rate code purely based on their subjective interpretation of readability. However, while the original Buse and Weimer survey involved undergraduates from the same institution, we find that crowdsourcing leads to a much broader diversity of annotator expertise which must be accounted for to learn a useful model.

We began by assessing the utility of general crowd worker responses for this task. We selected 100 test cases from our eight benchmark projects, including developer-written as well as automatically generated tests, and collected 2,388 human ratings for these test cases

[1] http://aws.amazon.com/mturk/, accessed 03/2015.

(i.e., each annotator rated 15–32 test cases). To evaluate the quality of these responses, we measured the inter-annotator agreement by calculating the average Pearson's correlation between each annotator and the average test scores. The inter-annotator agreement in this data set, generated from participants with no expertise requirements or filtering, is only weak, with a value of 0.25.

This low correlation could arise for many reasons, including an unsuitable choice of test cases or insufficient qualification of the human annotators. To investigate this issue, we manually selected 50 test cases that are examples of either very high quality (e.g., concise, well documented, well formatted) or very low quality (e.g., long, complex, badly formatted). The selected set was refined by iterations, and retaining only tests with unison agreement. For these test cases we again gathered human annotator scores and measured the inter-annotator agreement. The results confirmed the need to require annotator expertise. The agreement was even lower than on the first set of tests: The inter-annotator agreement in this experiment was only 0.2, which leads us to the conclusion that the more likely explanation is insufficient qualification of the human annotators.

4.2.2 Final Annotation Data Set

Based on these pilot studies, we designed our final experiment to use a qualification test. This qualification test consisted of four questions of understanding based on example Java code. Only human annotators who correctly answered three out of the four questions were allowed to participate. Our observations that crowdsourced participants can be fruitfully used for such human studies [109, 182] provided that care is taken to avoid participants who are simply trying to "game the system" [78] is consistent with previous work.

Our final experiment gathered data on 450 human- and machine-written test cases, ultimately obtaining 15,669 human readability scores. We conducted the experiment in stages, initially focusing on all tests equally but subsequently gathering additional annotations on particular tests to ensure that each test feature considered (see Section 4.2.3) had enough annotations for machine learning purposes. Restricting attention to qualified participants increased the inter-annotator agreement substantially, to 0.5. In addition, we generated 200 pairs of tests using the EvoSuite tool, such that each pair had the same coverage quality but a different textual representation. For these pairs of tests we gathered a separate set of forced-choice judgments (i.e., annotators were asked to select which of the two tests were the most readable) for use in feature selection (see Section 4.2.5).

Figure 4.1 shows the underlying frequency distribution of readability scores for the 450 tests. The histogram shows that there are few tests with very high or very low scores, and the majority of scores is in the range from 3.0-4.0. We note that our distribution of test readability

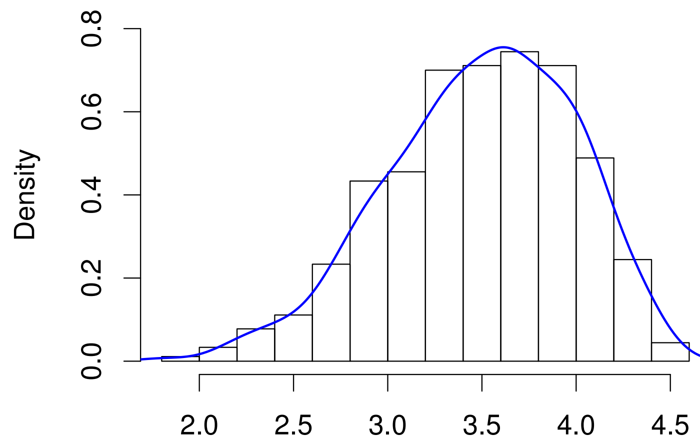


Fig. 4.1 Score distribution for the human test annotation dataset.

is quite different from the bimodal distribution of source code readability observed by Buse and Weimer [38], motivating the need for a test-specific model of readability.

4.2.3 Features of Unit Tests

The readability of a unit test may depend on many factors.

We aggregated features by combining structural, logical complexity, and density code factors. All features used by Buse and Weimer [38] are included in this set, as well as the entropy and Halstead features (see 2.7.0.2) used by Posnett al. [154] (we tend to collect as many features as possible and filter them if they are poor predictors of readability). Additionally, we included the following new features:

Assertions: This feature counts the number of standard JUnit assertions. Additionally we included a binary feature *has assertions* which has value 1 if a test case contains assertions, and 0 otherwise. Figure 4.2 shows a sample test case with *assertions* equal to 2 and *has assertions* equal to 1.

```
BitSet bitSet0 = new BitSet();
bitSet0.set(0, 6);
bitSet0.flip(5);
BitSet bitSet1 = bitSet0.get(6, 6);
assertEquals(8, bitSet0.size());
assertEquals("{0, 1, 2, 3, 4}", bitSet0.toString());
```

Fig. 4.2 Example test case with two assertions

Exceptions: We propose a feature to measure exceptions since exceptional behavior is handled differently from regular behavior in tests. As we did not encounter any tests with

more than one expected exception, we use a binary feature *has exceptions*. For example, Figure 4.3 shows a test case with *has exceptions* feature equal to 1 (true).

```
BitSet bitSet0 = new BitSet();
bitSet0.set(4);
// Undeclared exception!
try {
 BitSet bitSet1 = bitSet0.get(0, 4);
 fail("Expecting exception: ArrayIndexOutOfBoundsException");

} catch(ArrayIndexOutOfBoundsException e) {
 //
 // 0
 //
}
```

Fig. 4.3 Example test case throwing an exception

Unused Identifiers: Unit tests are typically short and contain few variables. In our anecdotal experience with EvoSuite we found that developers do not prefer tests that define but never use variables. As test cases may include variables that do not contribute to the tested assertion at all, developers prefer minimized tests for the sake of clarity.

For example, Figure 4.4 shows a test case that has one identifier (*bitSet1*) which is not used and makes the *unused identifiers* feature equal to 1.

```
BitSet bitSet0 = new BitSet();
BitSet bitSet1 = (BitSet)bitSet0.clone();
boolean boolean0 = bitSet0.equals((Object) (BitSet)bitSet0.clone());
bitSet0.set(0, true);
assertEquals(8, bitSet0.size());
assertEquals(false, bitSet0.isEmpty());
```

Fig. 4.4 Example test case with unused identifier

Comments: We count the number of lines for single-line ("//") and multi-line ("/* ... */") comments. However, as EvoSuite and other tools generate comments mainly within the catch block of an expected exception, where it shows the error message of the exception, we refined the comments feature to two versions, one that counts regular comments, and one that counts the comments within catch blocks. For example, Figure 4.5 shows a test case containing both regular comments and exception comments. Thus, in this case both of the features will have value 1 and 3 respectively. Comments are removed before calculating

code-specific features (e.g., features based on numbers, classes) but included for general presentational features (e.g., line length).

```
BitSet bitSet0 = new BitSet();
// Undeclared exception!
try {
 bitSet0.set(24);
 fail("Expecting exception: ArrayIndexOutOfBoundsException");

} catch(ArrayIndexOutOfBoundsException e) {
 //
 // 3
 //
}
```

Fig. 4.5 Example test case with comments

Token features: We identified several additional common syntactic features not captured in past readability models. In particular, we observe that unit test generation tools often tend to include defensive (sometimes redundant) casts. Furthermore, we propose a feature to count the overall tokens identified by the parser. Finally, we refined the "single character occurrence" feature used by Buse and Weimer, which counts the number of occurrences of different special characters (parenthesis, quotation marks, etc.) with a feature that counts all the special characters (*single characters*). Figure 4.6 is a test case with 26 tokens, and 17 single characters.

```
 XMLElement xMLElement0 = new XMLElement();
 xMLElement0.addChild((IXMLElement) xMLElement0);
 assertEquals(false, xMLElement0.isLeaf());
```

Fig. 4.6 Example test case

Datatype features: Different primitive data types have a possible impact on readability. Hence, we propose features based on the occurrence of the value null, Boolean values (true and false), array accesses, type constants (e.g., Object.class), floating point numbers, digits, strings, characters, and the length of strings. Figure 4.7 presents a test case containing 2 strings (line 3 and 4) with length equal to 24, one null (line 6) value, and two boolean values (line 5 and 7). While the same test case does not contain number or floating point values, test case in Figure 4.5, has one number in line 4. We also added a feature measuring the "English-ness" of string literals, using the language model of Afshan et al. [5]. Test case in Figure 4.7 contains two strings which in total have a low score of "English-ness" (0.000025).

```
DateLocaleConverter dateLocaleConverter0 = new DateLocaleConverter();
Locale locale0 = Locale.UK;
DateLocaleConverter dateLocaleConverter1 = new DateLocaleConverter((Object)
    dateLocaleConverter0, locale0, "=K6>D]*@uR>]");
DateLocaleConverter dateLocaleConverter2 = (DateLocaleConverter)
    dateLocaleConverter1.convert((Object) dateLocaleConverter0, "=K6>D]*@uR>]");
DateLocaleConverter dateLocaleConverter3 = new DateLocaleConverter((Object)
    dateLocaleConverter0, true);
DateLocaleConverter dateLocaleConverter4 = null;
assertEquals(false, dateLocaleConverter3.isLenient());
assertNull( dateLocaleConverter4.isLenient());
```

Fig. 4.7 Example test case with different datatypes

Statement features: We propose features to count different types of statements, in particular constructor calls, field accesses, and method invocations. Figure 4.8 shows a test case that contains a field access in line 2, a method invocation in line 5, and a class constructor in line 3.

```
Class<Object> class0 = Object.class;
Locale locale0 = Locale.CANADA_FRENCH;
DateLocaleConverter dateLocaleConverter0 = new DateLocaleConverter((Object) ",hz*
    AABm7s6p/", locale0, "xT\rqbbLS/", true);
String string0 = (String)dateLocaleConverter0.convert(class0, (Object) locale0, ",
    hz*AABm7s6p/");
assertEquals(",hz*AABm7s6p/", string0);
```

Fig. 4.8 Example test case with different statements

Class and method diversity: In addition to diversity, as captured by entropy features at the level of tokens (distribution of tokens in the code) and bytes (distribution of characters in the code), we also propose features to capture diversity in terms of the classes, methods, and identifiers used. For each of these, we include a feature counting the unique number as well as the ratio of unique to total numbers. For example, Figure 4.9 contains two method calls and two class constructors which both are unique (*unique classes* and *unique methods* will have value 2), while the *class ratio* and *method ratio* are equal to 1.

```
CommandLine commandLine0 = new CommandLine();
Option option0 = new Option("", false, "");
commandLine0.addOption(option0);
boolean boolean0 = commandLine0.hasOption('-');
assertTrue(boolean0);
```

Fig. 4.9 Example test case with class constructor and method call

Table 4.1 Predictive power of features based on correlation and one feature at a time analysis, and optimized regression model.

	Correlation			One feature a time		
Feature name	total	max	avg	total	max	avg
identifier length	**-0.50**	**-0.42**	**-0.46**	0.50	0.41	0.45
commas	-0.15	-0.20	-0.15	0.13	0.14	0.13
line length	**-0.45**	**-0.50**	-0.43	0.4	0.49	0.41
Halstead difficulty	-0.15	-	-	-	-	-
constructor calls	**-0.45**	-	-0.24	0.44	-	0.20
has exceptions	**0.15**	-	-	0.11	-	-
byte entropy	-0.39	-	-	0.31	-	-
identifier ratio	**0.15**	-	-	0.11	-	-
unique identifiers	**-0.37**	-0.25	-0.14	0.36	0.22	0.11
method invocations	-0.14	-0.06	-0.03	0.09	-0.00	-0.07
identifiers	**-0.36**	-0.23	-0.29	0.36	0.20	0.27
string length	-0.14	-0.24	**-0.20**	0.09	0.23	0.19
assignments	-0.33	-	-0.16	0.32	-	0.13
arrays	-0.14	-0.05	-0.06	0.11	-0.05	-0.01
casts	-0.33	-0.33	-0.28	0.32	0.32	0.26
indentation	-0.13	0.08	-0.01	0.10	0.02	-0.25
parentheses	-0.31	-	-0.28	0.30	-	0.26
field accesses	-0.13	-0.14	-0.17	0.11	0.11	0.15
keywords	-0.30	-0.28	-0.17	0.28	0.27	0.14
Halstead effort	-0.13	-	-	0.13	-	-
Halstead volume	-0.27	-	-	0.18		
assertions	**-0.12**	-	-0.04	0.09	-	-0.10
distinct methods	**-0.27**	-0.05	0.00	0.26	-0.02	-0.18
additional assertions	-0.12	-	-	0.09	-	-
single characters	-0.26	-0.32	-0.24	0.17	0.22	0.16
nulls	-0.12	**-0.20**	**-0.15**	0.06	0.19	0.12
periods	-0.25	-0.22	-0.17	0.24	0.20	0.13
class ratio	-0.10	-	-	0.04	-	-
comparison operations	-0.25	-	-0.23	0.24	-	0.23
blank lines	0.08	-	0.21	0.03	-	0.19
tokens	-0.24	-0.28	-0.21	0.15	0.17	0.14
unused identifiers	-0.07	-	-	0.02	-	-
digits	-0.24	-0.24	-0.19	0.21	0.22	0.161
string score	0.06	-	-	0.01	-	-

token entropy	**-0.24**	-	-	0.17	-	-
strings	-0.05	-0.19	-0.13	0.00	0.17	0.10
floats	**-0.22**	-0.21	-0.17	0.20	0.19	0.14
excep. comments	0.05	-	0.15	-0.00		0.12
comments	0.20	-	0.04	-	0.18	-0.01
arithmetic operations	-0.04	-	**0.03**	-0.07	-	-0.05
test length	-0.19	-	-	0.13	-	-
branches	0.04	-	**0.06**	0.02	-	0.02
numbers	**-0.19**	-0.18	-0.07	0.18	0.17	0.04
types	-0.02	-0.08	-0.04	-0.11	0.05	-0.02
spaces	-0.19	-	-0.17	0.13	-	0.13
has assertions	**0.01**	-	-	-0.14	-	-
loops	0.19	-	**0.18**	0.17	-	0.17
method ratio	**0.01**	-	-	-0.21	-	-
booleans	-0.16	-0.17	-0.01	0.14	0.13	-0.23
characters	0.00	**-0.03**	0.04	-0.15	-0.08	-0.01

$$
\begin{aligned}
\text{test_score} = & -0.0001 \times \text{total_line_length} - 0.0021 \times \text{max_line_length} + 0.0076 \times \\
& \text{total_identifiers} - 0.0004 \times \text{total_identifier_length} - 0.0067 \times \text{max_identifier_length} \\
& -0.005 \times \text{avg_identifier_length} + 0.0225 \times \text{avg_arithmetic_operations} + 0.9886 \times \\
& \text{avg_branches} + 0.1572 \times \text{avg_loops} + 0.0119 \times \text{total_assertions} - 0.0147 \times \\
& \text{total_has_assertions} + 0.1242 \times \text{avg_characters} - 0.043 \times \text{total_class_instances} - 0.0127 \\
& \times \text{total_distinct_methods} + 0.0026 \times \text{avg_string_length} + 0.1206 \times \text{total_has_exceptions} \\
& -0.019 \times \text{total_unique_identifiers} - 0.0712 \times \text{max_nulls} - 0.0078 \times \text{total_numbers} \\
& +0.1444 \times \text{avg_nulls} + 0.334 \times \text{total_identifier_ratio} + 0.0406 \times \text{total_method_ratio} - \\
& 0.0174 \times \text{total_floats} - 0.3917 \times \text{total_byte_entropy} + 5.7501
\end{aligned}
\tag{4.1}
$$

Table 4.1 lists all the candidate features, showing which of the features we used in terms of the **total** value for the unit test, the **average** value per line, and its **max** value in any line in the test case. In total, we considered 116 candidate features (in Section 4.2.5 we use feature selection to build our model from the 24 most relevant).

To analyze the influence and predictive power of the individual features, Table 4.1 shows the Pearson's correlation between each feature and the average test scores (ranging from a weak positive correlation of 0.21 for average blank lines to a strong negative correlation of -0.5 for the maximum line length and total identifier length), and the result of a one-feature-at-a-time analysis. For the latter, we train a linear regression model using only one feature,

and determine the correlation with 10-fold cross validation. We also considered a leave-one-feature-out analysis, where one measures the effect of a feature by in terms of the difference between a model learned using all features and with all but the feature under consideration; however, leave-one-out analyses are not applicable in the presence of feature overlap and due to our very large set of related features the results are not representative. Finally, we also applied the Relief-F method [161], which agrees with the one-feature-at-a-time analysis and is thus omitted for brevity.

4.2.4 Feature Discussion

Considering that a unit test is often simply a sequence of calls, one would expect the length of that sequence to be one of the main factors deciding on the readability. However, as shown in Table 4.1, the number of statements (test length) on its own surprisingly only has weak predictive power. However, other features related to the length have a larger influence on the readability. In particular, the line length plays an important role, both in terms of maximum line length as well as the total line length. The maximum line length presumably is important because a test case can have bad readability even if most lines are short and only a single line is very long. The "total" line length essentially amounts to the total number of characters in the test and thus is a better representation of length than the number of statements.

We furthermore observe that the identifiers in a test have a large influence on its readability. This refers to features related to the number of identifiers, their length, and their diversity, and is a challenge for test generation tools, which typically use simple heuristics to derive names for variables. The diversity in general results in important features, for example captured by byte and token entropy [154].

Only a few features are positively correlated with test readability: comments have a weak positive correlation, as does the ratio of blank lines (avg. blank lines). Surprisingly, exceptions also have very weak positive correlation. We expected assertions to show a strong influence on readability, but there is no correlation between assertions and the test score, and the predictive power is weak.

The small influence of loops and conditional statements to some extent may be attributed to our choice of test cases for annotation: Many of the tests are generated by EVOSUITE, which generates only tests that are sequences of calls. The manually written tests with loops and conditional statements included in the dataset tend to be short and well-formatted, contributing to the small but positive influence of these features.

4.2.5 Feature Selection

Feature selection is a widely used technique that reduces the dimension of a dataset with respect to a given value [45]. Selecting a subset of potential inputs from the total feature set can help on compacting relevant information, and removing the noise in prediction [120]. To improve the learning process and the generality of the resulting model, it is thus desirable to reduce the number of features using feature selection techniques.

Feature selection techniques are classified [61, 65] in two main categories, called filter and wrapper models. A filter model typically consists of removing features that are shown to have low predictive power. For example, as a baseline we considered the use of correlation and the Relief-F filter model method [161], which select 43 and 21 features leading to a correlation of 0.65 and 0.7 respectively. We desire a higher quality feature set, however, and thus focus on wrapper model feature selection.

A wrapper model selects subsets of variables considering the learning method as a black box, and scoring inputs based on their predictive power. We considered forward and backward feature selection; in forward selection one starts from an empty set of features, and iteratively adds features based on the resulting predictive power. In backward selection the starting point is the full set of features, and one iteratively removes individual features.

We used a steepest ascent hill climbing algorithm to perform this feature selection. That is, for forward selection we start with a randomly chosen feature, create a regression model using only that feature, and calculate the correlation using 10-fold cross-validation. Then, for each other feature, we determine the correlation of a model trained using this and the first feature. The pair with the highest correlation is the new starting point, and we explore all possible variants to add another feature. This is done iteratively, until there exists no feature that can be added while increasing the correlation value. In our experiments, forward feature selection achieved substantially better results than backward feature selection and is used for our model.

To avoid overfitting the model to the training set, in addition to standard cross-validation we used the set of 200 pairs of test cases with human annotation data described in Section 4.2.2. For each of the tests in a pair we predicted the readability score, and then ranked the paired tests based on their score (represented with 0 or 1). Then, we measured the agreement between the user preference (i.e., 0 or 1, depending on whether more human annotators preferred the first or second test in the pair), using Pearson's correlation.

We applied the forward feature selection 1,000 times (see Figure 4.10a), and the best configuration consisting of 16 features achieved a correlation of 0.86. However, when measuring user agreement, the correlation of the same configuration is only 0.49, which suggests a certain degree of overfitting to the training data. To counter this, we re-ran feature

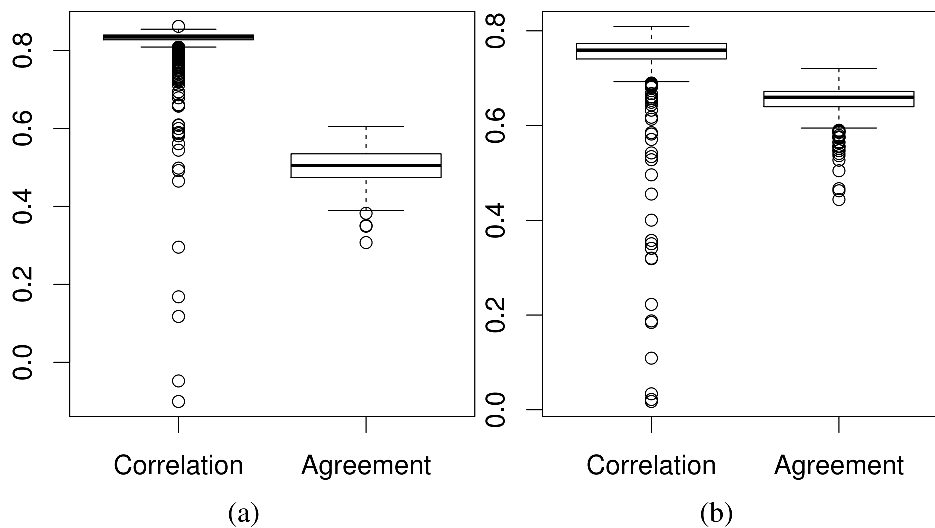


Fig. 4.10 Results of the feature selection measured in terms of Pearson's correlation with 10-fold cross validation, and agreement with user preferences for test pairs.

selection, but rather than using the correlation to guide the hill-climbing, we used the sum of correlation and user agreement. As Figure 4.10b shows, this reduces the achieved correlation slightly, but increases the agreement substantially.

The most frequently selected feature is "total identifier length", which occurred in 90% of all runs; the second most frequently selected feature is "max line length" (82% of runs), showing the importance of this feature. Interestingly, "total exceptions" was selected in 56% of the runs, suggesting that although the predictive power of this feature on its own is not so strong, it appears independent from other features. Byte entropy (52%) is selected more frequently than token entropy (19%), which suggests that token entropy is correlated with other features.

In the end, the overall best configuration consists of 24 features, shown in bold in Table 4.1, and the resulting regression model is shown below the table.

This combination of features achieves a correlation of 0.79 with a root relative squared error rate of 61.58%, and has a high user agreement (0.73).

4.3 Generating Readable Tests

Given a predictive model of test readability, we would now like to apply this model to improve automated unit test generation. As search-based testing is a common technique to generate unit tests, in principle it would be possible to simply include the readability prediction as a second objective in a multi-objective optimization, and thus optimize tests

Algorithm 3 Test Case Optimization

Input: Test case t, coverage objective c
Output: Optimized test case t'

```
 1: procedure OPTIMIZE(t, c)
 2:    T ← GENALTERNATIVES(t, c, length(t))
 3:    t' ← select highest ranked t' in T
 4:
 5:    return t'
 6: end procedure
 7: procedure GENALTERNATIVES(t, c, start)
 8:    T ← {t}
 9:    for p ← start down to 1 do
10:       s ← statement at position p in t
11:       for all s' ∈ get all replacements for s do
12:          t' ← replace s with s' in t
13:          if t' satisfies c then
14:             T ← T ∪ GENALTERNATIVES(t', c, p − 1)
15:          end if
16:       end for
17:    end for
18:    return T
19: end procedure
```

towards both coverage and readability at the same time. However, there is a dichotomy between the need for search techniques to include redundancy in the tests to explore the state space (i.e., statements that do not contribute to the code coverage), and the detrimental effects of this redundancy on the readability. Therefore, we use a post-processing technique to optimize unit tests, which has the additional benefit that it is independent of the underlying test generation technique, allowing our approach to apply to any such black box unit test generator.

