MASTER'S THESIS 2024

Comparative Analysis of Conventional Approaches and AI-Powered Tools for Unit Testing within Web Application Development

Emad Aldeen Issawi, Osama Hajjouz

ISSN 1650-2884
LU-CS-EX: 2024-48

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-48

Comparative Analysis of Conventional Approaches and AI-Powered Tools for Unit Testing within Web Application Development

Jämförande analys av konventionella och AI-drivna testmetoder inom webbapplikationsutveckling

Emad Aldeen Issawi, Osama Hajjouz

Comparative Analysis of Conventional Approaches and AI-Powered Tools for Unit Testing within Web Application Development

Emad Aldeen Issawi
em5771is-s@student.lu.se

Osama Hajjouz
os6283ha-s@student.lu.se

June 30, 2024

Master's thesis work carried out at IKEA Of Sweden AB.

Supervisors: Qunying Song, qunying.song@cs.lth.se
Mohajer Bahareh, bahareh.mohajer@inter.ikea.com

Examiner: Elizabeth Bjarnason , elizabeth.bjarnason@cs.lth.se

Abstract

As software development evolves, integrating Artificial Intelligence (AI) technologies, which simulate human intelligence in machines designed to think and learn like humans, offers a promising avenue to enhance testing efficiency. This thesis investigates the effectiveness of AI-based tools for unit testing in web application development, focusing on IKEA's ILA web application. By comparing AI-based Testing Tools (AITT) with manually implemented (MI) unit tests, the study evaluates the quality, accuracy, efficiency, and associated challenges of each approach. The research employs a literature review and 19 evaluation criteria to assess the quality of tests generated by seven selected AITTs and one MI method. Additionally, the Multi-Criteria Decision-Making method, using the Weighted Sum Model (WSM) score, is used to compare the evaluation results of the two testing approaches. Our findings indicate that while AITT can significantly enhance the speed and coverage of unit testing, their accuracy and quality vary compared to MI tests. Interviews with IKEA engineers provide practical insights into the challenges and benefits of integrating AI tools into existing workflows. The thesis concludes that MI unit tests achieved the highest WSM score, with a close second place by an AITT. Our analysis offers recommendations for effectively utilizing AITT to complement MI testing methods as a hybrid approach, thereby improving overall software development quality and efficiency.

Keywords: unit tests, AI-based testing tools (AITT), manually implemented (MI) tests, evaluation criteria, multi-criteria decision-making (MCDM)

Acknowledgements

We would like to thank our supervisor at IKEA, Bahareh Mohajer, for helping us accomplish this research. Her support facilitated a smooth and efficient process in accomplishing this thesis. We also thank our supervisor at Lund University, Qunying Song, for guiding us through this thesis during its minor and major milestones. Additionally, we are grateful to our examiner, Elizabeth Bjarnason, for her crucial roadmap tips at the beginning of the thesis. Our thanks also go to Filip Gustafsson for initiating this research study. We appreciate IKEA's senior software engineers for the interview on test quality evaluation and all valuable feedback. Lastly, we thank IKEA's tech leader at IKEA, for the interview on the challenges associated with adopting AI-based testing tools.

Contents

Chapter 1
Introduction

This chapter provides an overview of the research, beginning with background information on the research area, including an overview of Artificial Intelligence (AI) and Machine Learning (ML) in software testing. AI refers to the simulation of human intelligence in machines designed to think and learn like humans [32], while ML refers to the study of algorithms and statistical models that enable computers to improve their performance on tasks by learning from data without being explicitly programmed[38]. Additionally, it includes a description of IKEA AB's web application ILA and the motivation for integrating AI. Following this, definitions and terminology used in the thesis are made clear to ensure clarity and understanding for the reader. Subsequently, the goal and the primary problem of the research are described, along with the research questions. Furthermore, the method's steps are provided, along with the achieved results of the thesis. Moreover, the contributions of this study are outlined to underscore its significance and potential impact. Finally, a structured outline of the thesis report is presented to provide a road map for navigating through the subsequent chapters.

1.1 Background

Artificial Intelligence (AI) and Machine Learning (ML) integration into software testing improves efficiency while reducing time and cost [34]. AI-based testing tools are improving processes like test case generation and bug analysis [19] [35]. Initial hypotheses suggest that AI-generated unit tests could enhance code coverage efficiency, though they may not match the quality of manually implemented tests due to developers' in-depth system knowledge.

