SCUOLA DI DOTTORATO
UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Dipartimento di / Department of

Informatics, Systems and Communication

Dottorato di Ricerca in / PhD program Computer Science Ciclo / Cycle XXXV

Curriculum in (se presente / if it is)

EVIDENCE BASED SOFTWARE TESTABILITY MEASUREMENT

Cognome / Surname Guglielmo Nome / Name Luca

Matricola / Registration number 780713

Tutore / Tutor: Prof. Paola Bonizzoni

Cotutore / Co-tutor:
(se presente / if there is one)

Supervisor: Prof. Giovanni Denaro
(se presente / if there is one)

Coordinatore / Coordinator: Prof. Leonardo Mariani

ANNO ACCADEMICO / ACADEMIC YEAR 2021/2022

Acknowledgments

I would like to extend my thanks to all the people that have helped and supported me during this long journey.

In particular, I would like to express my deepest appreciation to my supervisor Prof. Giovanni Denaro for his invaluable supervision, help and support during the course of my PhD.

Additionally, I would also like to thank my fellow colleagues with which I shared many joyful moments.

Finally, I would like to express my gratitude to my family, for their presence and support, without which it would not have been possible to complete this journey.

Abstract

Software testing is a key activity of the software life-cycle that requires time and resources to be effective. One of the key aspects that influences the cost of the testing activities and ultimately the effectiveness of those activities for revealing the possible faults is software testability. Software testability expresses how difficult or easy it is to test a software artifact. The availability of estimates on the testability of the software under test and the components therein can support test analysts in anticipating the cost of testing, tuning the test plans, or pinpointing components that should undergo refactoring before testing. Several studies have been performed since the 1990 on the topic; the first ones focused more on giving an appropriate definition for software testability, while later ones focused on making software more testable and finding ways to measure software testability reliably. It is on this last aspect that we focused for this research work. Research on measuring software testability has the main objective of evaluating the testability of software components with the final goal of improving their testability, and better estimate the effort need in the testing phase. The current approaches proposed for estimating testability can mostly be categorized in: estimating testability by analyzing the fault-sensitivity of a software and estimating testability by analyzing the structure of the code of a software. Analyzing fault-sensitivity was really popular in the 90s, but the research on it has progressively dwindled in favor of the approach of using static software metrics, as representatives to the design characteristic of a software, to estimate the testability of the software. These approaches have some intrinsic limitations, admittedly highlighted in the studies, such as being costly or focusing on design characteristics that estimate testability only indirectly. In this research work we introduce a new technique for estimating software testability in which the novelty is to exploit automated test case generation to investigate to which extent a program may or may not suffer of testability issues. In a nutshell, our technique consists in executing (possibly multiple times) a test generator of choice against a program under test, and then automatically analyzing the outcomes of the test generation activity to extract evidences that the generated test cases are fostering effective (or ineffective) testing, due in particular to reasons that can be specifically reconciled with design choices that characterize the current program. We regard to testability issues as design choices that hamper the easiness of achieving effective testing. The higher the amount of the evidences our technique can collect for a given program in favor of the presence or the absence of testability issues in the program, the higher or the lower, respectively, the testability estimate that our technique will be reporting for that program. To validate our proposal, we developed a tool to concretely obtain the testability values of software artifacts and we performed many empirical experiments with the aim of finding if our technique is able to highlight testability issues reliably. Moreover, we compared our results against some of the most popular metrics that are currently suggested as potential estimators of testability. The results show the potential of our technique for measuring software testability even when compared against other proposed metrics.

Contents

1 Introduction **1**

2 State of the art **5**

2.1 Freedman and the boom of Testability studies 5

2.2 Introduction to software testability 6

2.3 Measurement of testability 10

2.3.1 Estimating testability by analyzing the structure of the code of a software 10

2.3.2 Estimating testability by analyzing the fault-sensitivity of a software 16

2.4 Open problems 18

3 Evidence-Based Testability Estimation **21**

3.1 Foundations for evidence-based testability measurement 22

3.2 Evidence-based testability measurement via automatically generated test cases and seeded faults 26

3.2.1 Sampling the test space with automated test generation 26

3.2.2 Sampling the fault space with mutation-based fault seeding 26

3.2.3 Sampling-based testability measurements 28

3.3 Our proposal for evidence-based testability measurement . . . 30

3.3.1 Extrapolating the set of baseline mutants 31

3.3.2 Obtain testability scores (final definitions) 32

3.3.3 Conclusiveness of the testability scores 39

3.4 Prototype 40

3.4.1 API augmentation and test cases pruning 41

3.4.2 Test Generation 41

3.4.3 Mutation Analysis 45

4 Experiments and results 49

4.1 Subjects 51

4.2 Qualitative study 52

4.2.1 Experimental Settings 52

4.2.2 Results 53

4.3 Quantitative study on historical data 59

4.3.1 Experimental setting 59

4.3.2 Ground Truth 61

4.3.3 Results 63

4.4 Controlled study with developers 75

4.4.1 Experimental settings 75

4.4.2 Results 77

4.5 Summary of the findings 83

4.6 Threats to validity 84

5 Conclusions 87

5.1 Future works 88

Bibliography 91

Chapter 1

Introduction

Software testing is a key activity of the software life-cycle that requires time and resources to be effective. One of the key aspects that influences the cost of the testing activities and ultimately the effectiveness of those activities for revealing the possible faults is software testability. Software testability expresses how difficult or easy it is to test a software artifact. The availability of estimates on the testability of the software under test and the components therein can support test analysts in anticipating the cost of testing, tuning the test plans, or pinpointing components that should undergo refactoring before testing.

The concept of testability is not new to other domains, in fact the first works on software testability took inspiration from testability applied to hardware [1], dynamical systems and automata [2] and logic circuits [3]. Several studies have been performed since the 1990 on software testability, in particular, the first ones focused more on giving an appropriate definition for software testability, while later ones focused on making software more testable (Design for testability) and finding ways to measure software testability reliably. This last aspect is at the core of our research work.

Research on measuring software testability has the main objective of evaluating the testability of software components with the final goal of improving their testability, especially in safety-critical systems, and better estimate the effort need in the testing phase. The current approaches proposed for estimating the testability, most of the times, rely on the analysis of the probability of revealing faults (fault sensitivity analysis) or derive potential testability issues from the design characteristics of a software (quantified software metrics, like lines of code and number of method calls), respectively. Analyzing fault-sensitivity was really popular in the 90s, but the research on it has progressively dwindled in favor of the approach of using software metrics, as representatives to the design characteristic of a software, to estimate the testability of the software. Both approaches have some intrinsic limitations, admittedly highlighted in the studies, such as being costly or focusing on design characteristics that estimate testability only indirectly.

In this research work we introduce a new technique for estimating software testability in which the novelty is to exploit automated test case generation to investigate to which extent a program may or may not suffer of testability issues. In a nutshell, our technique consists in executing a test generator of choice against a program under test, and then automatically analyzing the outcomes of the test generation activity to extract evidences that the generated test cases are fostering effective (or ineffective) testing, due, in particular, to reasons that can be specifically reconciled with design choices that characterize the current program. The higher the amount of the evidences our technique can collect for a given program in favor of the presence or the absence of testability issues in the program, the lower or the higher, respectively, the testability estimates that our technique will be reporting for that program. Compared to pursuing testability estimations based on measuring design characteristics that may correlate with testability, our technique differs in that it aims at measuring testability directly, that is, by experiencing with pursuing actual test cases and test objectives against the target program. Compared to estimating fault sensitivity, our technique is natively designed to work with sampled test suites, whereas fault sensitivity requires test cases designed to extensively traverse the possible execution flows in the program, which practical test generators can unlikely provide. In practice, our technique relies on a search-based test generation tool to automatically generate test cases [4], and refer to mutation-based fault seeding to sample possible faults [5]. We then refer to the generated test cases and the seeded faults to extrapolate the testability evidences. Our testability metric is fully automatic since it depends only on automatically generated test cases and automatically seeded faults. Moreover, our testability metric is flexible and can be estimated for software artifacts at different levels of granularity, e.g., methods, classes or larger components.

To validate our proposal, we performed a set of empirical experiments with the aim of finding if our technique is able to highlight testability issues reliably. The goal of the first experiment was to manually validate that, a set of classes for which our metric show low (resp. high) testability scores, effectively contains elements that lead to higher (resp. lower) difficulties during testing. In practice, we selected three open source Java projects and computed the class level testability scores for all their classes with our metric. For each project we extrapolated two set of classes for which our metric showed the lowest and highest testability score, respectively. For each one these classes we tried to identify which were the elements that contributed to the obtained testability score, if any. Our findings were that the classes that our approach highlights as having testability issues effectively contain elements that lead to potential difficulties in the testing phase. In the second experiment, we empirically studied the effectiveness of our testability estimates with respect to many methods of three large software projects in Java. In particular, we analyzed to what extent our estimates correlate with the potential test effort of the methods that were available in the considered projects, expressed as a set of code level metrics extracted from the test cases associated to a method. We compared the correlation yielded by our estimates with the ones yielded by a selection of popular software metrics for object-oriented programs. Our main findings were that our testability estimate contribute to explain the variability in the development effort of the test cases, while capturing a different phenomenon than the size of the software, whereas the software metrics that correlated with the test effort were tightly correlated with the size of the software as well. Furthermore, motivated by such findings, we explored the combination of our metric with the software metrics, revealing significant synergies to improve the testability estimations. In the third experiment we focused once again on the correlation with respect to the test effort, measured with a different approach. In this experiment we provided to a third-party developer a set of methods to test from an open source Java project. The third-party developer was given a set of instructions that specified how testing should be conducted and what information they needed to report for each of the testing activities. The reported information for each tested method needed to include: a subjective evaluation of the difficulty of testing, the final instruction and branch coverage scores and finally, the time taken for reaching maximum statement and branch coverage. The results of the study reaffirm the findings of the previous experiments, highlighting that our metric is able to capture elements that lead to having testability issues in a software.

The results of the experiments show the potential of our technique for measuring software testability even when compared against other proposed metrics. On the other hand, being a new technique there is still space for many improvements on it, especially related to automatic test generation, that could be improved to better fit our needs. We interpret our findings as supporting the research hypothesis that it is viable and useful to estimate testability based on empirical observations collected with automatically generated test cases. We underscore that we do not blame the use of software metrics for testability predictions, rather we aim at complementing that relevant approach, as the research done so far has ignored the relevant dimension of the testability problem that we are addressing in this work.

This manuscript is organized as the following:

- Chapter 2 will focus on the state of the art for software testability, giving an overview of the aspects on which research has focused on, and going more in details regarding the studies that focus on the measurement of software testability.
- Chapter 3 will introduce our novel technique for measuring software testability and provide a concrete way for computing it.
- Chapter 4 contains a set of empirical experiments that we performed with the aim of validating the effectiveness of our technique, with comparisons against some of the metrics that have often been suggested in the state of the art studies.

Chapter 2

State of the art

2.1 Freedman and the boom of Testability studies

The earlier works on software testability derived their ideas from the work already done on hardware testability (the testability of electronic circuits). Freedman [6] defined a testability measurement method called "domain testability". Programs that are domain testable have no input-output inconsistencies, consequently they are "easily testable". Domain testability is defined in terms of two properties: *controllability* and *observability*. Informally, a software component is controllable if, given any desired output value, an input exists which "forces" the component output to that value, consequently controllability is the ease of producing a specified output from a specified input. Conversely, a component is observable if distinct outputs are observed for distinct inputs, consequently observability is the ease of determining if specified inputs affect the outputs. These two concepts were not new and came from the fields of dynamical system and automata [2] and hardware testability [1]. In practice Freedman proposed to measure the number of bits required to implement observable and controllable extensions to obtain an index of observability and controllability, and consequently a measure of testability. Extensions are pieces of code that are used to adapt the specification of a component so that it becomes observable and controllable. Observable extensions add inputs to account for previously implicit states in the component. Controllable extensions modify the output domain such that all specified output values can be generated. The paper proposed a study on a limited subset of ADA-like expressions and procedures for which a measure of observability and controllability were derived, and the results were validated through a survey-like based approach in which 8 students were asked to test two different specifications of the same program, one less "domain testable" than the other.

2.2 Introduction to software testability

Following Freedman study, we could say software testability research became a hot topic and several works followed across the years. Research on testability can be categorized on three different aspects:

- Definition of software testability and factors affecting it
- Design for testability
- Measurement of testability

Definition of software testability and factors affecting it

Literature gives many definitions for Software Testability [7], but they can be categorized mainly in two groups: definitions that focus on the facilitation of testing (test efficiency) and definitions that focus on the facilitation on revealing the faults (test effectiveness). These definitions come either from standards or published papers. Some definitions that focus on the facilitation of testing aspect are:

- IEEE standard 610.12-1990 [8]-*Attributes of software that bear on the effort needed to validate the software product.*
- ISO standard 12207:2008 [9]-*Extent to which an objective and feasible test can be designed to determine whether a requirement is met.*
- Yeh et al. [10]-*A program's property that is introduced with the intention of predicting efforts required for testing the program.*

Some definitions that focus on the facilitation on revealing the faults are:

- Voas et al. [11]-*Software testability is the tendency of code to reveal existing faults during random testing.*

- Bertolino et al. [12]-*The testability of a program is the probability that a test of the program on an input diagram from a specified probability distribution of the inputs is rejected, given a specified oracle and given that the program is faulty.*
- Yu et al. [13]-*Probability that existing faults will be revealed by existing test cases.*

In rare cases, some definitions take into account both aspects such as Le Traon et al. [14] that defined testability as the following: *Testability is a property of both the software and the process and refers to the easiness for applying all the [testing]steps and on the inherent of the software to reveal faults during testing.* Some papers give some specialized definitions of testability. For example [15, 16] defined testability in terms of controllability and observability. In particular, Poston et al. [15] wrote the following: *Domain testability refers to the ease of modifying a program so that it is observable and controllable.*

As we can see there is not an established universal definition to testability. Each study typically focuses only on one, or a part, of the potential characteristics that should represent the concept of testability. This issue is exacerbated by the fact that the literature mentions several factors that could affect testability, but the studies focus only on a part of them. In particular some of the most mentioned factors that could affect testability are: observability, controllability, complexity, dependency, understandability and inheritance. While each one of these factors is defined in a slightly different way in each of the studies, we can give the following general definitions to each of them:

- Observability - *Observability determines how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment, and other hardware and software components.*
- Controllability - *Controllability determines how easy it is to provide a program with the needed inputs to exercise a certain condition or path, in terms of values, operations, and behaviors.*
- Complexity - *Software complexity is a term that encompasses numerous properties of a piece of software (similar to testability), all of which affect internal interactions. Simplicity is seen as the opposite of complexity,*

which is defined as the degree to which a software artifact has a single, well-defined responsibility.

- Dependency - *It mainly refers to cohesion and coupling. Cohesion refers to the degree to which the elements inside a module belong together. Coupling is the degree of interdependence between software modules.*
- Understandability - *Understandability is the degree to which a software artifact is documented or self-explaining.*
- Inheritance - *It measures the depth of a class in the inheritance tree.*

Design for testability

Design for testability is the name used to identify the set of techniques proposed for designing software that prevents the insurgence of testability issues. Design for testability is not a concept new to software design, in fact it antedates it and it is of great importance in the field of logic circuits development [17]. The studies on design for testability for logic circuits, such as the one from Fujiwara [3], established design for testability as an essential phase of the development process of hardware logic circuits since, if we compare them with software, it is impossible to fix hardware bugs once found, so it is essential that logic circuits can be easily tested before entering mass production.

When it comes to software, Binder [18], besides doing a recap of what was known about software testability at the time, proposed some architecture schema that could be applied to object-oriented systems with the goal of improving their testability. His idea was to interpose a driver component between the class under test and the test cases, this driver would allow for an easier interfacing with the class under test thereby raising the testability by making test cases easier to develop.

Across the years other researchers proposed their ideas to make the testing of software easier [14, 19–21]. The common point in these works is that they are often solutions specific to some environment and would need adaptations in other contexts. Some studies focused on the identification of testability anti-patterns: patterns that should be avoided in testable design. Baudry et al. [22–25] published a series of studies on the subject of UML testability. These studies focused on identifying testability anti-patterns by analyzing the object-to-object dependencies in the class diagram. In particular, two testability anti-patterns have been identified: class interactions, which are interactions between different classes, and self-usage, that corresponds to a class that uses itself by transitive usage dependencies. The papers define a model that is derived from the class diagram and from which it is possible to derive the testability anti-patterns and their complexity. Mouchawrab et al. [26] proposed a framework to assess testability of designs modeled with the UML. They also proposed a set of operational hypotheses for each attribute that can explain its expected relationship with testability; but the hypotheses are not empirically validated. Gao and Shih [27] said that testability should consider five dimensions that should be taken into account during component analysis and design: controllability, observability, understandability, traceability and test support capability. It is only a preliminary work applied on a small number of examples and cites the need of a more systematic solution to be used in practice. In some rare cases, in particular in the field of safety-critical systems, programming languages have been created taking design for testability into account. One such example is the SCADE programming language.

Measurement of testability

Research on measuring software testability has the main objective of evaluating the testability of software components with the final goal of improving their testability, especially in safety-critical systems, and better estimate the effort need in the testing phase. Several approaches have been proposed across the years that aim at satisfying both or only one of the two aspects of testability (efficiency/effectiveness). The approaches used for estimating testability can mostly be categorized in: estimating testability by analyzing the fault-sensitivity of a software and estimating testability by analyzing the structure of the code of a software. These estimates, most of the times, are concretely performed by analyzing the probability of revealing faults and using static software metrics, respectively. Analyzing fault-sensitivity was really popular in the 90s, but the research on it has progressively dwindled (mainly due to its drawbacks in terms of performance) in favor of the approach of using static software metrics to estimate the testability of the software.

Chapter 2.3 will detail the techniques that have been proposed across the years to measure testability, while chapter 2.4 gives a summary of what are the current limitations of the proposed techniques.

2.3 Measurement of testability

This section will go in detail about the techniques that have been proposed for measuring software testability at the state of the art. The first subsection (2.3.1) will focus on the approaches that propose to estimate testability by analyzing the structure of the code of a software, while the second subsection (2.3.2) will discuss the approaches that suggest the estimation of testability by analyzing the fault-sensitivity of a software.

2.3.1 Estimating testability by analyzing the structure of the code of a software

Extracting structural information from software static metrics

Most of the studies focusing on the aspect of measuring software testability [28–43], especially in the field of object-oriented programming, suggest that it can be derived from static software metrics that capture the static structure of software artifacts. The most common static metric is the number of lines of code, which is referenced in almost all of the studies. The need of more complex and structured metrics emerged when Object Oriented languages started becoming popular. Several measures for Object Oriented languages have been proposed, but the most frequently referenced ones are from Chidamber and Kemerer [44], that later gave the name to this set of metrics: the CK metrics. Included in this set of metrics we find: Weighted Methods per Class (WMC), Response For a Class (RFC), Lack of Cohesion Methods (LCOM), Coupling Between Objects (CBO), Dept of Inheritance Tree (DIT) and Number Of Children (NOC). Table 2.1 shows some of the most widely studied metrics in this research field.

The studies that aim at estimating software testability by extracting structural information from software static metrics do so by trying to find a correlation between these metrics and the testability of the analyzed software. The objective is to identify which are the metrics that are best predictors of testability. For the ground truth, since testability cannot be directly measured, all the studies make use of information that can be derived from the test cases provided in the analyzed software projects. In particular, typical proxy measures used as substitutes of testability are the size of the test suite (number of test cases and number of lines of code) or characteristics of the test cases such as the number of assertions. Some studies [29] made use of other test effort approximators such as the time required for testing.

The first studies focusing on using static metrics as proxies to testability used a small subset of static metrics on imperative programming languages; one of them was the one from Khoshgoftaar et al. [45, 46]. Khoshgoftaar et al. combined control flow analysis metrics such as number of edges, number of nodes, etc. with static metrics such as McCabe cyclomatic complexity by feeding them to a neural network to be able to create a model with the aim of predicting testability. In this case, the dependent variable (testability) was measured following the indication given by Voas [47]. The proposed technique had only been validated on a single software project.

