ISLAMIC UNIVERSITY OF TECHNOLOGY

Exploring The Effect of Code Coverage And Maintainability for Identifying Software Testability

By

Md. Fahim Abrar (180042101)

Muntasir Bin Alam (180042132)

Supervisor

Lutfun Nahar Lota

Assistant Professor

Dept. of CSE, IUT

A thesis submitted in partial fulfilment of the requirements for the degree of B.Sc. in Software Engineering

Academic Year: 2021-2022

Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A Subsidiary Organ of the Organization of Islamic Cooperation.

Dhaka, Bangladesh.

May 2023

Declaration of Authorship

We, Md. Fahim Abrar, Muntasir Bin Alam, declare that this thesis titled, 'Exploring The Effect of Code Coverage And Maintainability for Identifying Software Testability' and the work presented in it is our own. We confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Any part of this thesis has not been submitted for any other degree or qualification at this University or any other institution.
- Where we have consulted the published work of others, this is always clearly attributed.

Submitted By:

(Signature of the Candidate)
Md. Fahim Abrar
May 2023

(Signature of the Candidate)
Muntasir Bin Alam
May 2023

Exploring The Effect of Code Coverage And Maintainability for Identifying Software Testability

Approved By:

Lutfun Nahar Lota
Thesis Supervisor,
Assistant Professor,
Department of Computer Science and Engineering,
Islamic University of Technology (IUT), Dhaka, Bangladesh.

Abstract

The ability of code to reveal its flaws, especially during automated testing, is known as software testability. The program being tested must be able to withstand testing. The coverage of the test data provided by a specific test data generation algorithm, on the other hand, is what determines whether a test will be successful. To clarify whether and how software testability affects test coverage. However little empirical evidence has been presented. In this article, we suggest a technique to clarify this issue. The testability of programs is determined using a variety of source code metrics, and our suggested framework builds machine learning models using the coverage of Software Under Test (SUT) provided by various automatically generated test suites.The cost of additional testing is decreased because the resulting models can anticipate the code coverage offered by a particular test data generation algorithm before the algorithm is even run.To measure the testability of source code, a concrete proxy called predicted coverage is used. The correlation between code coverage and maintainability is crucial in assessing the testability of software, as high code coverage combined with well-maintained code facilitates the creation of comprehensive test cases and ensures thorough testing of critical paths and edge cases.

Acknowledgements

We would like to express our heartfelt thanks to Allah Subhanu Wata'ala for giving us the strength to finish this study and being with us when no one else was.

We are grateful to our loved ones for supporting us from the beginning to the end of our research work. We also extend our sincere gratitude to our thesis supervisor Lutfun Nahar Lota for her constant encouragement and support throughout our study.

We would not have been able to complete this research without the guidance and assistance of several individuals who contributed in various ways. We would like to acknowledge and appreciate their valued help.

Contents

Chapter 1

Introduction

Software testability metrics primarily assess how well a software artifact—such as a software com- ponent, system, or set of requirements—supports testing in a test environment [1]. Finding sys- tem flaws (if any) through testing is more comfortable if the software artifact is highly testable. The process of creating faultfree programs is time- consuming and expensive because software test- ing is an irrational problem [2]. Before any testing is done, testability provides a prediction of how much software testing will aid in identifying flaws. However, due to various subjective definitions, measuring testability is difficult.A recent study found that it is unclear how to measure testability and how to deal with relevant problems like quan- tifying [3]. Regardless of the actual conditions and results of software testing, numerous formu- lations and metrics for measuring testability have been proposed [4]–[6]. The majority of them focus on evaluating the testability of implementation (source code) [5]. Despite numerous studies on software testability, we found that the connection between testability and test adequacy standards when automated software testing tools are being used has not been investigated. We can better un- derstand the current challenges and issues with automated testing by measuring and predicting testability. Because testability must be directly related to how simple software testing is, we are compelled to suggest a useful method for calcu- lating and predicting testability using the results of testing.Testing effort is the first step in estab- lishing a connection between testability and soft- ware metrics (TE)In 1972, Edsger W. Dijkstra made the seminal quote: "Program testing can be used to show the presence of bugs, but never to show their absence!"[7]. In order to assess the effective- ness of testing, test adequacy criteria like code coverage have emerged [2].A sufficient test suite satisfies a specified adequacy criterion and offers sufficient test data to guarantee the accuracy of he software under test