The integration of AI and ML into software testing automates numerous complex tasks, significantly enhancing the efficiency of the testing process within a shorter time frame [34]. AI and ML technologies are introducing new ways of solving the challenges associated with software quality assurance. By utilizing AI techniques and solutions, AI-based testing tools have

brought an improvement in traditional testing methodologies. These innovations automate a wide range of testing activities, including test case generation, execution, and bug analysis, thereby enhancing the efficiency of the testing process [19] [35]. Moreover, software testing encompasses numerous tasks like analyzing requirements, managing tests, planning tests, generating tests, executing tests, evaluating tests, and reporting. These activities are complicated, require a lot of time, and need experienced engineers to complete them. As a result, there's an increasing focus on employing AI and ML to automate these processes, thereby enhancing the testing efficiency of testing [41].

IKEA AB, the case company of this thesis, is developing a web application called ILA to improve label management and they want to explore AI-based tools for efficient and effective testing of this web application. ILA is designed to streamline the creation and editing of product labels. A key feature of ILA is its robust label creation and editing module. Currently, the challenges the ILA team faces are the limited time and resources to effectively implement unit tests. Although the present testing method involves manual implementation by developers, developers need to concentrate on development rather than testing. Therefore, the motivation behind exploring and integrating AI in software testing of ILA stems from the potential to achieve enhanced testing efficiency as well as reduce costs and resources for the testing process. Before initiating this research, we hypothesized that the application of AI-based testing tools (AITT) could result in more efficient and extensive code coverage, as unit test cases are rapidly generated and may cover numerous testing scenarios due to the use of pre-trained models. Additionally, it was hypothesized that the unit tests generated by AITT would not match the quality of the manually implemented (MI) unit tests since developers possess detailed knowledge of the application's architecture, source code, and edge cases. This knowledge might enable them to write more thorough and accurate tests compared to AI-based tools, which may lack an in-depth understanding of the system's complexities.

1.2 Definitions and Terminology

In this section, we explain our terms and their definitions that is used in this thesis to ensure clarity and understanding for the reader.

The term ***Artificial Intelligence*** **(AI)** was introduced by John McCarthy in 1955 at a conference held by the Dartmouth Conference [32]. It was defined as "programming systems in which the machine is simulating some intelligent human behaviour". John McCarthy described it as "The science and engineering of making intelligent machines, especially intelligent computer programs" [32]. In our thesis, the simulated human behavior involves the creation of unit tests.

AI-based Testing Tools **(AITT)** refer to software testing applications that utilize AI technologies to automate the process of detecting and diagnosing software bugs or errors. These tools leverage AI algorithms and Machine Learning techniques to improve testing efficiency, accuracy, and coverage. AI-based testing tools can automatically generate test cases and allow for more comprehensive, adaptive, and efficient testing compared to traditional manual testing methods.

Manually implemented **(MI)** testing approaches refer to the process where software tests are manually designed and executed by human testers. This traditional method relies on the tester's expertise, and experience to manually navigate through the application, identify potential issues, and evaluate its behavior under various conditions.

Unit testing, also referred to as component or module is defined as a test that checks a small piece of code (a unit), does so quickly, and operates in an isolated manner. This means that the core attributes of unit tests are distinguished from other forms of testing, focusing on the speed of execution, the scope of the code being tested, and the requirement for tests to be performed independently of external influences or dependencies [33]. Unit tests are used to ensure a particular section of code works correctly [28].

1.3 Problem Statement

The goal of this thesis is to investigate whether unit tests generated by AITT can replace or supplement MI tests to enhance efficiency and quality in web application development such as ILA, focusing on three research questions about the performance, effectiveness, and challenges of both testing methods.

The primary problem is **whether the MI unit tests can, either completely or partially, be replaced with unit tests that are generated by AITT**. The objective of this thesis is to analyze the strengths and weaknesses of AITT compared to MI testing methods to enhance testing efficiency. Moreover, this study aims to delve into the challenges that might arise from the adoption of these tools within the context of web application development for example, for IKEA's ILA program. The focus of this thesis is on unit testing within the ILA web application as developers typically must spend a part of their time implementing this task, despite needing to concentrate on development. Therefore, we limit ourselves to the tools that claim to use AI, are available for individual users (although not limited to open-source), are pre-trained (no further training is required), and are applicable for generating unit test cases.