Most of the studies focusing on using static metrics as a proxy of testability have been performed more recently in object-oriented languages. Bruntik and van Deursen [28, 30] were the first to properly apply the concept of using static metrics to estimate testability in the field of object-oriented languages, following studies improve upon it with proposing additional metrics, evaluation methods and empirical studies. Gupta et al. [29] proposed to integrate RFC, CBO, DIT and CC in a unique measurement with the help of a fuzzy model. The idea was that the resulting unified metric should represent testability better compared to using each metric independently. Singh et al. [31] created an artificial neural network model to predict testability starting from several software static metrics. Alshawan et al. [41] proposed a set of static metrics specific to web applications that should help in identifying testability issues. These metrics focused heavily on web forms. For example, in the case of fields inside of the forms, they took into consideration the number of unbounded fields (NUF) and the number of possible combination of choices for enumerable fields (NEC). Moreover, they counted the number of forms and form fields that invoke JavaScript functions that change the interface and made a distinction between the ones that modify only test and display elements, the ones that modify existing fields and the ones that modify the form structure. Khalid et al. [33] proposed two additional static metrics that aimed at better estimating the complexity of an object (AHF, Attribute Hiding Factor and MHF, Method Hiding Factor) and evaluated their performance in the task of testability prediction. AHF is defined as the portion of hidden attributes in the class, MHF is defined as the portion of hidden methods in a class. In this case, hidden means that the attributes/methods are encapsulated (e.g. private methods and fields). Badri et al. [32,35] defined a different way of measuring LCOM with the objective of better representing the lack of cohesion between methods and measured its correlation and the correlation of other static metric with respect to the size of the test cases. Zhou et al. [37] made use of a linear regression model to combine several software static metrics with the aim of better predicting testability. Da Cruz and Medeiros Eler [38] reasoned that other possible proxies of testability could be the line and branch coverage and mutation score. In detail, they use the test suite provided in a set of selected software and performed a correlation study to be able to identify if some static metrics correlates with code coverage and mutation score. Terragni et al. [40] expanded upon this concept by making use of coverage data to apply a normalization to the static metrics before performing a correlation study with respect to the size of the test cases.

What emerged from these studies is that the metrics that seem to be better predictors of testability of a class are LOC, WMC, DIT, NOC, FOUT, CBO, RFC, LCOM. But there are contrasting results between different studies: i)Bruntink et al. [28, 30] do not identify WMC and LCOM as good predictors differently to other studies [29, 31, 35, 36, 38] ii)NOC is identified as a good predictor only by Singh et al. [31], while others have not found such correlation [28, 30, 36, 40] iii)DIT is identified as a good predictor by Singh et al. [31] and Gupta et al. [29], but not by other studies [28, 30, 36, 40]. As several studies highlight [28,30,39,40], a common threat to validity is that the correlation experiments have been performed on a limited number of projects which could skew the results.

There exist a set of studies that, while making use of software metrics to derive testability, focus on specific subjects. Yu et al. [13] proposed a way of measuring testability for concurrent programs, since they introduce a series of complications in the testing phase, like non-determinism. To solve this problem, they defined a series of concurrency-related metrics that are evolutions of existing static metrics, such as Concurrency Cyclomatic Complexity, or straight up new ones, such as Synchronization Point Count that measure the number of nodes involving synchronization operations in a function. At the end they performed an empirical study on 8 concurrent programs and show that their metrics are better predictor of testability when used for concurrent programs. Tahir et al. [48] proposed a new way to estimate coupling that instead of using static analysis (like the FAN IN and FAN OUT metrics) made use of dynamic analysis. In particular, they defined a new metric called Dynamic Coupling which takes into account two elements: i)when a class is accessed by another class at runtime, ii)when a class accesses other classes at runtime. They performed a correlation study with respect to the size of the test cases provided in three analyzed software projects, which shows a low/medium correlation depending on the project.

Table 2.1. Table containing the most popular static metrics that have been analyzed during the years with the aim of finding if they are good proxies to testability.

Acronym	Name	Description
DIT	Depth of Inheritance Tree	Number of ancestors for a particular class.
CC	McCabe Cyclomatic Complexity	Number of linearly independent paths for a particular class.
FIN	Fan In	Number of classes that reference a particular class.
FOUT	Fan Out	Number of other classes referenced by a particular class.
LCOM	Lack of Cohesion Of Methods	Number of pairs of methods in a class, having no common attributes, minus the number of pairs of methods having at least one common attribute.
LOC	Lines Of Code	Number of lines of code in a particular class.
NOC	Number Of Children	Number of successors for a particular class.
NOF	Number Of Fields	Number of fields for a particular class.
NOM	Number Of Methods	Number of methods for a particular class.
RFC	Response For Class	Number of unique method invocations in a class.
WMC	Weighted Methods Per Class	Number of branch instructions in a class.
CBO	Coupling Between Objects	Number of other classes to which a class is coupled.

Extracting structural information from data and control flow analysis

Another way of extracting structural information from the code, in addition to the counting the static elements, is the analysis of the control and data flows. Bache and Mullenburg [49] and later Badri and Toure [50, 51] focused on measuring testability by analyzing the control flow of the software under evaluation. Bache and Mullenburg relied on flowgraph models to derive their metric. In this model the statements are the nodes and the edges represents the flow of control. Testability would express the effort needed for testing these flowgraph models. In practice, they measured the minimum number of paths required to have a full coverage of the model. The main drawback highlighted by the authors is that testability and complexity problems could be hidden in data, consequently it is necessary to know additional testability measures for data flow, predicate and computation testing.

Badri and Toure defined a new metric called the Quality Assurance Indicator (Qi) metric which is used as an approximator of testability. The Qi metric is based of control call graphs, which are a reduced form of control flow graphs in which all instructions not containing methods calls are removed. The Qi metric estimates the probability that the control flow will go through a method without any failure. To estimate the probability of a failure in a method Mi, the Qi uses information such as Cyclomatic Complexity and unit testing coverage. In case the method Mi invokes another method Mj, the Qi of method Mj is added to the Qi of method Mi. The main drawbacks of these studies, as pointed out by the authors, was the lack of extensive studies and the fact that testability is affected by several factors other than the ones considered.

Yeh et al. [10] based their metric on the data-flow analysis technique. The aim of data-flow analysis is to obtain variables' relationships given a program's flow graph. Within a program, data flow is the occurrence of variables that are classified as definition (def), computation-use (c-use) or predicate-use (p-use). A life cycle of a variable is between the variable's definition and redefinition or between the variable's definition and destruction along a program's control graph. Their technique focused on the variables' definition use (d-u) pairs among different blocks, so local variables are excluded. The idea was that the more the non-local variables exist in blocks the more works to do in testing. The work showed only some small examples on how testability could be computed in such a way, without providing any theoretical or empirical way of verifying the obtained results.

Jungmayr [52] proposed to identify the test-critical dependencies between software components since they should have a large impact on testability. He defined 4 new metrics that should help in identifying test-critical dependencies: i)average component dependency, which averages the number of components that a component depends on directly and transitively, ii)number of feedback dependencies, which is defined as a set of dependencies which, when removed, makes the graph representing dependencies between components acyclic, iii)number of stubs needed to break dependencies cycles, iv)number of components within dependency cycles. At the end he performed a manual analysis of the test-critical dependencies identified in four software projects, concluding that most of them are due to poor design decisions.

Le Traon et al. [53–55] focused on the testability of data flow designs. They defined testability as the ease of testing a piece of software design using structural testing strategies. This "easiness" is both intrinsic (a characteristic of the software itself) and dependent on the chosen testing strategy. They made a distinction of global and local level testability. In particular, at the global level they affirm that testability is influenced by:

- Global test cost: size of the test set, difficulty of finding the proper test data and the difficulty on deciding the validity of the run results). This factor depends on the testing context since the test set is dependent on the chosen test method.

- Global controllability: the easiness of selecting relevant test data to exercise internal components of the design.

- Global observability: the easiness of detecting faulty results by observing the outputs.

On the other hand, at the local level, the overall test cost is not definable since it is related to the testing process, while controllability and observability can be defined by focusing on the single components. They are defined as:

- Local controllability: the easiness of carrying a given value from the design entries to the internal design component.

- Local observability: the easiness of observing the effects of this component execution at the design outputs.

In practice, they measured local controllability as the portion of the input domain of the component which can be covered from the inputs of the design; or in other words, is the total information quantity the component can receive compared to the information quantity that can be generated at its inputs when isolated. In an isolated component C, the input domain coincides with the input design, and consequently it will have maximum controllability. Conversely, when the component C belongs to a flow, the total information quantity the component can receive is smaller than the information quantity that can be generated at its inputs when isolated. Similarly, local observability measures the portion of the specified output domain of the component which leads to different outputs values of the design; or in other words, is the maximum information quantity the outputs of the flow may receive from the component compared to the total information quantity the component may produce on its outputs. Finally, they computed global measures as specified in the following. Global test cost is the sum of the components that are tested in each test flow. The global controllability of a design is the minimum local controllability of the components in it, similarly global observability is the minimum local observability of the components existing in the design. Le Traon et al. used these concepts to concretely perform an empirical case study on aerospace industrial systems that are developed in a data flow oriented rigorous semi-formal design language (SAO).

Nguyen et al. [56] use data flow analysis to analyze the information flow from the inputs to the outputs. Testability is computed based on the controllability and the observability of a module for each flow of the software. The controllability measure estimates the information quantity available on the inputs of a module from the inputs of the software through the considered flow. Respectively, the observability measure estimates the information quantity available on the outputs of the software from the outputs of a module. This last step is similar to the proposal of Le Traon et al. [53–55].

2.3.2 Estimating testability by analyzing the fault-sensitivity of a software

Voas [11, 47, 57–59] focused on the aspect of testability of being able to easily detect faults within a software. His technique, called PIE, was the basis of his studies. PIE stands for propagate, inject and execute; in practice this means that for a failure to occur and be observed, three things must happen: the fault must be executed, an incorrect data state must be created (and the original data state becomes “infected”), and the incorrect state must be propagated to a discernible output. These three stages are the foundation upon which he designed a technique called sensitivity analysis, which had the aim of computing a fault sensitivity measure. Fault sensitivity expresses the probability of failure that would be induced in the program by a single fault. The higher this probability the higher the software testability and vice-versa.

In practice, sensitivity analysis injects simulated faults into the code and evaluates what is their effect on the observable outputs. Its efficacy relies heavily on the tests used to take this measurement, and so it is important that the test suite covers all possible locations. Sensitivity analysis is broken into three independent processes, each of which estimates the likelihood of one of the three events: execution analysis, infection analysis, and propagation analysis. To estimate the likelihood of an event, each process divides the number of times the event occurred by the number of attempts to force that event. For example, if the propagation event occurs 10 out of 100 times, the propagation probability estimate is 0.1. The result of sensitivity analysis is the estimated probability of failure that would result if a particular location had a fault. This estimate is obtained by multiplying the means of the three estimates from the analysis phases. If you take the minimum over all three estimates and then obtain a product, you can obtain a bound on the minimum probability of failure that would result if this location had a fault. The strength of sensitivity analysis is that the results are based on the observed effects from actual faults; the weakness is that the faults injected and observed are only a small set from what might be an infinite class of faults. An indirect downside is on the performance side; since the number of input data that needs to be provided to cover all possible events is really high even for a small program such as the one used in Voas papers (solving of a quadratic equation).

The assumption under which Voas said that fault sensitivity could be a good proxy of testability is the *competent programmer hypothesis*, which affirms that a competent programmer will write code that is reasonably close to being correct. In particular, he makes use of a variation of the competent programmer hypothesis called the *simple fault assumption* that says that a fault exists in a single location, not distributed throughout the program, and that this fault is equally likely to be at any location in the program. Consequently, since fault sensitivity measures the probability of finding a single fault in a single location it respects the *simple fault assumption*.

Bertolino and Strigini [12] performed a study that analyzed the impact of improving the testability of a software with respect to its reliability. The study is based on the testability measurement proposed by Voas, and it highlights that small increases in Voas' testability metric often can be detrimental to reliability, on the other hand great increases in testability seemingly goes hand in hand with improvements to reliability.

Lin et al. [60] proposed to use a modified version of the PIE technique, in particular in the propagation aspect, which aimed at improving the performance in terms of time compared to Voas technique. In their case, instead of considering all input variables, they only considered variables that are *used* in a while or if condition. A *used* variable means that it is referenced in an instruction. This altered version was evaluated against a simple program that solves a quadratic equation. While requiring less resources (input data) to be executed compared to Voas' version, it still maintained a steep performance cost.

Zhao [61] proposed a metric called Fault Detection Probability that expresses the ratio of suites satisfying a specific test criterion that can detect a particular fault. This was a preliminary work that had the aim of showing that different faults have different test difficulty. In detail, by focusing on the ability to predict the failure rate caused by different faults under different testing criterion, it could be possible to obtain a prior indication of software testability. This research thought did not have a proper follow-up in the field of testability studies.

2.4 Open problems

As seen in the previous chapters, the current ways of measuring testability have some open problems that hinders accurate or/and efficient measurements. In the case of studies which focuses on analyzing the structure of the code of a software (Section 2.3.1), we can see that:

- *Testability is obtained in an indirect way with respect to software testing.* The metrics that should be good proxies for testability are derived from the correlation between the static metrics of the source code and the static metrics of the associated test cases. This is an indirect way of measuring testability that potentially misses real testability issues and instead is potentially biased by the size of the source code and associated test cases. We need to remark that a software module having many lines of code or many methods does not necessarily entail that it is not testable. Ideally, testability should not be influenced by the size of the tested module and should instead highlight only the potential difficulties that can be encountered during testing.

- *Correlation with weak ground truth in static metric studies.* The so called "best predictor" metrics in static metric studies are chosen based on the correlation with the test cases static metrics. A potential threat to validity that can arise is the fact that there is no information at all about the development process of the used test cases. So there is basically no knowledge on the test objectives of the existing tests, if they exist, and can be biased by the arbitrary decisions of the testers (e.g., decisions on designing few or many test cases, or aiming to high code coverage or ignoring code coverage). Moreover, many of the studies that suggest that static metrics could be good predictors of testability are performed only on a sample, oftentimes small, of projects and this could lead to generalization problems. The combination of these two factors may lead to the false belief that a source metric is a good predictor of testability in all situations while in reality the correlation that is seen is simply valid for that specific set of subject projects. These factors may be what lead to the discrepancies seen in the papers about which are the best predictor metrics.

- *Usage of the test cases without considering their quality aspects.* Most of the studies on static metrics do not consider the coverage of the provided test cases in their analysis. This may introduce severe biases dictated by test cases that may cover only a small part of the associated source code. This aspect has been mitigated in some of the latest works [38,40]

Our research work has in common with studies that estimate testability through fault-sensitivity estimation (Section 2.3.2) the goal of estimating testability by observing the execution of the software though actual testing. It aims at overcoming the main problem of these studies, that is the *costly measurements and difficulties in their applications in complex software.* In fact, fault sensitivity is costly and not easily scalable. As we saw in the case of Voas and his PIE technique and successive evolutions, the cost for generating all possible inputs for all the analyzed modules, creating appropriate tests and injecting real faults would be astronomical and not usable in practice.

Chapter 3

Evidence-Based Testability Estimation

This chapter introduces a novel technique for estimating software testability. The core novelty of our proposal is to exploit automated test case generation to investigate to which extent a program may or may not suffer of testability issues. In a nutshell, our technique consists in executing (possibly multiple times) a test generator of choice against a program under test, and then automatically analyzing the outcomes of the test generation activity to extract evidences that the generated test cases are fostering effective (or ineffective) testing for reasons that can be tracked to the design characteristics of the current program. In other words, we think of testability issues as design characteristics that hamper the easiness of achieving effective testing (i.e., characteristics that hamper the detection of faults. For an example see Listings 3.1, 3.2, 3.3 and their explanation in Section 3.3.1), and we rely on automatic test generation to investigate whether that is indeed the case for the given program under test. The higher the amount of the evidences that our technique can collect for a given program in favor of the presence or the absence of testability issues in the program, the lower or the higher, respectively, the testability estimates that our technique will be reporting for that program.

Compared to pursuing testability estimations based on measuring design characteristics that may correlate with testability (as in the approaches based on software metrics, as we surveyed in Section 2.3.1), our technique differs in that it aims at measuring testability directly, that is, by experiencing with pursuing actual test cases and test objectives against the target program.

Compared to estimating fault sensitivity (as in the approaches inspired by the work of Voas and colleagues, as we surveyed in Section 2.3.2), our technique is natively designed to work with sampled test suites (as the ones that we can achieve with an automated test generator), whereas fault sensitivity requires test cases designed to extensively traverse the possible execution flows in the program, which practical test generators can unlikely provide.

This chapter presents our technique for testability estimation in three incremental steps. First (Section 3.1), we characterize the type of measurements that our technique aims to address; we do that by referring to a purely idealistic scenario that assumes: i) the availability of an exhaustive-testing test suite, full knowledge of all possible test objectives, and a hypothetical tool that can provide perfect, objective and automatic judgments on which test cases can be considered hard or easy to be identified by test analysts, respectively. This idealistic scenario cannot be practically achieved but allows us to define the foundation of the measurements that we aim to. Next (Section 3.2), we explain how we can instantiate our measurement technique concretely by relying on test cases generated automatically, and sample test objectives rendered as seeded faults. Finally (Section 3.3), we finalize the definition of the technique by defining a measurement procedure that takes into account and mitigates the subjectivity of the measurements that may arise from the specific abilities and limitations of the specific test generator being used.

3.1 Foundations for evidence-based testability measurement

Our technique leverages automatically generated test cases to discern the presence (or the absence) of testability issues, that is, evidences that the program under test, in the current shape of its design, hampers or simplifies the derivation of effective test cases. For reasoning on the effectiveness of the test cases, our technique refers to the possible faults of the target program. The underlying intuition is that it would be wrong to judge a software module as highly testable simply because we empirically observe that it facilitates the generation of arbitrary test suites, which in fact might in some cases include only test cases that exercise that software just superficially. Rather, we would like to judge as highly testable a software module that facilitates the generation of test suites that work well for proper correctness-checking.

The optimal (though only idealistic) scenario for our technique would be having in advance full knowledge of:

- all the test cases that are needed to exhaustively test the software module,
- which faults may generally exist in the software module under test, and
- some criterion to analyze the test cases and argue whether they are easy or hard to be identified.

In this idealistic scenario, our technique would quantify the relative testability of a set of software modules based on the portion of easy-to-test and hard-to-test faults in each module. Intuitively, the higher the portion of hard-to-test faults in a software module, the higher the likelihood that the module may come with testability issues (since apparently it was not designed in a way that made its own testing simple), and vice-versa.

Drawing on these intuitions, we provide a definition for the testability score of a software module starting from the following predicates:

- Let M be a software module of a program P
- Let T be the set of possible test cases for P
- Let F be the set of *executable*[1] faults in P
- Let also $F(M) \subseteq F$ denote the faults located in M
- Let $\mathit{Reveal} : F \times T$ be the relation between faults and test cases that reveal them
- Let $\mathit{Exec} : F \times T$ be the relation between faults and test cases that execute them

The we can define testability as the following:

Definition 1. Testability score (in the optimal, idealistic scenario) The portion of faults for which there exists evidence (i.e., at least a test case) that those faults can be revealed with easy test cases. Formally:

[1]Non-executable faults are irrelevant for testing and testability.

Let $Hard : T \to \{true, false\}$ be a criterion (a predicate) to decide whether a test case is or is not hard to be identified. Accordingly, let $F_{hard}(M)$ be the set of faults in M that are hard to identify, that is, $F_{hard}(M) \equiv \{f \in F(M) | \forall t : Reveal(f, t) \Rightarrow Hard(t)\}$.[2]

Then the testability of module M is quantified as:

$$Testability(M) = 1 - \frac{|F_{hard}(M)|}{|F(M)|}.$$

We further adapt the above definition to discriminate between *controllability* and *observability* issues, inspired by the suggestions of the PIE (propagate-infect-execute) model [47], i.e., that the ability of testing a given fault in a program depends both on the ability of setting suitable test data (controllability), as necessary for *executing the faulty locations* in the program to *determine infected program states*, and on the ability of making those infected program states *propagate up to some observable outputs* (observability). Thus, maximum controllability corresponds to having sufficient control for setting any testing-relevant program input, and maximum observability corresponds to being able to observe any testing-relevant program output.

To measure controllability and observability we consider that each test case consists of two parts, namely, the test driver and test oracle. The test driver is the part of a test case that sets proper inputs for the module under test, aiming to drive its execution in specific way. The test oracle is the part of a test case that evaluates the outputs against the specification to exclude or pinpoint malfunctions.