(SUT).The effort (budget) necessary to locate and run a test suite that satis- fies a particular adequacy criterion can be defined as testing effort when taking test adequacy into consideration.Hence testability of a component, X, is approximately equaled to the required testing effort, RTE(X): T(X) RTE(X) (1) The required effort, including the test data generation and test execu- tion, is directly related to the percentage of the adequacy criterion, C(X), satisfied by that effort, i.e.:RTE () C(X) (2)The required effort may be un- limited and unquantifiable because there may not be a test suite to satisfy a given criterion on a spe- cific program. For instance, the infinite number of paths or existing infeasible paths may prevent the path coverage criterion from being fully sat- isfied with programs containing loop constructs. Only the portion of the adequacy criterion will be satisfied in these circumstances because the testing budget limits the testing effort in the case of an unsatisfiable adequacy criterion.The transi- tive property forrelations (1) and (2) results:T(X) C(X) (3)The degree to which a test adequacy crite- rion could be satisfied following testing is known as software testability. A crucial component of measuring testability is the relationship between runtime data, such as the per- centage of tests that satisfy the test adequacy criterion, and the static properties of the program, such as source code metrics. On a sizable dataset of experimental data, a machine learning approach is used to achieve this goal. Our approach involves two steps: First, automated testing is used to find the code cov- erage of actual software projects. Second, using a machine learning approach, a set of met- rics specific to each piece of software are mapped to the relevant coverage data.The resulting model is then used to forecast the Software Under Test (SUTtestability)'s prior to the application of tools for automatic test data generation and it aids the developer in understanding accessible code cov- erage.

1.1 Motivation

Achieving high testability and maintainability in software development holds nu- merous motiva- tional aspects that contribute to overall project success. These aspects not only impact the qual- ity and reliability of the software but also lead to increased efficiency and cost-effectiveness.

Testability serves as a powerful motivation by providing developers with the means to thor- oughly assess and validate the functionality of their code. When software is designed with testa- bility in mind, it becomes easier to identify and isolate defects, thereby reducing the time and ef- fort required for debugging. Robust

test suites enable developers to quickly detect issues, stream- line the debugging process, and ensure that the software meets the desired specifications. This motivation leads to enhanced confidence in the software's behavior and empowers developers to make changes without fear of unintended conse- quences.

Similarly, maintainability acts as a driving force throughout the software development life- cycle. By focusing on maintainability, developers ensure that the software remains adaptable and flexible to accommodate future changes and enhancements. When code is well-organized, mod- ular, and follows established coding conventions, it becomes easier to understand, modify, and ex- tend. This promotes efficiency in both bug fixes and feature additions, as developers can quickly locate and address specific areas of the codebase without causing ripple effects. Maintaining clean and maintainable code encourages collaboration among development teams, as multiple develop- ers can work on different components concur- rently, reducing bottlenecks and accelerating de- velopment cycles.

From a financial perspective, the motivational impact of testability and maintainability is signifi- cant. A highly testable software system saves costs by minimizing the risk of software failures and reducing the need for extensive manual testing. Comprehensive test suites not only catch bugs early but also provide a safety net during system upgrades and changes. With efficient maintain- ability practices in place, businesses can respond promptly to market demands and introduce new features or enhancements swiftly, giving them a competitive edge. Furthermore, maintenance costs are reduced as developers spend less time deciphering convoluted code and more time on productive tasks.

In conclusion, the motivational aspects of testability and maintainability in software devel- opment are far-reaching. They promote software quality, increase efficiency, and lead to cost sav- ings. By prioritizing testability, developers can ensure rigorous testing and confident code changes, while maintainability empowers them to adapt and evolve the software with ease. Embracing these aspects fosters a culture of excellence, where teams take pride in delivering high-quality, robust, and adaptable software solutions.

1.2 Problem Statement

Testability and maintainability are critical aspects of software development, yet they often pose significant challenges that hinder the efficient and effective delivery of high-quality software solutions.