We further divide the problem statement into three research questions:

- **RQ1: How do MI testing methods perform when testing web applications?** Regarding:
  - time to set-up and learn to use the tool?
  - time to prepare and perform the tests?
  - the quality of the testing including code coverage?
- **RQ2: How do existing AITT perform when testing web applications? (same aspects as listed in RQ1)**
- **RQ3: What are the challenges of using AITT in the context of web applications?**

1.4 Research Method

In this thesis, we explore AITT, evaluating the MI unit tests along with those generated by AITT, conducting interviews to improve the evaluation, and using a systematic approach to compare the results.

We start with a literature review to acquire a thorough understanding of unit testing, setting the foundation for the principles of implementing good unit tests and a comprehensive understanding of the Multi-Criteria Decision-Making approach used to compare the AITT with the

MI unit testing. Subsequently, we review 18 AITT and selected 7 of them, providing detailed documentation for each. Following this, we analyze the code of ILA and condense the literature on best practices of unit testing into a list of 18 criteria to use in the evaluation process, with one additional criteria evaluated later during an interview with an IKEA engineer. Later, we evaluate the existing MI unit tests, followed by evaluating the AI-generated unit tests for each selected AITT. Moreover, we interview a tech leader at IKEA to address the challenges of adopting the AITT within IKEA's web development context. Finally, we use the Multi-Criteria Decision-Making approach to compare the AITT and MI evaluation results and draw a final ranking.

1.5 Research Results

This thesis explores 7 effective AITT that generated unit tests for the ILA web application. Additionally, the thesis evaluates the existing MI unit tests along with those generated by AITT based on 18 existing criteria. An interview with an IKEA engineer is conducted to gather an additional important criterion for the evaluation, which is the quality of test cases. Another interview was conducted with IKEA's tech leader to address the challenges of adopting AITT. Thereafter, we use a systematic approach to compare the evaluation results and draw a final ranking, finding that the MI unit testing approach remains the gold standard, although some tools score surprisingly high, nearly matching the MI approach's score. Lastly, we analyze the results and recommend the use of a hybrid approach to save costs and achieve an efficient unit testing strategy. In this hybrid model, AI tools can handle routine, repetitive tests, generating a comprehensive suite of unit tests quickly. Human developers can then review these tests, focusing on complex edge cases and refining the tests as needed. This approach leverages the speed and coverage capabilities of AI while ensuring the accuracy and depth provided by human expertise.

1.6 Contributions

This thesis provides an evaluation of AITT, comparing their strengths and weaknesses with MI testing. It offers recommendations regarding the adoption of AITT to improve testing efficiency, coverage, quality, and cost-effectiveness.

We empirically evaluate several AITT and provide valuable insights to both academia and industry on how AITT can influence testing processes, including testing efficiency, test quality, and potential cost reductions. The thesis compares the strengths and weaknesses of AITT with those of MI testing, covering aspects such as efficiency, coverage, quality of test cases, test accuracy, test structure, and simplicity. Additionally, our thesis provides recommendations for software developers in the ILA team regarding the future adoption of AITT, advising on the selection of tools. These recommendations could potentially enable future developers to concentrate more on their development tasks rather than managing the entire testing process by utilizing AITT, thus minimizing distractions and enhancing focus on development efforts.

1.7 Outline of Thesis

Chapter 2 discusses related work, covering the evolution of unit testing, best practices, automation, and AI's role in software testing. It also addresses challenges in manual testing, AI tool integration, and principles of multiple-criteria decision-making (MCDM). Chapter 3 details the methodology, describing the exploration of the AITT along with their evaluations. The chapter also includes the step for selecting and documenting seven AITT, analyzing IKEA's ILA application code, defining 19 evaluation criteria, and interviewing IKEA engineers regarding test case quality and challenges with the adoption of the AITT adoption. As for Chapter 4, it presents our results of evaluating the MI unit tests and those generated by the selected AITT as well as a comprehensive comparison between them. The chapter also includes insights from two interviews with IKEA engineers and Tech Leaders respectively. Later in Chapter 5 we discuss our findings, comparing the performance of MI unit testing and AITT based on the evaluation criteria. The chapter highlights the strengths and weaknesses of each approach and discusses the results of the comparisons. Moreover, we discuss the practical implications of integrating AITT, and potential challenges developers might face. Lastly, Chapter 6 concludes the thesis with a recommended testing strategy that combines the strengths of both MI and AITT methods and suggests areas for future research to improve the AITT. At the end, References are included to support our research.