Algorithm 3 describes this post-processing algorithm: We assume a test case t is a sequence of statements $t = \langle s_1, s_2, \ldots, s_l \rangle$ of length l, where each statement is either a method call, a constructor call, some other form of assignment (e.g., primitive values, public fields, arrays, etc.) or an assertion. The algorithm is given a test case t generated for coverage obligation c. It is assumed that t is minimized with respect to c; that is, removing any of the statements in t means that c is no longer satisfied.

Given these inputs, we generate the set of alternative versions of t that still satisfy c as follows: We iterate over the statements in the test from the last statement to the first statement (Line 9). For each statement we determine the possible set of replacement statements

```
ElementName elementName0 = new ElementName("", "");
Class<Object> class0 = Object.class;
VirtualHandler virtualHandler0 = new VirtualHandler(elementName0, (Class) class0);
Object object0 = new Object();
RootHandler rootHandler0 = new RootHandler((ObjectHandler) virtualHandler0, object0
    );
ObjectHandlerAdapter objectHandlerAdapter0 = new ObjectHandlerAdapter((
    ObjectHandlerInterface) rootHandler0);
assertEquals("ObjectHandlerAdapter", objectHandlerAdapter0.getName());
```

Fig. 4.11 Test case generated without readability optimization feature in EvoSuite

```
ElementName elementName0 = new ElementName("", "");
ObjectHandlerAdapter objectHandlerAdapter0 = new ObjectHandlerAdapter((
    ObjectHandlerInterface) null);
assertEquals("ObjectHandlerAdapter", objectHandlerAdapter0.getName());
```

Fig. 4.12 Test case generated with readability optimization feature in EvoSuite

(Line 11), consisting of all possible method or constructor calls that generate the same return type. This restriction ensures that the variable defined at the statement (if any) still exists after the replacement. We only consider replacements for statements calling constructors or methods, but in the future, other types of transformations could also be integrated. Any additional parameters of the replacement call are assigned randomly chosen existing variables of the desired types, the value null, or if no variable of the required type exists, then an instance can also be generated by recursively inserting a random generator for that type and satisfying its dependencies (e.g., based on the statement insertion in EvoSuite [70]).

For each candidate replacement t' we determine if it still satisfies coverage objective c by executing t' and observing its coverage. If it does, then we recursively apply the replacement algorithm to t' starting at the position preceding the modified statement, and keep all valid replacements. In the end, T contains the set of valid replacements for t that still satisfy c. The tests in T are then sorted by readability, and the most readable test in T is selected.

For example, consider the test case in Figure 4.11, which was generated to cover the constructor of `ObjectHandlerAdapter`: Alternative generation would start with the last statement, which is an assertion, and thus is not modified. The next statement considered is the constructor call. The class `ObjectHandlerAdapter` has four different constructors, but as the one called in the original test is the coverage objective of the test, replacements with the three other constructors no longer satisfy this objective and are discarded. Because the algorithm is randomized it also attempts to replace the constructor call with a new parameter assignment. Assume it satisfies the parameter with a null reference: The coverage obligation is still satisfied, and minimization can now remove the first five statements of this alternative,

as they are no longer used in the constructor call, and thus not needed in order to satisfy the coverage goal. The new test has no more statements to modify, so no further alternatives can be generated from this test, and the algorithm continues generating alternatives with the next statement of the original statement, which is the constructor call of `RootHandler`. In the end, the alternative that calls the constructor with a null value has the highest readability value (3.67 vs. 3.30 for the original test), and is chosen as replacement, and Figure 4.12 presents the test case optimized for readability.

4.4 Empirical Evaluation

This section contains an empirical evaluation of default test cases (i.e, test cases generated with default parameters) and test cases optimized for readability. In particular, we empirically aim to answer the following research questions:

RQ1: How does the test readability metric compare to code readability metrics?
RQ2: Can our test readability metric guide improvement of generated unit tests?
RQ3: Do humans prefer readability optimized tests?
RQ4: Does readability optimization improve human understanding of tests?

4.4.1 Experimental Setup

4.4.1.1 Unit Test Generation Tool

We have implemented the algorithm described in Section 4.3 in the EvoSuite [70] tool for automatic unit test generation. EvoSuite uses search-techniques to derive test cases with the aim to maximize coverage of a chosen target criterion (e.g., line coverage or branch coverage). After the generation, EvoSuite applies several post-processing steps to improve readability: For each individual coverage objective (e.g., branch) a minimized test case is generated; that is, removing any statement from the test will lead to the coverage objective no longer being satisfied. In these minimized tests, primitive values are inlined to reduce the number of variables, and then the primitive values are minimized (i.e., strings are shortened, and numbers are decremented as close as possible to 0 without violating the coverage objective). Finally, assertions are added to the tests, and minimized using an approach based on mutation analysis [77]. The readability optimization algorithm from Section 4.3 was integrated as a further step of this chain of optimizations.

4.4.1.2 Experiment Procedure

For RQ1, we used the public dataset by Buse and Weimer [38], our dataset of 450 annotated test cases (Section 4.2), and the set of 200 test pairs used to support feature selection (Section 4.2), and measured the correlation and agreement of different readability models.

For RQ2-4, we manually selected 30 classes from open source projects, with the criteria that they (1) are testable by EVOSUITE with at least 80% code coverage, (2) do not exhibit features currently not handled by EVOSUITE's default configuration such as GUI components, and (3) have less than 500 non-comment source statements (NCSS) and few dependencies, such that they are non-trivial yet understandable in a reasonable amount of time. For each of the chosen classes we generated 10 tests for each coverage objective (i.e., branch) to account for the randomness of the test generation approach, with and without the readability optimization introduced in this chapter. Furthermore, we generated an additional 10 test cases per branch per class with both configurations, but modified EVOSUITE to generate failing assertions (i.e., during the assertion generation using mutation analysis [74] the assertions were chosen to pass on the mutants rather than the original class). To answer RQ2, we compare the default and optimized tests in terms of their readability score as predicted by our test readability model. We used the Wilcoxon-Mann-Whitney statistical symmetry test, and the Vargha-Delaney $\hat{A}_{ab}$ statistics to evaluate the significance of the optimization [15].

For RQ3, we selected three random pairs of tests for each class from the RQ2 dataset; each pair consisting of one test generated with and one without the readability optimization, both cover the same branch, and they differ in readability score. This resulted in a total of 90 pairs of tests, and we used a forced-choice questionnaire on Amazon Mechanical Turk (see Section 4.2) to determine for each pair which test is preferred by users.

For RQ4, we selected 10 out of the 30 classes with large differences in readability of its tests, and for each class chose either a pair of passing or failing tests. (Note that our procedure to generate failing tests did not guarantee failing tests, hence the pass or fail status within pairs is not always identical.) To recruit students for RQ4, we invited all computer science students (undergraduate and postgraduate) at the University of Sheffield and asked them to perform a pre-qualification quiz. This quiz consisted of the four questions from the Mechanical Turk qualification plus one JUnit specific question, and we selected 30 students who answered at least 3 questions correctly. The experiment was conducted in the computer lab of the university's Department of Computer Science. All 30 selected participants received a short introduction to the experiment, and then answered 10 questions in a web browser based quiz. Each question showed a test case and provided the source code of all classes required by the test case, and asked the students to select if the test would pass or fail. After

60 minutes the students were asked to submit the answers they had produced up to that point, filled in a short survey, and were paid a fee of GBP10.

4.4.1.3 Threats to Validity

Construct: For RQ4, we use time and correctness of pass/fail decision to measure understanding. It is possible that using a different task that requires understanding would give a different result. For example, Ceccato et al. [47] reported a positive effect when using random tests during debugging.

Internal: For all experiments involving humans, the tests were assigned randomly. To avoid learning effects for RQ4, we ensured that no two tests shown to one participant originate from the same project. Participants without sufficient knowledge of Java and JUnit may affect the results; to avoid this problem we only accepted subjects who passed a qualification test. Experiment objectives may have been unclear to participants, at least for RQ4; to counter this threat we tested and revised all our material on a pilot study, and interacted with the students during experiment to ensure they understood the objectives.

External: All our experiments are based on either Amazon Mechanical Turk users, or students, and thus may not generalize to all developers [109, 182]. The set of target classes used in the experiment is the result of a manual but systematic selection process, aiming to find classes that are understandable in the short duration of the expriment (RQ4). The chosen classes are not small, but it may be that the readability optimization is more important for classes with more dependencies. Thus, to which extent our findings can be generalised to arbitrary programming and testing tasks remains an open question. We used EVOSUITE for experiments and to support the generation of our data set, and tests produced by EVOSUITE and other tools may lead to different results. However, the output of EVOSUITE is similar to that of other tools aiming at code coverage.

Conclusion: The human study to answer RQ4 involved 30 human subjects, which resulted in significance in only two out of the 10 test pairs. However, obtaining more responses per test pair by reducing the number of pairs was not possible, as this would have implied that students would have to answer several questions related to the same class, which would have led to undesired learning effects.

4.4.2 RQ1: Test vs. Code Readability

As a baseline for the success of our domain specific readability model, we used the code readability model by Buse and Weimer [38], as well as the extended version by Posnett et al. [154]. Both models are originally classification models, and we replicated them as

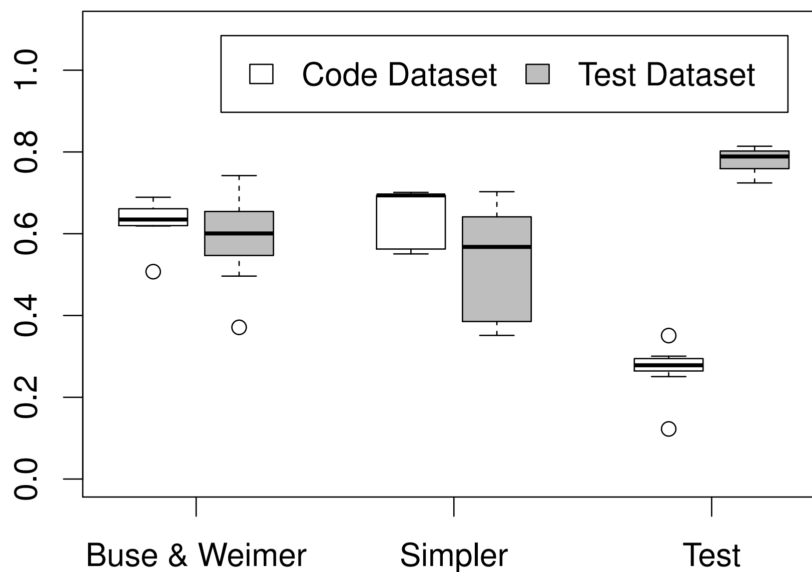


Fig. 4.13 10-fold cross-validation of code and test readability models using different learners and data sets.

regression models. We created two versions for each model, one trained with our dataset of 450 test cases with human annotations, and Buse and Weimer's original dataset of 100 code snippets with 1,200 human judgments. This allows us to distinguish between the effects of the choice of features and the training data.

Figure 4.13 shows the performance using seven different learners (Linear Regression, Multilayer Perceptron, SMOreg, M5Rules, M5P, Additive Regression, Bagging) on the *Buse and Weimer* code readability model, Posnett et al.'s *Simpler* model (which includes Halstead- and entropy-based features), and our *Test* readability model, in terms of the correlation using 10-fold cross validation. The Buse and Weimer and the Simpler models both perform better on code snippets than on test snippets. In contrast, our Test model shows a poor performance on the code dataset while achieving the overall highest correlation on the test dataset with a median value 0.79, which suggests that our choice of features is well adapted to the specific details influencing test readability. These results demonstrate the importance of our domain-specific model of software readability.

To compare these models with respect to their agreement with human judgement, we used the dataset of 200 pairs (Section 4.2.2) and measured the agreement between each model and the majority preference of the human annotators (i.e., percentage of matching choices). Table 4.2 shows the tests used to calculate the model–user agreement. Our test readability model trained with test case snippets achieves a high inter-rater agreement (kappa=0.73,

Table 4.2 Model prediction agreement with user choices

Each model is trained with two available datasets and tested with 200 pairs of test cases. Agreement shows the percentage of choices predicted by model agreeing with the overall user choice. Cohen's kappa [43] shows inter-rater agreement between the model prediction and user choices.

	Code dataset			Test dataset		
Model	Cohen's kappa	*p-value*	Agreement	Cohen's kappa	*p-value*	Agreement
Buse & Weimer	0.37	<0.01	0.685	0.33	<0.01	0.665
Simpler	0.57	<0.01	0.790	0.38	<0.01	0.695
Test Readability	0.31	<0.01	0.655	0.73	0	0.865

p-value=0), with a correctness ratio of 0.865 over 200 test pairs. As the Table shows, our test readability model significantly outperforms previous code readability models at this task.

RQ1: Our test readability model performs better on test snippet datasets, achieving a higher agreement with human annotators than previous work (kappa +28%).

4.4.3 RQ2: Improved Test Generation

To evaluate the success of the test optimization technique, we applied test generation to 30 classes with 10 repetitions each, with and without optimization, and compared the tests per branch in terms of their readability score. That is, each pair of tests is generated for the same coverage objective, but differs in readability score. We find that 56% of test cases on which optimization was applied had at least one alternative to choose from; for these tests, on average there were 5.1 alternatives. Table 4.3 summarizes the overall results: On all but three classes there is a significant increase in the readability score, by an average of 1.9% (2.5% when considering only tests that had alternatives). This increase may seem small, but recall that readability scores have low variance (cf. the narrow range of values between 3.0 and 4.0 in Figure 4.1), and the syntactic differences for improvement steps are small and incremental.

The largest improvement is observable for classes where the largest numbers of alternatives can be generated. As our algorithm (see Algorithm 3) is based on varying method and constructor calls, generation of alternatives works best when the dependency classes have many different constructors and parameters. Furthermore, we observe that more alternatives are generated for classes with more and simpler methods. For example, `math3.complex.Complex` has 196 branches, but these are spread over 48 methods,

Table 4.3 Readability value for the 30 classes selected based on 10 runs per branch.

For each class we report its version, the number of branches, and the readability value using *default* configuration. For the *optimized* configuration we also report its readability value, the effect sizes ($\hat{A}_{12}$ and relative average improvement) compared to the *default* configuration, and average number of optimized alternatives generated. Effect sizes $\hat{A}_{12}$ that are statistically significant are reported in bold.

Class	Version	Branches	Readability Default	Readability Optimized	$\hat{A}_{12}$	Relative Improvement	Average Number of Alternatives
java2.util2.BitSet	-	288	3.8701	3.9091	**0.54**	+1.0%	2.03
net.n3.nanoxml.StdXMLReader	2.2.1	96	3.4257	3.4347	**0.52**	+0.2%	1.14
net.n3.nanoxml.XMLElement	2.2.1	165	3.6655	3.7376	**0.70**	+1.9%	9.72
net.xineo.xml.handler.ObjectHandlerAdapter	1.1.0	23	3.4762	3.5492	**0.62**	+2.0%	17.42
nu.xom.Attribute	1.2.10	201	3.7734	3.7882	**0.63**	+0.3%	9.79
org.apache.commons.beanutils.locale.converters.DateLocaleConverter	1.9.2	51	3.6238	3.8021	**0.78**	+4.9%	64.03
org.apache.commons.chain.impl.ChainBase	1.2	33	3.6404	3.6754	**0.56**	+0.9%	25.62
org.apache.commons.cli.CommandLine	1.2	48	3.6985	3.7176	**0.57**	+0.5%	0.68
org.apache.commons.cli.Option	1.2	97	3.7459	3.7639	**0.54**	+0.4%	0.79
org.apache.commons.cli.PosixParser	1.2	46	3.5414	3.5912	**0.64**	+1.4%	20.62
org.apache.commons.codec.language.Metaphone	1.1	211	3.7523	3.7539	0.50	+0.0%	0.0044
org.apache.commons.codec.language.Soundex	1.1	40	3.7959	3.8312	**0.65**	+0.9%	1.74
org.apache.commons.collections4.comparators.ComparatorChain	4	52	3.4294	3.5163	**0.68**	+2.5%	11.85
org.apache.commons.collections4.comparators.FixedOrderComparator	4	73	3.1959	3.2772	**0.62**	+2.5%	11.01
org.apache.commons.collections4.iterators.FilterIterator	4	21	3.7577	3.8228	**0.65**	+1.7%	3.12
org.apache.commons.collections4.iterators.FilterListIterator	4	51	3.5880	3.6604	**0.60**	+2.0%	3.92
org.apache.commons.collections.primitives.ArrayIntList	1	28	3.7554	3.7707	0.52	+0.4%	1.56
org.apache.commons.configuration.tree.MergeCombiner	1.1	29	3.2433	3.3241	**0.67**	+2.4%	10.66
org.apache.commons.digester3.plugins.PluginRules	3.2	47	3.7284	3.7878	**0.56**	+1.5%	8.49
org.apache.commons.digester3.RulesBase	3.2	39	3.5806	3.6379	**0.60**	+1.6%	14.79
org.apache.commons.lang3.CharRange	3.3.2	58	3.9201	3.9363	**0.62**	+0.4%	4.80
org.apache.commons.math3.complex.Complex	3.4.1	196	3.6501	3.9748	**0.83**	+8.8%	91.89
org.apache.commons.math3.fraction.Fraction	3.4.1	118	3.7660	3.8892	**0.74**	+3.2%	34.66
org.apache.commons.math3.genetics.ListPopulation	3.4.1	25	3.6245	3.6353	**0.53**	+0.2%	1.26
org.apache.commons.math3.stat.clustering.DBSCANClusterer	3.4.1	32	3.1619	3.1763	0.53	+0.4%	2.63
org.jdom2.Attribute	2.0.5	74	3.7439	3.8507	**0.71**	+2.8%	41.93
org.jdom2.DocType	2.0.5	21	3.8127	3.9344	**0.67**	+3.1%	9.62
org.joda.time.Months	2.7	69	3.8716	3.9982	**0.68**	+3.2%	39.38
org.joda.time.YearMonthDay	2.7	71	3.5456	3.6278	**0.68**	+2.3%	32.69
org.magee.math.Rational	2005-11-19	36	3.7897	3.9391	**0.85**	+3.9%	9.84
Average			3.6953	3.7284	0.63	+1.9%	5.09

and this results in the overall largest readability improvement (+8.8%), with 92 alternatives per test on average. Similar examples are the class `org.joda.time.Months` and `beanutils.locale.converters.DateLocaleConverter`. The first one has five constructors plus many methods that also return `Months` instances, and the second has twelve different constructors. This leads to a large number of alternatives (39 for `Months` and 64 for `DateLocaleConverter`), and readability improvements of +3.2% and +4.9%, respectively.

On the other hand, class `cli.Option` has three constructors and methods that mainly take primitive values as parameters, and thus results in just 0.79 alternative tests on average (with a significant readability improvement of +0.4%). An extreme example is given by class `codec.language.Metaphone`, where the number of alternatives is close to 0.0 (see Table 4.3). `codec.language.Metaphone` is one of the largest classes, with 211 branches, but 184 of those branches are in the same method `public String metaphone(String txt){...}`, which receives and returns a String object. This arises because there is no other method for creating a Metaphone string. For this specific class, a better way to optimize readability may be by directly optimizing the strings using a language model [5].

Table 4.4 Readability value for the 30 classes selected based on the top three pairs that maximize the difference between *default* and *optimized* configurations.

For each class and pair we report the readability value of each configuration and the percentage of users that agree *optimized* test cases are better in terms of readability.

	Pair 1			Pair 2			Pair 3			
	Readability			Readability			Readability			Average
Class	Default	Optimized	Agree	Default	Optimized	Agree	Default	Optimized	Agree	Agree
java2.util2.BitSet	3.78	3.92	71.25%	3.73	3.88	62.79%	3.83	3.90	60.71%	64.92%
net.n3.nanoxml.StdXMLReader	3.15	3.87	70.00%	3.17	3.69	67.02%	3.34	3.81	69.44%	68.82%
net.n3.nanoxml.XMLElement	3.34	3.80	87.18%	3.33	3.73	73.00%	3.40	3.78	63.00%	74.39%
net.xineo.xml.handler.ObjectHandlerAdapter	3.48	3.78	71.95%	3.39	3.67	70.45%	3.22	3.56	78.41%	73.60%
nu.xom.Attribute	3.33	3.81	69.57%	3.24	3.83	75.00%	3.24	3.75	61.96%	68.84%
org.apache.commons.beanutils.locale.converters.DateLocaleConverter	3.22	3.79	81.25%	3.34	3.85	65.69%	3.38	3.74	68.09%	71.67%
org.apache.commons.chain.impl.ChainBase	2.99	3.75	74.39%	3.27	3.86	73.81%	3.31	3.86	68.57%	72.26%
org.apache.commons.cli.CommandLine	3.34	3.75	75.00%	3.49	3.82	82.14%	3.41	3.76	69.74%	75.63%
org.apache.commons.cli.Option	3.48	4.01	70.00%	3.53	3.87	68.18%	3.59	3.92	57.14%	65.11%
org.apache.commons.cli.PosixParser	3.39	3.68	67.39%	3.35	3.72	73.81%	3.49	3.73	63.41%	68.21%
org.apache.commons.codec.language.Metaphone	3.59	3.88	77.55%	3.61	3.87	72.34%	3.67	3.82	64.58%	71.49%
org.apache.commons.codec.language.Soundex	3.76	3.92	51.96%	3.63	3.92	68.89%	3.60	3.89	75.00%	65.28%
org.apache.commons.collections4.comparators.ComparatorChain	3.27	3.92	72.22%	3.36	3.79	69.44%	3.51	3.92	66.67%	69.44%
org.apache.commons.collections4.comparators.FixedOrderComparator	2.70	3.34	69.44%	2.80	3.44	72.34%	2.63	3.12	74.42%	72.07%
org.apache.commons.collections4.iterators.FilterIterator	3.56	3.99	71.62%	3.33	3.83	67.35%	3.64	3.95	65.85%	68.27%
org.apache.commons.collections4.iterators.FilterListIterator	3.36	3.84	80.21%	3.08	3.69	84.21%	3.12	3.69	81.71%	82.04%
org.apache.commons.collections.primitives.ArrayIntList	3.35	3.72	70.41%	3.79	3.94	53.06%	3.58	3.79	66.00%	63.16%
org.apache.commons.configuration.tree.MergeCombiner	3.32	3.68	73.17%	3.28	3.55	72.50%	3.32	3.59	55.41%	67.03%
org.apache.commons.digester3.plugins.PluginRules	2.49	3.46	67.44%	3.09	3.73	57.69%	3.08	3.64	61.54%	62.22%
org.apache.commons.digester3.RulesBase	3.23	3.88	78.57%	3.13	3.71	76.60%	3.17	3.73	67.59%	74.25%
org.apache.commons.lang3.CharRange	3.82	4.04	77.27%	3.80	3.97	63.27%	3.91	4.05	76.19%	72.24%
org.apache.commons.math3.complex.Complex	3.39	3.97	78.05%	3.38	3.94	60.47%	3.54	4.01	68.18%	68.90%
org.apache.commons.math3.fraction.Fraction	2.98	3.60	63.04%	3.65	3.83	66.67%	3.53	3.88	75.53%	68.41%
org.apache.commons.math3.genetics.ListPopulation	3.43	3.83	75.00%	3.44	3.75	67.05%	3.48	3.83	72.34%	71.46%
org.apache.commons.math3.stat.clustering.DBSCANClusterer	3.28	3.65	65.91%	3.49	3.65	67.44%	3.13	3.36	76.09%	69.81%
org.jdom2.Attribute	3.57	3.86	73.40%	3.78	3.86	73.33%	3.66	3.86	63.00%	69.91%
org.jdom2.DocType	3.49	3.75	73.86%	3.71	3.85	57.45%	3.57	3.85	67.86%	66.39%
org.joda.time.Months	3.35	4.05	64.00%	3.44	4.02	61.70%	3.46	3.99	62.16%	62.62%
org.joda.time.YearMonthDay	3.25	3.81	78.95%	3.32	3.81	71.11%	3.15	3.70	75.00%	75.02%
org.magee.math.Rational	3.86	3.87	61.54%	3.68	3.95	45.00%	3.67	3.83	50.00%	52.18%
Average										69.19%

RQ2: Alternatives were generated for 56% of the unit tests, resulting in a readability improvement of +1.9% on average.