Testability Problem: One of the major challenges in software develop- ment is achieving adequate testability. Many software systems are complex and tightly coupled, making it difficult to isolate individual components for testing. Lack of proper design considerations and poor code structure can lead to code that is hard to test, result ing in inadequate test coverage and potentially undetected defects. Insufficient testability leads to increased debugging time, decreased confidence in the software's correctness, and a higher likelihood of releasing faulty software to production.

Maintainability Problem: Maintaining software over its lifecycle is crucial for adapting to changing requirements, fixing bugs, and adding new features. However, maintaining software can be challenging, especially when it lacks proper maintainability practices. Soft ware systems with tangled code, lack of documentation, and a lack of adherence to coding standards become in creasingly difficult to modify, leading to increased effort, time, and costs for maintenance tasks. Moreover, when software is not designed with maintainability in mind, it becomes prone to technical debt, making it harder to evolve and hindering agility in response to market demands.

Existing QA datasets are designed to answer questions over a single paragraph or document as the context, thus failing to test a system's ability to answer complex questions spanning multiple contexts.

Interdependency and Impact: Testability and maintainability are interdependent. Poor testability often goes hand in hand with low maintainability, as code that is hard to test is usually challenging to maintain as well. The lack of clear separation of concerns, excessive coupling, and inadequate documentation hampers the ability to isolate and modify specific functionalities without unintended consequences. The interdependency between testability and maintainability creates a vicious cycle, where difficulties in maintaining the software also impede efforts to improve its testability.

Impact on Software Quality and Cost: The deficiencies in testability and maintainability have a direct impact on software quality and cost. Inadequate testability results in lower test coverage, leaving critical parts of the software untested and increasing the probability of defects in production. Similarly, poor maintainability slows down bug fixes, feature enhancements, and software updates, leading to longer time-to-market and increased development costs. Technical debt accumulated due to poor maintainability further exacerbates the situation, as the effort required to address the debt increases over time.

Chapter 2

Dataset Comparison

Variability in results obtained from different datasets, highlighting the impact of factors such as the size of the software system, its complexity, test coverage, types of tests conducted, and the outcomes of those tests. In this analysis, we have explored available datasets, focusing on parameters like lines of code, number of classes and methods, object-oriented metrics such as inheritance depth and polymorphism, test coverage percentage, and the various types of tests performed, including unit tests, integration tests, and acceptance tests. Additionally, we have considered the results of these tests, encompassing the number of failures and the severity associated with each failure.

However, the issue arises from the observed discrepancies and inconsistencies when comparing results across different datasets. Despite the efforts to collect and analyze various datasets, we face challenges in reconciling the differing outcomes. These variations in results suggest that the testability and maintainability of software systems are influenced by multiple factors, which can have a significant impact on the overall performance and reliability of the software.

One significant factor contributing to divergent outcomes is the size of the software system. The size, measured in terms of lines of code, number of classes, and methods, can vary significantly from one system to another. The complexity of the software system also plays a crucial role. Object- oriented metrics such as inheritance depth and polymorphism provide insights into the intricacy of the codebase. Complex systems may exhibit different behaviors and require specific testing approaches to ensure adequate coverage. Test coverage, another vital factor, represents the percentage of code exercised by the tests. It is essential to have

comprehensive test coverage to identify potential issues and ensure the reliability of the software. However, different datasets may exhibit varying levels of coverage, leading to disparate results. A dataset with higher test coverage may yield more accurate and reliable outcomes compared to datasets with lower coverage. Moreover, the types of tests conducted influence the results obtained. Unit tests, integration tests, and acceptance tests each serve distinct purposes in assessing the functionality and reliability of the software. The choice of test types can impact the testability and maintainability assessment, as different types of tests may reveal differ- ent aspects of the software's behavior. Furthermore, the results of the tests, including the number of failures and their severity, con- tribute to the overall understanding of the software's testability and maintainability. Datasets with a higher number of failures or severe failures indicate potential weaknesses in the software system, requiring closer attention and further analysis. Conversely, datasets with fewer failures may indicate better testability and maintainability. To address these challenges, it is crucial to establish a well-designed and comprehensive dataset that captures a diverse range of software systems, covering different sizes, complexities, test coverage levels, test types, and outcomes. Such a dataset would provide a more accurate representation of real-world scenarios and enable the development of more reliable and effective machine learning models. Additionally, by obtaining useful insights from the data, we can improve our understanding of the factors influencing testability and maintainability, paving the way for enhanced software development practices and more robust systems. In conclusion, the problem statement emphasizes the variability in results obtained from different datasets, highlighting the importance of factors such as the size and complexity of he software system, test coverage, types of tests conducted, and the outcomes of those tests. Overcoming these challenges requires the establishment of a well-designed and comprehensive dataset to develop accurate machine learning models and gain valuable insights into testability and maintainability in software systems.