Chapter 2

Related Work

In this chapter, we delve into the evolution and practices of unit testing in software development, highlighting advancements in automation, frameworks, and methodologies, while addressing the challenges faced in manual testing and the adoption of these practices across various organizations. We examine good practices for structuring unit tests, the principles underlying effective unit testing, and the use of coverage metrics to assess test quality. Additionally, we emphasize the importance of maintaining accurate and clean tests. The chapter also explores the impact and benefits of automation in software testing, the integration of AI, and the associated challenges. Lastly, the chapter provides a foundation for comparing unit testing tools and methodologies using a multiple-criteria decision-making approach that systemically evaluates complex scenarios.

2.1 Unit Testing

In this section, we examine existing research on the status and advancement of unit testing in software development. The studies reveal a transition towards structural and automated testing, although the widespread adoption of these practices continues to be difficult. The content also considers the educational advantages of testing frameworks and the move towards agile methodologies. Future development includes innovations such as parameterized unit testing and advanced features that improve code coverage and support larger components. Additionally, challenges in manual testing are addressed, underscoring the necessity for a balanced approach to maintain software quality across various organizational sizes.

2.1.1 Unit Testing in Practice

Surveys [15] [44] on unit testing practices reveal that while unit testing is recognized as crucial in software development, challenges persist in automation and practice standardization across organizations.

Runeson conducted a survey [44] among companies to analyze their unit testing practices. The survey aimed to go beyond standard definitions of unit testing, exploring practitioners' actual references and practices. It revealed a consistent view of unit testing's scope, though there was disagreement on whether the test environment should be an isolated harness or part of a partial software system. The survey highlighted that unit testing is predominantly a developer issue, both practically and strategically, with tests typically being structural or white-box based, yet completeness is rarely measured in terms of structural coverage. Most companies need automation in unit testing but faced challenges in spreading good practices across the organization.

Daka and Fraser conducted a survey [15] among 225 software developers from 29 countries to investigate current practices and challenges in unit testing. Their findings underscore the importance of unit testing in software development, highlighting a significant interest among developers in automated unit testing tools to improve testing efficiency. The study identifies future research areas, such as test maintenance, and evaluates the effectiveness of online platforms for conducting software engineering surveys.

2.1.2 Unit Testing Frameworks

Exploration of unit testing frameworks in educational settings reveals significant benefits in program design and development, while evolving testing models in the industry, including the "V model" and test-driven development, emphasize varied testing approaches across software development phases [7] [40].

Noonan and Prosl [40] explored unit testing frameworks through the lens of a student-written skeleton program for a Computer Graphics course. Their investigation highlighted the effectiveness of frameworks like JUnit for enhancing unit testing practices. They observed that incorporating unit testing frameworks could significantly improve program design and development, suggesting that unit testing has broader educational benefits beyond bug detection.

The testing process has undergone significant refinement over the years, with various models proposed for industrial adoption. The "V model" emerged as one of the most popular frameworks, emphasizing testing at different levels of software development, including unit testing. However, more agile approaches, such as test-driven development (TDD), which consists of writing the unit tests before implementing the code, have challenged the traditional phased and documented testing process by advocating for iterative and collaborative testing practices [7].

2.1.3 Unit Testing Automation

Recent advances in unit testing include parameterized unit testing for higher code coverage and systematic testing enhancements like memory leak detection [57] [4]. These methodologies integrate with existing frameworks to improve automated test generation and maintain code quality in the face of frequent changes.

Xie et al. [57] discussed the advances in unit testing through parameterized unit testing (PUT), which extends traditional unit tests by allowing tests to accept parameters. This approach separates the specification of external behaviors from the generation of internal test inputs, facilitating higher code coverage and enabling tools to generate test inputs efficiently. Their work highlights the integration of PUT with various testing frameworks and tools, offering a methodology that enhances developer testing practices by combining strong test oracles with automated test generation.

Artho et al. [4] extended basic unit testing with innovations like low-overhead memory leak detection, verified logging, and integrated coverage measurement. Their approach allowed scaling unit tests for larger software components and utilizing unit test infrastructure for system-wide tests. They found that systematic unit testing maintains code quality amidst frequent developer changes and that their enhancements to unit testing practices could be adapted to other frameworks.

2.1.4 Addressing Challenges in Manually Implemented Testing

Software testing requires a combination of functional methods, which assess the software's operations against its specifications, and structural methods, which examine the internal workings and code structure, along with improved test automation in organizational settings for effective evaluation and adaptation [30] [3].