Keeping in mind the predicates that we used to define the testability score, we define the controllability and observability scores of a software module as follows:

Definition 2. Controllability score (in the optimal, idealistic scenario) The portion of faults or which there exist evidence (i.e., at least a test case) that those faults can be *executed* with test cases comprised of *easy test drivers*. Formally:

Let $Hard\_d : T \to \{true, false\}$ be a criterion to decide whether or not

[2]The set $F_{hard}(M)$ includes also the faults that, although being executable, cannot be revealed with any test case. Executable, non-revealable faults are arguably symptoms of testability issues.

the driver part of a test case is hard to be identified. Correspondingly, let $F_{hard\_d}(M)$ be the set of faults in M that can be executed only with test drivers that are hard to identify, that is, $F_{hard\_d}(M) \equiv \{f \in F(M) | \forall t : Exec(f, t) \Rightarrow Hard\_d(t)\}$.

Then the controllability of module M is quantified as:

$$Contr(M) = 1 - \frac{|F_{hard\_d}(M)|}{|F(M)|}.$$

Definition 3. Observability score (in the optimal, idealistic scenario) The portion of revealable faults for which there exist evidence (i.e., at least a test case) that those faults can be *revealed* with test cases comprised of *easy test oracles*. Formally:

Let $F_r(M) \subseteq F(M)$ be the set of faults that can be revealed with some test case, that is, $F_r(M) \equiv \{f \in F(M) | \exists t : \mathit{Reveal}(f, t)\}$. Let $Hard\_o : T \to \{true, false\}$ be a criterion to decide whether or not the oracle part of a test case is hard to be identified. Correspondingly, let $F_{hard\_o}(M)$ be the set of faults in M that can be revealed only with oracles that are hard to identify, that is, $F_{hard\_o}(M) \equiv \{f \in F_r(M) | \forall t : \mathit{Reveal}(t, f) \Rightarrow Hard\_o(t)\}$.

Then the observability of module M is quantified as:

$$\mathit{Obs}(M) = 1 - \frac{|F_{hard\_o}(M)}{|F_r(M)|}.$$

As mentioned, these definitions capture an idealistic scenario, in which we know all the potential faults and all the possible test cases in advance, which is, of course, unrealistic. In the next subsection, we present a measurement framework that proxies these idealistic metrics by referring to concrete faults, concrete test cases and objective decisions on evaluating the hardness of test cases and faults.

3.2 Evidence-based testability measurement via automatically generated test cases and seeded faults

Our technique to estimate software testability exploits automatically generated test cases, and faults automatically seeded in the form of synthetic program mutations. Thus, whereas the idealistic measurements that we introduced in the previous section (unrealistically) assume the availability of all possible test cases and all possible faults, the concrete instance of our measurement technique refer to sampled test cases and sampled faults. In fact, relying on sampling is a common characteristic of most estimation processes in statistics. In particular, we sample the test space by using automatic test case generation and, sample the fault space through mutation-based fault seeding. The following subsections go more in detail about this choice. Finally, the last subsection will focus on explaining how we use these samples to derive a testability metric which is more concrete compared to the previously presented idealistic measurements.

3.2.1 Sampling the test space with automated test generation

Our intuition is that we can rely on an automatic test generator to estimate the testability degree of a software by using the generated test cases as samples that highlight the possible difficulties that a developer can encounter while developing test cases. Working with automatically generated test cases is indeed the main distinctive characteristic of our methodology. This radically differs from evaluating internal characteristics of the target components, in terms of the static structure of the code, or monitoring their execution spectrum dynamically, which are the mainstream approaches investigated so far.

3.2.2 Sampling the fault space with mutation-based fault seeding

Knowing all possible faults and reproducing them is self-evidently impossible. To solve this problem, we decided to rely on synthetic *mutation-based fault* *seeding* [62,63]. Mutation-based fault seeding instruments programs with possible faults by using mutation operators, each describing a class of code-level modifications that may simulate faults in the program. Through mutation-based fault seeding we can sample the faults that may be encountered in the target software module. To give an idea of a possible fault generated in such a way, *replacing numeric literals* is a mutation operator that produces a faulty version of the software module (called *mutant*) by changing a numeric literal in the program with a compatible literal.

After generating all possible mutants for a software module, a step called mutation analysis is run. During this step all provided test cases are run against the original and mutated version of the software module; if there is any discrepancy between the two runs, they get reported. In particular, we say that a test case that execute the portion of code modified by a mutation operator is said to *execute (or cover) the mutant m*. Additionally, a test case that has a different outcome when executed against the original program and its mutant *m* is said to *kill (or reveal) the mutant m*, meaning that it reveals the sample fault that the mutant represents. Finally, a mutant which is neither executed nor killed is called a *non-covered mutant m*.

Our approach grounds on the large body of scientific literature that argue that mutants are valid representative of real faults [64–67]. Andrews et al. [64, 65] performed a study comparing real faults and mutants. Their analysis suggests that mutants, when using carefully selected mutation operators and after removing equivalent mutants, can provide a good indication of the fault detection ability of a test suite. They also noted that mutants are not easier or harder to detect compared to real faults. This fact lends support to the “competent programmer assumption” that underlies the theory of mutation testing; that is, the assumption that faults made by programmers will be detected by test suites that kill mutants. Just et al. [66] affirms that mutants are intended to be used as practical replacements for real faults in software testing research and this is valid only if a test suite’s mutation score is correlated with its real fault detection rate. Their study empirically confirms that such a correlation generally exists by examining a collection of real faults in open source programs using developer written and automatically generated tests. In particular, it highlighted that conditional operator replacement, relational operator replacement, and statement deletion mutants are more often

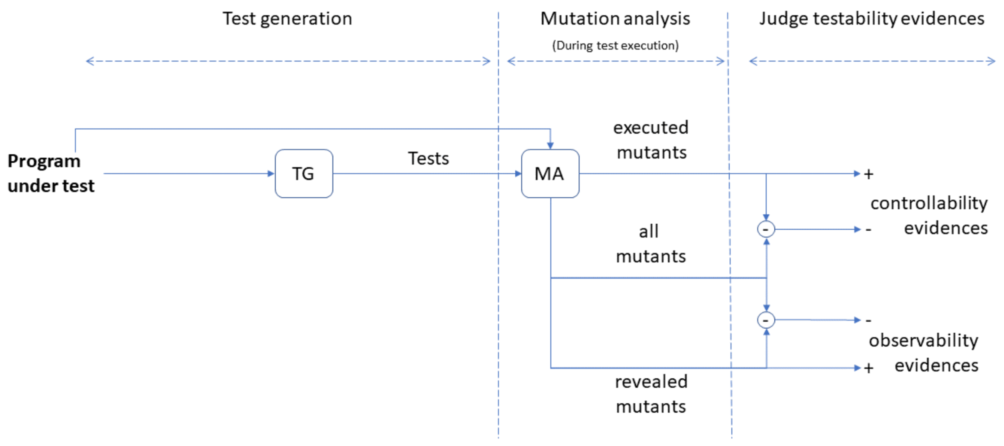

Figure 3.1. Workflow to estimate testability evidences using tests and faults sampling

coupled to real faults than other mutants. All these results support our choice of relying on mutants as samples of real faults.

3.2.3 Sampling-based testability measurements

Figure 3.1 illustrates the workflow by which we can exploits automated test generation (left part of the figure) and mutation analysis (middle part of the figure) in order to judge testability evidences (right part of the figure). The input is a given program under test, which is indicated at the top-left corner in the figure, and the results are sets of testability evidences, classified as either controllability evidences or observability evidences, as indicated at the right-most side of the figure. The block named *TG* indicate the automatic test generation activity. The block named *MA* indicate the mutation analysis activity, which consists of mutants' injection followed by test cases execution, with the goal of determining which mutants are killed, executed or non-covered. The circles that contain the $-$ symbol indicate that we remove the executed mutants (resp. revealed mutants) from the set of all mutants obtained during the mutation analysis to identify the set that highlight controllability issues (resp. observability issues). The results are yielded as positive or negative testability evidences ($+$ and - symbols at the rightmost side, respectively). The arrows specify the inputs and the outputs of each activity. From the schema, we can see that the first step consists in automatically generating the test cases followed by the mutation analysis step that will provide us with information about the ability of the test cases to execute and reveal the faults. Once we know this information for all the generated mutants, we can make use of them to judge two types of testability evidences, namely, *controllability* and *observability* evidences.

For each mutant in which the sampled test suites can execute (resp. cannot execute) the seeded fault, we judge a controllability (resp. non-controllability) evidence. We could define the set of mutants that we cannot cover with the generated test cases as hard-to-execute (set F_{hard_d} of Definition 2). For each mutant in which the sampled test suites can reveal (resp. cannot reveal) the seeded fault also based on actual outputs of the target program, we judge an observability (resp. non-observability) evidence. We define the set of mutants that cannot be revealed through actual outputs of the target program as hard-to-reveal(set F_{hard_o} of Definition 3). In summary, let $\hat{F}$ be the set of mutants that were generated for the program P, $\hat{F}_{kill}$ be the set of mutants that the generated test cases kill and $\hat{F}_{executed}$ the set of mutants that the generated test cases execute; our suggested testability measurement framework makes the following estimates related to the sets in Definitions 2 and 3:

- $\hat{T}$, all automatically generated test cases for the target program P.
- $\hat{F} \equiv \hat{F}_r$, all the generated mutants for the target program P. $\hat{F}_r$ is equivalent to $\hat{F}$ since the test generator is able to potentially reveal all faults.
- $\hat{F}_{hard\_d} \equiv \hat{F} - \hat{F}_{executed}$, the mutants that were not executed (non-covered).
- $\hat{F}_{hard\_o} \equiv \hat{F} - \hat{F}_{kill}$, the mutants that can be executed, but not killed.

We then recast the idealistic controllability and observability scores defined in the previous section to achieve concrete estimations of those scores as follows:

Definition 4. Sampling-based controllability score The portion of seeded faults (mutants) that we successfully executed with at least an automatically generated test case. Formally, for a software module M

$$Contr'(M) = 1 - \frac{|\hat{F}_{hard\_d}(M)|}{|\hat{F}(M)|} = \frac{|\hat{F}_{executed}(M)|}{|\hat{F}(M)|}.$$

Definition 5. Sampling-based observability score The portion of seeded faults (mutants) that we successfully revealed with at least an automatically generated test case. Formally, for a software module M

$$Obs'(M) = 1 - \frac{|\hat{F}_{hard\_o}(M)|}{|\hat{F}_r(M)|} = \frac{|\hat{F}_{kill}(M)|}{|\hat{F}(M)|}.$$

Definition 6. Sampling-based testability score The cumulated score yielded by the combination of the controllability and observability scores through the application of the geometric mean: For a software module M

$$Testability'(M) = \sqrt{Contr'(M) \times Obs'(M)}$$

Note that in these definitions we are conservatively assuming that all seeded faults are both *executable* and *revealable.* We will deal with mitigating the impact of this assumption when refining and finalizing the definition of our measurement technique in the next section.

3.3 Our proposal for evidence-based testability measurement

The assumption that all mutants necessarily correspond to faults that, at least in principle, can be executed and revealed with the tests generator of choice is generally too strong when working with practical test generators and practical mutation analysis tools. For example, a test generator that is not able to construct some types of data structures or does not handle test data from files and network streams, will systematically miss test cases for any fault that depends on those types of test data, independently from the actual testability of the software under test. Indeed, although testing those faults is hard for the test generator at hand, it might be relatively simple for other test generators or human testers. Similarly, missing test cases for functionally-equivalent mutants, including the case in which the mutated code is infeasible in the target program, conveys no useful information on the non-testability of the software.

Thus, when using automatically generated test suites and mutation-based fault seeding, we must pay attention that the quality of our estimations could be jeopardized by the intrinsic limitations of the approaches (and the tools) to which we refer for generating the test cases and computing the mutations, respectively. In particular, we would like to avoid non-testability judgments that can derive from either test cases that cannot be generated because of intrinsic limitations of the considered test generator or infeasible mutants that the test generator cannot execute. For these reasons, instead of focusing on any seeded fault provided by the mutation-analysis tool at hand, our approach constructively discriminates the subset of seeded faults for which we have sufficient evidence (not necessarily a proof) that (i) they do not correspond to infeasible mutants, and (ii) are not out of the scope of the considered test generator either. The resulting subset of mutants provides the baseline for us to judge the testability evidences.

3.3.1 Extrapolating the set of baseline mutants

We concretely extrapolate the set of baseline mutants as follows. We empower the test generator with the capability to directly assign any input and state variable in the program, by augmenting the program under test with custom setters for all state variables of any component in the program. In our specific implementation, we focus on custom setters, but this step may be generalized at different levels. Furthermore, the addition of custom setters may lead to think that this has an impact only on the class level but, in reality, the scope is wider. For example, custom setters may potentially be used in classes that are used as parameters for the method of another class which is the target of testing. Next, we execute the test generator against the augmented program, and we track all executed mutations, regardless of whether or not they propagate to the outputs. We refer to this setting of the testing problem as *testing the program with testability-facilitated APIs*, underscoring the ability of freely assigning the possible inputs and the assumption that any program state could be somehow inspected during testing. The seeded faults that are executable with the testability-facilitated APIs convey the information that the test generator at hand could in principle generate suitable test data. If it should not, it likely would depend on the constraints built in the original APIs of the program, not on limitations of the test generator with the required type of test data. Similarly, if those seeded faults, once executed, should not propagate to observable outputs, it likely would depend on the internal characteristics of target software, rather than on the infeasibility of the seeded faults.

Let us take the code shown in Listing 3.1 as an example and assume that we want to test the method at line 14 and that we have generated a mutated version program that changes the value of the return at line 19 as shown in Listing 3.2. If we want to detect the mutant in line 19, we would need to have an appropriate value for *field*, which is assigned in the private method *someOtherMethod*. In particular, to be able to access the else branch of *targetMethod* at line 19, we would need to somehow satisfy the true condition at line 7 of *someOtherMethod*, a task which is not trivial considering that the method is private and we have no knowledge on how difficult it would be to obtain a value of *par1* that satisfies the true condition at line 7 and at the same time have a value equal or less than 100 for *par2*. With the help of a facilitated API (in this case the setter) at line 23 of Listings 3.3 executing and killing the above mentioned mutant would instead become trivial.

The mutants which are identified as executed after this step are our set of baseline mutants. We are aware that, technically speaking, using the testability-facilitated APIs may lead us to generate some input states that are illegal for the original program. Nonetheless, we embrace this approach heuristically: observing faults that the test generator can execute only with the testability-facilitated APIs is a sign of restrictive designs, which may pinpoint testability issues.

3.3.2 Obtain testability scores (final definitions)

Once we have identified the baseline mutants, we refer to each of those mutants to once again judge the two types of testability evidences, namely, *controllability* and *observability* evidences, based on the test suites that we sampled with the test generator. In particular, the workflow that leads to the extraction of these evidences is shown in figure 3.2. As we can see from the schema in the figure, it is a direct evolution of the schema presented in figure 3.1. Again, we can see that our technique exploits automated test generation (left part of the figure) and mutation analysis (middle part of the figure) in order to judge testability evidences (right part of the figure). The input is a given program under test, which is indicated at the top-left corner in the figure, and the results are sets of testability evidences, classified as either controllability

```
 1 public Class Example{
 2   private int field;
 3 
 4   ...
 5 
 6   private someOtherMethod(int par1, int par2){
 7     if (par1 > 1000000) {
 8       field = par2;
 9     } else {
10       field = (abs(par2) + 1) * 100;
11     }
12   }
13 
14   public int targetMethod() {
15     int result = field;
16     if (result > 100) {
17       return result;
18     } else {
19       return -1;
20     }
21   }
22 }
```

Listing 3.1. Example class with potential testability issue

```
 1 public Class ExampleMutated{
 2   private int field;
 3 
 4   ...
 5 
 6   private someOtherMethod(int par1, int par2){
 7     if (par1 > 1000000) {
 8       field = par2;
 9     } else {
10       field = (abs(par2) + 1) * 100;
11     }
12   }
13 
14   public int targetMethod() {
15     int result = field;
16     if (result > 100) {
17       return result;
18     } else {
19       return -100;
20     }
21   }
22 }
```

Listing 3.2. Mutated example class at line 19

```
 1 public Class ExampleFacilitated{
 2   private int field;
 3 
 4   ...
 5 
 6   private someOtherMethod(int par1, int par2){
 7     if (par1 > 1000000) {
 8       field = par2;
 9     } else {
10       field = (abs(par2) + 1) * 100;
11     }
12   }
13 
14   public int targetMethod() {
15     int result = field;
16     if (result > 100) {
17       return result;
18     } else {
19       return -1;
20     }
21   }
22 
23   public setField(int par3) {
24     field = par3;
25   }
26 }
```

Listing 3.3. Example class with facilitated API

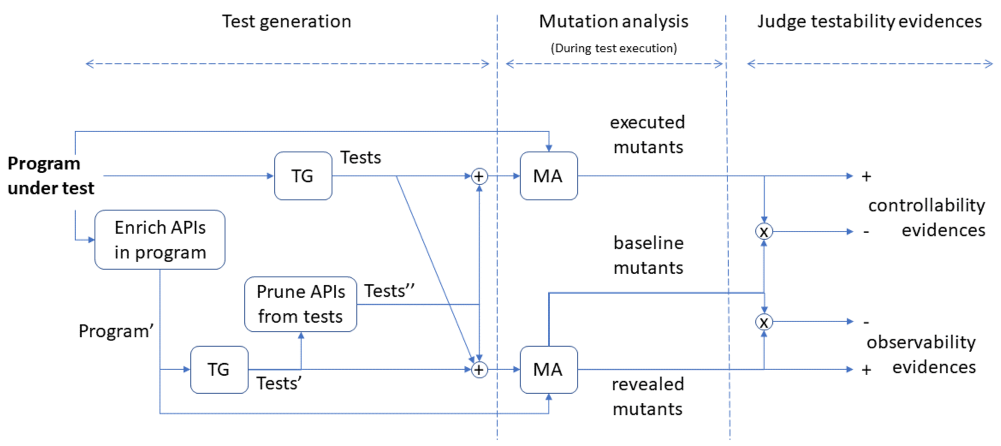

Figure 3.2. Workflow of our technique to estimate testability evidences

evidences or observability evidences, as indicated at the rightmost side of the figure. The blocks named *TG* indicate test generation activities. The block named *MA* indicate the mutation analysis activities, which consists of mutants' injection followed by test cases execution, with the goal of determining which mutants are killed, executed or non-covered. The blocks named *Enrich APIs in program* and *Prune APIs from tests* indicate the pre-processing of the program under test and the post-processing of generated test cases, respectively. Enrich API aims at augmenting the program under test with the *testability-facilitated APIs*, as we introduced in the previous subsection. Prune API aims at removing the calls to those APIs from the generated test cases, as explained later in this section. The circles that contain the + symbol indicate post-processing for merging the generated test suites into a single test suite. These steps, which differs greatly from the schema presented in figure 3.1, are detailed later in this section. The circles that contain the × symbol indicate the comparison of the set of executed mutants (resp. revealed mutants) against the baseline mutants, with the goal of judging the testability evidences, as described in the Judging testability evidences subsection. The results are once again yielded as positive or negative testability evidences (+ and - symbols at the rightmost side, respectively). The arrows specify the inputs and the outputs of each activity. Now let us go more in detail about the changes that we actuated to mitigate the potential intrinsic limitations of the used approaches compared to the workflow previously presented in figure 3.1.

Test generation

With reference to the *TG* blocks in Figure 3.2, our technique runs the automatic test generator against both the program under test and its augmented version *Program'*. Our implementation of the block *Enrich APIs in program* obtains the augmented program *Program'* by enriching the interfaces of all classes with custom setters for any class variables declared in the code. We denoted as *Tests* and *Tests'* the test suites generated as result of those automatic test generators runs, respectively. The test suite *Tests'* generated against *Program'* indicates program behaviors that the automatic test generator could provably exercise, possibly with the help of facilitated APIs. At the same time, the test suite *Tests'* implicitly captures the program behaviors that the test generation algorithm of automatic test generator is unlikely to exercise, since it failed even when facilitated to set the input state independently from the constraints encoded in the program APIs. In conclusion, the comparison between the test suites *Tests'* and *Tests* aims at highlighting program behaviors that arguably were hard to exercise specifically due to the constraints encoded in the APIs, that is, behaviors that potentially show real testability issues (see Listings 3.1, 3.2, 3.3 and their explanation) instead of limitations of the automatic test generator.