Chapter 3

Methodology

3.1 Proposed Method

Our proposed framework for measuring testability encompasses a systematic five-step approach. Figure 1 visually illustrates these steps, which involve gathering software repositories, computing code coverage, calculating software metrics, training a machine learning model, and testing the accuracy of the model. To conduct our research, we utilized the SF110 corpus [15], a comprehensive collection of more than 23,000 classes derived from 110 different Java open-source projects hosted on SourceForge, as the primary source of software repositories.

To measure the code coverage of various code segments within the repositories, we employed two well-established tools: EvoSuite [15] and JDART [16]. These tools were utilized to generate and execute test data, thereby enabling us to obtain precise measurements of code coverage. By encompassing both automated generation and execution of test cases, we ensured thorough code coverage analysis.

Additionally, we performed a static analysis of the source code for each project within the repository to compute relevant software metrics. These metrics were carefully selected to capture essential aspects that influence the testability of software systems. By assessing various factors such as the size of the software system in terms of lines of code or number of classes and methods, as well as the complexity measured through object-oriented metrics like inheritance depth or polymorphism, we gained insights into the impact of these metrics on testability.

The computed metrics, alongside the runtime information obtained from code

coverage analysis, were combined to form a comprehensive dataset. This dataset served as the foundation for training a machine learning model, enabling us to establish a meaningful relationship between the identified software metrics as static properties and the code coverage as a measure of testability. By leveraging machine learning techniques, we aimed to uncover patterns and correlations within the data, enabling us to predict the testability of software based on its inherent characteristics.

To validate the effectiveness and accuracy of the trained machine learning model, we conducted rigorous testing on a set of previously unseen data. This testing phase allowed us to assess the model's performance when exposed to new and unfamiliar software instances, ensuring its reliability in real-world scenarios. By comparing the predicted code coverage against the actual coverage achieved, we could ascertain the model's ability to accurately classify testability levels, ranging from very low to very high.

In conclusion, our framework for measuring testability involved a comprehensive and systematic approach, combining the utilization of software repositories, code coverage analysis, software metric computation, machine learning mod- eling, and rigorous testing. By employing these steps, we aimed to develop an effective method- ology for assessing the testability of software systems, ultimately contributing to the creation of robust and reliable software solutions.

3.1.1 Related Work

The use of source code metrics as a means to measure testability has become a common practice in software development[1]. These metrics provide valuable insights into the structural characteristics of the codebase that may impact its testability. However, determining which metrics are most relevant to testability often involves the application of static knowledge and established conventions.

Among the various metrics used to quantify testability, several complexity metrics have gained prominence. Cyclomatic complexity (CC), weighted method per class (WMC), lack of cohesion of a method (LCOM), tight class cohesion (TCC), and loose class cohesion (LCC) are some examples. These metrics aim to capture different di- mensions of complexity within the codebase, and researchers have

explored their relationship with testability. However, despite their widespread usage, the precise connection between these complexity metrics and testability in real-world software remains unclear.

In an attempt to shed light on this relationship, researchers proposed a metrics-based model for object-oriented design testability (MTMOOD) [6]. They conducted a study involving three medium sized projects and manually analyzed the relation- ship between specific design metrics and testability. Linear regression analysis was employed to establish a connection between the metrics and testability[2]. While this manual approach provided some insights, it is prone to errors and its generalization to other projects can be questioned.

To overcome the limitations of manual computation, alternative approaches have been explored. Controllability and observability concepts have emerged as key factors in assessing software testability. The COTT framework [3] offers developers a structured framework for instrumenting object-oriented software to achieve desired levels of controllability and observability. By instrumenting the system under test (SUT) and monitoring its behavior during execution, developers can gather valuable information about its controllability and observability. However, applying this framework to large-scale source code can be challenging and resource-intensive.