Manual testing faces several challenges, including test maintenance and the efficient execution of tests. Jorgensen [30] provided a comprehensive exploration of software testing, emphasizing the necessity of both functional and structural testing methods for a thorough evaluation. Jorgensen described the foundational concepts of errors, faults, failures, and incidents to create a structured understanding of software testing. Jorgensen argued that neither functional nor structural testing alone suffices; a judicious combination of both is essential for identifying unimplemented specified behaviors and uncovering implemented but unspecified behaviors, thus achieving a balance between confidence and fault detection.

Runeson et al. conducted a qualitative survey [3] on software test processes within the context of product evolution across 11 companies in Sweden. They found that larger organizations place a significant emphasis on documented processes as a key asset, whereas smaller organizations rely more on experienced individuals. The study also noted a desire among organizations to improve test automation, particularly for supporting evolutionary development, with a common challenge being the management of legacy parts of the product and associated documentation.

2.2 Good Practices of Unit Testing

In this section, we conduct extensive research on the best practices of unit testing. Our thorough analysis of this literature aims to facilitate the evaluation of unit tests for both MI and AITT. By adopting this approach, we ensure a comprehensive and fair comparison, ultimately leading to superior results. This section explores the essential principles and strategies that define good

practices in unit testing, emphasizing their importance in achieving high quality, maintainability, and efficiency. This section is summarized in a list of criteria in Section 3.2 which is later used for the evaluation of the unit tests in this thesis.

2.2.1 How To Structure a Unit Test

A well-structured unit test should apply a pattern known as the AAA pattern. This pattern is used to implement unit tests structurally which includes *arrange*, the *act*, and the *assert* sections [33]. Moreover, a well-structured unit test avoids both multiple cycles of AAA within the unit test and the usage of 'if' statements and tries to minimize the size of each AAA section.

1. **Implement with AAA (Arrange, Act, Assert) Pattern**

   Vladimir [33] structured the unit test into three parts known as the AAA pattern. The AAA pattern streamlines all tests in a suite with a consistent format that, once understood, eases both reading and maintenance. This method reduces the continuous costs associated with test management. The structure is as follows:

   - **Arrange**: sets up the system and dependencies.
   - **Act**: executes the system's methods and records results.
   - **Assert**: checks if the outcomes are as expected, based on return values, system state, or interactions with dependencies.

   Vladimir [33] gave a good example for the AAA pattern where each section of the arrange, act, and assert is separated in that order, see Figure 2.1.

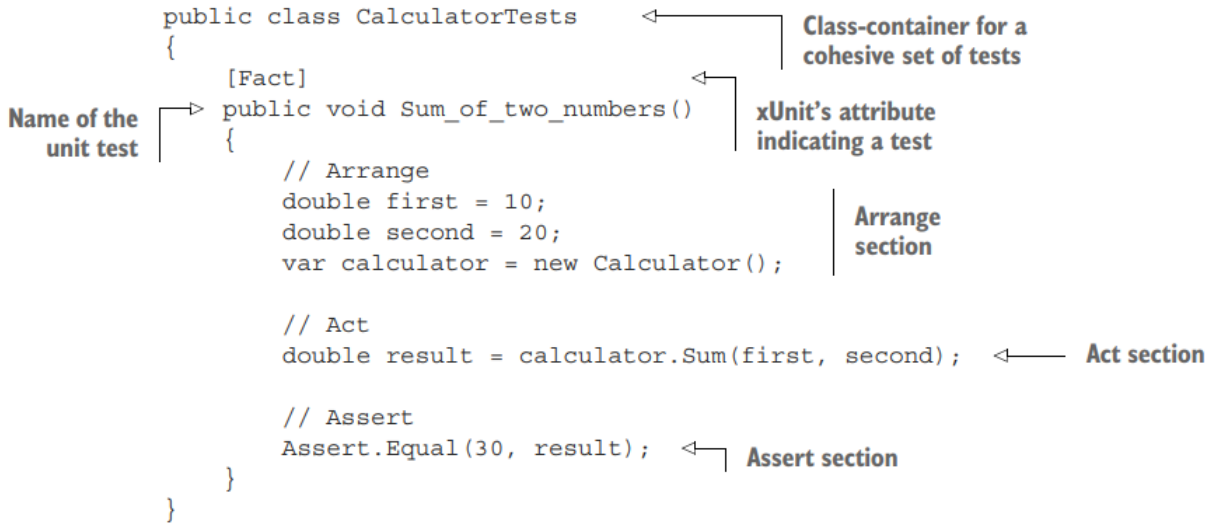

Figure 2.1: An example of AAA, a re-print from Vladimir [33].