4.4.4 RQ3: Do Humans Prefer Readability Optimized Tests?

To evaluate whether humans prefer the readability optimized test cases to the default tests generated by EvoSuite, we applied test generation to 30 classes, with and without optimization. For each class we chose the three pairs of tests with the largest difference in readability score, and used a forced-choice survey to let human annotators select which test cases they think are more readable (see Figure 4.17). That is, for each pair, both tests cover the same branch of the same class, but differ only in their readability score. Table 4.4 shows the details of the pairs used for this experiment, and shows the joint probability of agreement, which is 69% overall. On average, for all 30 classes there is a preference for the optimized tests. The average pair-wise agreement (fraction of pairs rated by both raters on which they agree) is 0.58; a one-sided Wilcoxon signed-rank shows that this is significantly better ($p < 0.001$) than a random choice (assuming agreement of 0.5 for random choices).

The highest agreement is observed for the first pair for class `net.n3.nanoxml.XMLElement` (87%). Here, the optimized test expects an exception and has only three statements. In contrast, the default test has four statements and six assertions, expects no exception, and uses random strings (e.g., “`7I%d7W5Y(Ta+`”) (Figure 4.14).

```
XMLElement xMLElement0 = new XMLElement(" 7I%d7W5Y(Ta+", "!P8/tp(", " 7I%d7W5Y(Ta+",
     1458);
xMLElement0.setAttribute(" 7I%d7W5Y(Ta+", "!P8/tp(", " 7I%d7W5Y(Ta+");
String string0 = xMLElement0.getAttributeType("!P8/tp(");
assertNull(string0);

String string1 = xMLElement0.getAttribute(" 7I%d7W5Y(Ta+", (String) null, "!P8/tp("
    );
assertEquals(" 7I%d7W5Y(Ta+", xMLElement0.getName());
assertEquals("!P8/tp(", string1);
assertEquals(1458, xMLElement0.getLineNr());
assertEquals("!P8/tp(", xMLElement0.getNamespace());
assertEquals(" 7I%d7W5Y(Ta+", xMLElement0.getFullName());
```

```
XMLElement xMLElement0 = new XMLElement(" ", " ", " ", 407);
xMLElement0.setAttribute(" ", " ");
// Undeclared exception!
try {
 int int0 = xMLElement0.getAttribute(" ", (String) null, 407);
 fail("Expecting exception: NumberFormatException");

} catch(NumberFormatException e) {
 //
 // For input string: \" \"
 //
}
```

Fig. 4.14 Test cases generated without and with readability optimization for class XMLElement

In class `org.magee.math.Rational` there are two pairs for which the users prefer the default test case. For pair 2 (Figure 4.15), the optimized test expects a `NullPointerException`, and we have generally observed that exceptional tests tend to get slightly higher readability scores. However, in this case the users disagree, possibly influenced by the rather meaningless comment in the catch block stating that there is “no message”. For pair 3, the tests are very similar, and it is possible that domain knowledge influences the choice: The default test

```
Rational rational0 = new Rational((long) (-576), (long) (-576));
Rational rational1 = rational0.divide(rational0);
assertEquals(1.0F, rational0.floatValue(), 0.01F);
assertEquals("1 / 1", rational1.toString());
```

```
Rational rational0 = new Rational((-1L), (-1L));
// Undeclared exception!
try {
 Rational rational1 = rational0.divide((Rational) null);
 fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
 //
 // no message in exception (getMessage() returned null)
 //
}
```

Fig. 4.15 Second test pair generated without and with readability optimization for class Rational

subtracts a number from itself, whereas the optimized test performs a syntactically similar, but mathematically slightly more complex calculation (Figure 4.16.).

RQ3: Our experiment showed an agreement of 69% between human annotators and the readability optimization.

```
Rational rational0 = new Rational((-1042L), (-1042L));
Rational rational1 = rational0.subtract(rational0);
assertEquals(1.0, rational0.doubleValue(), 0.01D);
assertEquals("0 / 1085764", rational1.toString());
```

```
Rational rational0 = new Rational((-1005L), (-1005L));
Rational rational1 = rational0.subtract((-1005L));
assertEquals("1006 / 1", rational1.toString());
```

Fig. 4.16 Third test pair generated without and with readability optimization for class Rational

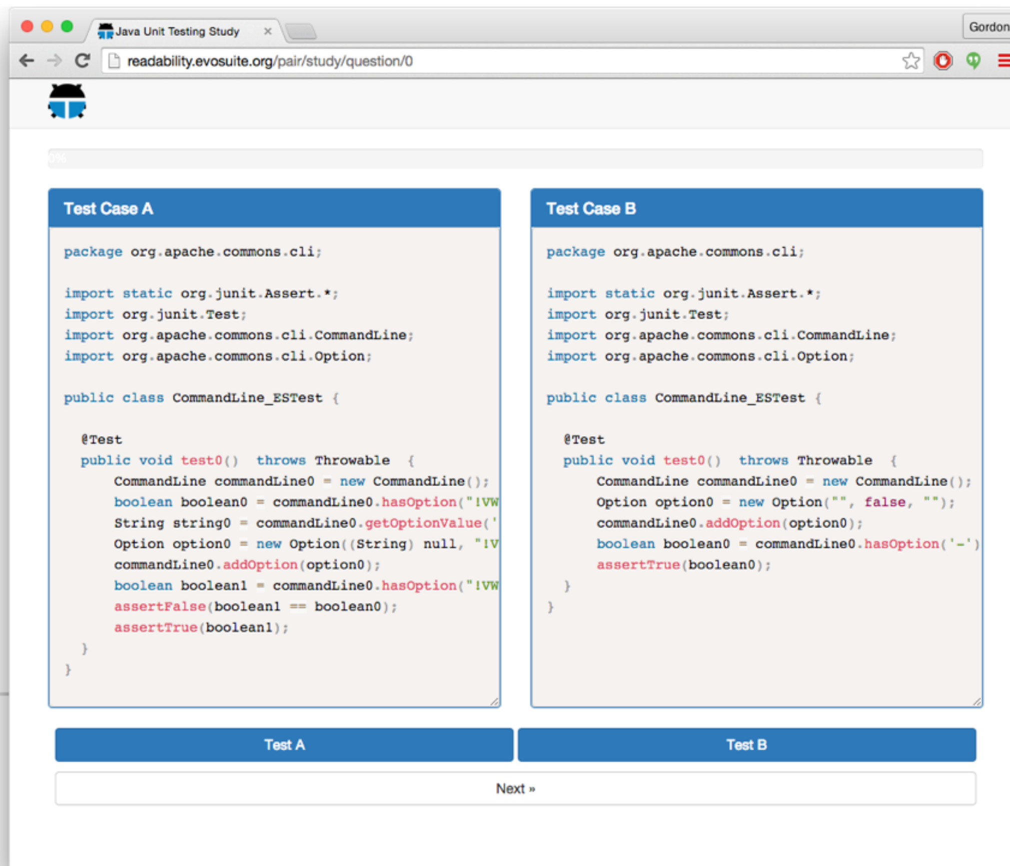


Fig. 4.17 Example question of agreement in empirical study

4.4.5 RQ4: Does Readability Optimization Improve Human Understanding of Tests?

Table 4.5 summarizes the results of the controlled experiment to answer RQ4. For seven out of the ten classes, the time participants required to make a decision about the pass/fail status of a test was lower for the optimized tests (see Figure 4.27). The average time spent on the non-optimized tests was 4.7 minutes, compared with 4 minutes for the optimized tests. Overall, this suggests that improved readability helps when making this software maintenance decision.

On the other hand, there are five classes where the ratio of correct responses (developer correctly determines that the test should pass or fail) increases, and five where the ratio decreases, suggesting that there are other factors influencing the difficulty of understanding

Table 4.5 Human understanding results of tests for the 10 classes randomly selected.

For each class selected we report the branch covered, the test result of each individual (pass/fail), number of asserts or fail keywords, average time to answer, and percentage of correct responses. Effect sizes of statistically significant differences ($p < 0.05$) are shown in bold.

Class	Branch	Oracle		Readability		Assert/Fail		Time (min)			Correct Answers	
		Def.	Optim.	Def.	Optim.	Def.	Optim.	Def.	Optim.	$\hat{A}_{12}$	Def.	Optim.
net.n3.nanoxml.StdXMLReader	67	pass	fail	3.40	3.79	0 / 1	0 / 1	4.35	3.76	0.57	60.00%	61.11%
nu.xom.Attribute	91	pass	pass	3.33	3.81	0 / 1	0 / 1	4.69	5.75	0.43	60.00%	38.46%
org.apache.commons.chain.impl.ChainBase	5	pass	pass	2.99	3.75	1 / 0	1 / 0	4.04	3.80	0.61	76.47%	54.55%
org.apache.commons.cli.Option	67	fail	pass	3.55	4.01	5 / 0	0 / 1	2.21	2.69	0.34	100.00%	75.00%
org.apache.commons.collections4.comparators.FixedOrderComparator	22	pass	pass	2.63	3.30	3 / 0	2 / 0	4.92	3.31	0.70	64.29%	23.08%
org.apache.commons.collections4.iterators.FilterListIterator	4	fail	fail	3.10	3.73	2 / 0	1 / 0	4.75	3.00	**0.73**	71.43%	85.71%
org.apache.commons.digester3.plugins.PluginRules	25	pass	pass	2.49	3.46	1 / 0	1 / 0	6.43	6.04	0.51	72.73%	62.50%
org.apache.commons.digester3.RulesBase	3	pass	pass	2.80	3.76	1 / 0	1 / 0	4.77	3.24	0.68	90.91%	100.00%
org.apache.commons.lang3.CharRange	15	pass	pass	3.82	4.04	2 / 0	1 / 0	4.35	1.93	**0.92**	50.00%	100.00%
org.joda.time.YearMonthDay	66	pass	pass	3.25	3.81	3 / 0	0 / 1	6.06	6.49	0.48	30.77%	61.54%
Average				3.14	3.75			4.66	4.00	0.60	67.66%	66.19%

a test that are not captured by our readability model. For example, the tests for class `cli.Option` (see Figure 4.18) have a substantial difference in readability (3.55 default, 4.01 optimized), but the default one contains regular assertions, whereas the optimized one expects an exception to be thrown. While all responses for the non-optimized test were correct, only 75% of the responses for the optimized test were correct, and the average time for responses increased from 2.21min to 2.69min. The likely explanation for this is that, even though the readability model suggests that exceptions improve readability, it may be more difficult to understand exceptional control flow.

```
Option option0 = new Option("1P01Ca", "1P01Ca");
String string0 = option0.getDescription();
assertNull(string0);

Option option1 = new Option((String) null, " (", false, "1P01Ca");
String string1 = option1.getKey();
assertNull(string1);
assertEquals(-2, option1.getArgs());
assertEquals(false, option1.hasArgName());
assertEquals("[ option: null ( [ARG...] :: 1P01Ca ]", option1.toString());
```

```
Option option0 = new Option((String) null, " ");
// Undeclared exception!
try {
  int int0 = option0.getId();
  fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
  //
  // no message in exception (getMessage() returned null)
  //
}
```

Fig. 4.18 Default and optimized test case for class `cli.Option`.

This conjecture is supported by the tests for class `org.joda.time.YearMonthDay` (see Figure 4.19), where again the optimized test leads to an expected exception. Here, the time to response increases from 6.06min to 6.49min on average. However, in contrast to `cli.Option`, the percentage of correct responses increases by 31%. Possibly, this improvement is influenced by the error message of the expected exception, which is included as a comment: The optimized test calls the constructor of class `YearMonthDay(day, month, year)` with value 0 and the exception message was *"Value 0 for monthOfYear must not be smaller than 1"*.

```
BuddhistChronology buddhistChronology0 = BuddhistChronology.getInstance();
YearMonthDay yearMonthDay0 = new YearMonthDay((Chronology) buddhistChronology0);
YearMonthDay.Property yearMonthDay_Property0 = new YearMonthDay.Property(
    yearMonthDay0, 0);
YearMonthDay yearMonthDay1 = yearMonthDay_Property0.addWrapFieldToCopy(0);
assertEquals("2557-02-14", yearMonthDay1.toString());
assertEquals("2557-02-14", yearMonthDay0.toString());
assertEquals(-292268511, yearMonthDay_Property0.getMinimumValueOverall());
```

```
YearMonthDay yearMonthDay0 = null;
try {
 yearMonthDay0 = new YearMonthDay(0, 0, 0);
 fail("Expecting exception: IllegalFieldValueException");
} catch(IllegalFieldValueException e) {
  //
  // Value 0 for monthOfYear must not be smaller than 1
  //
}
```

Fig. 4.19 Default and optimized test case for class `org.joda.time.YearMonthDay`.

The third class with an increase in time, `nu.xom.Attribute` (see Figure 4.20), tests exceptional behavior in both versions. Here, the percentage of correct responses is only 39%, compared to 60% in the default version. This reduction may again be related to the specific error message, as the default test complains about the use of an *"Illegal initial scheme character"* in a URI parameter, which apparently is easier to understand than the *"Missing scheme in absolute URI reference"* message of the optimized test.

```
Attribute.Type attribute_Type0 = Attribute.Type.CDATA;
Attribute attribute0 = Attribute.build(" for the http://www.w3.rg/XML/1998/namesace
     namesace URI", " for the http://www.w3.rg/XML/1998/namesace namesace URI", "
    for the http://www.w3.rg/XML/1998/namesace namesace URI", attribute_Type0, "
    for the http://www.w3.rg/XML/1998/namesace namesace URI");
// Undeclared exception!
try {
 attribute0.setNamespace(" for the http://www.w3.rg/XML/1998/namesace namesace URI"
     , " for the http://www.w3.rg/XML/1998/namesace namesace URI");
 fail("Expecting exception: MalformedURIException");

} catch(MalformedURIException e) {
 //
 // Illegal initial scheme character
 //
}
```

```
Attribute attribute0 = null;
try {
 attribute0 = new Attribute(".Vx:A1B", "j", "j");
 fail("Expecting exception: MalformedURIException");

} catch(MalformedURIException e) {
 //
 // Missing scheme in absolute URI reference
 //
}
```

Fig. 4.20 Default and optimized test case for class `nu.xom.Attribute`.

Both default and optimized test expect a null pointer exception for class `net.n3.nanoxml.StdXMLReader`. Although the rate of correct responses is comparable, here the time spent on the optimized test is lower on average (4.35min default vs, 3.76min optimized). This is likely related to the difference in size; the default test has seven statements, the optimized one only two.

```
PipedReader pipedReader0 = new PipedReader();
PushbackReader pushbackReader0 = new PushbackReader((Reader) pipedReader0, 519);
LineNumberReader lineNumberReader0 = new LineNumberReader((Reader) pushbackReader0,
     519);
StdXMLReader stdXMLReader0 = new StdXMLReader((Reader) lineNumberReader0);
String string0 = stdXMLReader0.getEncoding("#|e(y");
StdXMLReader stdXMLReader1 = null;
try {
 stdXMLReader1 = new StdXMLReader((String) null, "#|e(y");
 fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
 //
 // no message in exception (getMessage() returned null)
 //
}
```

```
StdXMLReader stdXMLReader0 = null;
try {
 stdXMLReader0 = new StdXMLReader("", "#");
 fail("Expecting exception: NullPointerException");

} catch(NullPointerException e) {
 //
 // no message in exception (getMessage() returned null)
 //
}
```

Fig. 4.21 Default and optimized test case for `net.n3.nanoxml.StdXMLReader`.

For `comparators.FixedOrderComparator` (see Figure 4.22), the reduction of correct responses may be a result of uncertainty arising from how null-values are handled in the maps underlying the class; the default test case does not use null, the optimized one does. The same may also hold for `digester3.plugins.PluginRules` (see Figure 4.23), where the optimized test case uses several null values.

```
FixedOrderComparator.UnknownObjectBehavior[]
    fixedOrderComparator_UnknownObjectBehaviorArray0 = FixedOrderComparator.
    UnknownObjectBehavior.values();
LinkedList<FixedOrderComparator<Object>> linkedList0 = new LinkedList<
    FixedOrderComparator<Object>>();
FixedOrderComparator<FixedOrderComparator<Object>> fixedOrderComparator0 = new
    FixedOrderComparator<FixedOrderComparator<Object>>((List<FixedOrderComparator<
    Object>>) linkedList0);
LinkedList<Object> linkedList1 = new LinkedList<Object>();
FixedOrderComparator<Object> fixedOrderComparator1 = new FixedOrderComparator<
    Object>((List<Object>) linkedList1);
boolean boolean0 = fixedOrderComparator0.add(fixedOrderComparator1);
FixedOrderComparator<Object> fixedOrderComparator2 = new FixedOrderComparator<
    Object>((Object[]) fixedOrderComparator_UnknownObjectBehaviorArray0);
boolean boolean1 = fixedOrderComparator0.addAsEqual(fixedOrderComparator1,
    fixedOrderComparator2);
assertEquals(FixedOrderComparator.UnknownObjectBehavior.EXCEPTION,
    fixedOrderComparator2.getUnknownObjectBehavior());
assertEquals(FixedOrderComparator.UnknownObjectBehavior.EXCEPTION,
    fixedOrderComparator1.getUnknownObjectBehavior());
assertTrue(boolean1);
```

```
String[] stringArray0 = new String[11];
FixedOrderComparator<String> fixedOrderComparator0 = new FixedOrderComparator<
    String>(stringArray0);
boolean boolean0 = fixedOrderComparator0.addAsEqual((String) null, " not known to
    ");
assertEquals(FixedOrderComparator.UnknownObjectBehavior.EXCEPTION,
    fixedOrderComparator0.getUnknownObjectBehavior());
assertTrue(boolean0);
```

Fig. 4.22 Default and optimized test case for `comparators.FixedOrderComparator`.

```
FactoryCreate.DefaultObjectCreationFactory
    factoryCreate_DefaultObjectCreationFactory0 = new FactoryCreate.
    DefaultObjectCreationFactory();
Digester digester0 = factoryCreate_DefaultObjectCreationFactory0.getDigester();
PluginRules pluginRules0 = new PluginRules();
Class<SetPropertiesRule> class0 = SetPropertiesRule.class;
Attributes2Impl attributes2Impl0 = new Attributes2Impl();
PluginRules pluginRules1 = new PluginRules((Digester) null, "/^ tYLy){^^<\fmXKHo",
    pluginRules0, class0);
List<Rule> list0 = pluginRules1.match("%s[methodName=%s, paramType=%s,
    paramTypeName=%s, useExactMatch=%s, fireOnBegin=%s]", "%s[methodName=%s,
    paramType=%s, paramTypeName=%s, useExactMatch=%s, fireOnBegin=%s]", "%s[
    methodName=%s, paramType=%s, paramTypeName=%s, useExactMatch=%s, fireOnBegin=%s
    ]", (Attributes) attributes2Impl0);
assertEquals(0, list0.size());
```

```
PluginRules pluginRules0 = new PluginRules((Rules) null);
Class<String> class0 = String.class;
PluginRules pluginRules1 = new PluginRules((Digester) null, "", pluginRules0,
    class0);
AttributesImpl attributesImpl0 = new AttributesImpl();
List<Rule> list0 = pluginRules1.match(", rule type: ", ", rule type: ", (String)
    null, (Attributes) attributesImpl0);
assertEquals(true, list0.isEmpty());
```

Fig. 4.23 Default and optimized test case for class `digester3.plugins.PluginRules`.

These results are reflected by the participants' opinions. In free response questions at the end of our survey, almost every user stated that a readable test case must be (1) minimal (no more than 5 lines if possible); (2) have short lines; (3) not dependent on too many classes, as for example indicated by the comment that "due to deep inheritance and lots of underlying methods to construction, it was somewhat hard to understand some classes under test."

For `chain.impl.ChainBase` the slight reduction in correct responses is surprising, as the optimized test case has a trivially true assertion (`assertNotSame` with two objects resulting from two different constructor invocations; cf. Figure 4.24). However, this assertion form is less common and might have been interpreted by several participants as the more common `assertNotEquals`, which may contribute to the increase in wrong responses.

```
CatalogFactoryBase catalogFactoryBase0 = (CatalogFactoryBase)CatalogFactory.
    getInstance();
DispatchLookupCommand dispatchLookupCommand0 = new DispatchLookupCommand((
    CatalogFactory) catalogFactoryBase0);
HashMap<Object, CopyCommand> hashMap0 = new HashMap<Object, CopyCommand>();
ContextBase contextBase0 = new ContextBase((Map) hashMap0);
Set set0 = contextBase0.keySet();
ChainBase chainBase0 = new ChainBase((Collection) set0);
RemoveCommand removeCommand0 = new RemoveCommand();
Command[] commandArray0 = new Command[4];
commandArray0[0] = (Command) removeCommand0;
commandArray0[1] = (Command) dispatchLookupCommand0;
commandArray0[2] = (Command) dispatchLookupCommand0;
commandArray0[3] = (Command) chainBase0;
ChainBase chainBase1 = new ChainBase(commandArray0);
assertFalse(chainBase1.equals((Object)chainBase0));
```

```
ChainBase chainBase0 = new ChainBase();
Command[] commandArray0 = chainBase0.getCommands();
ChainBase chainBase1 = new ChainBase(commandArray0);
assertNotSame(chainBase1, chainBase0);
```

Fig. 4.24 Default and optimized test case for class `chain.impl.ChainBase`.

Finally, for `FilterListIterator` (see Figure 4.25) and `CharRange` (see Figure 4.26) there are large improvements in the response time and in the correctness of responses. For `FilterListIterator` the default test case uses a confusing chain of calls to construct the instance of the class under test; for the `CharRange` default test case this also holds, but here maybe a different factor also plays a role. As we can see in Figure 4.26 the default test case starts by constructing a negated `CharRange` over a single character "#", and later checks whether a different character is contained in the `CharRange`, whereas the optimized test case uses the same character twice, when setting up the `CharRange` and when querying `contains`.

RQ4: In our experiments, the optimized tests reduced the response time by 14%, but did not directly influence participant accuracy.

```
Class<Object> class0 = Object.class;
InstanceofPredicate instanceofPredicate0 = new InstanceofPredicate(class0);
NullIsTruePredicate<Object> nullIsTruePredicate0 = new NullIsTruePredicate<Object
    >((Predicate<? super Object>) instanceofPredicate0);
NullIsExceptionPredicate<Boolean> nullIsExceptionPredicate0 = new
    NullIsExceptionPredicate<Boolean>((Predicate<? super Boolean>)
    nullIsTruePredicate0);
FilterListIterator<Boolean> filterListIterator0 = new FilterListIterator<Boolean>((
    Predicate<? super Boolean>) nullIsExceptionPredicate0);
assertEquals(false, filterListIterator0.hasNext());

ListIterator<? extends Boolean> listIterator0 = filterListIterator0.getListIterator
    ();
assertEquals(0, filterListIterator0.previousIndex());
```

```
FilterListIterator<Integer> filterListIterator0 = new FilterListIterator<Integer>()
    ;
ListIterator<? extends Integer> listIterator0 = filterListIterator0.getListIterator
    ();
assertEquals(1, filterListIterator0.nextIndex());
```

Fig. 4.25 Default and optimized test case for class `FilterListIterator`.

```
CharRange charRange0 = CharRange.isNot('#');
Character character0 = Character.valueOf('#');
CharRange charRange1 = CharRange.isNotIn('\"', (char) character0);
char char0 = charRange1.getStart();
assertEquals('\"', char0);

boolean boolean0 = charRange0.contains('\"');
assertTrue(boolean0);
```

```
CharRange charRange0 = CharRange.is(' ');
boolean boolean0 = charRange0.contains(' ');
assertTrue(boolean0);
```

Fig. 4.26 Default and optimized test case for `lang3.CharRange`.