Another aspect worth considering is the role of runtime testing in assessing testability. While static metrics provide insights into the structural characteristics of the code, runtime testing allows for the evaluation of how the code behaves in different scenarios. Runtime testing involves executing the SUT and observing its behavior during various test cases. By analyzing the SUT's response to different inputs and stimuli, developers can gain insights into its testability.

In conclusion, the measurement of testability using source code metrics is a well-established practice in software development. Complexity metrics, such as cyclomatic complexity and cohesion metrics, have been widely used to quantify testability. However, the precise relationship between these metrics and testability in real-world software is not yet fully understood. The MT- MOOD model attempted to establish a connection between design metrics and testability, but its manual approach may introduce errors and limit its generalizability. Controllability and observability concepts have also been explored, but applying them to large codebases can be challenging. Additionally, runtime testing provides valuable insights into testability by evaluating the behavior of the code during execution. Further

research and exploration are needed to gain a deeper understanding of the relationship between metrics and testability, and to develop more comprehensive and automated approaches for assessing testability in software systems.

To test their approach, the authors conducted a study in which they collected data on a set of software systems and their corresponding tests, and used this data to build a model for predicting testability. The model was trained using a number of object-oriented measures as features, as well as information about the coverage and quality of the tests. The authors found that their model was able to accurately predict testability and that the combination of measures they used was more effective than any individual measure alone[4]. The authors conclude that incorporating information about test quality into measures of software testability can provide a more accurate and complete picture of the testability of a software system. They suggest that this approach could be useful for developers looking to identify testing challenges and improve the testability of their software Callahan et al., 2000"Automated Software Testing Using ModelChecking" is a paper that discusses the use of model-checking, a technique for automatically verifying the correctness of software systems, for the purpose of testing. The authors argue that model-checking can be an effective approach to automated testing, as it can provide comprehensive coverage of the behavior of a software system and can identify defects that might be missed by traditional testing techniques[5].The authors describe the use of model checking for testing as follows: First, a model of the software system is created, which represents the behavior of the system and the conditions under which it is expected to operate. The model is then checked against a set of properties or requirements that the system is expected to satisfy. If the model-checker is able to find a counterexample to one of the properties, it means that the system does not behave as expected and a defect has been identified. The authors present several case studies in which model-checking was used to test software systems, including a real-time control system and a network protocol. They report that modelchecking was able to identify defects that had not been detected by other testing methods, and that it was able to provide comprehensive coverage of the behavior of the systems. Overall, the authors conclude that model checking is a useful technique for automated testing and can be an effective complement to traditional testing methods. They suggest that it can be particularly useful for testing safety critical or mission-critical systems, where the consequences of defects can be severe.

Dataset Analysis. To conduct a data analysis on maintainability and code coverage, you would need a dataset that includes relevant metrics for both maintainability and code coverage for a set of software projects[6]. Let's assume we have such a dataset containing information on various soft- ware projects. We can proceed with the following steps for the data analysis:

Data Preparation: Collect the necessary dataset that includes metrics for maintainability (e.g., cyclomatic com- plexity, code duplication, coupling) and code cov- erage (e.g., line coverage, branch coverage). En- sure the dataset is clean and organized, removing any irrelevant or missing data points. Exploratory Data Analysis (EDA):

Perform descriptive statistics on the maintainability metrics (mean, median, standard deviation) to gain insights into the overall maintainability of the projects. Calculate summary statistics for code coverage metrics, such as the average code coverage percentage and the distribution of coverage across different components. Visualize the distributions and relationships between maintainability metrics and code coverage metrics using histograms, scatter plots, or box plots. Correlation Analysis: Calculate correlation coefficients (e.g., Pearson correlation) between maintainability metrics and code coverage metrics to measure the strength and direction of the relationship. Identify which maintainability metrics have a significant correlation with code coverage and vice versa. Visualize the correlations using a correlation matrix or heat map to identify strong positive or negative associations between the metrics[7]. Statistical Analysis:

Perform statistical tests, such as t-tests or ANOVA, to determine if there are significant differences in code coverage between projects with different levels of maintainability (e.g., high maintainability vs. low maintainability). Conduct regression analysis to explore the relationship between maintainability metrics and code coverage, considering other potential confounding factors.[1] Machine Learning Modeling (Optional):