2. **Avoid Multiple Cycles of Arrange, Act, and Assert Steps**
   Tests with multiple arrange, act, or assert steps often indicate that they are testing more than one behavior, classifying them as integration tests, not unit tests. However, unit tests should always aim for single, focused actions to maintain speed and clarity [33].

3. **Avoid if Statements in Tests**
   Using 'if' statements in unit tests is not recommended because it complicates the test by introducing multiple pathways. Tests should be straightforward without branching. If one encounters conditional logic in a test, it is usually a sign to split it into multiple, more focused tests [33].

4. **How Large Should Each Section Be?**

   Arrange section often is the largest section, sometimes equaling the act and assert sections combined in size. However, if it grows substantially larger, it is advisable to trim it by moving the setup into private methods within the test class or outsourcing it to a dedicated factory class [33].

   Act section is usually just one line. If it needs more than one line, this might mean there is a problem with how the system or program one is testing is designed for use. [33].

   Assert section has a common guideline suggesting one assertion per test is based on the mistaken belief that tests should focus on the smallest code unit. In reality, a 'unit' refers to a behavior, not the code itself. It is acceptable for a test to have multiple assertions if they are evaluating different outcomes of the same behavior. However, if one finds himself writing a lengthy series of assertions, it may indicate a need for better abstraction in the code. For instance, rather than checking each property of an object individually, implement a proper equality comparison in the object's class to allow for a single, concise assertion against an expected object [33].

2.2.2 The Four Pillars of a Good Unit Test

Four essential principles of unit testing contribute to a robust and efficient testing strategy [33]. These principles ensure that tests can detect and protect against errors, adapt to changes in the code without failure, provide quick feedback to developers, and remain easy to maintain and update.

The four foundational principles of effective unit testing, as articulated by Vladimir, are as follows [33]:

Protection against regressions: A good unit test should safeguard the code against regressions, which are bugs that occur when existing functionality breaks due to new changes or additions to the code base. The primary objective of this pillar is to ensure that the application continues to function as expected as the code evolves. To assess the effectiveness of a test in providing this protection, consider the amount of code executed during the test, the complexity of that code, and the domain significance of the tested functionality.

Resistance to refactoring: Tests should be designed to withstand changes in the code's implementation without failing, provided the external behavior of the code remains unchanged.

This quality ensures that the tests do not become a hindrance to improving the code's structure, readability, or performance. A test's resistance to refactoring is measured by its ability to avoid generating false positives, which occur when a test fails despite the application working correctly from an end-user's perspective.

Fast feedback: Unit tests should execute quickly to provide immediate insights into whether the code changes have introduced any errors. Fast feedback is crucial for maintaining high development velocity, as it allows developers to promptly identify and address issues, thereby reducing the time spent debugging and validating changes.

Maintainability: The tests themselves should be easy to understand and modify. This involves clear naming conventions, a well-organized structure, and minimal coupling between tests and the code under test. Maintainable tests are easier to update alongside the code they test, ensuring that the test suite remains effective and relevant over time.

2.2.3 Using Coverage Metrics to Measure Test Quality

Coverage metrics, including statement, line, and branch coverage, assess the thoroughness of a test suite. While higher coverage percentages can indicate better quality, they are not definitive measures and should be considered alongside other quality indicators to ensure comprehensive testing.

A coverage metric assesses how thoroughly a test suite executes the source code, with values ranging from zero to 100 percent coverage. Various types of coverage metrics, including statement, line, and branch coverage, are commonly used to measure the effectiveness of a test suite [33]. It is often assumed that higher coverage percentages indicate better quality. However, the situation is more nuanced. While coverage metrics provide valuable insights, they are not reliable indicators of the quality of a test suite. They are useful as indicators when coverage is low, for example at 10 percent, suggesting insufficient testing. Nevertheless, achieving 100 percent coverage does not ensure a high-quality test suite. Even with full coverage, a test suite may still be ineffective [33]. According to Vladimir [33], it is dangerous to aim at a specific coverage percentage as optimal coverage since the coverage should not be a positive indicator, but instead, it should be considered as a negative one. However, insufficient coverage is what is below 60% [33].

The primary and widely utilized coverage metric is code coverage [33], also referred to as test coverage or line coverage. This metric indicates the proportion of code lines that have been executed by at least one test, compared to the total number of lines in the code. Therefore, to calculate the code coverage percentage, one would need to divide the lines of executed code by the total number of lines. It helps one understand if any parts of the code are not being executed at all during testing, which could indicate missed test cases. High code coverage suggests a potentially lower chance of undetected software bugs, whereas low code coverage indicates areas of the code that have not been tested and might contain hidden errors [27].