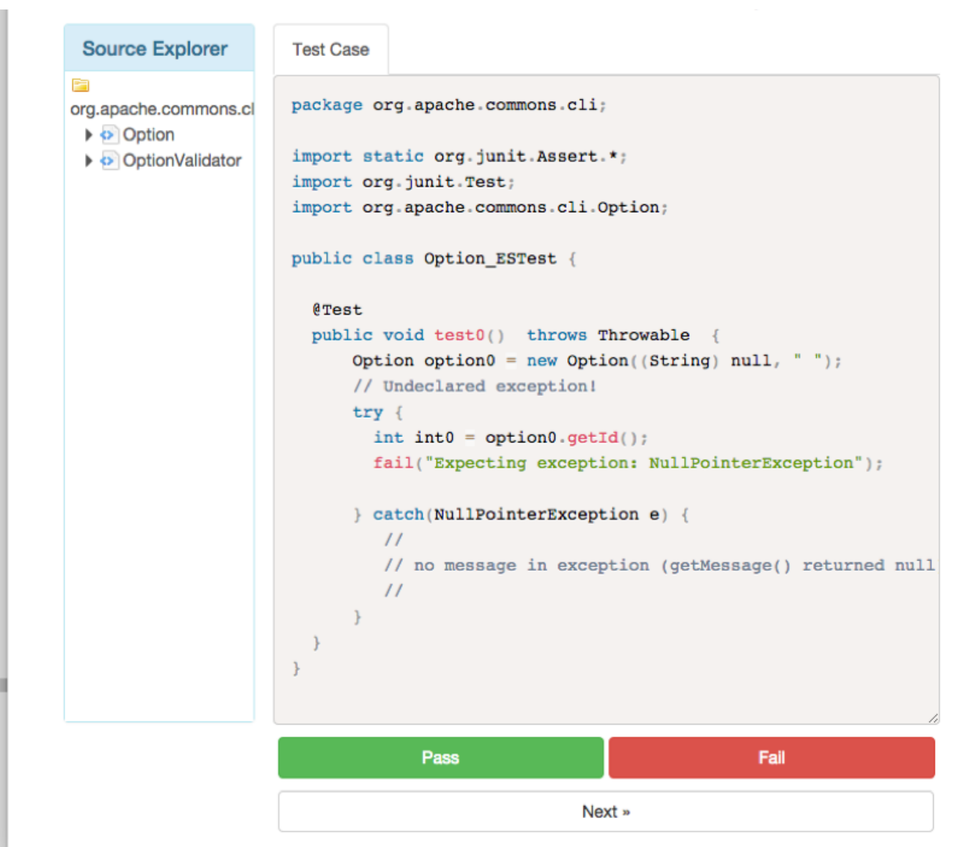


Fig. 4.27 Example question of understanding in empirical study

4.5 Generating Readable Tests for Guava

To study in detail the test readability model, we selected Guava library classes and generate test cases. EVOSUITE [70] uses a genetic algorithm to evolve individual unit tests or sets of unit tests, typically with the aim to maximize code coverage of a chosen test criterion. Over the time, we have collected ample anecdotal evidence of aspects developers disliked about the automatically-generated tests, and EVOSUITE now by default applies a range of different optimizations to the tests generated by the genetic algorithm. For example, redundant statements in the sequences of statements are removed, numerical and string values are minimized, unnecessary variables are removed, etc. However, the readability of a test may

be an effect of the particular choice of parameters and calls, such that only generating a completely different test, rather than optimizing an existing one, would maximize readability.

To integrate the unit test readability model into EVOSUITE, we explored the following approaches: (1) As code coverage remains a primary objective for the test generation, the readability model can be integrated as a secondary objective. If two individuals of the population have the same fitness value, during rank selection the one with the better readability value is preferred. (2) Because readability and code coverage may be conflicting goals (e.g., adding a statement may improve coverage, but decrease readability), classical multi-objective algorithms (e.g., NSGA-II [63]) can be used with coverage and readability as independent objectives.

However, EVOSUITE's post-processing steps may complicate initial readability judgments: An individual that seems unreadable may become more readable through the post-processing (and vice-versa). Therefore, we consider the following solutions: (1) Measure the readability of tests not on the search individuals, but on the result of the post-processing steps. That is, the fitness value is measured in the style of Baldwinian optimization [196] on the improved phenotype, without changing the genotype. This can be applied to the scenario of a secondary objective as well as to multi-objective optimization. (2) Optimize readability as another post-processing step, using an algorithm that generates alternative candidates and ranks them by readability [56].

4.5.1 Generating Readability Optimized Tests

After selecting five Guava classes randomly, and generated tests with both approaches mentioned in previous section, Figure 4.28 summarizes the overall results (over 5 runs) in terms of the modeled readability scores for these five different classes with and without optimization with both post-processing and no post-processing techniques. Furthermore, the readability values of the manually-written tests for these classes are included for reference.

The first three boxes of each plot show substantial improvement over the default configuration by including the readability model as a secondary objective or as a second fitness function. In all five classes, the multi-objective optimization achieves the most readable tests. However, note that without post-processing, these tests do not yet have assertions (which according to the model have a negative effect on readability). Despite this, in all five classes the average readability of the manually written tests (fourth box) is slightly higher.

Boxes 5–7 show a similar pattern when applying EVOSUITE's post-processing steps. However, we can see the large effects EVOSUITE's many post-processing steps have, as the generated tests approach the readability of manual tests. For `Splitter` and `DoubleMath` the improvement over the default is significant at $\alpha = 0.05$ (calculated by using Wilcoxon test),

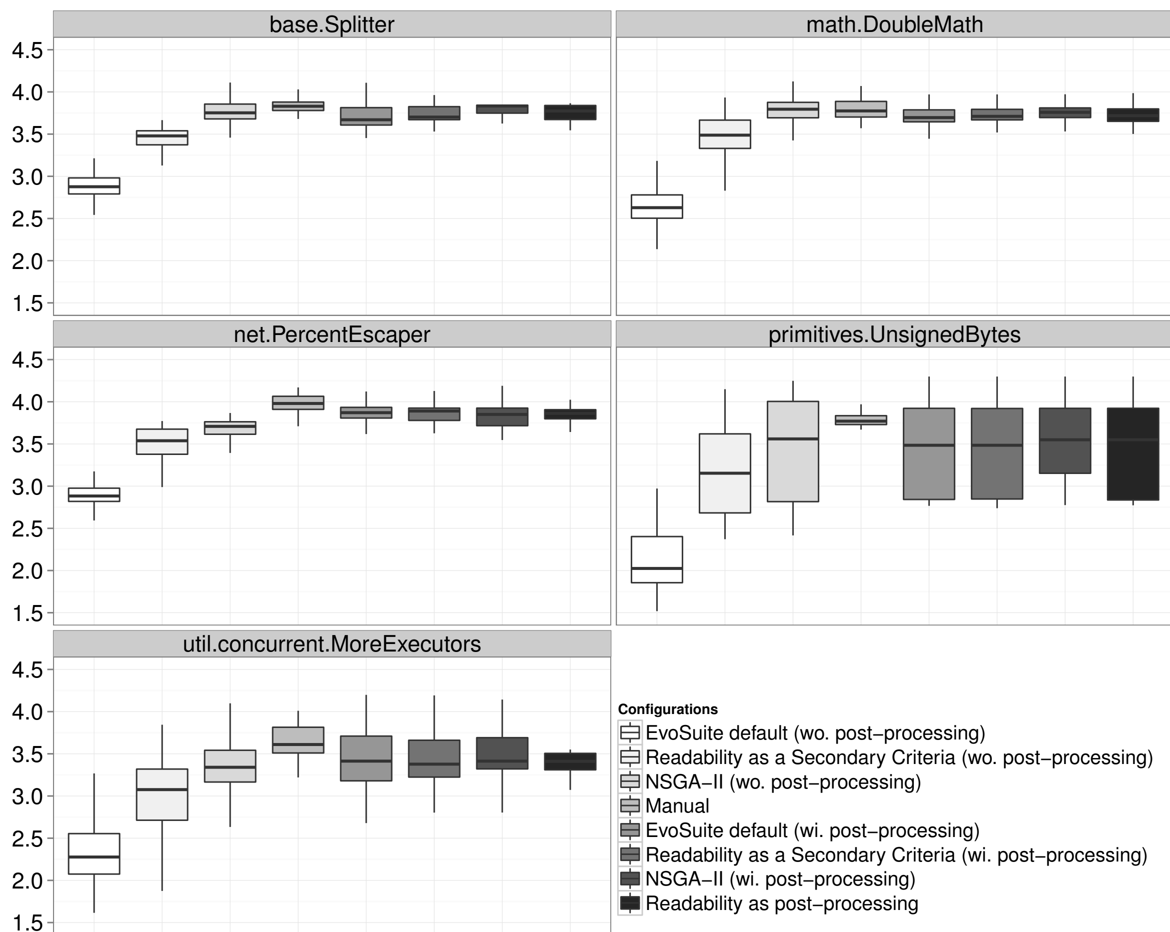
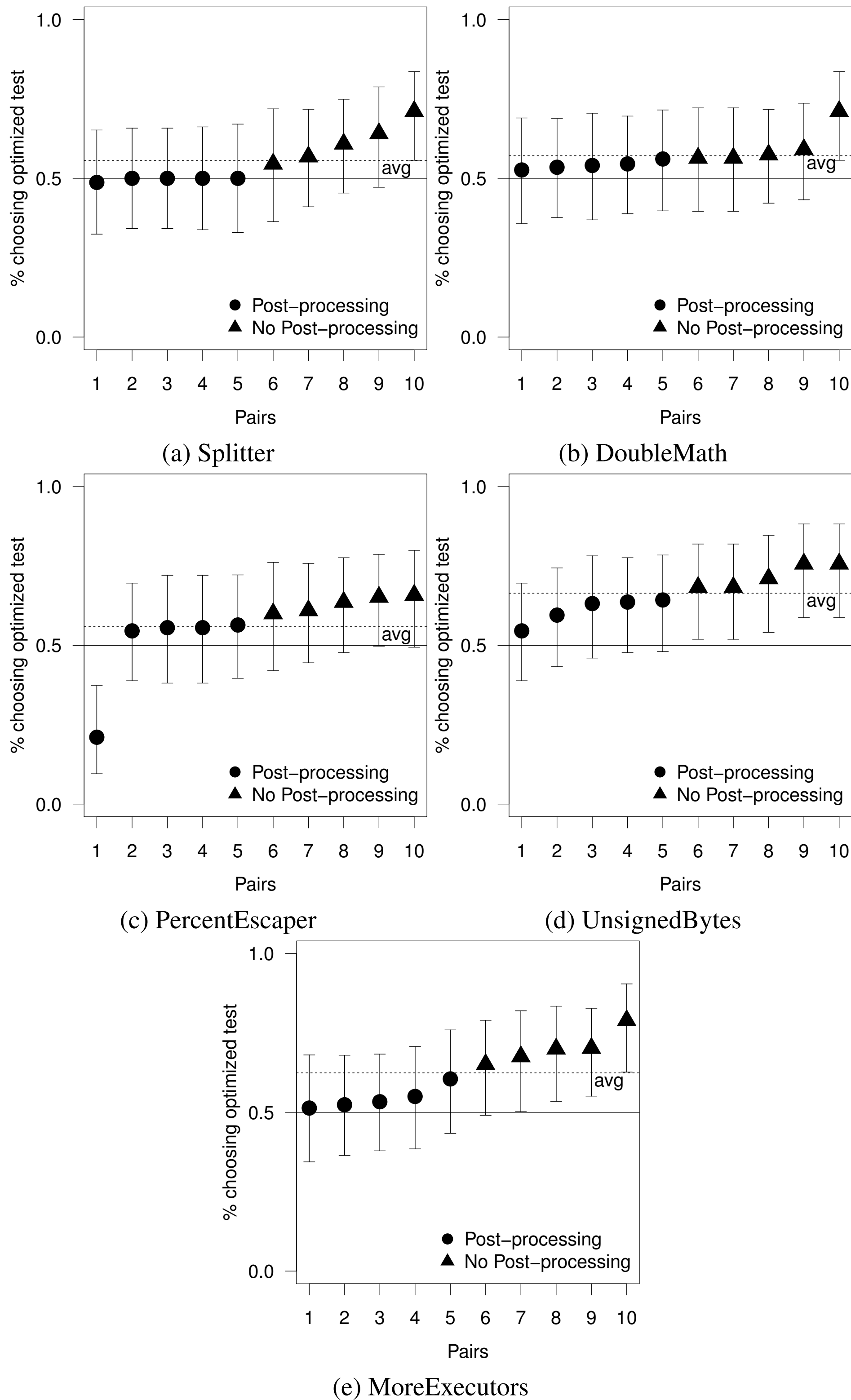


Fig. 4.28 Readability scores of manually-written and automatically-generated test cases in 7 different configurations.

on `UnsignedBytes` and `MoreExecutors` there is an improvement although not significant. On `PercentEscaper` there is no significant difference. The final box shows the results of a post-processing step driven by the readablity model, which is generally slightly below NSGA-II, but on the other hand is computationally much cheaper. We note that using the readability model in a post-processing step is generally on par with the multi-objective optimization.

These results demonstrate that our search-based approach can produce test cases that are competitive with manual tests in terms of modeled readability.



(a) Splitter (b) DoubleMath (c) PercentEscaper (d) UnsignedBytes (e) MoreExecutors

Fig. 4.29 Percentage of users preferring the optimized test cases

4.5.1.1 User Agreement

To validate whether users agree with these optimizations, we selected 50 pairs of test cases for the selected 5 classes, where the tests in each pair cover the same coverage objective, and each pair consists of one test generated using EVOSUITE's default configuration, whereas the other one is optimized. Half of the pairs were selected from the configurations that do use post-processing, and half from the configurations that do not. We used Amazon Mechanical Turk to run a forced-choice survey, showing a random subset of pairs to each participant. As when building the model, participants were required to pass a Java qualification test.

Figure 4.29 summarizes the 2,250 responses we received from 79 different participants. The error bars indicate the 95% confidence interval around the rate at which the participants preferred the optimized test. Overall, the participants preferred the optimized test 59% of the time ($p < 0.01$ calculated with Fleiss' kappa test). In four of the classes (`Splitter`, `DoubleMath`, `UnsignedBytes`, and `MoreExecutors`) we can see this preference at the level of individual tests. For example, we have 95% confidence that participants preferred the `UnsignedBytes` tests generated without post-processing at a rate higher than random chance. For pairs generated without post-processing the preference is generally clearer than for those with post-processing, where for these classes the difference in readability is generally small.

Notably, there is one pair of tests for class `PercentEscaper` where the users preferred the default version to the one optimized using the readability model. The model predicts that the shorter, optimized test is preferable to the longer test produced by EVOSUITE's default configuration, which contains an exception. However, this exception has a clear and easily to interpret message shown in a comment in the test, which the users seem to count as readable — which is not something a syntactic readability model could do.

Although human preference for our tests is modest, it is present, and our readability improvements are orthogonal to the structural coverage of the generated test suite.

4.5.1.2 Test Suite Generation

To see how results generalize, we generated test suites for all 359 top-level, public classes in Guava. Because the use of post-processing steps during fitness evaluation has high computational costs, we applied the readability optimization as a post-processing step, and compare the result to the default configuration with the regular post-processing steps. Calculated after 20 repetitions, there are 235 out of the 359 classes where the optimization leads to higher average readability values, and 162 cases are significant at $\alpha = 0.05$; there are 38 classes where readability is worse, with 8 of them being significant. On average, the

readability score (averaged over all tests in a test suite) is increased by 0.14 ($\hat{A}_{12} = 0.76$), without affecting code coverage.

4.6 Conclusion

A unit test may be written once, but it is read and interpreted by developers many times. If a test is not readable, then it may be more difficult to understand it. This problem is exacerbated for unit tests generated automatically by tools, which in principle are intended to support developers in generating high coverage test suites, but in practice tend to generate tests that do not look as nice as manually-written ones. To address this problem, we have built a predictive model of test readability based on data of how humans rate the readability of unit tests. We have applied this model in an automated unit test generation tool, and validated that users prefer our readability optimized tests to non-optimized tests.

Our experiment about the effects of readability on test understanding has demonstrated the boundaries between readability and understandability: Not all tests that look nice are also easy to understand. The inclusion of *semantic* features such as code coverage, may lead to an improvement of the readability model. It is even conceivable to use our human maintenance question data to learn a model of understandability, rather than readability.

In summary, this chapter proposed a domain-specific model of test readability and an algorithm for producing more readable tests. In general, it is found that this model outperforms previous work in term of agreement with humans on test case readability (0.87 vs. 0.79); this approach generated alternative optimized tests that were 2% more readable on average; humans prefer optimized tests 69% of the time; and humans can answer questions about readable tests 14% faster with no change in accuracy.

Finally, our technique to increase the readability of unit tests is still quite limited in the scope of its changes to test appearance. For example, variable names are chosen according to a fixed strategy (i.e., class name in camel case, with lower caps first letter, and numeric ID attached at the end). Our feature analysis suggests that identifiers have a very strong influence on readability, and indeed the participants of our experiment mentioned the choice of variable names repeatedly in the post-experiment survey. Therefore, the next chapter will focus in generating unit test cases with descriptive identifier names. The strategy that is described intent on helping developers on understanding the function of the test easier by building names based on tests coverage information.

Chapter 5

Generating Unit Tests with Descriptive Names

The content of this chapter is based on the published work during this PhD, and presented in the International Symposium on Software Testing and Analysis - ISSTA [57].

5.1 Introduction

We have seen in the previous chapter that readable test cases are preferred by developers as they can be understood more quickly, which makes the developers' job easier. Moreover, having suitable variable names is shown to be one of the most important features that affect readability (Table 4.1 list in bold important features such as identifiers and identifier length). Developers frequently interact with unit tests: When trying to understand code, unit tests can be consulted as usage examples. When maintaining code, unit tests help to identify undesired side-effects. When changing code, unit tests need to be updated to reflect the changed behaviour. Providing tests with good names simplifies all these tasks, which is important considering the substantial costs and effort of software maintenance [28]. For example, consider the artificial example class `ShoppingCart` (Figure 5.1), which has two methods `addPrice` and `getTotal`. Given a test named `addPriceThrowsIllegalArgumentException` it can be immediately seen that, even without using the test's source code, what the purpose of the test is (call `addPrice` with an argument that makes it throw an `IllegalArgumentException`), which part of the code it uses (method `addPrice`), and it is reasonable to assume that the test provides an example of unintended usage of the class `ShoppingCart`. Tests named `getTotalReturningZero` and `getTotalReturningPositive` would immediately reveal with their name that they provide two different scenarios for the `getTotal` method. When modifying the `getTotal` method, a developer would know that these tests are the first ones

to run, and when one of these tests fails during continuous integration, the developer would know immediately where to start debugging.

```
public class ShoppingCart {
private int total = 0;
private final static int MAX = 1000;

public boolean addPrice(int cost) throws IllegalArgumentException {
if (cost <= 0)
throw new IllegalArgumentException("Cost cannot be negative");
if (cost < MAX) {
total += cost;
return true;
} else {
return false;
}
}
public int getTotal() {
return total;
}
}
```

Fig. 5.1 Example class: A shopping cart that keeps track of the money spent, but has a limit for individual purchases.

Although automated test generation tools can produce tests that achieve high code coverage, these tests typically come without meaningful names (e.g., EvoSuite [70], Randoop [146]). To overcome this problem, in this chapter we propose a novel technique to generate descriptive names for automatically generated unit tests. This technique is based on the insight that, while an individual generated test might not have a clearly discernible purpose on its own, the context of the test suite is embedded in providing sufficient information to derive names which (a) describe the test's code, (b) uniquely identify the test within its test suite, and (c) provide a direct link from source code to test name. Our technique first identifies all possible descriptive elements that are identifiable at the level of the test code, then selects a minimal set of these elements for each test in a test set, and finally uses this minimal set to synthesize a descriptive, unique name for each test.

In detail, the contributions of this chapter are as follows:

- A technique to synthesize descriptive names for generated unit tests in terms of their observable behaviour at the level of test code.
- An open source implementation of the test naming technique, as an extension to the open source EvoSuite test generation tool [70].

- An empirical evaluation comparing manually derived names with automatically generated names in terms of developer agreement, ability to identify tests by their names, and ability to identify tests based on the code under test.

In our study, participants agreed with the names synthesized with our technique. The synthesized names are as descriptive as manually written names, participants were slightly better at matching tests with synthesized names than they were at matching tests with manually written names. Finally, participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.

5.2 Naming Tests Based on Coverage

The main challenge in naming automatically generated tests is that these tests usually lack a real scenario or purpose that could be used to derive meaningful names. However, it is still possible to generate names that provide some of the benefits good names for manually written tests also offer. We identify the following three requirements for good names for generated tests:

R1 Test names should be descriptive of the test code; there should be an understandable, intuitive relation between the test code and its name.

R2 Test names should uniquely distinguish tests within a test suite, such that developers can use them to navigate the test suite.

R3 Test names should have a clear relation to the code under test; they should allow developers to identify relevant tests and which part of the code each test exercises without having to inspect the test code.

In this section, we describe a naming approach tailored to a scenario where tests are generated for code coverage — perhaps the most common test generation scenario. In general, a code coverage criterion can be represented as a finite set of distinct goals, such that a test suite is considered adequate if for each goal there exists at least one test that covers it. The proposed approach uses coverage goals as the basis for name generation. While coverage goals do not describe a real test intent, they serve as a reasonable approximation since they can describe what the test *does*.

At a high level, the name synthesis first uses dynamic analysis (i.e., test execution) to determine, for a selection of relevant criteria, the set of goals each test in a test suite covers (assuming that during test execution tool will first use the default test name and then rename it). Then, a name is constructed by selecting a minimal subset of the covered goals that is sufficient to create a unique name for each test, such that no other test in the same test suite

has the same name. Finally, each test name will start with the prefix "test" followed by its unique covered goal (e.g., testCreatesShoppingCard).

5.2.1 Coverage Criteria for Test Naming

Table 5.1 Input/output coverage goals for different datatypes.

Datatype	Goals
Boolean	`True`, `False`
Numeric	`Negative`, `Zero`, `Positive`
Character	`Alphabetic`, `Digit`, `OtherChar`
Array	`Null`, `EmptyArray`, `NonEmptyArray`
List	`Null`, `EmptyList`, `NonEmptyList`
Set	`Null`, `EmptySet`, `NonEmptySet`
Map	`Null`, `EmptyMap`, `NonEmptyMap`
String	`Null`, `EmptyString`, `NonEmptyString`
Object	`Null`, *Goals derived from observers*

Not all coverage criteria are suitable for generating names. For example, assume a test that covers lines 27 to 36 of a class under test (CUT), which could be named `testLine27to36`. While the name satisfies the link between code under test and name (R3) and very likely is unique in the context of a test suite (R2), the name is hardly descriptive of the test code (R1). To satisfy this requirement, coverage goals need to be based on the *observable* behavior. That is, any values, calls, exceptions that exist in the scope of the test code, represent information that a developer could understand even without having to consult the code under test. Conceptually, we can think of these aspects as high-level coverage criteria. For each such high-level coverage criterion, we define a pattern to convert coverage goals into text, and the overall test name is a result of combinations of such names. In particular, we consider the following criteria, which are sometimes used by search-based test generation tools when generating tests [162]:

Method coverage: The most general criterion is given by the methods of the class under test; for each public method or constructor, there is one coverage goal. For a method coverage goal, the text representation simply is the method name. In the case of constructor calls, there is no method name, therefore we use the name of the class to generate a name. For example, a coverage goal for a constructor generating an instance of `ShoppingCart` would result in the text `CreatesShoppingCart`.