Build a predictive model using machine learning algorithms to predict code coverage based on maintainability metrics. Split the dataset into training and testing sets, train the model on the training set, and evaluate its performance on the testing set using appropriate evaluation metrics (e.g., accuracy, precision, recall). Interpretation and Conclusion:

Summarize the findings from the data analysis, highlighting any significant relationships or patterns observed between maintainability and code coverage. Discuss the implications of these findings for software development practices, emphasizing the importance of maintaining code quality and the impact it has on test coverage. Provide recommendations for improving maintainability and code coverage based on the insights gained from the analysis. By following these steps, you can conduct a comprehensive data analysis on maintainability and code coverage, revealing insights into the relationship between these two factors and their impact on software quality.

3.1.2 Literature Review

Zakeri-Nasrabadi and Parsa, 2021Predicting software testability involves using machine learning techniques to build a model that canaccurately predict the testability of a software system based on certain characteristics or features of the system. This can be useful for developers, as it can help them identify potential testing challenges early in the development process and take steps to address them. To build a model for predicting testability, developers will typically first gather a dataset of software systems and their corresponding testability scores. These scores can be determined using object-oriented measures or other techniques for evaluating testability. The developers will then use this dataset to train a machine learning model to predict testability based on the features of the software systems in the dataset. Once trained, the model can be used to predict the testability of new software systems by inputting the relevant features of the system into the model.

Predicting software testability can be a complex task, as it involves understanding the relationships between various characteristics of a software system and its testability. However, with sufficient data and the right machine learning techniques, it is possible to build accurate models that can help developers identify testing challenges and improve the testability of their software. Terragni et al., 2020a"Measuring Software Testability Modulo Test Quality" is a paper that discusses the use of object-oriented measures to evaluate the testability of software systems. The authors argue that traditional measures of software testability, which

focus on the characteristics of the software itself, are insufficient for accurately predicting testability. Instead, they propose using a combination of measures that take into account both the characteristics of the software and the quality of the tests that have been designed for it. The paper describes a study in which the authors collected data on a set of software systems and their corresponding tests, and used this data to build a model for predicting testability. The model was trained using a number of object-oriented measures as features, as well as information about the coverage and quality of the tests. The authors found that their model was able to accurately predict testability and that the combination of measures they used was more effective than any individual measure alone. The authors conclude that incorporating information about test quality into measures of software testability can provide a more accurate and complete picture of the testability of a software system. They suggest that this approach could be useful for developers looking to identify testing challenges and improve the testability of their software. Oluwatosin et al., 2020"Object-Oriented Measures as Testability Indicators: An Empirical Study" is a paper that investigates the use of object-oriented measures as indicators of software testability. The authors conducted a study in which they collected data on a set of software systems and their corresponding test results, and used this data to build a model for predicting testability. The model was trained using a number of objectoriented measures as features, and the authors used the model to evaluate the effectiveness of these measures as indicators of testability. The authors found that several objectoriented measures, including coupling, cohesion, and inheritance depth, were significantly correlated with testability. They also found that using a combination of these measures was more effective at predicting testability than using any single measure alone. The authors conclude that object-oriented measures can be useful indicators of software testability, and that using a combination of these measures can provide a more accurate picture of the testability of a software system. They suggest that this information could be useful for developers looking to improve the testability of their software and identify testing challenges. Fraser and Arcuri, 2014 Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite" is a paper that presents the results of a study on the effectiveness of using EvoSuite, an automated unit test generation tool, to improve the testability of software systems. The authors used EvoSuite to generate unit tests for a large dataset of Java programs, and compared the results to a baseline set of tests that had been manually written by developers. The authors found that EvoSuite was able to generate tests that covered a significant portion of the code in the programs in their dataset, and that the tests generated

by EvoSuite had a higher rate of statement coverage (a measure of the proportion of the code that was exercised by the tests) than the baseline tests. They also found that the tests generated by EvoSuite were more effective at detecting defects in the programs, as they were able to uncover a higher number of defects compared to the baseline tests[8]. The authors also conducted a user study in which they asked developers to evaluate the quality of the tests generated by EvoSuite. The developers reported that the tests generated by EvoSuite were of similar or higher quality compared to the baseline tests. Overall, the authors found that EvoSuite was effective at generating high-quality unit tests that improved the testability of the programs in their dataset. They suggest that automated test generation tools like EvoSuite could be a useful tool for improving the testability of software systems and helping developers to write more effective tests. Terragni et al., 2020 "Measuring Software Testability Modulo Test Quality" is a paper that discusses the use of object-oriented measures to evaluate the testability of software systems. The authors argue that traditional measures of software testability, which focus on the characteristics of the software itself, are insufficient for accurately predicting testability. Instead, they propose using a combination of measures that take into account both the characteristics of the software and the quality of the tests that have been designed for it.[**?**]