Branch coverage [33] [56], an alternative metric to code coverage, offers more nuanced insights by focusing on control structures like 'if' and 'switch' statements, rather than simply counting lines of code. This metric provides a clearer understanding of how many of these control structures are exercised by at least one test within the suite. To calculate the branch coverage percentage, one would need to divide all the traversed branches by the total number of branches.

Statement coverage is similar to line coverage where it measures the percentage of executable statements in the code that have been executed by the tests. While they seem similar, line coverage and statement coverage can give slightly different results in languages where multiple statements can be on a single line. Line coverage measures if a line is executed at all, while statement coverage ensures each statement is executed [43]. To calculate the statement coverage percentage, one would need to divide the number of executed statements by the total number of statements.

2.2.4 Accuracy

Enhancing test accuracy in software testing involves strategically managing false positives and false negatives, focusing on regression protection to capture actual defects, and refactoring resistance to prevent incorrect failure alerts [33].

Vladimir [33] outlined the concepts of test accuracy in software testing, particularly unit testing, and how it is related to the concepts of true positives, true negatives, false positives (type I error), and false negatives (type II error) as shown in Figure 2.2. Vladimir emphasized two critical attributes of a good test: protection against regressions and resistance to refactoring. Protection against regressions focuses on minimizing false negatives, meaning the test should catch as many real bugs as possible. Resistance to refactoring is about minimizing false positives, ensuring that the test does not fail when the functionality is correct but has been refactored.

To increase test accuracy, two methods are suggested:

1. Increasing the Signal: Improving the test's ability to detect actual bugs (minimizing false negatives).
2. Reducing the Noise: Ensuring the test does not raise unnecessary alarms (minimizing false positives).

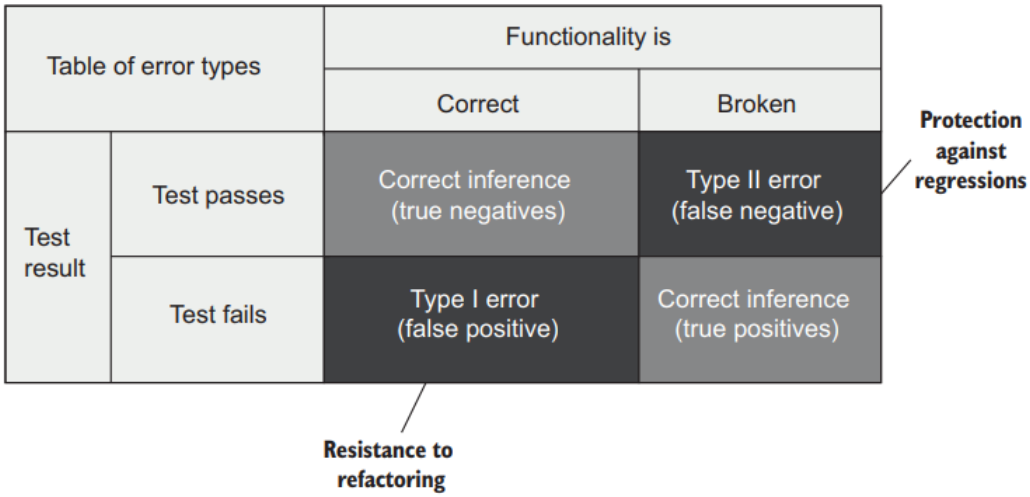

Figure 2.2: A re-print figure that shows the relation between protection against regression to avoid false negatives (type II errors) and resistance to refactoring to reduce false positives (type I errors)[33].

Buffardi et al. [9] composed a conference paper where they discuss measuring unit test accuracy through a dual corpus methodology. This method evaluates the performance of unit tests by applying them to a set of both acceptable and unacceptable solutions. Accuracy is determined based on the unit tests' ability to correctly classify these solutions—approving acceptable solutions and rejecting unacceptable ones. This method not only focuses on fault detection (as with traditional bug identification rates) but also highlights the ability of the unit tests to validate correct implementations accurately. This comprehensive measure provides an enhanced perspective on the effectiveness of unit tests in distinguishing between correct and incorrect code implementations.