Exception coverage Exceptions are generally treated special in unit tests: First, the source code structure is typically different (using a try/catch statement), and second, they often represent the main target behavior of a test. Exception coverage defines a coverage goal for each declared exception of every method of a class under test. In addition, any undeclared exceptions arising during test generation are also typically retained in a generated test suite, and can thus be counted as coverage goals. An exception coverage goal is converted to a string simply by concatenating the method name, the keyword `Throws`, and the name of the exception class (without package name). For example, method `addPrice` of class `ShoppingCart` can throw an `IllegalArgumentException`, and this would be represented as `AddPriceThrowsIllegalArgumentException`. For constructors, we use the classname again; for example, if a constructor throws an exception while trying to instantiate the `ShoppingCart` class, this coverage goal would result in the name `FailsToCreateShoppingCartThrowsIllegalArgumentException`.

Output coverage The concept of output coverage [7] is based on the intuition that uniquely different output values are suitable as test adequacy criteria. Table 5.1 summarizes different output partitions for different possible return values of tested methods. For objects, we include output goals based on observers, i.e., methods that (1) do not change the object state, (2) take no parameters, and (3) return a primitive (numeric, character, string) type. For each observer, we define output coverage goals as listed in Table 5.1. For example, if a return value is a `Set`, then the observer `isEmpty` would result in two coverage goals (true, false), and the observer `size` would result in three output goals (<0, 0, >0). Names for output coverage goals are created by concatenating the name of the method with the keyword `Returning`, followed by the descriptor of the actual return value (as shown in Table 5.1). For example, a coverage goal that represents `getTotal` returning 0 would be named `GetTotalReturningZero`. When coverage goals are based on observers, then the name consists of the method name, the keyword `Where`, the name of the observer, the keyword `Is`, and a textual description of the return value of the observer. For example: `FooWhereIsEmptyIsTrue`.

Input coverage The same partitioning of value ranges described above for outputs can also be applied to all input parameters. Names for input coverage goals are created by concatenating the method name with the keyword `With`, followed by a descriptor of the parameter value. For example: `AddPriceWithZero`.

Note that, in a unit testing scenario, we assume there is a dedicated CUT, and hence only method calls on instances of the CUT, exceptions thrown in calls to the CUT, as well as inputs and outputs of CUT methods are considered for coverage goals.

Table 5.2 Coverage goal list for the `ShoppingCart` class, and their string representation.

	Goals in class ShoppingCart	Text representation
Method	`ShoppingCart()`	`CreatesShoppingCart`
	`int getTotal()`	`GetTotal`
	`boolean addPrice(int)`	`AddPrice`
Exc.	`boolean addPrice(int)` → `java.lang.IllegalArgumentException`	`AddPriceThrows IllegalArgumentException`
Output	`int getTotal()` → `== 0`	`GetTotalReturningZero`
	`int getTotal()` → `< 0`	`GetTotalReturningNegative`
	`int getTotal()` → `> 0`	`GetTotalReturningPositive`
	`boolean addPrice(int)` → `False`	`AddPriceReturningFalse`
	`boolean addPrice(int)` → `True`	`AddPriceReturningTrue`
Input	`boolean addPrice(int)` → `== 0`	`AddPriceWithZero`
	`boolean addPrice(int)` → `> 0`	`AddPriceWithPositive`
	`boolean addPrice(int)` → `< 0`	`AddPriceWithNegative`

5.2.2 Finding Unique Method Names

Table 5.3 Method names for overloaded methods.

Method signature	Method name
`void foo(int x)`	`FooTakingInt`
`void foo(int x, int y)`	`FooTakingTwoInts`
`void foo(int x, String z)`	`FooTakingIntAndString`
`void foo(int x,int y,int z)`	`FooTaking3Arguments`
`Foo()`	`CreatesFooTakingNoArguments`
`Foo(int x)`	`CreatesFooTakingInt`

Method names are part of all criteria listed above. However, if methods are overloaded, then method names are not unique, and thus not sufficient to uniquely identify which method a name refers to. Consequently, we include aspects of the method signature to generate a

unique method name. If the number of arguments is sufficient to uniquely identify a method, we include this in the name. Table 5.3 shows an overloaded method `foo` with five different variants. If there is no argument, then the suffix `TakingNoArgument` is appended to the method name. If there is exactly one argument, then its type name is appended to the method name. If there are two or more parameters but no other method with the same name and same number of arguments, then `TakingXArguments` represents the name. Finally, if there are several methods with the same name and the same number of arguments, then the name is constructed from the actual signature. To abbreviate names, for each type we use the number of arguments that have this type to construct the name (e.g., `FooTakingTwoInts`). If there are overloaded constructors, then the same principle applies, except that the argument descriptor string is appended to the `CreatesFoo` string. (Note that, and this is different from input goal based naming that is taken into consideration only if method, output, and exception goals are not unique for a specific test, and there is no overloaded method.)

5.2.3 Synthesizing Coverage-based Test Names

Table 5.4 Coverage goals for the test suite generated by EvoSuite (Figure 5.2); unique coverage goals are highlighted.

Test	Coverage Goals
`test0`	Method `CreatesShoppingCart`
	Method `AddPrice`
	Output `AddPriceReturningFalse`
	Input `AddPriceWithPositive`
`test1`	Method `CreatesShoppingCart`
	Method `AddPrice`
	Exception `AddPriceThrowsIllegalArgumentException`
	Input `AddPriceWithZero`
`test2`	Method `CreatesShoppingCart`
	Method `AddPrice`
	Output `AddPriceReturningTrue`
	Input `AddPriceWithPositive`
`test3`	Method `CreatesShoppingCart`
	Method `GetTotal`
	Output `GetTotalReturningZero`

```
@Test
public void test0 / testAddPriceReturningFalse() {
 ShoppingCart cart0 = new ShoppingCart();
 boolean boolean0 = cart0.addPrice(2298);
 assertEquals(0, cart0.getTotal());
 assertFalse(boolean0);
}

@Test
public void test1 / testAddPriceThrowsIllegalArgumentException() {
 ShoppingCart cart0 = new ShoppingCart();
 // Undeclared exception!
 try {
  cart0.addPrice(0);
  fail("Expecting exception: IllegalArgumentException");
 } catch(IllegalArgumentException e) {
  // Cost cannot be negative
  verifyException("ShoppingCart", e);
 }
}

@Test
public void test2 / testAddPriceReturningTrue() {
 ShoppingCart cart0 = new ShoppingCart();
 boolean boolean0 = cart0.addPrice(1);
 assertEquals(1, cart0.getTotal());
 assertTrue(boolean0);
}

@Test
public void test3 / testGetTotal() {
 ShoppingCart cart0 = new ShoppingCart();
 int int0 = cart0.getTotal();
 assertEquals(0, int0);
}
```

Fig. 5.2 Test suite generated with EvoSuite for class `ShoppingCart`, with original names / descriptive names.

The overall technique to synthesize test names is summarized in Algorithm 4. The main input arguments of the algorithm are an automatically generated test suite and a set of coverage goals. These coverage goals are computed statically for the class under and represent the coverage criteria used to guide the test generation process; the third input controls the

length of the resulting test names. A basic dynamic analysis (labelled COVEREDGOALS in Algorithm 4) first determines which goals are covered by each test in the suite. If a test only covers one goal, then the name of the goal is an obvious candidate name for the test. Typically a test will cover multiple coverage goals at the same time, and the ideal name will be a combination of these goals. However, naively joining up goal names would easily result in too long names, and goals covered by multiple tests would make it more difficult to distinguish tests (R2). Therefore, this section describes how to create shorter, representative names given a test and the coverage goals it covers.

As a running example for the naming process, Figure 5.2 shows a test suite generated automatically using EvoSuite [70]. Table 5.4 lists the coverage goals for each of these tests. Note that `test0` and `test2` call `getTotal` but these are not listed as method coverage goals for these tests. We ignore these calls since they are not part of the sequence of calls a test generator would produce, but within assertions which are usually added in a post-processing step.

Algorithm 4 Test name generation.

Input: Test suite T, set of coverage goals G, and maximum number of goals in a name n

1: **procedure** NAMETESTS(T, G)
2: **for all** $t \in T$ **do**
3: $U \leftarrow$ COVEREDGOALS($\{t\}$, G) $\setminus$ COVEREDGOALS($T \setminus \{t\}, G$)
4: $top \leftarrow$ TOPLEVELGOALS(U, n)
5: $name \leftarrow$ MERGETEXT(top)
6: LABEL(t, $name$)
7: **end for**
8: **for all** $T' \subset T$ where all $t \in T'$ have the same name **do**
9: RESOLVEAMBIGUITIES(T')
10: **end for**
11: **for all** $t \in T$ **do**
12: **if** t has no name **then**
13: $C \leftarrow$ COVEREDGOALS($\{t\}$, G)
14: LABEL(t, Head(TOPLEVELGOALS(C, n))
15: **end if**
16: **end for**
17: **for all** $t \in T$ **do**
18: SUMMARIZETEXT($\{t\}$)
19: **end for**
20: **for all** $T' \subset T$ where all $t \in T'$ have the same name **do**
21: ADDSUFFIXES(T')
22: **end for**
23: **end procedure**

5.2.3.1 Identification of Unique Goals

In a scenario where tests are generated with the intent of covering code, it is reasonable to assume that each test in a test suite covers something that no other test in the test suite covers; a test that does not provide any additional coverage would not be retained during test generation, or else would be removed during post-processing of generated test suites. Under this assumption, each test will have at least one *unique* coverage goal that distinguishes it from all other tests, which will influence the name given to the test. Note that even without a unique coverage goal our approach will still come up with a descriptive name; however, if a test cannot be distinguished in terms of coverage, then whether or not the test can be clearly distinguished from other tests in the same test suite by name will be a result of chance.

Given a set of covered goals for each test in a test suite, determining the set of unique covered goals is a matter of calculating the set complement of goals covered by a test, and goals covered by all other tests (set U in Algorithm 4). For example, considering the coverage goals listed in Table 5.4, the test `test0` is the only test to cover the output goal `AddPriceReturningFalse`; `test1` is the only test to cover exception goal `AddPriceThrowsIllegalArgumentException` and input goal `AddPriceWithZero`; `test2` uniquely covers the output goal `AddPriceReturningTrue`; and `test3` uniquely covers the method `GetTotal` and the output goal `GetTotalReturningZero`.

5.2.3.2 Ranking Test Goals

A test will typically cover more than one goal. For example, if a method is covered, then it's return value will result in an output coverage goal that is covered, and any parameters will result in covered input coverage goals. To make sure that the name reflects only the minimum amount of information necessary to satisfy the three desired properties of a good name, we sort coverage goals based on the following ranking:

1. Exception coverage is at the highest level in the hierarchy. Intuitively, if a test leads to an exception, then this is information we will always want to see reflected in the name.
2. Method coverage is ranked next; even if there is no exception, we always want a test name to describe at least the method it targets.
3. Output coverage follows method coverage, since the output (return value) often represents the result of the scenario under test. Intuitively, if a test uniquely covers a method, then the method coverage is sufficient information. However, if there are several tests covering the same method, then the differences in method behavior are captured by different return values.
4. Input coverage represents information about the "scenario" of the test, and we consider this only if no other coverage type can be used to name a test.

Given the set of unique coverage goals for a test, the next step in the name generation is thus to select all goals of the highest rank (TopLevelGoals in Algorithm 4). That is, if there are exception goals covered, then these will be used for the name generation (for example, `test1` in Table 5.4). If not, then all method goals will form the basis for name generation (e.g., method `GetTotal` for `test3`). If there are no uniquely covered methods, we next look at uniquely covered output goals, since the values produced are representative of the expected behavior (e.g., `AddPriceReturningFalse` for `test0` and `AddPriceReturningTrue` for `test2`). Finally, if there are no output goals, then we select all unique input goals.

Note that it is possible for a test not to cover uniquely any coverage goals; this might be because the test suite is not minimized and the test is redundant, or it might be that there are several tests that cover different lines, branches, or other coverage entities not contained in our hierarchy.

5.2.3.3 Merging Test Goals

When any test case is the test suite is named, the prefix *test* will always be present (e.g., testCreatesShoppingCart). Next, we convert the set of highest level unique coverage goals to text (MERGETEXT in Algorithm 4). If there are no such coverage goals, then the initial name is an empty string. If there is exactly one goal, then the text representation of that goal is the name. For example, in Table 5.4, each test has exactly one top-level goal. If there is more than one goal, then we concatenate their text representation using the `And` keyword. During this concatenation, for any subset of coverage goals that target the same method (e.g., if a test covers two input coverage goals for the same method call), the method name will only be listed for the first one. If a coverage goal on a constructor is merged with a coverage goal on a method, then instead of just `And` these are merged using `AndCalls`; for example `testCreatesFooAndCallsBar`.

In order to avoid long names, we use a maximum of n goals for concatenation (in our case, $n = 2$). If there are more than n goals, we select a subset of n goals. For this selection we prioritize the coverage goals based on the methods they relate to. In particular, we prioritize methods that change the state of an instance of the class under test (i.e., impure method), and methods whose return values are involved in assertions. If there are more than n goals for the same method, we make a random selection of n goals.

5.2.3.4 Resolving Ambiguities

At this point, each test has a name, but the name might not be unique, for example if two or more tests only differ in terms of lower level coverage (e.g., branches) and thus an empty name. For each ambiguous name we select all tests that resulted in that name (T'). For each of these tests, RESOLVEAMBIGUITIES computes the set of unique coverage goals relative to the set of ambiguous tests (rather than the whole test suite), and then selects the top ranked goal out of that set.

If, at the end of this process, there exist tests that have no name, then we consider the set of non-uniquely covered goals for that test (C), and select the top ranked goal (Head(TOPLEVELGOALS(C))). Now that all tests have names, we use a basic postprocessing technique based on abstractive text summarisation algorithms (SUMMARIZETEXT). This

step simplifies common patterns to more natural versions. For example, the following test name is based on a `List` returned by method `foo`:

`testFooReturningListWhereIsEmptyIsFalse`

For collection classes, we reduce the use of observers such that the resulting test is named as follows:

`testFooReturningNonEmptyList`

Similarly, we reduce output goals on Boolean values. For example, `testEqualsReturningFalse` is changed to the simpler version `testNotEquals`.

Finally, we append indices to all names in any remaining set of ambiguous test names to resolve duplication (ADDSUFFIXES). Each test in Figure 5.2 shows the name resulting from the overall process.

5.3 Evaluation

To evaluate the effectiveness of the proposed technique to generate descriptive names *for automatically generated unit tests*, we conducted an empirical evaluation aimed at answering the following research questions:

RQ1 Do developers agree with synthesized test names?
RQ2 Can developers match unit tests with synthesized test names?
RQ3 Can developers use synthesized test names to identify relevant unit tests for a given piece of code?

5.3.1 Experimental Setup

5.3.1.1 Subjects

We recruited participants by direct email invitations to Computer Science and Software Engineering students at the University of Sheffield and the University of Leicester. All participants were prescreened using a Java and JUnit qualification quiz: they were required to answer correctly at least 3 out of 5 competency questions. As a result, a total of 47 unique applicants were accepted and took part in the study.

5.3.1.2 Treatments

We consider two treatments: test names synthesized by our coverage-based naming technique (*synthesized*) and names manually written by experienced developers (*manual*).

5.3.1.3 Tasks

The three research questions in the study are addressed by assigning participants the following tasks:

Agreement (RQ1). Given the code of a JUnit test and a suggested test name, indicate your level of agreement with the name.

Selection (RQ2). Given the code of a JUnit test, select the more appropriate, descriptive name from a given list of candidate names.

Understanding (RQ3). Given the source code of a Java class and a list of candidate test names, select the test name you think will execute a given line number in the source code.

5.3.1.4 Objects

We followed a systematic protocol to select objects from the SF110 corpus of open source Java projects [72]. SF110 consists of a statistically representative sample of 110 projects and includes 23,886 classes with more than 6.6 millions of lines of code in total. Our selection protocol consisted of the following steps:

1. Download the SF110 compilable sources and tests package.
2. Select classes for which a generated test suite exists in the package (i.e., for which test generation succeeds).
3. Select classes with a ratio of at least two generated unit tests per public method. Test naming is particularly relevant when there is more than one test per public method.
4. Generate test suites with descriptive, synthesized names for the selected classes.
5. Select target methods for which there are strictly three generated tests. This is a necessary design decision: When there is only one (or two) tests for a target method, test naming is trivial (or nearly trivial). On the other hand, more than three tests per method would overly complicate the tasks (more and longer test names to analyze by participants) without adding value to the study.
6. Run mutation analysis on the selected classes and test suites [100]. Select target methods for which at least one generated test uniquely kills at least one mutant. This is needed for the understanding task: a uniquely killed mutant provides the connection between a synthesized test name and a source code line.
7. Finally, randomly select one suitable target method per class, and for each target method, randomly select a test to be used to formulate agreement, selection and understanding tasks.

As a result of this systematic search, we selected ten target methods for ten different classes to be part of our study, each one with three automatically generated tests with descriptive, synthesized names. Table 5.5 summarizes this final object selection.

Baseline. We used test names written by experienced Java developers as baseline. Deciding this was the appropriate baseline to compare against was a challenging task. In principle, when looking at automatically generated tests, the obvious baseline should be enumerated test names – that is how generated tests are currently named. However, such a comparison would be pointless at the very least: there would be no true value in showing improvement over `test0` or `test1`. As an alternative, having seasoned human developers manually assign names to generated tests provides a realistic baseline and enables us to evaluate how the names synthesized using our coverage-based naming technique are perceived by developers in a less contrived scenario. Under this rationale, we recruited two PhD students with extensive programming experience and knowledge of JUnit, and asked them to create this baseline dataset. We presented them with the 30 selected tests (three for each of the ten selected target methods) and the respective classes under test. They were instructed to provide the most appropriate and descriptive name they could think of for all these tests. They worked independently, without time constraints, and did not receive any guidance to name the tests; they were not made aware of the purpose of the study, either. The resulting dataset was consolidated by two of the authors of this paper, who inspected each test and selected the best of the two names created by the experts. These names are shown in column "Manual names" of Table 5.5.

Table 5.5 List of experimental objects. NUKM stands for Number of uniquely killed mutants. For each class, the test selected for the study is highlighted. The other tests names are used to be presented as options for selection and understanding tasks.

Test ID	Project	Class.Method	Synthesized names	Manual names	NUKM
1	jsecurity	ValueListImpl.asBoolean	testAsBooleanThrowsNullPointerException	testNullList	0
			testAsBooleanReturningEmptyList	testListEmptyOnCreation	0
			testAsBooleanReturningNonEmptyList	**testAddNullValue**	**1**
2	vuze	PeerItem.convertSourceID	**testConvertSourceIDReturningZero**	**testConvertSourceIDTracker**	**1**
			testConvertSourceIDThrowsNullPointerException	testNullArgException	0
			testConvertSourceIDReturningNegative	testUnrecognisedSourceID	6
3	jsecurity	SimpleAuthorizingAccount.getSimpleRole	testGetSimpleRolesReturningEmptySet	testEmptySimpleRoles	0
			testGetSimpleRolesReturningNull	testNullSimpleRoles	0
			testGetSimpleRolesReturningNonEmptySet	**testMergeRoles**	**10**
4	squirrel-sql	StringUtilities.getByteArray	testGetByteArrayWithNull	testNullParameter	3
			testGetByteArrayWithEmptyArray	testParameterOfLengthZero	0
			testGetByteArrayReturningNonEmptyArray	**testNonZeroLengthNotNull**	**2**
5	glengineer	VWordPosition.equals	**testEquals**	**testWhetherAVWordPositionsIsEqualToItSelf**	**5**
			testNotEquals	testWhetherTwoDifferentVWordPositionsAreNotEqual	4
			testEqualsWithNull	testWhetherAValidVWordPositionIsEqualToNull	1
6	vuze	BitFlags.and	testAndThrowsNullPointerException	testAndNullPointerException	3
			testAndThrowsArrayIndexOutOfBoundsException	**testAndArrayIndexOutOfBoundsException**	**5**
			testAndReturningBitFlagsWhereSizeIsZero	testAndSameObjectEquals	14
7	noen	DaikonConstraint.stringFrom	**testStringFromReturningNonEmptyString**	**testConversionOfANonEmptyListIntoString**	**3**
			testStringFromReturningEmptyString	testConversionOfAnEmptyListIntoString	0
			testStringFromThrowsNullPointerException	testConversionOfANullListIntoString	0
8	jsecurity	StringUtils.hasText	testHasTextWithNull	testHasTextNullArg	4
			testHasText	testHasTextNonEmptyArg	4
			testNotHasText	**testHasTextEmptyArg**	**2**
9	vuze	StringPattern.digitWildcard	testDigitWildcardTakingNoArgumentsReturningNull	testSingleArgConstructor	8
			testDigitWildcardTakingNoArgumentsReturningNonNull	testTwoArgConstructorWildcard	11
			testDigitWildcardTakingCharacter	**testNonMatchingPattern**	**94**
10	jiprof	ClassWriter.visitAnnotation	**testVisitAnnotationWithNonEmptyStringAndFalse**	**testVisitNonVisibleAnnotation**	**4**
			testVisitAnnotationWithEmptyString	testVisitVisibleAnnotation	4
			testVisitAnnotationThrowsNullPointerException	testVisitAnnotationNullPointerException	1

5.3.1.5 Procedure

The study was implemented as an online survey. We designed a webpage containing strictly the necessary information to complete each of the three tasks. The webpage for *agreement* tasks contains a tab panel for the target test, a tab panel for the target test shown in context with the other two tests for the same target method (the name of these context tests are concealed), a Likert-like input component (strong disagree, disagree, neutral, agree and strong agree) (Figure 5.4). The webpage for the *selection* tasks is similar, except that the name of the target test is concealed, and three shuffled candidate names are presented as choices (plus “None”). The webpage for *understanding* tasks, in turn, shows the source code of a class under test instead of the test code, with a target line number highlighted (the line where a mutant is uniquely killed by a test). Similarly to the webpage for selection tasks, three candidate names plus “None” are presented as options. In the three webpages, a required free-text explanation field is included to elicit the reasoning behind each participant’s answer.

Questions. With ten target methods (with a selected test, as highlighted in Table 5.5), two treatments (synthesized and manual), and three tasks (agreement, selection, understanding), a total of 60 questions were formulated for the study and loaded on the survey.
Assignment. Upon login on the website, each participant was assigned ten questions, one per test, using a balanced random sampling algorithm. That is, for each test, a participant can see either manual or generated test names, and one of the three types of questions. The order in which question are presented is randomized.

5.3.1.6 Analysis

All data collected in the study is processed using well tested R scripts. For agreement tasks, we present Likert scales to reason about levels of agreement as a whole and per individual test. For the selection and understanding tasks, bar plots are used to present correctness results (categories “Correct”, “Don’t know”, and “Incorrect”). For the three tasks, we present boxplots and report on effect sizes and $p-$values for the comparison of response times for synthesized and manually written test names.