Chapter 4

Future Work

Examining the connection between testability and software smells using different source code met- rics opens up several avenues for future work. Here are some potential directions for further in- vestigation:

Correlation Analysis: Conduct a thorough cor- relation analysis between testability metrics (such as code coverage, fault density, or test execution time) and various software smell metrics. Explore whether specific types of smells, such as long methods, duplicated code, or excessive class com- plexity, are strongly correlated with reduced testa- bility. Investigate the strength and direction of these correlations to understand the relationship between smells and testability more precisely.

Smell Detection Tools: Develop or enhance existing tools that can accurately detect software smells. This includes expanding the range of sup- ported smells and improving the accuracy of smell detection algorithms. Utilize these tools to analyze a large corpus of software projects and collect data on the presence and severity of smells. Then, examine how these smells impact testability metrics and provide insights into which smells have the most significant effect on testability.

Predictive Modeling: Explore the possibility of building predictive models to estimate testability based on software smell metrics. Use machine learning techniques to train models on datasets containing both smell metrics and correspond- ing testability metrics. Investigate which combination of smells has the most significant impact on testability, and develop models that can accurately predict

testability based on smell information alone. Validate the models using cross- validation techniques and evaluate their performance on unseen projects.

Testability Metrics for Smells: Develop specialized testability metrics that specifically target software smells. These metrics should capture the impact of different smells on various aspects of testability, such as coverage, maintainability of test code, or ease of writing test cases. By quantifying the testability impact of individual smells, it becomes easier to prioritize and address smells that have the most adverse effect on testability. Empirical Studies: Conduct empirical studies to validate the findings from correlation analyses and predictive models. Design controlled experiments or case studies to investigate the impact of specific smells on testability in real-world software projects. Compare testability metrics before and after smell refactoring to assess the effectiveness of smell removal techniques in improving testability. Collect qualitative feedback from developers regarding the challenges they face in testing code with different smells.

Tool Integration: Integrate testability metrics and software smell detection tools into popular Integrated Development Environments (IDEs) or Continuous Integration (CI) systems. This integration will provide real-time feedback to developers about the testability implications of introduced smells, allowing them to take corrective actions early in the development process. Evaluate the effectiveness of such integrations through user stud- ies and assess the impact on overall code quality and testability.

By pursuing these future directions, researchers and practitioners can gain a deeper understanding of the connection between testability and software smells. This knowledge can inform the development of tools, guidelines, and best practices that enable developers to proactively address smells and enhance testability in software systems.

Bibliography

[1] N. Anwar and S. Kar, "Review paper on various software testing techniques strategies," *Global Journal of Computer Science and Technology*, pp. 43–49, 05 2019.

[2] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, pp. 1–42, 12 2014.

[3] T. Heričko and B. Šumak, "Exploring maintainability index variants for software maintainability measurement in object-oriented systems," *Applied Sciences*, vol. 13, 02 2023.

[4] N. Kasisopha, S. Rongviriyapanish, and P. Meananeatra, "Method evaluation for software testability on object oriented code," 09 2020, pp. 308–313.

[5] V. Terragni, P. Salza, and M. Pezzè, "Measuring software testability modulo test quality," 07 2020, pp. 241–251.

[6] O.-J. Oluwatosin, A. Balogun, S. Basri, A. Akintola, and A. Bajeh, "Object-oriented measures as testability indicators: An empirical study," *Journal of Engineering Science and Technology*, vol. 15, pp. 1092–1108, 04 2020.

[7] M. Zakeri-Nasrabadi and S. Parsa, "Learning to predict software testability," 03 2021, pp. 1–5.

[8] R. Sharma and A. Saha, "A systematic review of software testability measurement techniques," 09 2018, pp. 299–303.