The above presented approach may introduce a confounding effect due to differences between test suites *Tests'* and *Tests* that results just from the randomness of the automatic test generator. Basically, it may be possible, that the automatic test generator may cover additional program behaviors during the generation of test suite *Tests'*, compared to test suite *Tests*, not due to the additional APIs, but simply because of the random nature of its algorithm, which even if mitigated is always present. To solve this potential issue, we specifically focus on the program behaviors that can be exercised only with the testability-facilitated APIs, but not with the original program APIs, by post-processing the test suite *Tests'*. In particular, we create a new test suite called *Tests"* that we achieve from *Tests'* by pruning all calls to the custom setters added in *Program'* (Figure 3.2, block *Prune APIs from tests*). The custom setters pruning can be safely performed without breaking the syntactic validity of the test cases since they are the only additional element not existing in the original program. Listing 3.4 shows a sample test case for the API facilitated class in Listing 3.3 that makes use of the custom setter. Listing 3.5 contains the same test case after performing the pruning operation. In detail, all calls to the custom setters are commented out; while all the following method calls are extracted from the assertions (if inside any) and enclosed in try-catch blocks to prevent early test case termination due to exceptions that are raised by method calls on potentially null objects. We do not care about the semantic validity of the test case, since for our technique the only thing that matter is if the test generator is able to cover a certain program state. Consequently, a pruned test case is none other than a test method that the test generator could potentially easily cover even without custom setters.

```
public class TestFacilitated{
  ...
  @Test
  public testTarget{
    Example obj = new Example
   ();
    int result = obj.setField
   (55);
    assertEquals(-1, obj.
   targetMethod());
  }
}
```

Listing 3.4. Example test case of the API facilitated class

```
public class TestPruned{
  ...
  @Test
  public testTarget{
    Example obj = new Example
   ();
    //int result = obj.
   setField(55);
    try{
      obj.targetMethod();
    } catch (Exception ex) {}
  }
}
```

Listing 3.5. Pruned example test case of the API facilitated class

In Figure 3.2, the top one of the +-circles represents the test suite that we achieve by merging *Tests* and *Tests"* and that our technique thereon considers as representative of the program behaviors that automatic test generator could hit without facilitated APIs. For similar reasons, the test cases in *Tests* and *Tests"* must be accounted among the ones that the automatic test generator could generate also against *Program'*, and thus, to this purpose, our technique relies on the test suite that we achieve by merging *Tests*, *Tests'* and *Tests"*, as illustrated at the bottom +-circle in figure 3.2.

Mutation Analysis

Following the workflow presented in figure 3.1, after the test generation phase is complete, we move onto the mutation analysis phase. In figure 3.2, the two blocks *MA* indicate that our technique executes the mutation analysis tool for the test suites that we generated for both the program under test *Program* and its augmented version *Program'*. As result we collect:

- the *baseline mutants*, i.e., the mutants that are revealed or executed with the test suite generated for *Program'*. These mutants were provably executed with the automatic test generator, even if this could have been achieved with the help of the facilitated APIs available in *Program'*.
- the *executed mutants*, i.e., the mutants that are executed with the test suite generated for the original program. These mutants were provably executed with automatic test generator with the original program APIs as well.
- the *revealed mutants*, i.e., the mutants that are revealed with the test suite generated for *Program'*. These mutants were provably revealed with at least a test case in which they could be successfully executed. In facts we refer to the test suite generated for *Program'* because we are interested in whether the mutants were ultimately revealed, modulo having executed them anyhow.

Judging testability evidences

The final step consists in judging the testability evidences. We proceed in two steps: first, we judge a controllability and an observability evidence for each baseline mutant (that is, for each baseline mutant, we judge whether it testifies in favor or against controllability, respectively, and in favor or against observability, respectively) and then we aggregate the testability evidences related to the mutants that correspond to faults seeded at the same line of code. We refer to the testability evidences inferred at a line of code as *unitary testability evidences*. Specifically, we proceed as follows.

First, for each baseline mutant in which the sampled test suites can execute (resp. cannot execute) the seeded fault also against the target program with its original APIs, we judge a controllability (resp. non-controllability) evidence. In similar fashion, for each baseline mutant in which the sampled test suites can reveal (resp. cannot reveal) the seeded fault also based on actual outputs of the target program, we judge an observability (resp. non-observability) evidence. Then, for each line of code associated with at least a baseline mutant, we infer a unitary controllability (resp. observability) evidence if more than half of the associated baseline mutants vote as controllability (resp. observability) evidences; or we infer unitary non-controllability (resp. non-observability) evidence otherwise. Aggregating the testability evidences as unitary testability evidences prevents the unbalanced skewing of the results towards the instructions that offers more opportunities for applying mutations than others (e.g., when a high number of mutants is focused only on a single line of code).

Once we have collected the unitary testability and non-testability evidences as above, we refer to those unitary evidences to reason on the testability of given software modules that belong to the program under test. To this end, we first map each target module (e.g., class, method) to the subset of unitary evidences that relate with that module, and then aggregate those unitary evidences into a testability value measured in the interval $[0, 1]$, where 0 and 1 correspond the minimum and the maximum testability values that we can estimate for a module, respectively.

Let M be a software module that belongs to the program under test, and let $\mathit{contr}^+(M)$, $\mathit{contr}^-(M)$, $\mathit{obs}^+(M)$ and $\mathit{obs}^-(M)$ be the subsets of positive and negative controllability and observability evidences, respectively, that we mapped to the module M, out of the collected unitary evidences. Then, by referring to the size of those sets, we estimate the controllability and the observability and testability scores of the module M as follows:

Definition 7. Sampling-based controllability score (final definition) The portion of positive controllability evidences that we obtained from the software module M

$$Controllability(M) = \frac{|\mathit{contr}^+(M)|}{|\mathit{contr}^+(M)| + |\mathit{contr}^-(M)|},$$

Definition 8. Sampling-based observability score (final definition) The portion of positive observability evidences that we obtained from the software module M

$$Observability(M) = \frac{|obs^{+}(M)|}{|obs^{+}(M)| + |obs^{-}(M)|}.$$

Definition 9. Sampling-based testability score (final definition) We estimate the testability of the software module M as the combination of its controllability and its observability, namely, as the geometric mean of the two:

$$Testability(M) = \sqrt{Controllability(M) \times Observability(M).}$$

3.3.3 Conclusiveness of the testability scores

We acknowledge that the testability evidences collected with our technique can be sometimes insufficient to calculate reliable estimates for some program modules. In particular, we reckon this to be the case if our technique was unable to significantly sample the execution space of the module.

We recall that the unitary testability evidences collected with our technique refers to the lines of code that we associated with some baseline mutants, i.e., to the seeded faults that we were able to exercise with some automatically generated test cases. As a matter of facts, there can be some lines of code for which our technique was unable to exercise any mutant, thus ending up with not collecting any evidence (neither positive nor negative evidences) about their testability.

When reasoning on the testability of a module of the program under test, we mark our estimates as *inconclusive* if the portion of module's lines of code for which we successfully computed testability evidences were not a representative sample out of the module's lines of code that were associated with some mutants. We grounded our calculations on the classic theory of small sample techniques [68].

Definition 10. Determining sample size needed to be representative of a given population:

$$s = \frac{X^2 \times N \times P \times (1-P)}{d^2 \times (N-1) + X^2 \times P \times (1-P)}$$

Where:
s = required sample size.

X^2 = the table value of chi-square for 1 degree of freedom at the desired confidence level.
N = the population size.
P = the population proportion.
d = the accuracy of the sample proportions.

For our study we set P to 0.5 since we do not know the population proportions. The confidence level is set to 95%, this means that X = 1.96 and X^2 = 3.8416. Finally we set the accuracy to 15%.

We acknowledge that the possibility of producing inconclusive results for some modules is an intrinsic limitation of our technique. Depending on the actual implementations of the technique, the concrete manifestation of this limitation boils down to the characteristics of the tools with which we instantiate the test generation tool and mutation analysis phases. Explicitly pinpointing the conclusiveness of the estimates aims to alleviate the impacts of such limitation.

3.4 Prototype

We have currently implemented the entire workflow of Figure 3.2 for programs in Java as a fully automated process scripted in Bash and Java.

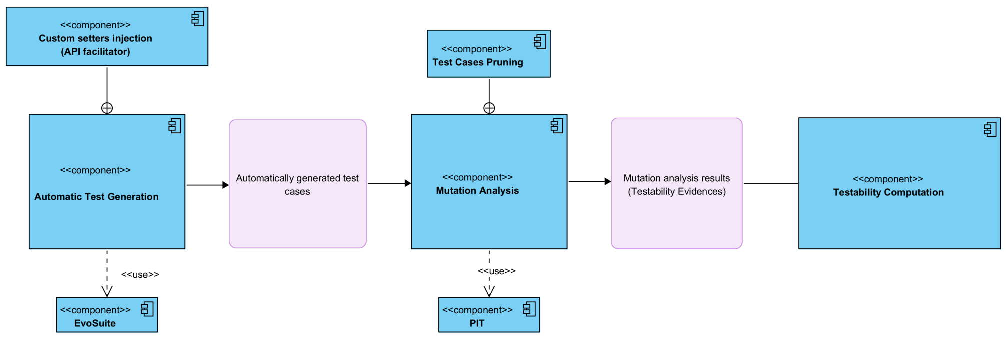

Figure 3.3. Component diagram of our prototype

Figure 3.3 shows the component diagram of our prototype. In the following subsections we will provide a brief explanation on how we concretely performed the API augmentation and test cases pruning (Subsection 3.4.1), we will introduce the automatic test generator (Subsection 3.4.2) and mutation analysis tool (Subsection 3.4.3) used in our technique.

3.4.1 API augmentation and test cases pruning

As already explained in the previous sections the API augmentation is performed through the automatic injection of custom setters. In practice, we developed a Java tool that through the help of the JavaParser [69] library. With the help of JavaParser we traverse the Java code of the analyzed class and extract all the field that are private or protected. Once this step is completed, we inject the code of the class with our custom setters for all the private or protected fields. For non-primitive fields we add a setter with two branches, one that is entered when the input parameter is set to *null*, the other one in all the other cases; this has the aim of steering the test generator tool to try to generate both *null* and non-null values for a specific field.

Similarly, for test case pruning, we use JavaParser to traverse the code of the automatically generated test cases until the first custom setter is invoked (if at all). The pruning is done by: (i) removing the line of code invoking the custom setter, (ii) adding a try-catch block which catches a generic exception, (iii) adding inside of it all the instructions following the custom setter invocation (excluding other possibly encountered custom setters). Since existing assertions would potentially break the newly created test cases (since we remove the custom setters), in step iii) we extract the instructions inside of the asserts and add them as normal calls. This will ensure that the newly created pruned test case will invoke exactly the same ordered list of instructions of the original test case, except for all the custom setters invocations.

3.4.2 Test Generation

Our current implementation generates test cases with the test generator EvoSuite that exploits a *search-based test generation algorithm* to generate test cases for Java classes [4].

Reference tool

EvoSuite is a tool that automatically generates test cases with assertions for classes written in Java code.

The first version of EvoSuite used an evolutionary search approach that evolved whole test suites with respect to an entire coverage criterion at the same time [70]. Optimizing with respect to a coverage criterion rather than individual coverage goals achieves that the result is neither adversely influenced by the order nor by the difficulty or infeasibility of individual coverage goals. A common systematic approach to test generation is to select one coverage goal for a given coverage criterion at a time, and to derive a test case that exercises this particular goal. A major flaw in this strategy is that it assumes that all coverage goals are equally important, equally difficult to reach, and independent of each other. Unfortunately, none of these assumptions holds since many goals could be infeasible or could be more difficult to satisfy than others. Therefore, given a limited amount of resources for testing, a lucky choice of the order of coverage goals can result in a good test suite, whereas an unlucky choice can result in all the resources being spent on only few test cases. To overcome these problems, whole test suite generation is an approach that does not produce individual test cases for individual coverage goals, but instead focuses on test suites targeting an entire coverage criterion. EvoSuite implements whole test suite generation as a search-based approach. In evolutionary search, a population of candidate solutions is evolved using operators imitating natural evolution such as crossover and mutation. Individuals are selected for reproduction based on their fitness and with each generation the fitness improves until a solution is found or the allotted resources are used up. In Evosuite, a candidate solution is a test suite, consisting of a variable number of individual test cases. Crossover between two test suites exchanges test cases based on a randomly chosen crossover position, while mutation of a test suite may add new test cases or mutate individual tests. To help the generation of appropriate input data, EvoSuite also makes use of focused local searches and dynamic symbolic execution after every predefined number of generations. The fitness of individuals is measured with respect to an entire coverage criterion, such as branch coverage. As longer method sequences make it easier to reach coverage goals, EvoSuite allows the search to dive into long sequences, but applies several bloat control techniques [71] to ensure that individuals do not become excessively large. At the end of the search, test suites are minimized such that only statements contributing to coverage remain.

EvoSuite evolved across the years and adapted the most recent state of the art techniques to improve its effectiveness. In the latest version, which we used for our study, instead of relying on the whole-suite approach, a many objective optimization algorithm has been introduced, called DynaMOSA. As already mentioned, the whole suite approach is indeed more effective than targeting one target at a time, but it suffers all the well-known problems of sum scalarization in many objective optimization, among which the inefficient convergence occurring in the non-convex regions of the search space, hence why new techniques have been researched with the aim of improving automatic test generation. DynaMOSA (Many-Objective Sorting Algorithm with Dynamic target selection) is an evolution of the many-objective many-objective genetic algorithm called MOSA [72]. Different from the whole suite approach, MOSA evolves test cases that are evaluated using the *branch distance* and *approach level* for every single branch in the CUT. Consequently, the overall fitness of a test case is measured based on a vector of n objectives, one for each branch of the production code. Thus, the goal is finding test cases that separately satisfy the target branches. To focus the selection towards tests reaching uncovered branches, MOSA proposes a new way to rank candidate test cases, called preference criterion. Panichella et al. [73] further improved the MOSA algorithm by presenting DynaMOSA. Relying on the control dependency graph (CDG), DynaMOSA narrows the search towards the uncovered targets free of control dependencies. New targets are then iteratively considered when their dominators are satisfied. In particular, the difference between DynaMOSA and MOSA is the following: at the beginning of the search, DynaMOSA tries to hit only the targets free of any control dependencies. Thenceforth, at every iteration, the current set of targets is updated based on the execution results of the newly generated offsprings. This approach does not impact the way MOSA ranks the generated solutions, but rather speeds up the convergence of the algorithm, while optimizing the size of the current objects. Empirical results show that DynaMOSA performs better than both the whole suite approach and MOSA in terms of branch, statement, and strong mutation coverage.

The test cases found in the generated test suites are in the form of sequences of calls to the constructors and the methods of both the class itself and the classes that define the types of the parameters needed (recursively) for instantiating those constructors and method calls.

Furthermore, Evosuite is able to automatically generate the oracles of the generated test cases. Given an automatically generated unit test, there is only a finite number of things one can assert, the choice of assertions is defined by the possible observations one can make on the public API of the UUT and its dependent classes. Consequently, synthesizing the possible assertions is easily possible. However, presenting all assertions to the developer is problematic, as there might be too many of them, and many of them will be irrelevant. EvoSuite uses mutation testing to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the test cases. These assertions highlight the relevant aspects of the current behavior in order to support developers in identifying defects, and the assertions capture the current behavior to protect against regression faults.

Usage in our implementation

For each of the *TG* blocks in Figure 3.2, our technique runs EvoSuite for a maximum time budget that depends on the size of the class, considering a minimum time budget of 2 minutes for the smallest classes in the considered project and a maximum time budget of 20 minutes for the largest classes, while linearly scaling the time budget for the classes of intermediate size.

Furthermore, our technique acknowledges the dependency of the search-based algorithm of EvoSuite from the different code coverage criteria that the tool allows as possible fitness functions, and from the intrinsic randomness that underlies several steps of the algorithm, for instance, when sampling the initial set of test cases or when evolving the test cases incrementally, which can naturally make EvoSuite generate different sets of test cases at different runs. Aiming to exercise as many program behaviors as possible, we set EvoSuite to address the following fitness functions: line coverage, branch coverage, exception coverage, weak mutation coverage, method-output coverage, top-level method coverage, no-exception top-level method coverage and context branch coverage.

Our technique makes use of the assertion-style oracles that EvoSuite generated in the test cases, in particular, we instructed EvoSuite to generate assertions for all program outputs encompassed in the test cases (Evosuite option `assertion_strategy=ALL`), instead of selecting only the ones that it judged as relevant based on its internal mutation testing, since we aim to reveal as many mutants as possible, even if the test cases could become large.

3.4.3 Mutation Analysis

We use the mutation analysis tool PIT [5] to both seed possible faults of the program under test and characterize the generated test suites according to their ability to execute and reveal those seeded faults. This information is used to judge the testability evidences that the generated test suites provided for the program under test.

Reference tool

Many mutation testing tools for Java code have been developed mainly to support research. In the past, among the Java tools, the most popular ones were MuJava [74] and Major [75]. Unfortunately, these tools were built to support research projects and thus, their practical use is limited. PIT offers the following three major advantages over the other tools: a) it is open source, b) it is well integrated with development tools, as it offers a Maven plugin and c) it is quite robust and actively maintained. PIT also operates on the latest version of Java while other tools have not been updated.

Since PIT was developed with the aim of being easily usable by everyone in any given context, it aims at:

- having a good integration with build tools (e.g., Maven, Ant, Gradle), integrated development environments (IDEs, such as, Eclipse or IntelliJ) and static code analysis tools (e.g., SonarQube). It is easy to start working with PIT using build tools.
- being fast. PIT uses three techniques to obtain its quick results: working on bytecode instead of source code, selecting the tests to run against the mutants and minimizing the number of mutant executions.
- providing a clear report of the tests' execution, which makes a navigation between source code and mutants easy and highlights the mutants that were not killed (those for which the testers need to investigate).

PIT generates mutants via bytecode manipulation. The approach taken offers significant performance advantages when compared with compiled mutant files as it practically reduces the mutant generation cost to zero (because bytecode manipulation is computationally inexpensive). The bytecode representation of the mutants does not require any program to be written on the disk but, instead to keep it in memory.

Mutant generation is a two stage process. An initial scan is performed in the main controlling process. During this phase all the possible mutation points in the provided classes are identified and memorized, these points are referred to as MutationIdentifiers. A MutationIdentifier consists of the precise location of the mutation and the name of the mutation operator. In the next phase, the tool assesses each mutant by running tests against it by creating child JVM processes. The MutationIdentifier and names of selected tests to run against the mutant are passed to the child by the controlling process. The mutant bytecode is then generated within the child process and inserted into the running JVM using the Java instrumentation API. Since creating a JVM is a very expensive operation, PIT tries to minimize the number that are created. This may affect the results of the tests when run against other mutants. A trade-off is therefore made between performance and isolation. However, PIT can be configured to give stronger guarantees (including launching a JVM per mutants) at the expense of performance.

PIT comes with a set of mutation operators that can be applied to the Java code that can be found in the documentation of the tool.[3]

Usage in our implementation

We chose PIT as the mutation analysis framework for our tool mainly because it offers compatibility with the latest versions of Java, it is constantly kept up-to-date, it is open source and widely used, and most important of all, its compatibility with EvoSuite is explicitly declared, and they provided a set of indications on how to configure the two tools to make them compatible. We instructed PIT to launch a single JVM per mutant with the objective of obtaining results that are as accurate as possible.

Following the schema in figure 3.2 we execute the mutation analysis with PIT against the subject software with both of the sets of test cases generated with the help of EvoSuite (obtained from the original and API facilitated version of the subject software) in addition to the set obtained with pruning.

In the usage of PIT for our tool, we specifically considered the set of

[3]`https://pitest.org/quickstart/mutators/`

mutation operators that PIT advises as the DEFAULT group plus the Remove Conditionals and Experimental Switch mutators, for a total of 13 mutation operators that address several types of mutations at the level of the arithmetic operators, the comparison operators, the logic operators, the return values and the if and switch statements in the programs. We did not consider the larger set of mutation operators that PIT refers to as the "all" group because either they are marked as *experimental* in the documentation, or our initial experiments showed that they result most often in duplicating mutants that we already obtain with the mutation operators of our choice. The full list of used mutators with a description of each of them can be found in table 3.1.

Table 3.1. Table detailing the mutators that we used from the ones available in PIT.