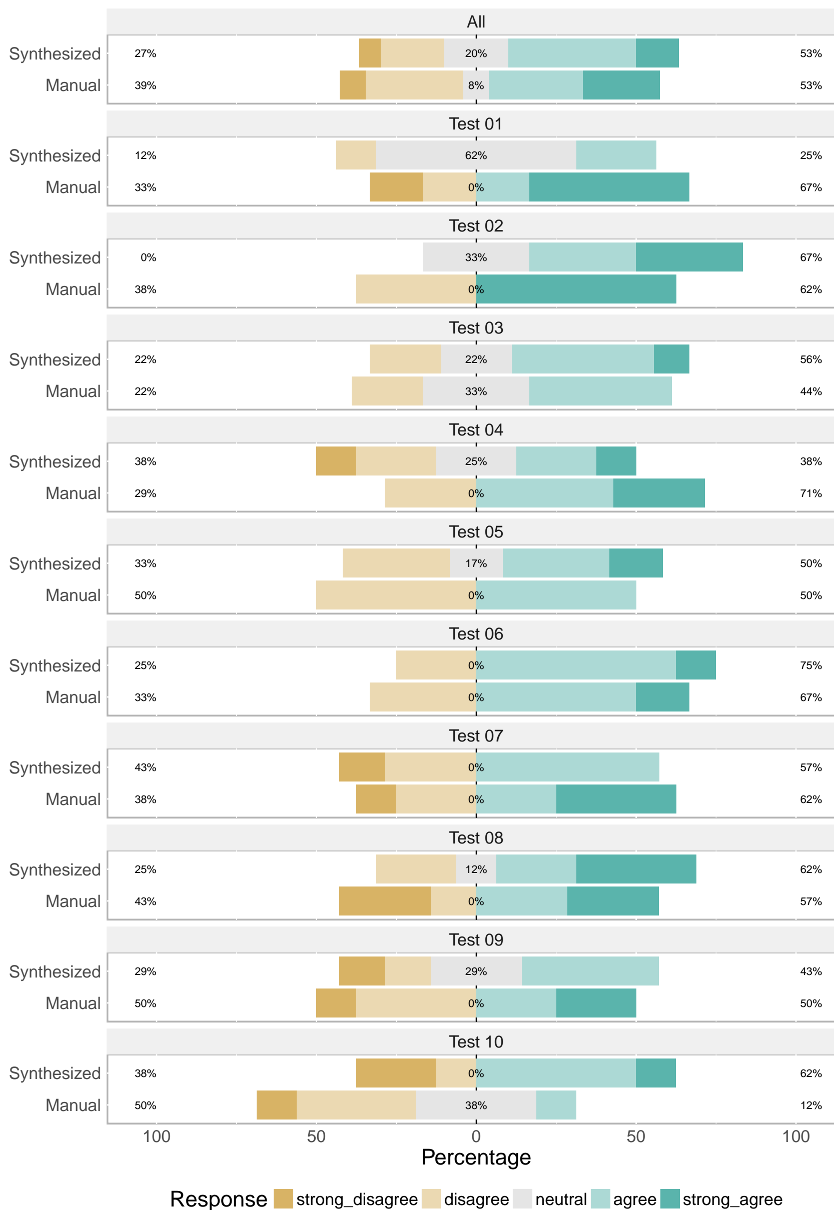
5.3.1.7 Threats to Validity

Objects. The Java methods and tests used in this study may not be representative of larger, more complex systems and test suites. We alleviated this by using a systematic object selection protocol, drawing methods and tests from a statistically significant sample of open source Java projects. Further replications will be needed to verify how our results generalize to more complex systems under test, where the ratio of generated tests to methods may be higher.
Participants. Most participants were students and hence may not be representatives of real-world experienced programmers [44, 93, 108]. This is a common issue encountered when conducting human studies, since it is nearly impossible to reach the number of professional developers needed to perform any statistical test on their data. However, the most recent research on this issue threat can be mitigated by carefully scoping of experimental goals [169]. Participants in the study may answer randomly, imposing a threat on the validity of conclusions drawn from their data. We alleviated this threat by requiring free-text justifications for all answers.
Context. We purposefully limited the context available to participants when answering the study questions. It may be the case that having other artifacts available would affect some

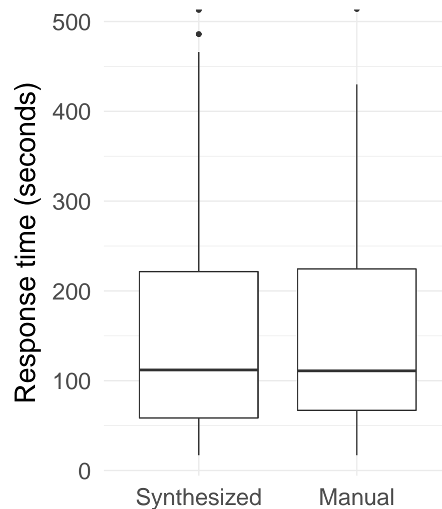
participants's answers. However that would have come at the unknown cost of introducing confounding variables in the experimental set up.

5.3.2 Results

5.3.2.1 RQ1: Agreement with Synthesized Test Names



(a) Likert agreement scales, aggregated at the top (labeled "All"), and individually for each of the ten tests in the study.



(b) Response times.

Fig. 5.3 Agreement results (RQ1).

Figure 5.3a summarizes the results for the agreement questions, and shows that there is generally more agreement than disagreement with synthesized test names. Over all answers for questions with synthesized names, 53% indicated some level of agreement, while only 27% showed some level of disagreement (top of Figure 5.3a). This suggests that participants perceived synthesized names as appropriate and descriptive for their respective unit tests.

Positive examples: The highest levels of agreement are observed for Tests 06 (75%) and for Test 02 (67%). In the case of Test 06, the synthesized name makes it clear that the test captures an exceptional behavior (`testAndThrowsArrayIndexOutOfBoundsException`), while for Test 02 the name reveals the expected output value of a call: `testConvertSourceIDReturningZero`. In both cases, the agreement answers were supported by positive free-text responses, e.g., *"The test name is perfect. It does exactly that, call the method and check if it returns zero"*, and relevant suggestions, e.g., *"The test name is a bit too long for me. I would have named it testArrayOutOfBound"*.

Negative examples: The highest levels of disagreement, on the other hand, was observed for Test 07 at 43%, and Tests 04 and 10, both at 38%. For the latter two tests, one of the reasons given by participants for disagreement is the length of the name. In the case of Test 07, although the name is nearly as long as the ones for Tests 04 and 10, no participant complained about the length, but rather considered the name confusing and insufficient. In spite of these cases, the general trend observed in the data is that participants only mildly and not very often disagree with synthesized names, with the best case being Test 02, with 0% level of disagreement.

Comparison to manually written names: The levels of agreement and disagreement observed for synthesized names constitute evidence of the effectiveness our coverage-based approach at generating descriptive test names, but to properly gauge the level of agreement we need to consider how the synthesized names compare to manually written test names. Figure 5.3a depicts this comparison for each of the tests used in our study, and aggregated for all. Overall, we observe similar levels of agreement, but less disagreement in the case of synthesized names (28% vs 39%).

Two of the synthesized names with the highest disagreement (the ones for Test 07 and Test 04) are also the worst-performing compared to manually given names. Both cases exemplify how developers are often capable of eliciting abstract knowledge and capturing it into their test names. The manually given name for Test 07, `testConversionOfANonEmptyListIntoString`, makes it explicit that the target method performs a type conversion from a populated list to a string object. It is not surprising that this name was more agreeable to participants than the synthesized `testStringFromReturningNonEmptyString`. Similarly, the manually given name `testNonZeroLengthNotNull` succinctly captures the flow of the test (not even mentioning the method being tested), and was more often agreed upon than its synthesized counterpart `testGetByteArrayReturningNonEmptyArray`.

We observe cases where participants clearly agreed more often with a synthesized name than with a manually given name, e.g., for Test 10 (manual `testVisitNonVisibleAnnotation` vs. synthesized `testVisitAnnotationWithNonEmptyStringAndFalse`). This case shows that the additional understanding developers can gather from the source code and build into their names is not always perceived as descriptive by other developers. Namely, the expert-given name `testVisitNonVisibleAnnotation` encodes information not evident in the test code itself (notion of non-visibility) which rendered the name *"unclear"*, *"unnecessarily long"* and *"poorly worded"*, according to participants.

For some cases, e.g., Test 02 and Test 08, participants showed agreement with both manual and synthesized names. For Test 02, for instance, although by different means, both test names are descriptive of what the test does. The manual name `testConvertSourceIDTracker`, captures a string constant value being used explicitly in the test code, whereas the synthesized name `testConvertSourceIDReturningZero` neatly describes the method under test and its expected return value.

Timing: Besides agreement levels, the time it took participants to answer agreement questions is also indicative of how descriptive, hence how easy to interpret, test names are. Our timing results (Figure 5.3b) suggest that synthesized names are slightly easier to interpret

than manually given names on average. The mean response time for synthesized names is 185.56” compared to 222.87” for manually given names, although there is no statistical significance between the two samples, with $A_{12} = 0.48$ and $p-$value=0.48.

***RQ1**: Participants in our study agreed similarly, and disagreed less, with synthesized test names than with manually given names.*

50% Complete

Question 6

For the following test, indicate your level of agreement with the suggested test name “testDigitWildcardTakingCharacter”.

Test | Same test in context

```
@Test
public void testDigitWildcardTakingCharacter()  throws Throwable  {
  StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
  Character character0 = Character.valueOf(':');
  stringPattern0.digitWildcard(character0);
  boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
  assertFalse(boolean0);
  assertFalse(stringPattern0.getIgnoreCase());
}
```

testDigitWildcardTakingCharacter is an appropriate name for this test.

Strongly disagree	Disagree	Neutral	Agree	Strongly agree
This test name is completely inappropriate and undescriptive.	This test name is somewhat inappropriate and undescriptive.	Neither agree nor disagree with this test name.	This test name is somewhat appropriate and descriptive.	The test name is completely appropriate and descriptive.

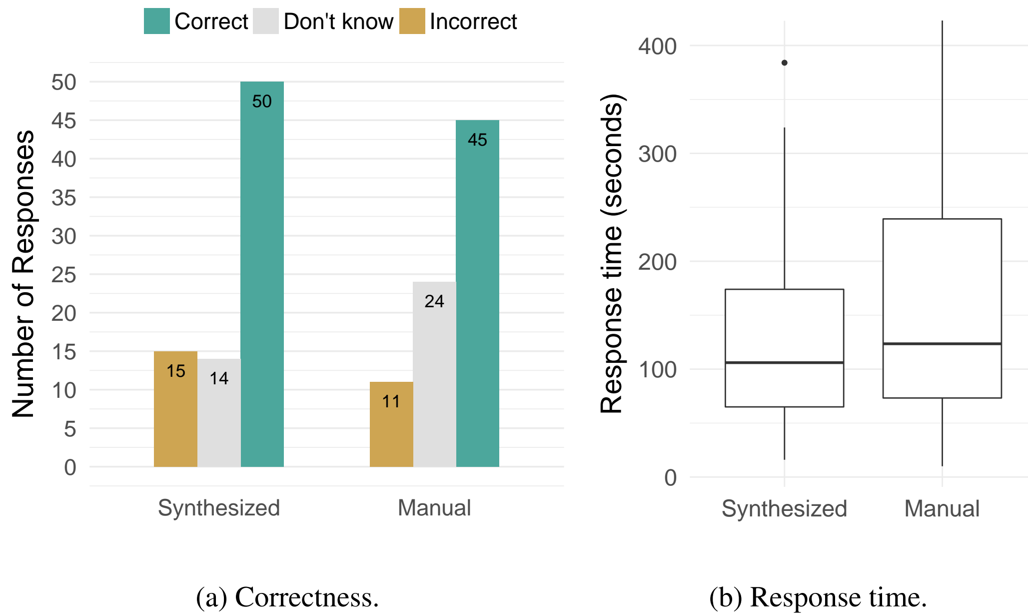
Why? Please explain your answer here:

For example: "That's exactly how I would have named the test", "It's not entirely clear by the name what method is being tested", "The name is unnecessarily long".

« Previous | Next »

Fig. 5.4 Example of agreement question as presented to participants in the survey website.

5.3.2.2 RQ2: Matching Unit Tests with Synthesized Names



(a) Correctness. (b) Response time.

Fig. 5.5 Selection results (RQ2).

One of the key aspects of our approach is that it captures the uniqueness of each generated test in the context of a test suite to synthesize a descriptive name for it (cf. Section 5.2, R2). The selection tasks in our study aim to validate this focus on uniqueness by giving participants a choice of names and asking them to select the most descriptive for a given test. Figure 5.5 shows the results of these selection tasks in terms of correctness (Figure 5.5a) and response time (Figure 5.5b). Our results show that participants were slightly more accurate at choosing the right name for a unit test when the name was synthesized, as opposed to manually created. A total of 50 correct selection answers were collected for synthesized names compared to 45 for manual ones. Furthermore, synthesized names seem to be indeed clearer in terms of uniqueness, as suggested by the lower number of *I don't know* answers (14 vs 24), although this positive observation is somewhat shadowed by the higher number of incorrect answers (15 vs. 11).

There is also a trend indicating that participants were faster at selecting synthesized test names (means 164.43” vs. 171.68”). Although the difference is not statistically significant ($A_{12} = 0.45$ and p-value=0.3.), it at least suggests that interpreting synthesized names does not represent extra effort to developers.

RQ2*: Participants of our experiment were slightly more accurate and faster at matching tests and synthesized names.*

40% Complete

Question 5

For the following test, select the test name that you think is most appropriate and best describes the test code.

Test | Same test in context

```
@Test
public void ___________() throws Throwable {
    StringPattern stringPattern0 = new StringPattern("2*#@}:*Q54)M!");
    Character character0 = Character.valueOf(':');
    stringPattern0.digitWildcard(character0);
    boolean boolean0 = stringPattern0.matches("2*#@}:*Q54)M!");
    assertFalse(boolean0);
    assertFalse(stringPattern0.getIgnoreCase());
}
```

Select a name for the highlighted test above:

- `testNonMatchingPattern`
- `testTwoArgConstructorWildcard`
- `testSingleArgConstructor`
- None of the above.

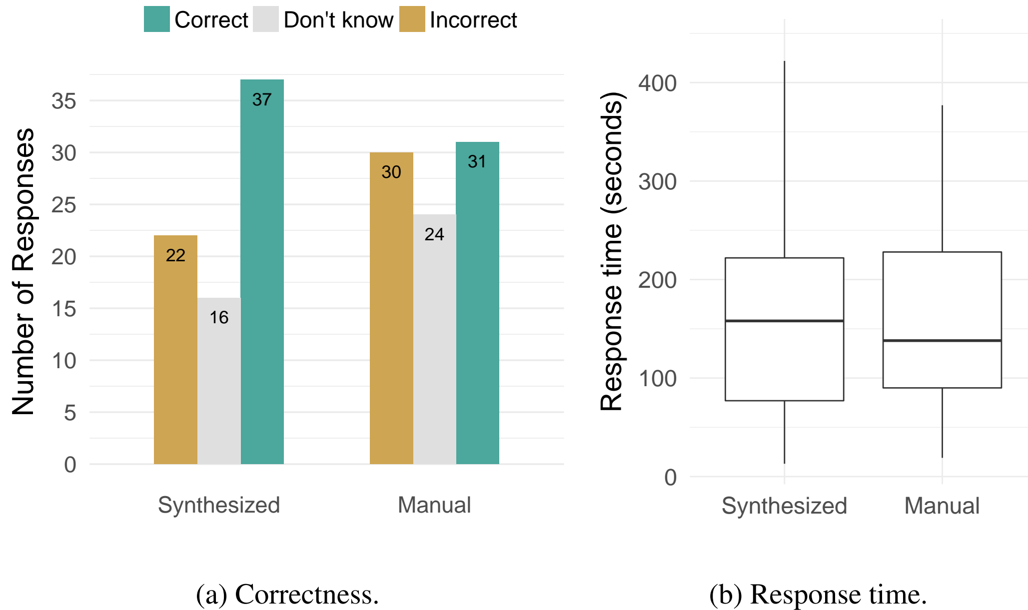
Why? Please explain your selection here:

For example: "That's exactly how I would have named the test", "It's the most descriptive name of the three", "The other names are just terrible".

« Previous | Next »

Fig. 5.6 Example of selection question as presented to participants in the survey website.

5.3.2.3 RQ3: Identifying Relevant Unit Tests by Names



(a) Correctness. (b) Response time.

Fig. 5.7 Understanding results (RQ3).

Beyond descriptiveness and uniqueness, test names should expose the relation between the unit test and the code under test (cf. R3 in Section 5.2). In other words, a test's name should ideally help developers identify which parts of the code under test will be exercised by a test. This is the scenario we present to participants in our *understanding* tasks. Figure 5.7 shows correctness and efficiency results for these tasks.

The trend in favour of synthesized names observed for selection tasks is even more pronounced for understanding tasks. As shown in Figure 5.7a, participants were more accurate at identifying the relevant tests for specific parts of the code when they were presented synthesized names. Synthesized names also reduced by a notable margin the number of incorrect and uncertain answers.

The higher accuracy at understanding tasks comes at a price of a marginally larger effort (Figure 5.7b). However, the average time participants needed with synthesized (186.08”) and manually given names (188.24”) are nearly the same, and the statistical comparison indicates no significant difference, with $A_{12} = 0.49$ and p-value=0.9.

***RQ3**: Participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.*

90% Complete

Question 4

If a change was made in line number 189 in the following class, which of the tests listed in the right panel would you inspect/run to test the changed behaviour?

Class Under Test

```
173                 SimpleRole role = new SimpleRole(roleName);
174                 add(role);
175             }
176         }
177
178         public void add(SimpleRole role) {
179             Set<SimpleRole> roles = getSimpleRoles();
180             if (roles == null) {
181                 roles = new LinkedHashSet<SimpleRole>();
182                 setSimpleRoles(roles);
183             }
184             roles.add(role);
185         }
186
187         public void addRoles(Set<String> roleNames) {
188             if (roleNames != null && !roleNames.isEmpty()) {
189                 for (String name : roleNames) {
190                     addRole(name);
191                 }
192             }
193         }
```

Please select one test:

- testMergeRoles
- testEmptySimpleRoles
- testNullSimpleRoles
- None of the above.

Why? Please explain your selection here:

For example: "The name suggests the test will cover that line", "I honestly tried, but I don't know.".

« Previous Next »

Fig. 5.8 Example of understanding question as presented to participants in the survey website.

5.4 Conclusions

In the previous chapters we have seen that developers prefer tests with meaningful variable names, and they repeatedly mentioned the choice of meaningful names that will further contribute to test understanding. There are many advantages to having well-named unit tests: Finding relevant unit tests becomes easier, the importance of test failures can be judged without reading the source code, and the test is easier to understand as the name summarizes its purpose. Automatically generated unit tests, however, often have no clear purpose, and thus generally are not given descriptive names.

In order to provide the benefits of well-named tests also to automated unit test generation, in this chapter we have presented a coverage based technique that will generate test names. This technique aims on naming tests based on what they do, with short and descriptive names. Furthermore, we evaluated this approach and our extensive experiments showed that developers agree with the tests, and the names are equally descriptive as manually derived test names. We even noticed an improvement in the ability of study participants to match

synthesized names with relevant tests and code. Consequently, our technique offers an easy and automated way to boost the usefulness of automatically generated tests.

The complete implementation of our coverage-based test naming technique has been developed as an extension to the open source EvoSuite test generation tool [70].

Chapter 6

Conclusions and Future Work

In this chapter, we draw together the various different strands of the work described in the earlier part of the thesis. The original challenge was motivated by a desire to support programmers in the maintenance part of the software development lifecycle, in particular to provide greater support in maintaining large unit test suites. Understanding saved tests is important (see Chapters 3, 5), both from the perspective of knowing when tests need to be modified, in response to code changes, and generally from the perspective of understanding the intent of the tested code, for which the saved tests are sometimes the only form of specification for what the code is supposed to do. With the advent of a multitude of new techniques for automatic test generation (see Chapter 2), the comprehension problem facing software developers has paradoxically become worse. Whereas automatic test generation algorithms relieve the programmer of the burden of having to think up the right tests to include in a compact and discriminating test suite, and whereas these algorithms usually do a better job than human test-authors in achieving test-coverage goals, the resulting test sequences are often incomprehensible. This is because the test generation algorithms are driven by optimising criteria that are opaque to the programmer. We wanted to find out how great a problem this posed in industry. For example, did programmers consider the comprehension of automatically-generated tests a problem; or would they be content simply to regenerate tests blindly every time that the tested software was modified? Did they consider it important to have software tests whose names and intent were manifestly clear? If so, could we do something to improve the state-of-the-art, by ensuring that automatic test-generators produced more comprehensible tests? This motivated the main objectives of the thesis, which included the following:

- To perform an empirical survey of programmers working in the software industry (see Chapter 3), in order to find out how widespread the practice of unit-testing has become;

and to find out the purposes for which they used their saved tests; and whether they considered test-maintenance a particular problem; and generally to find out what other issues or aspirations they had regarding unit testing;

- To create a new model of test-readability, using machine learning techniques, that faithfully reflected the perceptions of human software testers about test-readability (see Chapter 4); to find out whether test-readability is different from general code-readability; and to evaluate the test-readability model by applying it to collections of tests, seeking the level of agreement with human software testers;
- To create a new algorithm for test-name generation that made explicit the intent of the tests in the choice of test-names (see Chapter 5); to seek the level of agreement with human software testers on the choice of test-names; and to evaluate, by means of a comprehension test that asked programmers to match individual tests against properties of the tested code, whether the improved names would help in software maintenance.

6.1 Research Findings

During the course of this research, we discovered answers to the above questions; and also learned a number of ancillary lessons that were not directly relevant to the research questions, but which we feel are worthy of note.

6.1.1 Unit Testing Practices and Problems in Industry

We were keen to confirm how far the practice of unit-testing had penetrated the software industry. We wished to elicit the opinions of industry developers on the utility of unit-testing in general; to find out whether they thought that automatic test-generation techniques were useful; and whether they considered that test-suite maintenance was a problem. The responses to these questions would determine whether our work on test-naming improvements was timely. The survey that we conducted via AYTM (Ask Your Target Market) revealed the following (see Chapter 3):

- Unit-testing is widespread; and programmers carry out unit-testing from conviction, because they believe it improves the quality of their software, although many cite management pressure as the second-most influential factor, and customer satisfaction as relevant.

- Programmers spend one third of their time creating new software and two thirds on other aspects, including bug-fixing, code modification and, to a lesser extent, test-authoring. When investigating test failures, just over half ascribe the cause to out-of-date tests that needed maintaining.
- When asked about their testing goals, programmers preferred tests that exercised realistic scenarios over tests optimised according to other criteria, such as code coverage, or random testing. Made-up"toy" tests and random tests were distrusted and less likely to prompt code fixes.
- Programmers also prioritised maintainable tests, whether manually authored or generated by algorithm. In both cases, the tests were frequently integrated into the target codebase, such that test maintenance became a serious issue. Programmers needed to be able to identify out-of-date tests that no longer reflected the intent of the code.
- Programmers used automated test generation for a number of disparate reasons, to perform regression analysis, check for crashes, cover specifications, or verify sets of pre-defined assertions. Automated test generation helped answer the question of what to test, since the algorithms used internal criteria (such as code coverage), which was seen as an advantage.
- Desired improvements to automated test generation included the wish to generate more realistic tests that could be understood and maintained; and the need to identify what assertions to check at the end of a test (the test oracle problem), in those test-generation methods that did not use an independent specification for the test oracle.