Conditional Boundary	The conditionals boundary mutator replaces the relational operators <, <=, >, >=
Increments	The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.
Invert Negatives	The invert negatives mutator inverts negation of integer and floating point numbers.
Math	The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation.
Negate Conditionals	The negate conditionals mutator will mutate all conditionals found.
Void Method Calls	The void method call mutator removes method calls to void methods.
Empty Returns	Replaces return values with an ‘empty’ value.
False Returns	Replaces primitive and boxed boolean return values with false.
True returns	Replaces primitive and boxed boolean return values with true.
Null Returns	Replaces return values with null. Methods that can be mutated by the Empty Returns mutator or that are directly annotated with NotNull will not be mutated.
Primitive Returns	Replaces int, short, long, char, float and double return values with 0.
Remove Conditionals	The remove conditionals mutator will remove all conditionals statements such that the guarded statements always execute.
Experimental Switch	The switch mutator finds the first label within a switch statement that differs from the default label. It mutates the switch statement by replacing the default label (wherever it is used) with this label. All the other labels are replaced by the default one.

PIT monitors the execution of the test suites against the mutants that it computes according to the selected mutation operators, and classifies the mutants as either *revealed*, *executed* or *missed*. PIT classifies a mutant as revealed, if at least a test case produces a different result when executed against the original program or the mutant program, respectively. That is, the test case executes with no exception and raises an exception for either program, or it raises different exceptions for either program, or it passes all test oracles and fails for at least a test oracle for either program, or it fails with respect to different test oracles for either program. PIT classifies a mutant as executed, if it could not classify the mutant as revealed, but there is at least a test case that executes the code in which the corresponding fault was injected. PIT classifies a mutant as missed if it could not classify it neither as revealed nor as executed.

Chapter 4

Experiments and results

We validated our methodology for estimating software testability through a set of experiments. Our experiments were driven by two main research questions:

- RQ1, *Validity:* Do our testability estimates effectively pinpoint software modules that suffer or do not suffer of testability issues?
- RQ2, *Conclusiveness:* How frequently can we achieve conclusive estimates of testability with our current implementation of the technique?

We studied these questions with three types of experiments:

- EXP1: Qualitatively, by manually inspecting software modules for which we estimated the highest or lowest testability, respectively, trying to empirically validate the goodness of the indications provided by the testability estimates.
- EXP2: Quantitatively on historical data, by estimating the testability of many software modules, and then measuring the extent of correlation with the cost of testing those modules (extrapolating the testing costs by analyzing the test cases that were available in the project repositories).
- EXP3: Quantitatively in controlled experiments with developers, by estimating the testability of sample software modules, and then comparing our estimates with the testing cost evaluated by running testing sessions focused on those modules.

In all experiments we considered software modules out of open-source Java projects as benchmarks. In fact, our current prototype addresses Java programs. We addressed the research question RQ1 in all experiments, while we addressed the research question RQ2 only in experiments EXP1 and EXP2. Furthermore, in the experiments EXP1 we considered the Java classes as unitary software modules, while in the experiments EXP2 and EXP3 we focused on the methods of the Java classes. We explain these choices in more detail below.

The qualitative study (EXP1) addressed the research question RQ1 by manually analyzing the subset of classes for which our metric showed the lowest (resp. highest) testability scores with the aim of confirming if our metric highlights actual testability issues. The selected classes came from a previously determined subset containing only the classes for which we were able to obtain conclusive testability estimates (thus answering RQ2). This study considered Java classes as module units, since it was more reliable for us to manually characterize the possible testability issues with respect to design decisions made at class level, rather than reasoning on smaller units (i.e., the class methods) separately.

The quantitative studies (both EXP2 and EXP3) addressed RQ1 by focusing on the relation between the testability estimates and the actual testing costs. In these studies, RQ2 is addressed only in EXP2. In EXP3, due to its cost in terms of time and human resources, we decided to ask the third-party tester to run the test sessions on a set of methods for which we knew that our metric could provide conclusive results. We remark that the third-party tester could freely choose any method to test from the ones for which we could obtain conclusive estimates, so the only limitation is related to the conclusiveness. We feel like this is not a limiting factor since RQ2 was already studied in EXP1 and EXP2. The quantitative studies considered Java methods as module units, since it was more reliable to associate the available test cases with class methods or simulate the testing session for class methods, rather than entire Java classes. For the study EXP2, we processed the available test cases to derive a ground truth that reflects the test effort spent by the developers to test the considered Java methods, and we crosschecked the correlation between the test effort and our (conclusive) testability estimates, also comparing our results with the ones that could be obtained by using static software metrics as estimators of testability. Furthermore, we analyzed if combining our testability estimates and static software metrics improved over using those estimators separately. For EXP3, we involved a third-party developer to devise the test cases for a set of Java methods.

The following section will introduce the subjects considered in the experiments, while the further ones will go in detail over each of the above mentioned experiments.

4.1 Subjects

For the first two experiments (EXP1 and EXP2), we selected from GitHub three open-source Java projects that:

- use Maven as build tool, as this is a requirement of our current implementation of the technique,
- are representative of large projects comprised of at least 500 classes,
- are representative of different types of software developments, namely, a programming library, a software engineering tool and a business oriented application,
- include at least 250 methods that can be associated with test cases available in the projects (the motivations of this choice and how this is done are detailed in Section 4.3).

The three selected projects are:

- JFreeChart, a *programming library* that supports the display of charts,
- Closure Compiler, a *software engineering tool* that parses and optimizes programs in Javascript, and
- OpenMRS-Core, a *business-oriented application* for the healthcare domain.

Instead, for the third experiment (EXP3), that requires developers to simulate testing sessions, we selected the jsoup project, an open-source Java HTML Parser. We choose this project mainly due to its moderate size (it has "only" 64 classes) and well-documented code.

4.2 Qualitative study

We conducted a qualitative study on the classes of the three above presented projects, selecting the classes with the lowest and highest *Controllability* and *Observability* scores, to try to confirm our hypothesis that low and high scores according to our metric can be reconciled to actual testability issues and boosters, which can be revealed with manual inspection of the classes.

4.2.1 Experimental Settings

We instantiated our technique with EvoSuite, version 1.2.0, and PIT, version 1.8.1. In Section 3.4 we have already described the configuration of EvoSuite with respect to the fitness functions, the time budget and the generation of assertions, and the mutation operators used with PIT.

Table 4.1 shows how many classes belong to each project. The columns *All* refer to all the classes in the projects, while the columns *Considered* and *Subjects* refer, respectively, to the subset of classes for which we have at least a mutant, and to the further subset that we selected as actual potential subjects for our qualitative study.

Table 4.1. Number of classes in the three considered Java projects

	Number of classes		
Project	All	Considered	Subjects
JFreeChart	632	471	321
Closure Compiler	839	606	455
OpenMRS	735	542	267

The number of classes for which we have at least a mutant, is lower than the total number of classes due to the existence of interfaces and simple classes without mutants; such classes can be said to be testable by definition (testability score of 100%) since they include only basic instructions.

We further refined the set of subject classes for the qualitative study by excluding the classes for which PIT computed mutants for at most 9 lines of code. This has been done with the aim of excluding from the qualitative study the classes which have only a little amount of code with potential testability issues.

We discriminated inconclusive testability estimates by determining, for each subject class, the threshold for the minimal number of lines of code that we must sample with testability evidences out of the lines for which PIT identified at least a mutant. We computed the thresholds by referring to the classic approximation to the hypergeometric distribution [68], setting the confidence level to 95%, the population portions to 0.5 and the corresponding accuracy to 15%.

We performed the study on a total of 20 classes per each project, choosing them as in the following: (i) 5 classes with the lowest controllability, (ii) 5 classes with the highest controllability, (iii) 5 classes with the lowest observability, (iv) 5 classes with the highest observability.

For each one of the selected classes we report our findings on the controllability (resp. observability) issues or boosters that we identified manually.

4.2.2 Results

The subsections in this chapter will answer the two research questions presented at the start of this chapter 4. Since knowing for which classes we have obtained a conclusive testability estimation is a prerequisite to be able to answer RQ1, we will first present the results related to RQ2 in the following subsection.

Conclusiveness (RQ2)

Table 4.2 reports, for each of the three Java projects (column *Project*) and set of subject classes (column *Subjects*), the number of classes for which we achieved conclusive estimations (column *Conclusive*) and the corresponding portion (column *Portion*).

Table 4.2. Conclusive testability estimations

Project	Subjects	Conclusive	Portion
JFreeChart	321	269	84%
Closure Compiler	455	238	52%
OpenMRS	267	152	57%

The portion of inconclusive estimations is evidently not negligible, ranging between 16% and 48% across the three Java projects. The inspection of the classes with inconclusive estimations revealed that, as we expected, many parts of code of the subject classes were not hit with any test case from EvoSuite since they depended on inputs that EvoSuite cannot generate due to limitations of its current implementation. For example, we identified several methods that take files and streams as inputs (e.g., parameters of type *ObjectInputStream*) that EvoSuite does not currently handle.

We remark that EvoSuite is a research prototype, though very popular in the community of researchers that work on test generation, and we did not expect it to be perfect. Tuning our technique with further test generators or even ensembles of test generator (as well as experiencing with further mutation analysis tools other than PIT) is an important milestone for our technique to make its way to practice, and definitely the most relevant next goal in our research agenda. But we also underline the importance of studying the merit of our novel proposal for the cases in which we could indeed achieve conclusive results with the current implementation.

Does our metric highlight actual testability issues (RQ1)

To answer RQ1, we performed a manual analysis of a subset of the classes that have shown the highest and lowest *Controllability* and *Observability* scores across the three considered projects. Tables 4.3 to 4.5 contain the results of our qualitative analysis on controllability and observability for each project. The first two columns of Tables 4.3 to 4.5 indicate the class name (column *Class*) and the corresponding *Contr* (resp. *Obs*) score. The last column reports our findings on the controllability (resp. observability) issues or boosters that we identified manually in the classes. In case no issues or booster is identified we report it with the **(n.a.)** wording.
In the following we provide an explanation of each kind of issue (resp. boosters) for *Controllability* and *Observability* that we found in the analyzed classes.

Controllability Issues:

- **(ci1) Multi-step protocol**: The class interface induces an interaction protocol that requires test cases to call multiple methods in specific sequences to set the relevant input states, thus hardening the task of identifying tests.

- **(ci2) Complex structured inputs**: The class takes several inputs defined as complex data structures, and thus requires long test cases

Table 4.3. JFREECHART - Qualitative study of classes with lowest and highest *Controllability* and *Observability*

Controllability

Class	**Contr**	**Issue/Boost**
org.jfree.data.time.TimeSeriesCollection	91.80%	**(ci3)**
org.jfree.chart.editor.DefaultPolarPlotEditor	92.59%	**(ci2)**
org.jfree.data.jdbc.JDBCPieDataset	93.75%	**(ci1)**
org.jfree.chart.editor.DefaultLogAxisEditor	95.45%	**(ci1)**
org.jfree.chart.editor.DefaultNumberAxisEditor	96.15%	**(ci1)**
org.jfree.chart.date.SerialDate	100%	**(cb1)**
org.jfree.data.xy.XYDataItem	100%	**(cb1) (cb2)**
org.jfree.chart.renderer.Outlier	100%	**(cb1)**
org.jfree.data.gantt.Task	100%	**(cb1) (cb2)**
org.jfree.data.xy.OHLCDataItem	100%	**(cb1) (cb2)**

Observability

Class	**Obs**	**Issue/Boost**
org.jfree.chart.renderer.category.StandardBarPainter	0%	**(oi1) (oi2)**
org.jfree.chart.renderer.xy.StandardXYBarPainter	0%	**(oi1) (oi2)**
org.jfree.chart.renderer.category.GradientBarPainter	2.86%	**(oi1) (oi2)**
org.jfree.chart.renderer.xy.GradientXYBarPainter	6.06%	**(oi1) (oi2)**
org.jfree.chart.needle.MiddlePinNeedle	6.25%	**(oi2)**
org.jfree.data.time.Day	100%	**(ob2)**
org.jfree.data.time.SimpleTimePeriod	100%	**(ob1) (ob2)**
org.jfree.data.xy.XYDataItem	100%	**(ob1) (ob2)**
org.jfree.data.xy.XYCoordinate	100%	**(ob1) (ob2)**
org.jfree.chart.urls.CustomPieURLGenerator	100%	**(ob1)**

Table 4.4. CLOSURE COMPILER - Qualitative study of classes with lowest and highest *Controllability* and *Observability*

Controllability

Class	**Contr**	**Issue/Boost**
com.google.javascript.jscomp.CheckUnreachableCode	58.33%	**(ci1)**
com.google.javascript.jscomp.PeepholeFoldConstants	67.39%	**(ci1)**
PeepholeSubstituteAlternateSyntax[i]	68.67%	**(ci1)**
com.google.javascript.jscomp.lint.CheckConstantCaseNames	76.92%	**(ci1) (ci2)**
com.google.javascript.jscomp.FindModuleDependencies	77.78%	**(ci1) (ci2)**
com.google.javascript.jscomp.regex.RegExpTree	100%	**(cb1)**
com.google.javascript.jscomp.GoogleJsMessageIdGenerator	100%	**(cb1) (cb2)**
com.google.javascript.jscomp.JsMessage	100%	**(cb1) (cb2)**
com.google.javascript.jscomp.regex.CharRanges	100%	**(cb1) (cb2)**
com.google.javascript.jscomp.ModuleIdentifier	100%	**(cb1)**

Observability

Class	**Obs**	**Issue/Boost**
com.google.javascript.jscomp.JsonErrorReportGenerator	0%	**(oi2)**
com.google.javascript.jscomp.WarningLevel	5.41%	**(oi1)**
com.google.javascript.refactoring.FixingErrorManager	5.88%	**(oi1)**
com.google.javascript.jscomp.FindModuleDependencies	7.41%	**(oi1)**
SemanticReverseAbstractInterpreter[ii]	7.89%	**(oi1)**
com.google.javascript.jscomp.ModuleIdentifier	100%	**(ob1)**
com.google.javascript.jscomp.AccessorSummary	100%	**(ob1)**
com.google.javascript.jscomp.transpile.TranspileResult	100%	**(ob1) (ob2)**
com.google.javascript.jscomp.NodeNameExtractor	100%	**(ob1)**
com.google.debugging.sourcemap.SourceMapObject	100%	**(ob1) (ob2)**

[i] com.google.javascript.jscomp.PeepholeSubstituteAlternateSyntax
[ii] com.google.javascript.jscomp.type.SemanticReverseAbstractInterpreter

Table 4.5. OPENMRS - Qualitative study of classes with lowest and highest *Controllability* and *Observability*

Controllability

Class	**Contr**	**Issue/Boost**
org.openmrs.api.impl.DatatypeServiceImpl	57.89%	**(ci1)**
org.openmrs.module.ModuleClassLoader	82.35%	**(ci1)**
org.openmrs.util.DoubleRange	88.00%	**(n.a)**
org.openmrs.validator.PersonValidator	92.11%	**(ci3)**
org.openmrs.logging.MemoryAppender	94.12%	**(n.a)**
org.openmrs.logic.result.Result	100%	**(cb2)**
org.openmrs.ConceptStopWord	100%	**(cb1) (cb2)**
org.openmrs.BaseFormRecordableOpenmrsData	100%	**(cb1)**
org.openmrs.patient.impl.LuhnIdentifierValidator	100%	**(cb1)**
org.openmrs.Program	100%	**(cb2)**

Observability

Class	**Obs**	**Issue/Boost**
org.openmrs.api.handler.ConceptSaveHandler	0%	**(oi1)**
org.openmrs.validator.BaseAttributeTypeValidator	5.88%	**(oi1)**
org.openmrs.module.ModuleFactory	13.43%	**(oi1)**
org.openmrs.validator.DrugOrderValidator	16.67%	**(oi1)**
org.openmrs.validator.PersonValidator	16.97%	**(oi1)**
org.openmrs.person.PersonMergeLogData	100%	**(ob2)**
org.openmrs.logic.Duration	100%	**(ob1) (ob2)**
org.openmrs.DrugReferenceMap	100%	**(ob2)**
org.openmrs.logic.LogicTransform	100%	**(ob2)**
org.openmrs.scheduler.TaskDefinition	100%	**(ob1) (ob2)**

that go through sophisticated initialization sequences to set the relevant inputs.

- **(ci3) Preconditioned updates**: The interface methods for updating the class state are guarded with many preconditions, and thus the class challenges the testers to comply with the preconditions when specifying the test inputs.

Controllability Boosters:

- **(cb1) Simply-typed control inputs**: The class methods are fully controllable with inputs of primitive type, string type or types defined as flat data structures with only primitive fields and setters for all fields.

- **(cb2) Complete field-input mapping in constructors or setters**: The test cases can rely on class constructors and setters to assign all fields based on simply-typed inputs of the constructors or setters, one input for each field.

Observability Issues:

- **(oi1) Complex observer methods**: The observer methods depend on many parameters, or take complex data structures as input, thus hardening the task of specifying the test oracles.

- **(oi2) Output on system streams**: The class produces most output on system streams, e.g., it writes results to the console or in the GUI, hardening the task of specifying automatic test oracles on those results.

Observability Boosters:

- **(ob1) Output as simply-typed return values**: The class produces all relevant outputs as return values of simple types, which can be easily checked with test oracles.

- **(ob2) Full getter access to modified fields**: The class stores its outputs in fields that the test cases can easily access with provided getter methods.

By analyzing the tables, we can see that our metrics scored maximum values for classes that allow for controlling the execution with simple inputs of interface methods and constructors, and observing the results in return values or with getter methods. Conversely, controllability and observability issues depend on interfaces that hamper test cases from setting relevant input values and inspecting relevant outputs.

For the OpenMRS projects two classes that have been highlighted as having controllability issues are marked as false positive. The first point that should be noted is the fact that we can see that the two classes have relatively high controllability scores, and so should not be considered has having serious controllability issues. Having said this, let us focus on these two false positive to understand their causes. The false positives are highlighted as such since the potential faults that could be difficult to reach during testing are from part of the code that should not be reachable (dead code), as specified in the comments found in the code of the class. Basically, since our technique enrich the API of the software to be able to set any field in a class with the end goal of measuring testability, as specified in section 3.3.1, we are identifying a controllability issue that should be non-existent. Having said this, we may argue that leaving dead code in a software is a known code smell that should in any case be fixed.

In conclusion, from this preliminary qualitative study, we can affirm that our technique is seemingly able to highlight potential testability issues in a software.

4.3 Quantitative study on historical data

We investigated to which extent our estimates of software testability can capture the testing cost by analyzing the complexity of the test cases associated to the methods of a Java program, in a set of experiments with many methods and test cases out of the three Java projects described in section 4.1.

4.3.1 Experimental setting

We instantiated our technique with EvoSuite, version 1.2.0, and PIT, version 1.8.1. In In Section 3.4 we have already described the configuration of EvoSuite with respect to the fitness functions, the time budget and the generation of assertions, and the mutation operators used with PIT.

Table 4.6. Descriptive statistics of the subject methods in the three considered Java projects

	JFreeChart			**Closure Compiler**			**OpenMRS**		
	All	Tested	Subjects	All	Tested	Subjects	All	Tested	Subjects
Number of methods	8552	615	248	14723	301	112	9166	493	241
Total lines of code (LOC)	71703	6788	5199	110553	3203	2274	51412	6527	5402
Average LOC per method	8.38	11.03	20.96	7.51	10.64	20.30	5.61	13.21	22.41
Mininum LOC per method	1	3	5	1	1	1	1	3	6
Maximum LOC per method	288	188	188	433	246	246	221	121	121
Average mutants per method	2.45	3.70	7.42	2.85	3.64	7.56	1.33	4.70	8.28
Mininum mutants per method	0	1	3	0	1	3	0	1	3
Maximum mutants per method	104	74	74	617	66	66	74	46	46

Table 4.6 summarizes descriptive statistics about the Java methods that belong to each project, namely, the number of the methods (first row), their total and individual size (from the second to the fifth row), and the number of mutants in the methods (from the sixth to the eighth row). The columns *All* refer to all methods in the projects, while the columns *Tested* and *Subjects* refer, respectively, to the subset of methods that we were able to successfully associate with some test cases, and to the further subset that we selected as actual subjects for our study. We describe the procedure by which we selected these two latter subsets in the next section. The table shows that we selected methods with increasing size and increasing number of mutants at each selection step.

We discriminated inconclusive testability estimates by determining, for each subject method, the threshold for the minimal number of lines of code that we must sample with testability evidences out of the lines for which PIT identified at least a mutant. We computed the thresholds by referring to the classic approximation to the hypergeometric distribution [68], setting the confidence level to 95%, the population portions to 0.5 and the corresponding accuracy to 15%.

To compare the performance of our testability estimates with the performance of the estimates that can be done with traditional software metrics we used the tool CK[1] [76] to collect the 7 metrics *Loc*, *Rfc*, *Cbo*, *Fan-out*, *Fan-in*, *Cbo-modified* and *Wmc*, for each subject method. *Loc* is the number of lines of code in the method. *Rfc* is the number of unique method invocations done within the method. *Cbo* is the number of non-primitive data types used in

[1]the tool CK is available at `https://github.com/mauricioaniche/ck`

the method. *Fan-out* is the number of unique classes on which the method depends via method calls. *Fan-in* is the number of other methods that call the method within the same class. *Cbo-modified* is the sum of *Fan-out* and *Fan-in*. *Wmc* is the sum of complexities of methods defined in a class. It therefore represents the complexity of a class as a whole and this measure can be used to indicate the development and maintenance effort for the class. In this implementation, the measure of complexity of methods is McCabe's Cyclomatic Complexity. A method with no branches has a Cyclomatic Complexity of 1 since there is 1 arc. This number is incremented whenever a branch is encountered.

We evaluated the test complexity of the test cases associated with each subject method using 5 static metrics extracted from the test cases *Rfc*, *Loc*, *AvgLoc*, *NMC*, *NA*. We extracted them for the ensemble of the test cases associated to each subject method. The *Rfc* counts the number of unique method invocations across all the test cases associated to a method. This metric is the one that should best express the development complexity of the test cases since it allowed us to avoid complexity measures that depended directly on the amount of associated test cases, as multiple test cases may not necessarily indicate high testing complexity if they are all built with the same or similar sets of methods invocations. The *Loc* and *AvgLoc* metrics measure the sum of the lines of code of all the test cases associated to a method and the average number of lines of code of the associated test cases, respectively. The *NMC* metric counts the number of method invocations across all the associated test cases of each method, finally the *NA* metric counts the number of assertions across all the associated test cases of each method.

4.3.2 Ground Truth

Out of the Java methods in the considered projects (Table 4.6, columns *All*), we excluded all methods *hashCode* and *equals* that are usually generated automatically, and further selected only the methods that we could associate with a reference ground-truth, that is, available test cases that the programmers developed for those methods. This because we aimed at investigating the correlation between our testability estimates for the methods and the development complexity of the corresponding test cases, for methods and test cases designed by human programmers. We built on the *methods2test* tool [77] to associate the methods with the test cases available in the projects and selected only the methods for which we identified at least an associated test case (Table 4.6, columns *Tested*).

Methods2test heuristically infers the associations between the available test cases and the methods that are their main testing target. It originally relies on two heuristics, *name matching* and *unique method call*, but we extended it with three additional heuristics, *stemming-based name matching*, *contains-based name matching* and *non-helper unique method call*, which generalize the two original ones with the aim to increase the set of identified associations.

For each test case, which in the considered projects is a test method within a test class, *name matching* searches for a target method that both exactly matches with the name of the test case and belongs to a class that exactly matches with the name as the test class. *Stemming-based name matching* and *contains-based name matching* address the name matching with respect to either the stemmed names of methods and test cases, or whether the test name contains the method name, respectively. For example, *testCloning* and *testCloneSecondCase* will match with method *clone* after name stemming or by name containment, respectively. *Unique method call* further exploits the name-based association between a test class and a target class, by searching for test cases that call a single method of the target class. *Non-helper unique method call* re-evaluates the unique method check after excluding the calls to possible helper methods, such as setter methods, getter methods and the method *equals*.

After the association with the test cases, we further refined the set of subject methods by excluding the methods for which PIT computed mutants for at most two lines of code. For these methods our technique could distil the unitary testability evidences out of a too squeezed population of seeded faults, which results in yielding unbalanced estimates in most cases. We see this as a drawback of the fault models that we are currently able to consider by relying on PIT, rather than as a limitation of our idea of estimating testability based on automatically generated test cases, and we thus dismissed these methods from the current experiments on this basis. We ended with selecting the set of subject methods summarized in the columns *Subjects* of Table 4.6.

4.3.3 Results

The subsections in this chapter will answer the two research questions presented at the start of this chapter 4. Since knowing for which methods we have computed a conclusive testability estimation is a prerequisite to be able to answer RQ1, we will first present the results related to RQ2 in the first subsection.

Conclusiveness (RQ2)

Table 4.7 reports, for each of the three Java projects (column *Project*) and set of subject methods (column *Subjects*), the number of methods for which we achieved conclusive estimations (column *Conclusive*) and the corresponding portion (column *Portion*).

Table 4.7. Conclusive testability estimations

Project	Subjects	Conclusive	Portion
JFreeChart	248	208	86%
Closure Compiler	112	69	62%
OpenMRS	241	141	59%

Like in the previous experiment (EXP1), the portion of inconclusive estimations is evidently not negligible, ranging between 14% and 41% across the three Java projects. The inspection of the methods with inconclusive estimations revealed that, as we expected, many subject methods were not hit with any test case from EvoSuite since they depended on inputs that EvoSuite cannot generate due to limitations of its current implementation. For example, we identified several methods that take files and streams as inputs (e.g., parameters of type *ObjectInputStream*) that EvoSuite does not currently handle.

We remark that EvoSuite is a research prototype, though very popular in the community of researchers that work on test generation, and we did not expect it to be perfect. Tuning our technique with further test generators or even ensembles of test generator (as well as experiencing with further mutation analysis tools other than PIT) is an important milestone for our technique to make its way to practice, and definitely the most relevant next goal in our research agenda. But we also underline the importance of studying the merit of our novel proposal for the cases in which we could indeed achieve

Table 4.8. JFREECHART - Correlations between testability estimates, static metrics and test case complexity.

Legend: T=Our metric, L=Loc, R=Rfc, C=Cbo, CM=CboModified, FI=FanIn, FO=FanOut, W=Wmc.

JFreeChart

	T	L	R	C	CM	FI	FO	W
Loc	-0.16							
Rfc	-0.45	0.64						
Cbo	-0.50	0.41	0.66					
CboModified	-0.40	0.52	0.71	0.66				
FanIn	-	-	-0.20	-0.28	0.15			
FanOut	-0.44	0.50	0.77	0.77	0.93	-0.16		
Wmc	-0.18	0.85	0.51	0.34	0.48	-	0.46	
RfcTest	-0.51	0.23	0.51	0.46	0.45	-	0.50	0.18
LocTest	-0.22	0.23	0.34	0.22	0.30	-	0.32	0.18
AvgLocTest	-0.29	0.17	0.28	0.15	0.27	-	0.26	0.15
NmcTest	-0.20	0.27	0.34	0.16	0.26	-	0.27	0.22
NaTest	-	0.14	-	-	-	-	-	-

conclusive results with the current implementation, which admittedly is our main objective in this paper.

Correlation with cost of testing (RQ1)

To answer RQ1 we analyzed if our metric correlates with the cost of testing each method, expressed with 5 software metrics. Furthermore, we compare the performance obtained with our metric against 7 static software metrics that have been proposed in the state of the art. For this part of the experiment, we focused on the subject methods for which our technique yielded conclusive results.

Tables 4.8, 4.9 and 4.10 report the correlation (as the Spearman rank correlation coefficient[2]) between our testability estimations, the 7 static software metrics that we measured with the tool CK, and 5 software metrics expressing the testing cost (in the form of test case complexity) that have been proposed across different studies, for the subjects methods (for which we were able to reach a conclusive testability estimation) in each considered Java project.

[2]The Spearman rank correlation coefficient indicates the extent to which the ranking of the subjects with respect to an indicator produces a good approximation of the ranking with respect to the other indicator. The correlation value ranges between -1 and 1, being 1 an indication of perfect correlation (same ranking), -1 and indication of perfect anti-correlation (same inverse ranking) and 0 an indication of no correlation (completely different ranking).

Table 4.9. CLOSURE COMPILER - Correlations between testability estimates, static metrics and test case complexity.

Legend: T=Our metric, L=Loc, R=Rfc, C=Cbo, CM=CboModified, FI=FanIn, FO=FanOut, W=Wmc.

Closure Compiler								
	T	L	R	C	CM	FI	FO	W
Loc	-0.27							
Rfc	-	0.58						
Cbo	-	0.35	0.70					
CboModified	-0.33	0.26	0.45	0.39				
FanIn	-	-	-	-	0.56			
FanOut	-0.32	0.40	0.86	0.73	0.57	-		
Wmc	-	0.81	0.45	-	-	-	0.27	
RfcTest	-0.41	-	-	-	0.24	-	0.29	-
LocTest	-0.30	-	0.24	-	-	-	0.24	-
AvgLocTest	-	-	-	-	-	-	-	-
NmcTest	-	0.32	0.26	-	-	-	-	0.33
NaTest	-	0.37	-	-	-	-	-	0.29

Table 4.10. OPENMRS - Correlations between testability estimates, static metrics and test case complexity.

Legend: T=Our metric, L=Loc, R=Rfc, C=Cbo, CM=CboModified, FI=FanIn, FO=FanOut, W=Wmc.

OpenMRS								
	T	L	R	C	CM	FI	FO	W
Loc	-0.28							
Rfc	-0.42	0.72						
Cbo	-0.33	0.40	0.49					
CboModified	-0.27	0.43	0.60	0.28				
FanIn	0.20	-	-	-0.30	0.38			
FanOut	-0.44	0.51	0.78	0.57	0.74	-0.19		
Wmc	-	0.77	0.51	0.32	0.36	-	0.43	
RfcTest	-0.45	0.33	0.47	0.48	0.33	-0.17	0.53	-
LocTest	-0.44	0.32	0.39	0.50	0.19	-0.27	0.44	0.30
AvgLocTest	-0.22	-	0.23	0.35	0.17	-	0.31	-
NmcTest	-0.46	0.32	0.43	0.47	0.22	-0.21	0.46	0.27
NaTest	-0.42	0.24	0.26	0.28	-	-0.18	0.27	0.22

Each cell in the table represents the correlation between the metrics indicated in the titles of the corresponding column and row, respectively. For example, the column T represents the correlations between our testability estimates and all other metrics, and the row *RfcTest* represents the correlation of all possible metrics (including our testability estimates) with the Rfc metric computed at the test cases level.

All reported correlation values were computed with R. The missing correlation values (indicated as *dash* symbols in the table) refer to cases for which we did not find support for statistical significance (p-values greater than 0.05).

We observe that:

- *RfcTest* is the test complexity metric that generally express the highest correlation with all the considered metrics.
- Our testability estimations have a moderate negative correlation with *RfcTest* for the sets of subject methods of all three projects (-0.51, -0.41 and -0.45, respectively).
- Our testability estimates yielded the best correlation with *RfcTest* for the methods of *JFreeChart* and *Closure Compiler*, and the fourth best correlation for the methods of *OpenMRS*.
- Our metric has a moderate negative correlation with *LocTest*, *NmcTest* and *NaTest* for the sets of subject methods of the *OpenMRS* project and is consistently the best or second best at correlating with these metrics in this specific project.
- In most cases, *AvgLocTest* and *NaTest* are the test complexity metric which correlates less with the considered metrics. Notable exceptions are for *JFreeChart* the *AvgLocTest* that is arguably on par with *LocTest* and *NmcTest* and for *OpenMRS* the *NaTest* that is just a bit worse to *LocTest* and *NmcTest*.
- Our testability estimates have weak correlation with the size of the methods measured as the lines of code (top-left correlation value, row *Loc*, in the three value sets in the table).
- The static software metrics resulted in significantly higher correlations with *Loc* (columns L in the table) than *Testability*, with the only excep-

tions of *FanIn* which has basically no correlation with the test complexity metrics, and *Cbo* and *CboModified* in project *Closure Compiler* where however *Cbo* does not correlate at all with the test complexity metrics, while *CboModified* has a weak correlation only with *RfcTest*.

- The correlations between the testability estimator metrics and the test complexity for the Closure Compiler projects are overall comparatively weaker (sometimes not even significant) than the ones of the other two projects. This may be due to how the test cases for this project have been constructed. In particular, in this project, we find an extensive use of helper classes for testing that act as scaffolding for creating test cases. Since these helper classes (and their methods) are defined outside the test cases associated to a method, we potentially lose part of the test complexity expressed with the help of the considered test metrics.

In summary the findings confirm that our testability estimates may contribute to explain the variability in the complexity of the test cases, while capturing a different phenomenon than the size of the software. The other software metrics also correlate with the test complexity metrics, in particular with *RfcTest*, sometimes with comparable strength as our testability estimates, but their independence from *Loc* is questionable. Overall, these findings motivate us to explore the possible synergies between our testability estimates and the static metrics.

Synergy with Static Software Metrics (RQ1)

With the end goal of evaluating if our metric has a potential synergistic behavior with the static software metrics we conducted two types of experiments:

- Performing a study using logistic regression by combining our metric with static software metrics with the goal of predicting the test effort.
- Creating 7 new metrics derived from the combination of our testability metric with static software metrics and evaluating their performance against all testability estimation metrics.

In the following, we will describe each of these experiments and provide their results.

Logistic Regression

We used logistic regression to perform an empirical study with the aim of verifying the relationship between the testability estimator metrics (our metric and the classic static metrics) (independent variables) and the required test effort (dependent variable) in the three considered software project. We trained the model specifying as optimality criterion the ROC curve and we used cross-validation (10-fold cross validation) with the aim of avoiding over-fitting. To determine the testability, we used the RfcTest metric which, as shown in the correlation study, is the metric that has the strongest overall correlation with all testability estimator metrics across the three selected projects. In order to adapt the RfcTest metric to the logistic regression study, we transformed it into a categorical variable. We defined two categories: high required testing effort and low required testing effort. We categorized the tests associated to a method as requiring high testing effort if:

$$RfcTest > TrimmedMean(RfcTest)$$

We associated a low required testing effort otherwise. (NB: we use a 7% trimmed mean to reduce the impact of the outliers).

In our regression study we considered the following metrics: regression coefficient of the involved dependent variables, significance (p-value) of the coefficient, pseudo-R2 metric (McFadden) of the model, Area Under the Curve measure. We'll provide a brief explanation of the considered metrics in the following:

- Regression Coefficient (b). The larger the absolute value of a coefficient, the stronger the impact of the independent variable on the probability of detecting a high testing effort.

- Significance of the coefficient (p-value). The p-value (related to the statistical hypothesis) is the probability of the coefficient being different from zero by chance and is also an indicator of the accuracy of the coefficient estimate. In this study, to decide whether a metric is a statistically significant predictor of testing effort, we use the $\alpha = 0.05$ significance level to assess the p-value.

- Pseudo R2 (R2). Pseudo R2 (McFadden) is defined as the proportion of the total variance in the dependent variable that is explained by the model. The higher the R2 is, the higher the effect of the independent variables, and the more accurate the model.

- Area Under the Curve (AUC). The larger the AUC measure, the better the model is at classifying classes. A perfect model that correctly classifies all classes has an AUC measure of 1. An AUC value close to 0.5 corresponds to a poor model (the model equates to a random choice). An AUC value greater than 0.7 corresponds to a good model.

Tables 4.11, 4.12, 4.13 contain the results of the logistic regression study for each of the considered projects. The rows called Combined refers to the results obtained from combining our metric with the specified object-oriented metric through a multivariate regression, while the row called Original reports the results of the univariate regression to allow for easier comparison of the data.

Focusing on the *Original* rows (univariate regression), the results largely confirm what already emerged from the correlation study, with some slight differences probably induced by the fact that instead of considering the ranks like we did for correlation, in this case the test effort is approximated to a categorical variable. In particular, we can see that:

- The FanIn metric is not significant in any of the considered projects.
- In the Closure compiler project, no metric shows significant values. The transformation of RfcTest to a categorical variable has probably had a negative impact on this project.

Focusing on the *Combined* rows (multivariate regression), the results confirm our hypothesis that our metric captures a complementary dimension of testability compared to traditional static metrics and can be synergistically combined with them for the purpose of predicting software testability. In particular, we can see that:

- As expected, after seeing the univariate study we cannot say anything about the Closure Compiler project, since the results for the base metrics were not significative enough.

Table 4.11. Results of the logistic regression study - JFreeChart

		JFreeChart			
		p	b	R2	AUC
Combined	Loc	0.01	-0.04	0.18	0.76
	OurMetric	0.00	3.79		
Original	Loc	0.00	-0.05	0.07	0.65
Combined	Rfc	0.00	-0.21	0.22	0.79
	OurMetric	0.00	2.93		
Original	Rfc	0.00	-0.24	0.16	0.74
Combined	Cbo	0.00	-0.27	0.18	0.76
	OurMetric	0.00	2.96		
Original	Cbo	0.00	-0.40	0.12	0.70
Combined	CboMod	0.00	-0.30	0.25	0.82
	OurMetric	0.00	3.04		
Original	CboMod	0.00	-0.36	0.19	0.77
Combined	FanIn	0.26	-0.14	0.15	0.71
	OurMetric	0.00	4.24		
Original	FanIn	0.42	-0.08	0.00	0.47
Combined	FanOut	0.00	-0.32	0.25	0.81
	OurMetric	0.00	2.86		
Original	FanOut	0.00	-0.37	0.19	0.77
Combined	Wmc	0.13	-0.06	0.15	0.74
	OurMetric	0.00	3.91		
Original	Wmc	0.02	-0.10	0.03	0.61

Table 4.12. Results of the logistic regression study - Closure Compiler

		Closure Compiler			
		p	b	R2	AUC
Combined	Loc	0.42	-0.01	0.02	0.49
	OurMetric	0.27	1.12		
Original	Loc	0.37	-0.02	0.01	0.47
Combined	Rfc	0.90	0.00	0.01	0.51
	OurMetric	0.24	1.18		
Original	Rfc	0.95	0.00	0.00	0.34
Combined	Cbo	0.79	0.02	0.02	0.52
	OurMetric	0.24	1.21		
Original	Cbo	0.91	0.01	0.00	0.35
Combined	CboMod	0.94	0.00	0.01	0.51
	OurMetric	0.26	1.20		
Original	CboMod	0.75	0.00	0.00	0.48
Combined	FanIn	0.98	0.00	0.01	0.51
	OurMetric	0.26	1.18		
Original	FanIn	0.76	0.00	0.00	0.48
Combined	FanOut	0.88	0.01	0.02	0.50
	OurMetric	0.24	1.20		
Original	FanOut	0.98	0.00	0.00	0.48
Combined	Wmc	0.66	-0.02	0.02	0.50
	OurMetric	0.24	1.20		
Original	Wmc	0.70	-0.02	0.00	0.43

Table 4.13. Results of the logistic regression study - OpenMRS

		OpenMRS			
		p	b	R2	AUC
Combined	Loc	0.01	-0.05	0.13	0.69
	OurMetric	0.00	2.41		
Original	Loc	0.00	-0.05	0.06	0.66
Combined	Rfc	0.00	-0.17	0.18	0.75
	OurMetric	0.03	1.72		
Original	Rfc	0.00	-0.19	0.15	0.73
Combined	Cbo	0.03	-0.19	0.10	0.68
	OurMetric	0.00	2.23		
Original	Cbo	0.00	-0.25	0.05	0.67
Combined	CboMod	0.01	-0.11	0.12	0.69
	OurMetric	0.01	2.16		
Original	CboMod	0.00	-0.13	0.08	0.66
Combined	FanIn	0.28	0.08	0.08	0.63
	OurMetric	0.00	2.54		
Original	FanIn	0.10	0.13	0.02	0.23
Combined	FanOut	0.00	-0.17	0.15	0.73
	OurMetric	0.03	1.72		
Original	FanOut	0.00	-0.20	0.13	0.71
Combined	Wmc	0.05	-0.09	0.10	0.67
	OurMetric	0.00	2.55		
Original	Wmc	0.03	-0.09	0.03	0.58

- The best results for the JFreeChart project are obtained by combining our metric with the CboModified and FanOut Metrics, followed by the Rfc metric.
- For the OpenMRS project the best results are obtained with the Rfc metric followed by the FanOut metric.
- Combining traditional static metrics with our metric shows an improvement in all the cases in which we have significant coefficients.