Alongside these results, we also learned something about the suitability of online market research platforms, such as AYTM, for conducting software engineering surveys. These are run on a commercial basis, so the cost of using such platforms is relevant. We found that:

- AYTM gave access to a much larger pool of respondents from industry and academia (up to 20 million) than typically responded to message-board or newsgroup survey invitations; furthermore the respondents were pre-classified into useful demographic groups;
- The cost of AYTM increased with the number of pre-qualification questions asked; such that it was more cost-effective to ask just one public pre-qualification question, but include several secret pre-qualification questions in the main survey and disregard results from unreliable respondents;

- Respondents answering at least two out of three secret pre-qualification questions generated sufficiently reliable data for the purposes of our survey; and this was irrespective of the educational level reached by the respondent.
- AYTM is limited in the number of iterations for which a survey may be run, due to extra charges levied to exclude previous survey participants, after two survey iterations. This can drive down the size of the participant-pool and therefore indicates that surveys should be well planned in advance.

6.1.2 Developing a Model Predictor for Test Readability

The findings of the above survey indicated that programmers not only generate large legacy test suites, but also have to maintain these test suites manually (rather than by automatic means, such as re-generating the tests). This meant that there was a significant cost associated with test maintenance, in particular after test failures, where programmers had to identify whether faulty code or an out-of-date test needed fixing. This put a premium on the notion of test comprehension, which roughly equates to the ease and speed with which a programmer can understand the intent of a test, and match it up with the part of the software which it is testing. Test comprehension is closely related to test readability, so we sought to find an improved model predictor that would give judgements of test readability that closely mimicked the judgements of human software testers (see chapter 4). We found that:

- A simple linear regression learner for data mining [197] was eminently suitable, and learned quickly to make accurate judgements of test readability, based on a dataset of 450 human- and algorithm-generated tests and 15,699 human readability judgements, which were obtained via the Amazon Mechanical Turk crowdsourcing platform.
- The question of which features of the tests to measure was finessed by including all of the structural, logical complexity, and density code factors used by Buse and Weimer [38], as well as the entropy and possibly contentious Halstead features used by Posnett al. [154], which are thought unreliable by some (but see below for our compensation).
- The initial 116 features chosen were reduced, via iterative feature selection, to a compact set of 24 most predictive features, which together gave the best model performance and highest inter-rater agreement. This compensated for any negative effect of using contentious features, which were excluded if they eventually had no predictive power.
- Qualitatively, the readability of a test was found to depend on the maximum line-length and on the choice of identifiers. Identifier features that were strong predictors of

readability included the number of identifiers, their length and their diversity, as judged by byte- and token-entropy measures.

- Unexpectedly, the following had weak influence on test readability: code comments, line spacing ratios and the total number of statements in the test. The presence of exceptions, conditional statements and iterative loops likewise had only a very weak positive correlation with readability.

- Using 10-fold cross-validation performed on our dataset and on Buse and Weimer's original dataset [38] of 100 code snippets with 1200 human judgements, we confirmed that test readability was a distinct measure from general code readability. Our domain-specific model of test readability was finely tuned to test-code, rather than general code.

- It was possible to combine the test-readability predictor with automatic test generation, to generate more readable automated tests. The predictor was added as a second objective, as part of a multi-objective optimisation, in the search-based testing tool EVOSUITE; however we found that this created a dichotomy in the fitness function (see below).

- Whereas fitness should reward longer, redundant tests that eventually evolve to cover the whole state-space, readability rewarded shorter, non-redundant tests, reducing coverage. This was finessed by applying the readability predictor in a second post-processing phase, after coverage goals had been reached.

- In evaluations with human testers, 69% agreed with automatic judgements about test readability; and a test comprehension exercise (judging a pass/fail distinction) was completed 14% faster using the readability-optimised tests, to a similar accuracy.

- Comprehension of examples involving exceptional control flow was less accurate than normal control flow. Examples involving deeply-nested class hierarchies were challenging and took longer to answer. Respondents reported anecdotally that readable tests should be minimal (exclude redundant statements) and ideally consist of no more than five short statements involving few classes. Even for manually-authored tests, respondents highlighted the importance of making a judicious choice of descriptive variable- and method-names.

6.1.3 A Novel Technique for Test-Naming

Both the machine-learning experiments and the anecdotal comments from programmers revealed the impact of good names on test readability. In the feature clustering algorithm, identifier-related features were the ones that correlated most strongly with test readability. In the last phase of this research, we therefore sought to create a test-naming algorithm that could produce more evocative names for tests, which clearly indicated their intent. Since our goal was to support better test comprehension, we hypothesised that the most important aspects to record about a test were different kinds of coverage information: the operation that it was testing, the supplied inputs and expected output, or exception, or other goal for the test. This was intended to help programmers to match a test against the portion of code that it was exercising, thereby reducing the time and cost of test maintenance. We found that:

- Derived requirements for a readable test included that it should be uniquely distinguishable within a test suite; that its name should clearly describe what goal the test is carrying out; and that its name should bear a clear relationship to the portion of code under test.
- Assuming that test-names arecreated for methods in a JUnit test-driver, these had the typical form: "test" + method-name + "With" + (input-types | input-values) + ("Throws" + expected-exception |"Returning" + expected-outputs).
- Achieving test-distinctness produced longer names, whereas test-succinctness for the sake of readability preferred shorter names. The algorithm was tailored to generate test-names that were minimally distinct; test-names typically included the tested method and at least one test-goal indicator.
- Where tests had multiple coverage goals, and where only some of these should form part of the test name, these were ranked in the order: exception, method, output, input. This was expected to correspond with programmer's intuitions about test names.
- Evaluations were conducted with 47 programmers, using the SF110 open source Java software repository, from which we selected classes with suitable test-suites, for which we could generate comparable readability-optimised tests using EVOSUITE augmented with our novel name-generator.
- Samples were selected carefully following a 7-step protocol, to ensure that there were at least three distinct tests for each target method (requiring some name-complexity) and at least one test uniquely killed a mutant (supporting unique code-line identification).

- When asked to judge the appropriateness of a given test-name for a given body of test-code, respondents exhibited equal agreement over manual and automatic names, and disagreed less over automatic names, taking less time to respond for automatic names.
- When asked to select the most suitable test-name from three candidates for a given body of test-code, respondents identified suitable automatic names more accurately (compared to a baseline set by experts), partly because automatic names were more distinct.
- When asked to perform a comprehension test to identify which of three named tests exercised a particular line of target code, respondents were faster and more accurate in identifying the correct tests (judged against a baseline) with automatic names.

Overall, readability-optimised generated names were preferred over manually-chosen names. In the measures reported above, the results were statistically significant. In some other measures, such as the speed of selecting the best of three name-candidates, while there was a positive difference, this was not statistically significant. Finally, while this approach highlighted the particular importance of naming in test-readability and comprehension, a complete evaluation of test-readability should also take into account the benefits of test-minimisation (see Chapter 2).

6.2 In Support of the Thesis

The main hypotheses of this work are:

RQ1: Unit testing is an important step of software development in practice, and developers need more sophisticated tools that can generate maintainable unit tests.

The survey in Chapter 3 addresses this research question by performing an online questionnaire with developers working in industry. The survey is a result of 225 responses of different countries around the world revealing that unit testing is one way of improving software quality, and unit tests need to be easy maintainable (easy to detect updated code behavior, and change it).

RQ2: Automatically generated unit test cases can be improved in terms of readability, and help on test comprehension during software development and further software maintenance as a costly activity. Readability of unit test cases can be measured, and

added as an additional testing objective to improve the readability of automatically generated unit test cases.

This research question is investigated in Chapter 4, which proposes a domain specific model of readability based on human judgements that applies to object oriented unit test cases. This model is evaluated with an empirical study, and analysis of 116 syntactic features of unit tests shows that readability is not simply a matter of the overall test length; several features related to individual lines, identifiers, or entropy have a stronger influence. Our optimized model results with a strong correlation with the user data, which improves over the ability of general code readability models. Our technique to improve tests succeeds in increasing readability in more than half of all automatically generated unit tests, and validation with humans confirms that the optimized tests are preferred.

RQ3: Descriptive identifier names increase test comprehension and readability during software development and maintenance.

Third research question is covered in Chapter 5 that presents an extensive evaluation of a novel test naming technique. The results of this empirial evaluation show that participants agreed with the names synthesized with our technique. The synthesized names are as descriptive as manually written names, participants were slightly better at matching tests with synthesized names than they were at matching tests with manually written names. Finally, participants of our study were more accurate at identifying relevant tests for given pieces of code using synthesized test names.

6.3 Future Work

The idea of improving test readability and naming and reduce test oracle cost or human effort during test maintenance, opens several new research directions:

6.3.1 Extending Readability Model with a larger dataset

As future work we will have a new readability metric that will be based on a larger set of Java open-source projects containing manually written test cases, and a larger set of readability features (10 additional test readability features). From this dataset, in order to reduce the number of features and instances (test cases) for model training (reduce the noise in the model and improve its performance), feature clustering may be applied. With feature clustering we may be able to group similar features together such that redundant features are avoided and

only discriminating features remain. The proposed approach will apply correlation-based clustering and group highly correlated features with ($\rho > 0.5$).

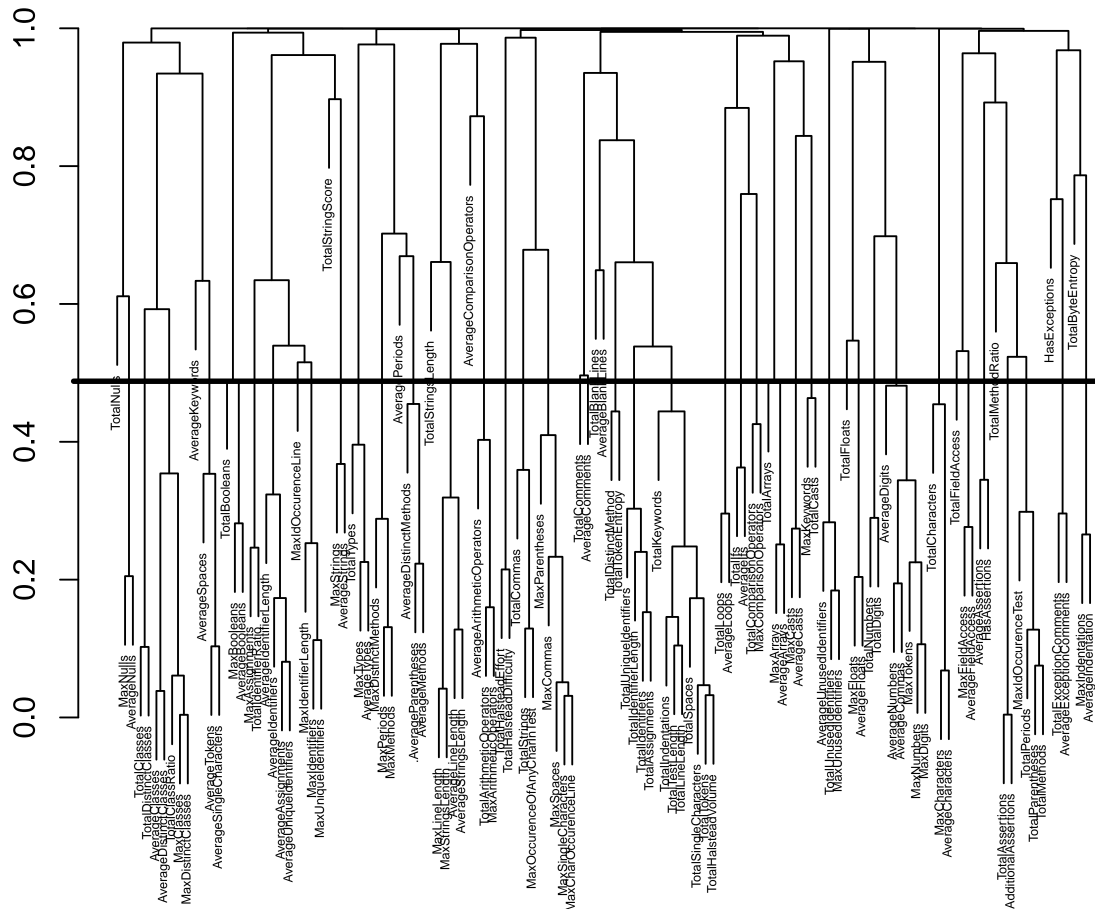


Fig. 6.1 Readability feature cluster - dendrogram

The application of feature clustering using *R-language* on 125 readability features, resulted on 53 clusters or 53 centroid features respectively. Centroid or representative feature is chosen the one that is mostly correlated with the rest of the features in the cluster, and it is used for readability model learning. Figure 6.1 is the dendrogram[1] of features. This plot represents feature differences (1-correlation_value) based on correlation, which means that the features with lower value have higher correlation and the cut-line (0.49) decides the clusters.

Furthermore, 53 centroid features (e.g., AverageLineLength, TotalNumbers, MaxNulls,...) are further used for test selection for the empirical study that will collect human judgements

[1] https://www.r-graph-gallery.com/dendrogram

of readability. The training set of test cases and their readability score will be used for model learning, which further will be used for readable test generation.

6.3.2 Understandability Model

In Chapter 4 the experiment about the effect of readability on test understandability has also demonstrated the boundaries between these two features. This means that, not all tests that look nice are also easy to understand. Therefore, a new understandability model will improve generated test cases enriched with aspects of understandability.

However, an understandability model for test generation will need semantic features, too. For example, features such as code coverage, class dependency, readability level, can measure the complexity in a test case. The more complex a class is, its generated tests will be larger, have more variables, more class instances, or more method calls. In general, this all will effect the time a developer will need to understand that test, Furthermore, additional understandability features can be such as time to answer the questions for a test, and correctness of questions for a test, which will also contribute in the final understandability model. Therefore, in future we are planing to work in an understandability model that can learn from test complexity and further contribute to simpler generated test cases.

6.3.3 Unit Test Name Improvement

In Chapter 5 we presented a process of descriptive test naming that achieved to generate improved test names acceptable from user. However, there is scope to improve the synthesized names further. In particular, our naming technique currently only uses assertions to rank coverage goals, but it would be possible to utilize them to derive better explanations of test purposes. A further possible improvement would be to adapt the synthesized names to the patterns and conventions used in the software project under test (cf. [93]). Furthermore, we plan to investigate whether our technique is also applicable for renaming manually written unit tests.

6.3.3.1 Generating Unit Tests with Descriptive Variable Names

Method names are not the only identifiers involved in unit test generation; as future work we will investigate how generating useful variable names can influence test understanding and maintainability. It is also conceivable to extend our technique to synthesize not only short method names, but more elaborate explanations or summaries [149, 208].

The main purpose of this future work is the further reduction of human effort during testing. The application of such a large test improvement model will mainly reduce the total software test time and its further costs.

6.4 Final Comments

Software testing is used to verify and validate software and provide evidence for its quality. Testing software manually is an effort and time requiring process, therefore, generating tests automatically is an improvement. However, as generated test cases usually aim at code coverage, they do not look as nice as manually written tests (in terms of readability and understandability), which raises the need of having maintainable test cases. As such, this thesis explores the problem of automatic test generation, and test readability that will help in producing readable/maintainable unit test case. We started this work by conducting a survey on unit testing practices and problems with developers in the industry. The results of the survey show how unit testing is an important part of software development, and suggest that there is indeed potential and need for research on automation of unit testing. Furthermore, they help us to identify the areas of importance such as maintenance of unit tests. Second, based on the last findings and in order to improve test generation, we trained a test readability model with 116 readability features that apply to object-oriented unit test cases. This model that is based on human judgements of readability was found to increase test readability, and affect user preference on choosing improved tests (user likes more improved tests). Although the resulting technique was successful, it is quite limited in the scope of its changes to test appearance. Therefore, in the last part of this thesis, a test naming algorithm is presented that can be applied to any test generation tool and change test appearance by producing descriptive test names. Overall the contributions of this thesis are a unit testing survey that identifies areas of importance for unit testing on which further research will be necessary, an approach that can automatically generate test suites with both high coverage and also improved readability, and an approach which generates descriptive names for automatically generated unit tests by summarizing API-level coverage goals. Finally, as unit testing is a common technique applied during development, automatically generating tests with less and shorter lines, less and no unused variables, more comments, shorter and meaningful strings, fewer assertions, shorter and meaningful test names, and more than 100 other improved readability features, will more likely be adopted by practitioners. Hence, if readable tests are adopted, they will be used more and the quality of the software will increase. This means that software programs of high quality will be used longer, require less maintenance, and overall will have fewer expenses.

Chapter 7

Appendix - A

7.1 Survey Questions and Data

This appendix lists in Table 7.1 all the questions that were asked in the survey on "Unit Testing Practices and Problems. Each question is followed with the options and the number of answers collected from users. The last rows specify the inter-rater-reliability, confidence interval, and p-value, respectively.

Table 7.1 Survey questions and response data.

Question	Answer							IRR	95 %-CI	p-value
Q1: What motivates you to write unit tests?								0.965	[0.903,0.996]	< 0.01
	Extremely influential	Very influential	Moderately influential	Somewhat influential	Slightly influential	Hardly influential	Not at all influential			
Required by management	41	36	45	28	12	2	7			
Demanded by customer	32	19	58	29	17	9	7			
Own conviction	37	39	47	28	14	3	3			
Peer pressure	15	21	40	29	27	24	15			
Other	17	21	36	57	21	8	11			
Q2:How do you spend your software development time (in percentages)? ***(Listing averages)***								0.972	[0.921, 0.997]	< 0.01
Writing new code	33.04%									
Writing new tests	15.8%									
Debugging/fixing	25.3%									
Refactoring	17.4%									
Other	8.4%									
Q3: When a test case fails, what do you typically do? (Percentage of test failures) ***(Listing averages)***								0.985	[0.952, 0.999]	< 0.01
Fix the code until the test passes	47.2%									
Repair the test to reflect changed behaviour	25.9%									
Delete or comment out the failing test or assertion	15.6%									
Ignore the failure	11.2%									
Q4:How important are the following aspects for you when you write new unit tests?								0.621	[0.082, 0.922]	0.0153
	Extremly important	Very important	Moderately important	Neutral	Slightly important	Low importance	Not at all			
Code coverage	30	41	51	34	12	2	1			
Execution speed	28	32	60	31	11	7	2			
Robustness against code changes (i.e. test does	32	36	57	31	10	4	1			
not break easily)	37	57	35	29	8	3	2			
How realistic the test scenario is	37	33	58	28	11	3	1			
How easily faults can be localised/debugged if	31	30	60	33	12	4	1			
the test fails	30	35	57	30	12	3	4			
How easily the test can be updated when the underlying code changes										
Sensitivity against code changes (i.e. test should detect even small code changes)										
Q5:Please select which techniques you apply when writing new tests (select "Never" if you don't know a technique)								0.921	[0.81, 0.984]	< 0.01
	Always	Very frequently	Frequently	Occasionally	Rarely	Very rarely	Never			
Automated test generation (e.g., test inputs are	22	28	43	40	15	5	18			
created automatically)	27	31	47	44	8	5	9			
Code coverage analysis	22	18	44	34	11	10	0			
Mutation analysis	24	26	46	44	13	12	6			
Test-driven development (i.e., tests are written	29	29	53	40	7	2	11			
before code)	25	20	40	41	18	10	17			
Systematic testing approaches (e.g., boundary	21	11	32	59	11	14	23			
value analysis)										
Mocking/stubbing										
Other										
Q6:What do you use automated unit test generation for (skip if you do not use automated unit test generation)?								0.04	-	0
Exercising specifications(e.g.,JML,code contracts)	51	Finding crashes and undeclared exceptions			63					
Exercising assertions in the code	58	Regression testing			58					
Exercising parameterised unit tests	58	To complement manually written tests			12					
Other	12									
Q7: Please rank the following aspects of writing a new unit test according to their difficulty.								0.477	[-0.496, 0.935]	0.113
	Position 1	Position 2	Position 3	Position 4	Position 5					
Determining what to check (i.e., finding good assertions)	35	32	32	35	37					
Finding and creating relevant input values	40	30	39	34	28					
Identifying which code/scenario to test	28	26	37	39	41					
Finding a sequence of calls to bring the unit under test into the target state	41	42	26	34	28					
Isolating the unit under test (e.g. handling databases, filesystem, dependencies)	27	41	37	29	37					
Q8: What makes it difficult to fix a failing test? Please rank by importance.								0.538	[-0.295, 0.944]	0.0713
	Position 1	Position 2	Position 3	Position 4	Position 5					
The test reflects outdated behaviour	40	45	29	33	24					
The test is difficult to understand	35	40	29	38	29					
The code under test is difficult to understand	38	27	32	32	42					
The test reflects unrealistic behaviour (e.g. unrealistic mock objects)	31	37	39	20	44					
The test is flaky (i.e., it fails nondeterministically)	27	22	42	48	0					

Q9: Please indicate your level of agreement with the following statements.								0.865	[0.653, 0.978]< 0.01
	Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly Disagree		
Writing unit tests is difficult	19	21	47	47	20	12	5		
I enjoy writing unit tests	30	22	35	41	19	18	6		
I would like to have more tool support when	34	27	65	32	7	4	2		
writing unit tests	27	26	42	43	21	7	5		
I would like to have more unit tests	26	14	56	40	21	9	5		
Maintaining unit tests is difficult	28	28	53	40	10	9	3		
I usually have sufficiently many unit tests									
Q10: How many years of professional programming experience (i.e. in your job) do you have?									
	Less than 1 year		1-3 years	4-10 years	More than 10 years		No prof. exp.		
	13		54	49	51		4		
Q11: How large is the typical software development team you work in?									
	1		2-5	6-10	More than 10				
	16		65	53	34				
Q12: Which programming languages do you commonly use?									
	Java		90	Python	41				
	C#		68	JavaScript	95				
	Objective C		33	Ruby	29				
	C		64	Visual Basic	74				
	Scala		24	Other	42				
Q13: Please select all statements that are true									
A unit test is a small piece of code testing a unit under test	126		(Correct = yes)						
Unit tests are written (or generated) by the developer	132		(Correct = yes)						
If all unit tests pass, the software is correct	133		(Correct = no)						

References

[1] Borda count. URL http://www.oxfordreference.com/view/10.1093/oi/authority.20110803095518715.

[2] Junit. URL http://junit.org/junit4/.

[3] Nunit. URL https://www.nunit.org/.

[4] Robin Abraham and Martin Erwig. Autotest: A tool for automatic test case generation in spreadsheets. In *VL/HCC*, pages 43–50. IEEE Computer Society, 2006. ISBN 0-7695-2586-5. URL http://dblp.uni-trier.de/db/conf/vl/vlhcc2006.html#AbrahamE06.

[5] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 352–361, 2013. doi: 10.1109/ICST.2013.11. URL http://dx.doi.org/10.1109/ICST.2013.11.

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

[7] Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1345–1348. IEEE Press, 2012.

[8] Sandra Aluísio, Lucia Specia, Caroline Gasperin, and Carolina Scarton. Readability assessment for text simplification. In *Proceedings of 5th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2010)*, Los Angeles, CA, USA, 2010.

[9] Scott Ambler. Quality in an agile world. *Software Quality Professional*, 7(4):34, 2005.

[10] Saswat Anand, Corina Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[11] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792771.

[12] Saswat Anand, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013. doi: 10.1016/j.jss.2013.02.061.

[13] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 144–153, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321654. URL http://doi.acm.org/10.1145/1321631.1321654.

[14] James H Andrews, Felix CH Li, and Tim Menzies. Nighthawk: A two-level genetic-random unit test data generator. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 144–153. ACM, 2007.

[15] Andrea Arcuri and Lionel Briand. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)*, 24(3):219–250, 2014. ISSN 1099-1689. doi: 10.1002/stvr.1486.

[16] Andrea Arcuri and Lionel C. Briand. Adaptive random testing: an illusion of effectiveness? In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 265–275. ACM, 2011. ISBN 978-1-4503-0562-4. URL http://dblp.uni-trier.de/db/conf/issta/issta2011.html#ArcuriB11.

[17] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, Nov 2014. ISSN 0098-5589. doi: 10.1109/TSE.2014.2342227.

[18] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62. IEEE Computer Society, 2011.