Combination metrics

We evaluated the performance of the 7 testability indicators obtained by combining each static software metrics with our testability estimates. For each static metric, we derived a new combined indicator assigned to each subject method. This new indicator is computed as the average of the rankings yielded by the static metric and by our testability estimate. For the static metrics that are anti-correlated with our testability estimate (all but *FanIn*, see Tables 4.8, 4.9 and 4.10) we reversed the testability rankings before computing the combined indicators. Table 4.14 contains an example that shows how the new metric is derived. Columns OurMetric and Rfc contain our testability estimation metric and the Rfc static metric. Columns RankT and RankR contain the compute rank for the two metrics. We can see that the rank for the Rfc metric is reversed. Column NewIndicator contains the new metric derived as the mean of the rankings of OurMetric and Rfc.

Table 4.14. Example showing the combination of the ranks

OurMetric	RankT	Rfc	RankR	NewIndicator
0,10	6	30	6	6
1,00	1	3	1	1
0,50	3	10	2	2,5
0,70	2	23	4	3
0,30	4	20	3	3,5
0,20	5	25	5	5

In this study we considered also the methods for which our technique resulted in inconclusive estimates. Since the static metrics are generally available for all methods, and we aim to evaluate if we can benefit from the static metrics in combination with the testability estimates, it makes sense to include those methods as well. For the methods with inconclusive testability estimates, the combined indicator takes into account only the rank derived from the static metric (that is, without any additional benefit from our testability estimate).

Table 4.15. Correlation with the combined testability indicators

JFreechart

	RfcTest		**LocTest**		**AvgLocTest**		**NMCTest**		**NATest**	
	base	comb	base	comb	base	comb	base	comb	base	comb
Loc	0.18	0.35	0.24	0.29	0.19	0.30	0.27	0.30	0.15	0.16
Rfc	0.47	0.53	0.30	0.29	0.24	0.27	0.28	0.26	-	-
Cbo	0.45	0.53	0.21	0.24	0.13	0.21	0.14	0.17	-	-
CboMod	0.46	0.54	0.22	0.22	0.20	0.20	0.18	0.16	-	-
FanIn	-	-0.37	-	-0.26	-	-0.31	-	-0.22	-	0.13
FanOut	0.51	0.58	0.25	0.24	0.19	0.23	0.20	0.20	-	-
Wmc	0.15	0.33	0.21	0.29	0.18	0.30	0.23	0.39	0.14	0.17

Closure Compiler

	RfcTest		**LocTest**		**AvgLocTest**		**NMCTest**		**NATest**	
	base	comb	base	comb	base	comb	base	comb	base	comb
Loc	-	-	0.27	0.32	-	-	0.30	0.31	0.34	0.29
Rfc	0.22	0.32	0.33	0.38	-	-	0.28	0.28	-	-
Cbo	0.26	0.33	0.21	0.29	-	-	-	-	-	-
CboMod	-	0.34	-	0.29	-	-	-	-	-	-
FanIn	-	-0.32	-	-0.23	-	-	-	-	-	-
FanOut	0.23	0.30	0.29	0.34	-	-	-	-	-	-
Wmc	-	-	0.24	0.32	-	-	0.35	0.35	0.33	0.28

OpenMRS

	RfcTest		**LocTest**		**AvgLocTest**		**NMCTest**		**NATest**	
	base	comb	base	comb	base	comb	base	comb	base	comb
Loc	0.38	0.46	0.31	0.37	0.13	0.18	0.33	0.39	0.19	0.25
Rfc	0.52	0.55	0.42	0.46	0.26	0.28	0.45	0.49	0.27	0.33
Cbo	0.46	0.51	0.48	0.50	0.29	0.29	0.47	0.51	0.33	0.38
CboMod	0.39	0.52	0.30	0.52	0.22	0.31	0.31	0.51	0.23	0.41
FanIn	-0.19	-0.32	-0.23	-0.24	-0.14	-0.18	-0.21	-0.29	-0.18	-0.27
FanOut	0.51	0.53	0.45	0.48	0.31	0.31	0.46	0.49	0.33	0.39
Wmc	0.28	0.43	0.34	0.45	-	-	0.33	0.45	0.21	0.31

Table 4.15 reports the correlation between the 5 test complexity metrics and the 7 combined testability indicators (columns *combined*) in comparison with the correlation obtained with respect to the base static metrics alone (columns *base*) for the subject's methods in each considered Java project. We report only the correlation values supported with statistical significance (p-value less than 0.05). The data in the table show that the correlation

yielded with the combined indicators consistently outperformed the correlation yielded with the corresponding static metrics alone for most of the test complexity metrics considered, in most cases with relevant deltas. The only cases in which the *combined* score is worse is in *Closure Compiler* for the *Loc* and *Wmc* metrics only when correlated with the number of assertions. In the *JFreechart* project we see slightly worse results for the *Rfc* and *FanIn* metrics when correlated with *LocTest* and for *Rfc* when correlated with *NumberMethodCallsTest*, but we can see that the delta is clearly negligible (difference of 0.01). This confirms our main research hypothesis that our testability estimates capture a complementary dimension of testability with respect to the traditional software metrics and can be synergistically combined with those metrics for the purpose of predicting software testability.

4.4 Controlled study with developers

We performed an experiment with the aim of measuring the performance of our metric against other software static metrics by making use of additional testing effort information (information that expresses the cost of testing) compared to the one used in the correlation study in the previous section (Section 4.3). These additional data regarding the cost of testing are derived from a testing activity performed on the jsoup project (Section 4.1) by a third-party developer that had to follow a set of given rules specified by us. During this activity the third-party developer had to report a set of information that are used as the testing effort estimators for this experiment.

4.4.1 Experimental settings

As with the two previous experiments, we instantiated our technique with EvoSuite, version 1.2.0, and PIT, version 1.8.1. In In Section 3.4 we have already described the configuration of EvoSuite with respect to the fitness functions, the time budget and the generation of assertions, and the mutation operators used with PIT.

Once again, we discriminated inconclusive testability estimates by determining, for each subject method, the threshold for the minimal number of lines of code that we must sample with testability evidences out of the lines for which PIT identified at least a mutant. We computed the thresholds by referring to the classic approximation to the hypergeometric distribution [68], setting the confidence level to 95%, the population portions to 0.5 and the corresponding accuracy to 15%.

Table 4.16. Number of all, considered, conclusive and subjects methods for the jsoup project

	All	Considered	Conclusive	Subjects
Number of methods	1643	803	640	51

Table 4.16 provides information about the overall number of methods in the jsoup project and the ones that we effectively used in our project. The column *All* refers to all methods in the project, the column *Considered* indicates the number of methods that have at least one applicable mutant in them, the column *Conclusive* refer to the number of methods for which we can compute our metric with a conclusive estimate and finally the column *Subjects* contains the number of methods from which we picked the methods for the controlled study. The *Subjects* method set has been extracted by selecting all methods from the jsoup project that have at least 5 lines of code containing a mutant. This choice has been done with the aim of focusing on non-trivial methods that are potentially more interesting for the study.

We provided to a third-party developer the list of *Subjects* methods from which they should pick from to develop the appropriate test cases. For each one of the selected methods the third-party developer followed the instructions listed below for developing the test cases:

- Devise and implement a test case for the method under test, stopping if time exceed 60 minutes.
- Aiming the testing at (i) statement coverage[3] as first objective and then (ii) to branch coverage, if the maximum time has not elapsed yet.
- Annotate the time needed for each activity.
- Report:
    - A subjective evaluation, expressing if the method was easy or hard to test.

[3]We measure statement coverage through the number of covered instructions

- The statement coverage and branch coverage scores obtained.
- The time elapsed while working to reach statement coverage, in minutes.
- The time elapsed while working to reach branch coverage, in minutes.

4.4.2 Results

In this experiment we analyzed the correlation of our proposed testability metric with the five test effort information that we asked the third-party developer to report. Additionally, we compared the results obtained with our metric against 7 static software metrics that have been proposed in the state of the art.

Table 4.17 contains the list of methods for which the third-party tester effectively developed a test (26 in total), table 4.18 contains, for each method, the following information: our testability score, the subjective evaluation of the tester expressing if he found difficulties in the development of the tests for the method, the final instruction and branch coverage scores reached, the time that was needed to reach statement coverage, the time that was needed to reach statement and branch coverage, the values of the 7 static software metrics used as competitors to our metric.

For some methods (1, 2, 21) we can see that both the instruction and branch coverage does not reach the maximum value, but the assigned time for testing (60 min) still has not elapsed. This is due to methods that contain statements that are unreachable.

From a first analysis of the reported test effort information, we can see that there is a relation between the subjective difficulty expressed by the tester and the time took for testing. As expected, the tests classified as hard to test are usually the ones that took more time to be completed. There are some exceptions, for example method 21 has been classified as easy to test by the third-party tester even if it took him 33 minutes to reach statement coverage and 45 minutes to reach branch coverage. From a brief interview with the third-party tester, we learned that this was due to lacking knowledge on his part about complex regular expressions; after studying them, he was able to quickly create the appropriate tests and consequently he judged the

Table 4.17. Methods considered in the study

ID	Class	Method
1	org.jsoup.select.QueryParser	byAttribute/0
2	org.jsoup.select.QueryParser	combinator/1[char]
3	org.jsoup.safety.Cleaner	createSafeElement/1[Element]
4	org.jsoup.parser.CharacterReader	matchesIgnoreCase/1[String]
5	org.jsoup.parser.CharacterReader	nextIndexOf/1[char]
6	org.jsoup.parser.TokenQueue	matchesAny/1[char[]]
7	org.jsoup.parser.Token$Tag	newAttribute/0
8	org.jsoup.safety.Safelist	removeProtocols/3[String,String,String[]]
9	org.jsoup.parser.CharacterReader	consumeTo/1[String]
10	org.jsoup.parser.CharacterReader	matches/1[String]
11	org.jsoup.parser.CharacterReader	scanBufferForNewlines/0
12	org.jsoup.select.NodeTraversor	filter/2[NodeFilter,Node]
13	org.jsoup.internal.StringUtil	isAscii/1[String]
14	org.jsoup.internal.StringUtil	isBlank/1[String]
15	org.jsoup.internal.StringUtil	isNumeric/1[String]
16	org.jsoup.helper.HttpConnection	data/1[String[]]
17	org.jsoup.safety.Safelist	addProtocols/3[String,String,String[]]
18	org.jsoup.safety.Safelist	addAttributes/2[String,String[]]
19	org.jsoup.select.QueryParser	consumeSubQuery/0
20	org.jsoup.select.NodeTraversor	traverse/2[NodeVisitor,Node]
21	org.jsoup.nodes.Attribute	getValidKey/2[String,Syntax]
22	org.jsoup.nodes.Attributes	indexOfKey/1[String]
23	org.jsoup.parser.CharacterReader	consumeDigitSequence/0
24	org.jsoup.parser.CharacterReader	consumeHexSequence/0
25	org.jsoup.safety.Safelist	addEnforcedAttribute/3[String,String,String]
26	org.jsoup.safety.Safelist	removeEnforcedAttribute/2[String,String]

Table 4.18. Methods testability estimation, test effort metrics and static metrics.

Legend: T=Our metric, D=Subjectively hard to test, CI=Instruction coverage, CB=Branch coverage, TS=Time for statement coverage, TB=Time for branch coverage, CboM=CboModified, FI=FanIn, FO=FanOut

ID	T	D	CI	CB	TS	TB	Cbo	CboM	FI	FO	Wmc	Rfc	Loc
1	0.00	T	89.00	93.80	52	52	11	18	1	17	9	11	19
2	0.33	T	86.40	83.30	19	19	11	15	1	14	11	10	1
3	0.41	T	0.00	0.00	60	60	7	16	1	15	5	12	16
4	1.00	F	100.00	100.00	17	17	1	2	1	1	4	4	11
5	1.00	F	100.00	100.00	4	4	1	2	1	1	3	1	7
6	0.91	F	100.00	100.00	9	9	2	2	1	1	4	2	7
7	0.47	T	100.00	100.00	30	51	2	6	2	4	9	7	21
8	0.58	T	100.00	100.00	40	60	5	6	0	6	4	15	20
9	0.94	F	100.00	100.00	17	17	1	6	3	3	2	3	11
10	1.00	F	100.00	100.00	17	17	1	2	1	1	4	3	7
11	0.49	T	52.20	70.00	60	60	1	4	2	2	6	6	14
12	0.56	T	53.30	40.60	60	60	3	10	3	7	17	7	35
13	1.00	F	100.00	100.00	11	11	1	2	1	1	3	3	10
14	1.00	F	100.00	100.00	6	7	1	7	6	1	5	3	8
15	1.00	F	100.00	100.00	3	4	0	1	1	0	5	3	8
16	0.85	F	100.00	100.00	6	10	5	6	0	6	3	6	12
17	0.53	T	100.00	100.00	40	60	5	8	3	5	4	12	29
18	0.71	F	100.00	100.00	3	5	4	8	3	5	4	11	20
19	0.82	T	100.00	100.00	24	24	2	8	1	7	6	10	11
20	0.68	T	72.30	50.00	60	60	3	20	11	9	12	9	41
21	1.00	F	97.90	91.70	33	45	2	3	3	0	7	3	11
22	1.00	F	100.00	100.00	13	13	1	10	9	1	3	2	7
23	0.89	F	100.00	100.00	12	13	1	3	1	2	4	2	10
24	1.00	F	100.00	100.00	12	13	1	3	1	2	8	2	10
25	0.77	F	100.00	100.00	6	6	5	5	1	4	2	9	18
26	0.63	F	100.00	100.00	4	14	5	3	0	3	4	9	12

method as easy to test. Furthermore, analyzing the subjective difficulty and coverage columns, we can see that almost all methods for which a maximum coverage was not reached have been classified as hard to test.

A problem that may impact the study is the fact that the two features related to the time spent testing may saturate when the target coverage (100%) is not reached, leading to the fact that two methods that are potentially hard-to-test with different levels (different coverage reached) are indistinguishable from one another. With the goal of mitigating this problem in our experiments, we introduced two features derived as the combination of the time required for reaching the specific coverage and the coverage value, called UTS (Time per unit of statement coverage) and UTB (Time per unit of branch coverage). In particular, these new metrics will express how much time the tester required to cover 1% of the target objective (statement or branch).

Table 4.19 presents the correlation study in which we analyzed the relationship between the considered test effort metrics and a set of testability estimators, including our proposed metric. The columns represent the test effort metrics, as for the rows, the ones in the top part of the table represent the test effort metrics, while the ones in the bottom part represent the considered testability estimators.

The correlations between the collected test effort metrics confirm our first analysis of the data. In particular we can see that:

- All test effort metrics have at least a moderate correlation between them.
- The two coverage metrics and the two original time metrics have a very strong correlation between themselves.
- The combined metrics UTS and UTB have a very strong correlation with the original two time metrics.
- The subjective evaluation shows a higher correlation with the time metrics (both original and combined) compared to the coverage metrics.

Let us now analyze the correlation between the testability estimation metrics and the test effort metrics. We can see that our metric has a moderate to strong correlation with all the considered test effort metrics, showing particularly good results especially with the subjective difficulty of the tests and the

Table 4.19. Correlation between testability estimation metrics and test effort metrics.

Legend: D=Subjectively hard to test, CI=Instruction coverage, CB=Branch coverage, TS=Time for statement coverage, TB=Time for branch coverage, UTS=Time per unit of statement coverage, UTB=Time per unit of branch coverage

	D	CI	CB	TS	TB	UTS	UTB
CI	-0.62						
CB	-0.61	1.00					
TS	0.81	-0.72	-0.71				
TB	0.83	-0.64	-0.64	0.96			
UTS	0.81	-0.72	-0.72	1.00	0.96		
UTB	0.82	-0.69	-0.69	0.96	0.99	0.96	
Our metric	-0.82	0.56	0.53	-0.56	-0.63	-0.56	-0.63
Loc	0.56	-	-	0.49	0.58	0.48	0.57
Rfc	0.70	-	-	0.44	0.56	0.45	0.55
Cbo	0.53	-	-0.39	-	0.41	-	0.41
CboModified	0.66	-0.53	-0.53	0.54	0.53	0.54	0.55
Fan In	-	-	-	-	-	-	-
Fan Out	0.74	-0.47	-0.46	0.50	0.55	0.50	0.57
Wmc	0.63	-0.65	-0.66	0.57	0.55	0.56	0.57

time metrics related to branch coverage. By analyzing the correlation between all considered testability estimators and test effort metrics we can see that:

- Our metric shows the best results when correlated with the subjective difficulty (D), other metrics that show particularly good but inferior results are Fan Out and Rfc.
- The metrics that show the best correlation with instruction coverage is Wmc followed by our metric and CboModified. All three of the mentioned metrics show moderate to strong correlation values.
- Wmc is the metric which shows the best correlation with the reached branch coverage score, followed by CboModified and our metric.
- The metric that offers the best correlation with the time taken for reaching maximum statement coverage is Wmc closely followed by our metric and CboModified which offer similar correlation values.
- Our metric has the best correlation with the time taken for reaching maximum branch coverage, all other metrics, except for Cbo and Fan In, show moderate correlation values.

- The results related to the combined time metrics (UTS and UTB) are really similar to the ones obtained with the original corresponding time metrics. In particular, we can see that there is almost no variation between the time taken for reaching statement coverage and time per unit of statement coverage. On the other hand, we can see that there is a general small improvement in the correlation values when considering the time per unit of branch coverage over the time for branch coverage alone.

- As already seen in experiment 4.3 the Fan In metric is the one that shows the lowest correlation values with respect to the test effort.

We performed a deeper analysis on the methods for which our technique showed the most discrepancies compared to the test effort metrics. In particular, method 26, and to a lesser degree methods 18 and 25, appears to not be the easiest to test based on our testability estimation, however test effort metrics show the opposite. By analyzing the test cases generated by EvoSuite in conjunction with the mutants created by Pit, we found out that this happens due to a particular characteristic of these considered methods. These methods accept as inputs some String parameters that, before being used to perform some calculation, are individually checked with a method from another class to make sure that the contained strings are not empty; in case a parameter contains an empty string, an exception is raised. In the worst cases, EvoSuite instead of creating a set of test cases in which each one of the parameters is the only empty one, created a single test case in which all or a part of them are empty. Using the set of generated test cases when performing mutation analysis with Pit lead to some of the generated mutants to be executed but not killed; in particular, Pit creates a mutant for each of the statements checking the non-emptiness of the string wherein the statement is removed. This leads to some of these mutants to not being killed because the test cases do not cover those situations. This can be seen as a limitation by EvoSuite that focused only on creating a test case that led to an exception, without considering that the same type of exception can be raised by different statements in the method.

From these initial results we can answer RQ1 and conclude that our testability estimation metric is potentially a very good predictor of testability issues, surpassing traditionally used test effort metrics.

4.5 Summary of the findings

The main goal behind the performed experiments was to answer the two research questions (RQ1 and RQ2) posed at the start of this chapter. We answered RQ1 (Do our testability estimates effectively pinpoint software modules that suffer or do not suffer of testability issues?) with all three performed experiments. In the qualitative study (EXP1) we were able to confirm that for the analyzed sample of classes, our technique was able to identify the elements that lead to have (resp. not have) potential difficulties in the testing phase. In the quantitative study on historical data (EXP2) we studied the performance of our technique both as the extent of correlation of our conclusive estimates with the development complexity of the test cases, expressed with 5 static metrics extracted from the test cases, and by looking into how well our estimates can combine with traditional software metrics. Thanks to this study we collected empirical evidence that (i) our metric is more than competitive with these other metrics, (ii) it captures a different testability dimension than the size of the software, and (iii) it can complement traditional software metrics to reason on software testability in synergistic fashion. Finally, the controlled study with developers (EXP3), complements the second experiment (EXP2) by making use of a smaller, but more reliable ground truth, built by observing the performance of a third-party developer while they test the target software. In this last experiment we focused on other test effort metrics that were not considered in the second experiment due to a lack of information, such as the time taken for developing the test cases. The collected empirical evidence highlight that our technique is able to identify real testability issues, as can be seen by the correlation with the subjective evaluation of difficulty expressed by the third-party tester and by the time needed to reach the maximum possible branch coverage.

Based on the results of these experiments, we can affirm that our approach is generally able to capture which software modules contains actual testability issues, even if the conclusiveness of the estimates (RQ2) needs to be taken into consideration. RQ2 (How frequently can we achieve conclusive estimates of testability with our current implementation of the technique?) was answered in EXP1 and EXP2. As we saw in the experiments, the amount of non conclusive estimates is often non negligible and the inspection of the software modules with inconclusive estimations revealed that, as we expected, many parts of code of the subject modules were not hit with any test case from EvoSuite due to its limitations. We remark once again that EvoSuite is a research prototype, though very popular in the community of researchers that work on test generation, and we did not expect it to be perfect. Further work should be done on this aspect (such as using further test generators or even ensembles of test generator), but we also underline the importance of studying the merit of our novel proposal for the cases in which we could indeed achieve conclusive results with the current implementation.