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *arXiv preprint arXiv:1610.00502*, 2016.

[20] Pawan Luthra Baljinder Singh. Study of lehman's laws and metrics during software evolution. 2015.

[21] Luciano Baresi and Matteo Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 281–284, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: 10.1145/1810295.1810353. URL http://doi.acm.org/10.1145/1810295.1810353.

[22] E.T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering (TSE)*, PP (99), 2014. ISSN 0098-5589. doi: 10.1109/TSE.2014.2372785.

[23] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 56–65. IEEE, 2012.

[24] K Beck and E Gamma. Junit test infected: Programmers love writing tests, 2001.

[25] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[26] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[27] Rebekah George Benjamin. Reconstructing readability: Recent developments and recommendations in the analysis of text difficulty. *Educational Psychology Review*, 24(1):63–88, 2012.

[28] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. pages 73–87, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336534. URL http://doi.acm.org/10.1145/336512.336534.

[29] Antonia Bertolino. Issta 2002 panel: Is issta research relevant to industrial users? *SIGSOFT Softw. Eng. Notes*, 27(4):201–202, July 2002. ISSN 0163-5948. doi: 10.1145/566171.566205. URL http://doi.acm.org/10.1145/566171.566205.

[30] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[31] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[32] Jürgen Börstler, Michael E Caspersen, and Marie Nordström. Beauty and the beast: on the readability of object-oriented example programs. *Software quality journal*, 24 (2):231–246, 2016.

[33] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013.

[34] P Bourque and R Dupuis. Guide to the software engineering body of knowledge, 2004 edition, vol. 1, 2004.

[35] Robert S Boyer, Bernard Elspas, and Karl N Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6): 234–245, 1975.

[36] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Symbolic execution of programs with heap inputs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 602–613. ACM, 2015.

[37] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 443–446. IEEE, 2008.

[38] Raymond P. L. Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.70. URL http://dx.doi.org/10.1109/TSE.2009.70.

[39] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 121–130, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390647. URL http://doi.acm.org/10.1145/1390630.1390647.

[40] Raymond PL Buse and Thomas Zimmermann. Information needs for software development analytics. In *Proceedings of the 34th international conference on software engineering*, pages 987–996. IEEE Press, 2012.

[41] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, New York, NY, USA, 2006. ACM. ISBN 1-59593-518-5. doi: 10.1145/1180405.1180445. URL http://doi.acm.org/10.1145/1180405.1180445.

[42] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008. URL http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[43] Jean Carletta. Assessing Agreement on Classification Tasks: The Kappa Statistic. *Computational Linguistics*, 22(2):249–254, 1996. ISSN 0891-2017.

[44] Jeffrey Carver, Letizia Jaccheri, Sandro Morasca, and Forrest Shull. Issues in using students in empirical studies in software engineering education. In *IEEE Int. Software Metrics Symposium*, pages 239–249, 2003.

[45] Silvia Cateni, Marco Vannucci, Marco Vannocci, and Valentina Colla. *Multivariate Analysis in Management, Engineering and the Sciences*. InTech, 2013-01-09.

[46] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. An industrial survey on contemporary aspects of software testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 393–401. IEEE, 2010.

[47] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D. Nguyen, and Paolo Tonella. An Empirical Study About the Effectiveness of Debugging when Random Test Cases Are Used. In *International Conference on Software Engineering (ICSE)*, pages 452–462, 2012. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337277.

[48] Shi Kuo Chang, GEORGE SPANOUDAKIS, and ANDREA ZISMAN. *Handbook of Software Engineering & Knowledge Engineering*. World Scientific Publ., 2005.

[49] Tsong Yueh Chen, Hing Leung, and I. K. Mak. In Michael J. Maher, editor, *ASIAN*, pages 320–329. Springer. ISBN 3-540-24087-X.

[50] Yih-Farn Chen, David S Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th international conference on Software engineering*, pages 211–220. IEEE Computer Society Press, 1994.

[51] Emilio Collar Jr and Ricardo Valerdi. Role of software readability on software development cost. Technical report, 2006.

[52] Christophe Croux and Catherine Dehon. Influence functions of the spearman and kendall correlation measures. *Statistical methods & applications*, 19(4):497–515, 2010.

[53] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004. ISSN 0038-0644. doi: 10.1002/spe.602. URL http://dx.doi.org/10.1002/spe.602.

[54] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 201–211. IEEE, 2014.

[55] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. Generating readable unit tests for guava. In *International Symposium on Search Based Software Engineering*, pages 235–241. Springer, 2015.

[56] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. pages 107–118. ACM, 2015.

[57] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67. ACM, 2017.

[58] E. Dale and J.S. Chall. A formula for predicting readability: Instructions. *Educational Research Bulletin*, 27(2):37–54, 1948.

[59] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. ReAssert: Suggesting Repairs for Broken Unit Tests. In *International Conference on Automated Software Engineering (ASE)*, pages 433–444, 2009. ISBN 978-0-7695-3891-4. doi: 10.1109/ASE.2009.17. URL http://dx.doi.org/10.1109/ASE.2009.17.

[60] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 207–218. ACM, 2010.

[61] Manoranjan Dash, Kiseok Choi, Peter Scheuermann, and Huan Liu. Feature Selection for Clustering - A Filter Solution. In *International Conference on Data Mining (ICDM)*, pages 115–122, 2002. ISBN 0-7695-1754-4. URL http://dl.acm.org/citation.cfm?id=844380.844731.

[62] Jonathan de Halleux and Nikolai Tillmann. Moles: Tool-assisted environment isolation with closures. In *International Conference on Objects, Models, Components, Patterns*, TOOLS'10, pages 253–270. Springer-Verlag, 2010. ISBN 3-642-13952-3, 978-3-642-13952-9. URL http://dl.acm.org/citation.cfm?id=1894386.1894400.

[63] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6:182–197, 2000.

[64] Lionel E Deimel Jr. The uses of program reading. *ACM SIGCSE Bulletin*, 17(2):5–14, 1985.

[65] Rakkrit Duangsoithong and Terry Windeatt. Relevance and Redundancy Analysis for Ensemble Classifiers. In *Machine Learning and Data Mining in Pattern Recognition*, volume 5632, pages 206–220, 2009. ISBN 978-3-642-03069-7. doi: 10.1007/978-3-642-03070-3_16. URL http://dx.doi.org/10.1007/978-3-642-03070-3_16.

[66] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[67] Ronan Fitzpatrick. Software quality: definitions and strategic issues. *Reports*, page 1, 1996.

[68] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[69] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[70] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025179. URL http://doi.acm.org/10.1145/2025113.2025179.

[71] Gordon Fraser and Andrea Arcuri. Whole test suite generation. 39(2):276–291, 2013.

[72] Gordon Fraser and Andrea Arcuri. A large scale evaluation of automated unit test generation using evosuite. 24(2):8, 2014.

[73] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the 9th International Workshop on Search-Based Software Testing*, pages 33–36. ACM, 2016.

[74] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0. doi: 10.1145/1831708.1831728. URL http://doi.acm.org/10.1145/1831708.1831728.

[75] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 80–89, 2011. doi: 10.1109/ICST.2011.53. URL http://dx.doi.org/10.1109/ICST.2011.53.

[76] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 28(2):278–292, 2012. ISSN 0098-5589.

[77] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. on Software Engineering (TSE)*, 38(2):278–292, 2012.

[78] Zachary P. Fry and Westley Weimer. A Human Study of Fault Localization Accuracy. In *International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010. ISBN 978-1-4244-8630-4. doi: 10.1109/ICSM.2010.5609691. URL http://dx.doi.org/10.1109/ICSM.2010.5609691.

[79] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 360–369. IEEE, 2013.

[80] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 360–369. IEEE, 2013.

[81] Vahid Garousi and Junji Zhi. A survey of software testing practices in Canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.

[82] Adam M Geras, MR Smith, and J Miller. A survey of software testing practices in alberta. *Electrical and Computer Engineering, Canadian Journal of*, 29(3):183–191, 2004.

[83] Patrice Godefroid. Compositional dynamic test generation. *SIGPLAN Not.*, 42(1): 47–54, January 2007. ISSN 0362-1340. doi: 10.1145/1190215.1190226. URL http://doi.acm.org/10.1145/1190215.1190226.

[84] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065036. URL http://doi.acm.org/10.1145/1064978.1065036.

[85] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012. ISSN 1542-7730. doi: 10.1145/2090147.2094081. URL http://doi.acm.org/10.1145/2090147.2094081.

[86] Arthur C. Graesser, Danielle S. Mcnamara, Max M. Louwerse, Zhiqiang Cai, Kyle Dempsey, Y Floyd, Phil Mccarthy, Yasuhiro Ozuru, Margie Petrowski, Srinivas Pillarisetti, Mack Reese, Mike Rowe, Jayme Sayroo, Kim Sumara, and Fang Yang Correspondence. Coh-metrix: Analysis of text on cohesion and language. In *M. Louwerse Topics in Cognitive Science*, page 27, 2004.

[87] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 322–331. IEEE, 2013.

[88] Michaela Greiler, Andy Zaidman, Arie van Deursen, and Margaret-Anne Storey. Strategies for avoiding text fixture smells during software evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 387–396. IEEE Press, 2013.

[89] Florian Gross, Gordon Fraser, and Andreas Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 67–77. ACM, 2012.

[90] Maurice H Halstead. Elements of software science. 1977.

[91] Paul Hamill. *Unit Test Frameworks: Tools for High-Quality Software Development*. " O'Reilly Media, Inc.", 2004.

[92] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. Is This a Bug or an Obsolete Test? In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–628. 2013. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8_25. URL http://dx.doi.org/10.1007/978-3-642-39038-8_25.

[93] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In *ECOOP 2009 – Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317. Springer, 2009. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_14. URL http://dx.doi.org/10.1007/978-3-642-03013-0_14.

[94] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, (4):266–278, 1977.

[95] IABG. Federal republic of germany government standard. 1992.

[96] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 425–428. ACM, 2007.

[97] ISO ISO. Iec 12207 information technology-software life cycle processes. *ISO, Geneva*, 1995.

[98] jtest. Parasoft JTest, 2016. URL www.parasoft.com/jtest. Last visited on 30.08.2016.

[99] Natalia Juristo, Ana M Moreno, and Wolfgang Strigel. Software testing practices in industry. *IEEE software*, 23(4):19–21, 2006.

[100] René Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. pages 433–436, 2014.

[101] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 201–209. IEEE, 2009.

[102] Mohd Ehmer Khan. Different approaches to black box testing technique for finding errors. *International Journal of Software Engineering & Applications*, 2(4):31, 2011.

[103] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 3(6), 2012.

[104] Mohd Ehmer Khan et al. Different approaches to white box testing technique for finding errors. *International Journal of Software Engineering and Its Applications*, 5 (3):1–14, 2011.

[105] Sarfraz Khurshid, Corina S. PĂsĂreanu, and Willem Visser. *Generalized Symbolic Execution for Model Checking and Testing*, pages 553–568. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-36577-8. doi: 10.1007/3-540-36577-X_40. URL http://dx.doi.org/10.1007/3-540-36577-X_40.

[106] James C King. A new approach to program testing. In *ACM SIGPLAN Notices*, volume 10, pages 228–233. ACM, 1975.

[107] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252.

[108] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. 28(8):721–734, August 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1027796.

[109] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing User Studies with Mechanical Turk. In *Conference on Human Factors in Computing Systems (CHI)*, pages 453–456, 2008. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357127. URL http://doi.acm.org/10.1145/1357054.1357127.

[110] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16 (8):870–879, August 1990. ISSN 0098-5589. doi: 10.1109/32.57624. URL http://dx.doi.org/10.1109/32.57624.

[111] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1759–1766. ACM, 2008.

[112] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. Flopsy-search-based floating point constraint solving for symbolic execution. In *IFIP International Conference on Testing Software and Systems*, pages 142–157. Springer, 2010.

[113] J Lee, S Kang, and D Lee. Survey on software testing practices. *IET software*, 6(3): 275–282, 2012.

[114] Meir M Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.

[115] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[116] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[117] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2482-6. doi: 10.1109/ISSRE.2005.28. URL http://dx.doi.org/10.1109/ISSRE.2005.28.

[118] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321698. URL http://doi.acm.org/10.1145/1321631.1321698.

[119] B. Li, C. Vendome, M. Linares-VÃ¡squez, D. Poshyvanyk, and N. A. Kraft. Automatically documenting unit test cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 341–352, April 2016. doi: 10.1109/ICST.2016.30.

[120] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012. ISSN 0928-8910. doi: 10.1007/s10515-011-0092-1. URL http://dx.doi.org/10.1007/s10515-011-0092-1.

[121] Bennet P Lientz and E Burton Swanson. Software maintenance management. 1980.

[122] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[123] Chang Liu. Platform-independent and tool-neutral test descriptions for automated software testing. In *Proceedings of the 22nd international conference on Software engineering*, pages 713–715. ACM, 2000.

[124] Zicong Liu, Zhenyu Chen, Chunrong Fang, and Qingkai Shi. Hybrid test data generation. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 630–631. ACM, 2014.

[125] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[126] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: An automated test generator using orchestrated program analysis. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 842–847, Nov 2015. doi: 10.1109/ASE.2015.102.

[127] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 436–439. IEEE, 2011.

[128] Salvatore Mamone. The ieee standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, 19(1):75–76, 1994.

[129] G Harry Mc Laughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.

[130] G Harry Mc Laughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.

[131] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[132] P. McMinn and M. Holcombe. Evolutionary testing using an extended chaining approach. *Evol. Comput.*, 14(1):41–64, March 2006. ISSN 1063-6560. doi: 10.1162/106365606776022742. URL http://dx.doi.org/10.1162/106365606776022742.

[133] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004. ISSN 0960-0833. doi: 10.1002/stvr.v14:2. URL http://dx.doi.org/10.1002/stvr.v14:2.

[134] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[135] James Miller, Marek Reformat, and Howard Zhang. Automatic test data generation using genetic algorithm and program dependence graphs. *Information and Software Technology*, 48(7):586–605, 2006.

[136] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2(3):223–226, May 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233818. URL http://dx.doi.org/10.1109/TSE.1976.233818.

[137] M. Mirzaaghaei, F. Pastore, and M. Pezzè. Supporting Test Suite Evolution through Test Case Adaptation. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 231–240, 2012. doi: 10.1109/ICST.2012.103.

[138] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[139] David Molnar, Xue Cong Li, and David Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *USENIX Security Symposium*, volume 9, pages 67–82, 2009.

[140] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[141] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[142] SP Ng, Tafline Murnane, Karl Reed, D Grant, and TY Chen. A preliminary survey on software testing practices in Australia. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 116–125. IEEE, 2004.

[143] Srinivas Nidhra and Jagruthi Dondeti. Blackbox and whitebox testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[144] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality*, pages 242–256. Springer, 2005.

[145] Manuel Oriol and Sotirios Tassis. Testing .net code with yeti. In Radu Calinescu, Richard F. Paige, and Marta Z. Kwiatkowska, editors, *ICECCS*, pages 264–265. IEEE Computer Society, 2010. ISBN 978-0-7695-4015-3. URL http://dblp.uni-trier.de/db/conf/iceccs/iceccs2010.html#OriolT10.

[146] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[147] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84. IEEE Computer Society, 2007. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.37. URL http://dx.doi.org/10.1109/ICSE.2007.37.

[148] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: what if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 130–141. ACM, 2016.

[149] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. pages 547–558, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884847. URL http://doi.acm.org/10.1145/2884781.2884847.

[150] V. Petridis, S. Kazarlis, and A. Bakirtzis. Varying fitness functions in genetic algorithm constrained optimization: the cutting stock and unit commitment problems. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 28(5):629–640, Oct 1998. ISSN 1083-4419.

[151] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 33. ACM, 2012.

[152] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Testevol: A tool for analyzing test-suite evolution. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1303–1306. IEEE Press, 2013.

[153] Strategic Planning. The economic impacts of inadequate infrastructure for software testing. 2002.

[154] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A Simpler Model of Software Readability. In *Working Conference on Mining Software Repositories (MSR)*, pages 73–82, 2011. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985454. URL http://doi.acm.org/10.1145/1985441.1985454.

[155] I. S. Wishnu B. Prasetya. T3, a combinator-based random testing tool for java: Benchmarking. In Tanja E. J. Vos, Kiran Lakhotia, and Sebastian Bauersfeld, editors, *FITTEST@ICTSS*, volume 8432 of *Lecture Notes in Computer Science*, pages 101–110. Springer, 2013. ISBN 978-3-319-07784-0. URL http://dblp.uni-trier.de/db/conf/pts/fittest2013.html#Prasetya13.

[156] I. S. Wishnu B. Prasetya. T3i: A tool for generating and querying test suites for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 950–953, 2015. ISBN 978-1-4503-3675-8.

[157] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *ACM Trans. Softw. Eng. Methodol.*, 21 (3):19:1–19:29, July 2012. ISSN 1049-331X. doi: 10.1145/2211616.2211622. URL http://doi.acm.org/10.1145/2211616.2211622.

[158] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 36–42. IEEE Press, 2012.

[159] Darrell R Raymond. Reading source code. In *Proceedings of the 1991 conference of the Centre for Advanced Studies on Collaborative research*, pages 3–16. IBM Press, 1991.

[160] Brian Robinson, Michael D Ernst, Jeff H Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *International Conference on Automated Software Engineering (ASE)*, pages 23–32. IEEE Computer Society, 2011.

[161] Marko Robnik-Šikonja and Igor Kononenko. Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning*, 53(1-2):23–69, 2003. ISSN 0885-6125. doi: 10.1023/A:1025667309714. URL http://dx.doi.org/10.1023/A:1025667309714.

[162] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *SSBSE 2015*, volume 9275 of *LNCS*, pages 93–108. Springer, 2015. ISBN 978-3-319-22182-3. doi: 10.1007/978-3-319-22183-0_7. Best Paper Award (Industry-relevant SBSE results).

[163] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on software engineering*, 22(8):529–551, 1996.

[164] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[165] Spencer Rugaber. The use of domain knowledge in program understanding. *Annals of Software Engineering*, 9(1-2):143–192, 2000.

[166] Per Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.

[167] Abdelilah Sakti, Yann-Gaël Guéhéneuc, and Gilles Pesant. Boosting search based testing by using constraint based testing. In *International Symposium on Search Based Software Engineering*, pages 213–227. Springer, 2012.

[168] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, 41(3):294–313, 2015.

[169] I. Salman, A. T. Misirli, and N. Juristo. Are students representatives of professionals in software engineering experiments? volume 1, pages 666–676, 2015. doi: 10.1109/ICSE.2015.82.

[170] I. B. Sampaio and L. Barbosa. Software readability practices and the importance of their teaching. In *2016 7th International Conference on Information and Communication Systems (ICICS)*, pages 304–309, 2016.

[171] Suguru Matsuyoshi Satoshi Sato and Yohsuke Kondoh. Automatic assessment of japanese text readability based on a textbook corpus. In Bente Maegaard Joseph Mariani Jan Odijk Stelios Piperidis Daniel Tapias Nicoletta Calzolari (Conference Chair), Khalid Choukri, editor, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco, may 2008. European Language Resources Association (ELRA). ISBN 2-9517408-4-0. http://www.lrec-conf.org/proceedings/lrec2008/.

[172] Carolina Scarton and Sandra M. Aluísio. Coh-metrix-port: a readability assessment tool for texts in brazilian portuguese. In *International Conference on Computational Processing of Portuguese (PROPOR 2010)*, Porto Alegre-RS, Brazil, 2010.

[173] Andy Schneider. Junit best practices. *Java World*, 12:181, 2000.

[174] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software*

Engineering, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: 10.1145/1081706.1081750. URL http://doi.acm.org/10.1145/1081706.1081750.

[175] Anthony JH Simons. Testing with guarantees and the failure of regression testing in extreme programming. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 118–126. Springer, 2005.

[176] Anthony JH Simons. A theory of regression testing for behaviourally compatible object types. *Software Testing, Verification and Reliability*, 16(3):133–156, 2006.

[177] Anthony JH Simons. Jwalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 14(4):369–418, 2007.

[178] Anthony JH Simons and Christopher D Thomson. Lazy systematic unit testing: Jwalk versus junit. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 138–138. IEEE, 2007.

[179] Anthony JH Simons and Christopher D Thomson. Feedback-based specification, coding and testing with jwalk. In *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference*, pages 69–73. IEEE, 2008.

[180] Anthony JH Simons and Christopher D Thomson. Benchmarking effectiveness for object-oriented unit testing. In *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*, pages 375–379. IEEE, 2008.

[181] Anthony JH Simons and Wenwen Zhao. Dynamic analysis of algebraic structure to optimize test generation and test case selection. In *Testing: Academic and Industrial Conference-Practice and Research Techniques, 2009. TAIC PART'09.*, pages 33–42. IEEE, 2009.

[182] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. Cheap and Fast—but is It Good?: Evaluating Non-expert Annotations for Natural Language Tasks. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 254–263, 2008. URL http://dl.acm.org/citation.cfm?id=1613715.1613751.

[183] Matt Staats, Gregory Gay, and Mats PE Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *International Conference on Software Engineering*, pages 870–880. IEEE Press, 2012.

[184] SES Subcommittee. Ieee standard for software maintenance. *IEEE Std*, pages 1219–1993, 1992.

[185] Yahya Tashtoush, Zeinab Odat, Izzat Alsmadi, and Maryan Yatim. Impact of programming features on code readability. 2013.

[186] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mseqgen: object-oriented unit-test generation via mining source code. 2009. URL http://dblp.uni-trier.de/db/conf/sigsoft/fse2009.html#ThummalapentaXTHS09.

[187] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *ACM SIGPLAN Notices*, volume 46, pages 189–206. ACM, 2011.

[188] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. *SIGPLAN Not.*, 46(10):189–206, October 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048083. URL http://doi.acm.org/10.1145/2076021.2048083.

[189] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79123-X, 978-3-540-79123-2. URL http://dl.acm.org/citation.cfm?id=1792786.1792798.

[190] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29 (4):119–128, July 2004. ISSN 0163-5948. doi: 10.1145/1013886.1007528. URL http://doi.acm.org/10.1145/1013886.1007528.

[191] Richard Torkar and Stefan Mankefors. A survey on testing and reuse. In *Software: Science, Technology and Engineering, 2003. SwSTE'03. Proceedings. IEEE International Conference on*, pages 164–173. IEEE, 2003.

[192] Arie Van Deursen and Leon Moonen. The video store revisited–thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002.

[193] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[194] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932. ACM, 2006.

[195] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[196] Darrell Whitley, V Scott Gordon, and Keith Mathias. Lamarckian evolution, the baldwin effect and function optimization. In *Parallel Problem Solving from Nature—PPSN III*, pages 5–15. Springer, 1994.

[197] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011. ISBN 0123748569, 9780123748560.

[198] W Eric Wong, Joseph R Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pages 264–274. IEEE, 1997.

[199] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and Applications*, pages 625–636, 1992.

[200] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *European Conference on Object-Oriented Programming*, pages 380–403. Springer, 2006.

[201] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

[202] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 359–368. IEEE, 2009.

[203] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635906. URL http://doi.acm.org/10.1145/2635868.2635906.

[204] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[205] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[206] Benwen Zhang, Emily Hill, and James Clause. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 625–636, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi: 10.1145/2970276.2970342. URL http://doi.acm.org/10.1145/2970276.2970342.

[207] Sai Zhang. Practical Semantic Test Simplification. In *International Conference on Software Engineering (ICSE)*, pages 1173–1176, 2013. ISBN 978-1-4673-3076-3.

[208] Sai Zhang, Cheng Zhang, and M.D. Ernst. Automated documentation inference to explain failed tests. pages 63–72, 2011. doi: 10.1109/ASE.2011.6100145.

[209] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 33–42. ACM, 2005.

[210] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.