4.6 Threats to validity

The main threats to the internal validity for all of our experiments depend on our current choices about the test generation and mutation analysis tools (EvoSuite and PIT) embraced in our current prototype implementation of the technique. On one hand, our results directly depend on the effectiveness of those tools in sampling the execution space and the fault space of the programs under test, respectively, and thus we might have observed different results if we experienced with different tools. On the other hand, our experiments suffered of several subject methods for which PIT failed to identify sufficient sets of mutants (in EXP2, the methods that belonged to the subsets *Tests* in Table 4.6, but that we excluded from the considered subsets *Subjects*) and EvoSuite failed to provide sufficient test cases (the subject methods that resulted in inconclusive estimates, see Table 4.2 for EXP1, see Table 4.7 for EXP2 and see Table 4.16 for EXP3). We mitigated the possible threats by focusing our analysis only on the methods that could be reasonably handled with PIT, and by explicitly pinpointing the methods for which EvoSuite allows us to compute conclusive results.

As for external validity, for the qualitative study (EXP1) the main concerns are (i) the the potential bias due to the fact that we performed the qualitative study and (ii) the limited number of projects and classes analyzed. The limitations come from the fact that this type of study is time consuming and labor intensive. For the quantitative study on historical data (EXP2) our findings may not generalize to other software projects other than the ones that we considered or to programming languages other than Java. In the future, we aim to replicate our experiments on further projects and implement our technique for additional programming languages. Finally, for the controlled study with developers (EXP3), we admit that the number of considered methods is limited and from one single software project and consequently our findings may not generalize to other software projects. This limitation came from the fact that we wanted the testing to be performed by a third-party developer to exclude any potential bias coming from our side; on the other hand, this meant that we had to outsource the testing job, that, as already mentioned, is costly in terms of time and human resources. In the future, we aim to replicate this experiment on further projects and with more people.

A final threat to validity is related to what our metric is able to capture. In particular, our testability estimation method does not cover all possible testability issues affecting the code and focuses in particular on the difficulty of executing and exposing faults aspect. There are other issues that may affect testability such as the lack of documentation and the skills of the tester.

Chapter 5

Conclusions

Software testing is a key activity of the software life-cycle that requires time and resources to be effective, and that is why being able to estimate the testability of a software is essential to control the cost and effectiveness of the test activities. So far researchers focused on estimating testability either by analyzing the fault-sensitivity of a software or by extrapolating information related to the structure of the code of a software. As we discussed in the state of the art chapter, the former approaches were lately abandoned because their high cost makes them impractical, while the latter approaches are intrinsically limited in that they estimate testability just indirectly, without doing any actual attempt or experiment with concretely testing the target software.

In this research work we introduce a new technique for estimating software testability in which the novelty is to exploit automated test case generation to investigate to which extent a program may or may not suffer of testability issues. In a nutshell, our technique consists in executing a test generator of choice against a program under test, and then automatically analyzing the outcomes of the test generation activity to extract evidences that the generated test cases are fostering effective (or ineffective) testing, due in particular to reasons that can be specifically reconciled with design choices that characterize the current program. We regard to testability issues as design choices that hamper the easiness of achieving effective testing. The higher the amount of the evidences our technique can collect for a given program in favor of the presence or the absence of testability issues in the program, the lower or the higher, respectively, the testability estimate that our technique will be reporting for that program.

We developed a prototype that realizes our technique to estimate the testability of software modules in Java. To automatically generate the test cases needed for our technique we rely on EvoSuite, while mutation-based fault seeding is performed thanks to PIT.

With the goal of validating our technique, we performed three empirical experiments based on the developed prototype. The research questions that the experiments aimed to answer where *if our metric highlight actual testability issues* and *how large is the portion of inconclusive estimates with our current implementation of the technique.*

Based on the results of the experiments, we can say that our approach is generally able to capture which software modules contains actual testability issues. In particular, the first experiment showed that the modules that our approach highlights as having testability issues effectively contain elements that lead to potential difficulties in the testing phase. The second experiment compared our testability metric with static software metrics in the task of predicting the testing effort, by referring to a ground truth derived by measuring the complexity the existing test cases associated to the software modules; the results show that our metric is more than competitive with these other metrics and captures a complementary testability dimension with respect to the static metrics. The third experiment complemented the second experiment by making use of a smaller but more reliable ground truth, built by observing the performance of a third party developer while they test the target software. In this last experiment we focused on other test effort metrics that were not considered in the second experiment due to a lack of information, such as the time taken for developing the test cases. The results show the effectiveness of our technique.

5.1 Future works

Our technique, while effective and promising, still has some open problems. From the experiments reported in Section 4.2 and in Section 4.3 we saw that our technique was not able to provide conclusive testability estimates for a non-negligible part of the considered software modules (respectively classes and methods). As already mentioned in the discussion of the experiments, this is mainly due to the intrinsic limitations of our chosen automatic test generator, EvoSuite. An aspect upon which further research is needed is the possible integration of our technique with other test generators or a specifically improved version of EvoSuite itself. Another approach that can be attempted for increasing the conclusiveness of the results is to find other ways to enrich the API of the analyzed software. This will allow the automatic test generator to have an easier access to additional parts of the software that will lead to an increase to the number of baseline mutants and will consequently increase the number of conclusive estimates.

As for the mutation analysis part of the technique, currently all the mutants for which we have evidence that are executed at least one time are considered as potential faults. We could obtain a more resilient baseline by making use of weak mutation testing. Weak mutation testing analyses the differences of the states between the original and the mutant software module. Instead, strong mutation testing focuses only on the differences on the outputs of the program. Currently available mutation testing tools, like PIT, perform only strong mutation testing, mainly due to the fact that weak mutation testing is costly (since instead of focusing only on the outputs of a software module, all intermediate states should be checked) and the benefit of weak mutation testing are potentially limited. Nevertheless, we think we could obtain more accurate testability estimates if we consider all the mutants that propagate their changes to a state which is distinguishable from the original program to create the baseline of our technique, instead of all the mutants that have been executed at least one time.

Regarding the judging of the testability evidences, in the future we want to explore the possibility of including additional information to corroborate them. In particular, we are currently evaluating the possibility of determining the effort spent by the automatic test generator in reaching a particular mutant. With this knowledge we could add a new variable that could improve the testability estimation abilities of our technique.

On the topic of experiments, we plan on extending them to further open-source projects, possibly written in languages other than Java (by matching suitable automatic test generators and mutation analysis tools) and execute other controlled experiments by involving further developers.

Bibliography

[1] E. J. McCluskey, *Logic design principles with emphasis on testable semi-custom circuits.* Prentice-Hall, Inc., 1986.

[2] R. E. Kalman, P. L. Falb, and M. A. Arbib, *Topics in mathematical system theory*, vol. 33. McGraw-Hill New York, 1969.

[3] H. Fujiwara, *Logic testing and design for testability.* MIT press Cambridge, MA, 1985.

[4] G. Fraser and A. 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*, pp. 416–419, 2011.

[5] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 449–452, 2016.

[6] R. S. Freedman, "Testability of software components," *IEEE transactions on Software Engineering*, vol. 17, no. 6, pp. 553–564, 1991.

[7] V. Garousi, M. Felderer, and F. N. Kılıçaslan, "A survey on software testability," *Information and Software Technology*, vol. 108, pp. 35–64, Apr. 2019.

[8] "IEEE Standard Glossary of Software Engineering Terminology," tech. rep., IEEE, 1990. ISBN: 9780738103914.

[9] ISO, "IEC 12207 Systems and software engineering-software life cycle processes," *International Organization for Standardization: Geneva*, 2008.

[10] Pu-Lin Yeh and Jin-Cherng Lin, "Software testability measurements derived from data flow analysis," in *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, (Florence, Italy), pp. 96–102, IEEE Comput. Soc, 1998.

[11] J. M. Voas and K. W. Miller, "Improving the software development process using testability research," 1991.

[12] A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *IEEE Transactions on Software Engineering*, vol. 22, pp. 97–108, Feb. 1996.

[13] T. Yu, W. Wen, X. Han, and J. H. Hayes, "Predicting testability of concurrent programs," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 168–179, IEEE, 2016.

[14] Y. Le Treon, D. Deveaux, and J.-M. Jézéquel, "Self-testable components: From pragmatic tests to design-for-testability methodology," in *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 (Cat. No. PR00275)*, pp. 96–107, IEEE, 1999.

[15] R. Poston, J. Patel, and J. Dhaliwal, "A software testing assessment to manage project testability," 2012.

[16] K. Karoui and R. Dssouli, "Testability analysis of the communication protocols modeled by relations," *Technical Report No. 1050*, vol. 1050, 1996.

[17] T. W. Williams and K. P. Parker, "Design for testability—A survey," *Proceedings of the IEEE*, vol. 71, no. 1, pp. 98–112, 1983. ISBN: 0018-9219 Publisher: IEEE.

[18] R. V. Binder, "Design for testability in object-oriented systems," *Communications of the ACM*, vol. 37, no. 9, pp. 87–101, 1994. ISBN: 0001-0782 Publisher: ACM New York, NY, USA.

[19] S. Jungmayr, "Design for testability," in *Proceedings of CONQUEST*, vol. 2002, pp. 57–64, 2002.

[20] B. Pettichord, “Design for testability,” in *Pacific Northwest Software Quality Conference*, pp. 1–28, 2002.

[21] T. Kanstren, “A study on design for testability in component-based embedded software,” in *2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pp. 31–38, IEEE, 2008.

[22] B. Baudry, Y. Le Traon, and G. Sunye, “Testability analysis of a UML class diagram,” in *Proceedings Eighth IEEE Symposium on Software Metrics*, (Ottawa, Ont., Canada), pp. 54–63, IEEE Comput. Soc, 2002.

[23] B. Baudry, Y. Traon, G. Sunye, and J.-M. Jezequel, “Measuring and improving design patterns testability,” in *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717)*, (Sydney, NSW, Australia), pp. 50–59, IEEE Comput. Soc, 2003.

[24] B. Baudry, Y. Le Traon, and G. Sunye, “Improving the testability of UML class diagrams,” in *First International Workshop onTestability Assessment, 2004. IWoTA 2004. Proceedings.*, (Rennes, France), pp. 70–80, IEEE, 2004.

[25] B. Baudry and Y. Le Traon, “Measuring design testability of a UML class diagram,” *Information and software technology*, vol. 47, no. 13, pp. 859–879, 2005. Publisher: Elsevier.

[26] S. Mouchawrab, L. C. Briand, and Y. Labiche, “A measurement framework for object-oriented software testability,” *Information and Software Technology*, vol. 47, pp. 979–997, Dec. 2005.

[27] J. Gao and Ming-Chih Shih, “A Component Testability Model for Verification and Measurement,” in *29th Annual International Computer Software and Applications Conference (COMPSAC’05)*, vol. 2, (Edinburgh, Scotland), pp. 211–218, IEEE, 2005.

[28] M. Bruntink and A. van Deursen, “Predicting class testability using object-oriented metrics,” in *Source Code Analysis and Manipulation,*

Fourth IEEE International Workshop on, (Chicago, IL, USA), pp. 136–145, IEEE Comput. Soc, 2004.

[29] V. Gupta, K. Aggarwal, and Y. Singh, "A fuzzy approach for integrated measure of object-oriented software testability," *Journal of Computer Science*, vol. 1, no. 2, pp. 276–282, 2005.

[30] M. Bruntink and A. van Deursen, "An empirical study into class testability," *Journal of Systems and Software*, vol. 79, pp. 1219–1232, Sept. 2006.

[31] Y. Singh, A. Kaur, and R. Malhotra, "Predicting testing effort using artificial neural network," in *Proceedings of the World Congress on Engineering and Computer Science (WCECS 2008) San Francisco, USA. Newswood Limited*, pp. 1012–1017, 2008.

[32] L. Badri, M. Badri, and F. Toure, "Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems," in *Advances in Software Engineering: International Conference, ASEA 2010, Held as Part of the Future Generation Information Technology Conference, FGIT 2010, Jeju Island, Korea, December 13-15, 2010. Proceedings*, pp. 78–92, Springer, 2010.

[33] S. Khalid, S. Zehra, and F. Arif, "Analysis of object oriented complexity and testability using object oriented design metrics," in *Proceedings of the 2010 National Software Engineering Conference*, pp. 1–8, 2010.

[34] Y. Singh and A. Saha, "Predicting testability of eclipse: a case study," *Journal of Software Engineering*, vol. 4, no. 2, pp. 122–136, 2010.

[35] L. Badri, M. Badri, and F. Toure, "An empirical analysis of lack of cohesion metrics for predicting testability of classes," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 2, pp. 69–85, 2011. Publisher: Citeseer.

[36] M. Badri and F. Toure, "Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes," *Journal of Software Engineering and Applications*, vol. 05, no. 07, pp. 513–526, 2012.

[37] Y. Zhou, H. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, and B. Xu, "An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems," *Science china information sciences*, vol. 55, no. 12, pp. 2800–2815, 2012. Publisher: Springer.

[38] R. C. da Cruz and M. Medeiros Eler, "An Empirical Analysis of the Correlation between CK Metrics, Test Coverage and Mutation Score," in *Proceedings of the 19th International Conference on Enterprise Information Systems*, (Porto, Portugal), pp. 341–350, SCITEPRESS - Science and Technology Publications, 2017.

[39] F. Toure, M. Badri, and L. Lamontagne, "Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software," *Innovations in Systems and Software Engineering*, vol. 14, pp. 15–46, Mar. 2018.

[40] V. Terragni, P. Salza, and M. Pezzè, "Measuring Software Testability Modulo Test Quality," in *Proceedings of the 28th International Conference on Program Comprehension*, (Seoul Republic of Korea), pp. 241–251, ACM, July 2020.

[41] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella, "Improving web application testing using testability measures," in *2009 11th IEEE International Symposium on Web Systems Evolution*, pp. 49–58, IEEE, 2009.

[42] R. A. Khan and K. Mustafa, "Metric based testability model for object oriented design (MTMOOD)," *ACM SIGSOFT Software Engineering Notes*, vol. 34, pp. 1–6, Feb. 2009.

[43] A. Kout, F. Toure, and M. Badri, "An empirical analysis of a testability model for object-oriented programs," *ACM SIGSOFT Software Engineering Notes*, vol. 36, pp. 1–5, Aug. 2011.

[44] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994. ISBN: 0098-5589 Publisher: IEEE.

[45] T. Khoshgoftaar, R. Szabo, and J. Voas, “Detecting program modules with low testability,” in *Proceedings of International Conference on Software Maintenance*, (Opio, France), pp. 242–250, IEEE Comput. Soc. Press, 1995.

[46] T. Khoshgoftaar, E. Allen, and Z. Xu, “Predicting testability of program modules using a neural network,” in *Proceedings 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, (Richardson, TX, USA), pp. 57–62, IEEE Comput. Soc, 2000.

[47] J. M. Voas, “PIE: A dynamic failure-based technique,” *IEEE Transactions on software Engineering*, vol. 18, no. 8, p. 717, 1992. Publisher: IEEE Computer Society.

[48] A. Tahir, S. G. MacDonell, and J. Buchan, “Understanding class-level testability through dynamic analysis,” in *2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 1–10, IEEE, 2014.

[49] R. Bache and M. Müllerburg, “Measures of testability as a basis for quality assurance,” *Software Engineering Journal*, vol. 5, no. 2, p. 86, 1990.

[50] M. Badri and F. Toure, “Empirical Analysis for Investigating the Effect of Control Flow Dependencies on Testability of Classes.,” in *SEKE*, pp. 475–480, Citeseer, 2011.

[51] M. Badri and F. Toure, “Evaluating the Effect of Control Flow on the Unit Testing Effort of Classes: An Empirical Analysis,” *Advances in Software Engineering*, vol. 2012, pp. 1–13, June 2012.

[52] S. Jungmayr, “Identifying test-critical dependencies,” in *International Conference on Software Maintenance, 2002. Proceedings.*, (Montreal, Que., Canada), pp. 404–413, IEEE Comput. Soc, 2002.

[53] Y. Le Traon and C. Robach, “Testability analysis of co-designed systems,” in *Proceedings of the Fourth Asian Test Symposium*, (Bangalore, India), pp. 206–212, IEEE Comput. Soc. Press, 1995.

[54] Y. Le Traon and C. Robach, "Testability measurements for data flow designs," in *Proceedings Fourth International Software Metrics Symposium*, (Albuquerque, NM, USA), pp. 91–98, IEEE Comput. Soc, 1997.

[55] Y. Le Traon, F. Ouabdesselam, and C. Robach, "Analyzing testability on data flow designs," in *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, (San Jose, CA, USA), pp. 162–173, IEEE Comput. Soc, 2000.

[56] Thanh Binh Nguyen, M. Delaunay, and C. Robach, "Testability analysis applied to embedded data-flow software," in *Third International Conference on Quality Software, 2003. Proceedings.*, (Dallas, TX, USA), pp. 351–358, IEEE, 2003.

[57] J. Voas, L. Morell, and K. Miller, "Predicting where faults can hide from testing," *IEEE Software*, vol. 8, no. 2, pp. 41–48, 1991. Publisher: IEEE.

[58] J. Voas and K. Miller, "Software testability: the new verification," *IEEE Software*, vol. 12, pp. 17–28, May 1995.

[59] J. M. Voas, "Object-Oriented Software Testability," in *Achieving Quality in Software* (S. Bologna and G. Bucci, eds.), pp. 279–290, Boston, MA: Springer US, 1996.

[60] J.-C. Lin and S.-W. Lin, "An estimated method for software testability measurement," in *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, pp. 116–123, IEEE, 1997.

[61] L. Zhao, "A new approach for software testability analysis," in *Proceedings of the 28th international conference on Software engineering*, (Shanghai China), pp. 985–988, ACM, May 2006.

[62] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978. ISBN: 0018-9162 Publisher: IEEE.

[63] M. Pezzè and M. Young, *Software testing and analysis: process, principles, and techniques.* John Wiley & Sons, 2008.

[64] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proceedings of the 27th international conference on Software engineering*, pp. 402–411, 2005.

[65] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using mutation analysis for assessing and comparing testing coverage criteria,” *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006. ISBN: 0098-5589 Publisher: IEEE.

[66] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 654–665, 2014.

[67] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, “Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 537–548, IEEE, 2018.

[68] R. V. Krejcie and D. W. Morgan, “Determining sample size for research activities,” *Educational and psychological measurement*, vol. 30, no. 3, pp. 607–610, 1970. ISBN: 0013-1644 Publisher: Sage publications Sage CA: Los Angeles, CA.

[69] D. v. Bruggen, F. Tomassetti, R. Howell, M. Langkabel, N. Smith, A. Bosch, M. Skoruppa, C. Maximilien, ThLeu, Panayiotis, S. Kirsch (@skirsch79), Simon, J. Beleites, W. Tibackx, j. p. L, A. Rouél, edefazio, D. Schipper, Mathiponds, W. y. w. t. know, R. Beckett, ptitjes, kotari4u, M. Wyrich, R. Morais, M. Coene, bresai, Implex1v, and B. Haumacher, “javaparser/javaparser: Release javaparser-parent-3.16.1,” May 2020.

[70] G. Fraser and A. Arcuri, “Evolutionary generation of whole test suites,” in *2011 11th International Conference on Quality Software*, pp. 31–40, IEEE, 2011.

[71] G. Fraser and A. Arcuri, “It is not the length that matters, it is how you control it,” in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 150–159, IEEE, 2011.

[72] A. Panichella, F. M. Kifetew, and P. Tonella, “Reformulating branch coverage as a many-objective optimization problem,” in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pp. 1–10, IEEE, 2015.

[73] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2017. ISBN: 0098-5589 Publisher: IEEE.

[74] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, “MuJava: a mutation system for Java,” in *Proceedings of the 28th international conference on Software engineering*, pp. 827–830, 2006.

[75] R. Just, “The Major mutation framework: Efficient and scalable mutation analysis for Java,” in *Proceedings of the 2014 international symposium on software testing and analysis*, pp. 433–436, 2014.

[76] M. Aniche, *Java code metrics calculator (CK)*, 2015. Available in https://github.com/mauricioaniche/ck/.

[77] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, “Methods2Test: A dataset of focal methods mapped to test cases,” *arXiv preprint arXiv:2203.12776*, 2022.