Accuracy Formula

The accuracy formula based on the paper [9] can be defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Where:

- TP (True Positives)
- TN (True Negatives)
- FP (False Positives)
- FN (False Negatives)

2.2.5 Unit Testing Principles

To ensure the effectiveness and maintainability of software testing, it is crucial to adopt principles that promote simplicity, specificity, test independence, and reliability, discourage overly protective or redundant assertions, and include both happy and sad scenario testing to maintain comprehensive coverage and quality [8].

Simplicity: Emphasizes the importance of maintaining simplicity in test code to ensure its maintainability. Complexity in tests increases the likelihood of errors and makes maintenance harder. The principle advocates for 'baby steps' in test case development to reduce error chances and simplify maintenance. Metrics such as test code size, number of assertions, and conditional logic can measure compliance [8].

Single Responsibility: Each test should verify a single behavior or condition to simplify diagnosis when a test fails. This principle ensures clarity in what each test aims to prove, allowing for precise root cause analysis. If multiple assertions are necessary, they should all focus on a single behavior observable through a singular class entry point [8].

Avoid Over-protectiveness: This principle warns against the inclusion of redundant assertions that do not contribute to the test's objective, leading to unnecessary complexity and potential maintenance challenges. For example, including asserting conditions unrelated to the test's main goal [8].

A Test Should Fail: It is necessary for tests to be able to fail to prove their value. A test that never fails does not contribute to the software's quality assurance. Including assertions and using a test's failure history can assess a test's ability to detect issues effectively [8].

Reliability: Tests should consistently yield the same results under unchanged conditions, free from randomness or non-determinism. This principle ensures that tests can be relied upon as a stable safety net for development, with randomness and non-determinism identified and eliminated [8].

Happy vs. Sad Tests: This principle encourages testing both expected (happy path) and unexpected (sad path) behaviors to counteract confirmation bias and ensure comprehensive coverage. This balanced approach helps uncover potential faults while verifying expected functionalities, aiming for an equal number of happy and sad test cases [8].

Test (in)dependency / Test Isolation: Asserts that tests should run independently of each other and in any order, without any dependencies that could affect their outcomes. This isolation ensures that the addition of new tests or changes to existing ones does not impact the overall test suite's integrity [8].

2.2.6 Assertions and Clean Tests

In promoting a balanced approach to software testing, the emphasis is placed on maintaining clarity by limiting assertions per test, yet not strictly adhering to a single assertion rule when multiple assertions are necessary for comprehensive evaluation. Additionally, the importance of clean test code is underscored, characterized by the F.I.R.S.T. principles — Fast, Independent, Repeatable, Self-Validating, and Timely — which are essential for effective and reliable unit testing practices [37].

Martin [37] suggested that while a single assertion per test can enhance clarity and focus, adhering too rigidly to this rule might not always be practical. They recognized scenarios where multiple assertions were necessary to fully evaluate a single concept or behavior within a test. Advocating for a balanced approach, Martin emphasized the importance of testing a single concept per test, which means emphasizing the importance of minimizing assertions to maintain test clarity but without enforcing a strict one-assertion rule. This flexible stance allows for more expressive and comprehensive testing, provided that the assertions collectively contribute to understanding the test's aim without introducing unnecessary complexity.

Martin also emphasized the significance of maintaining cleanliness in test codes to ensure they are efficient and effective. According to Martin, clean tests embody five critical attributes, concisely captured by the acronym F.I.R.S.T., which stands for Fast, Independent, Repeatable, Self-Validating, and Timely. These attributes form the cornerstone of well-structured and reliable unit testing practices [37].

Fast: Tests must execute rapidly to facilitate frequent runs, aiding early problem detection and resolution.

Independent: Each test should operate independently of others, allowing for random execution without interdependencies.

Repeatable: Tests should yield consistent results in any environment, ensuring reliability across different testing scenarios.

Self-Validating: The outcome of a test should be binary, either passing or failing, without the

need for manual interpretation of logs or results.

Timely: Tests should be written just before the production code they are meant to validate, ensuring the code is designed with testability in mind. However, this principle does not apply to this thesis, as it involves testing provided software, not developing new code.

2.3 Advancements and Impact of Automation in Software Testing

In this section, we review recent research that highlights the growing importance and benefits of automation in software testing, focusing on its potential to improve efficiency and accuracy while influencing industry practices.

Daka's and Fraser's survey [15] among software developers revealed a strong interest in automated unit testing tools, indicating a shift towards automation to enhance testing efficiency.

Sneha and Gowda [49] explored various software testing techniques and the evolution of software automation testing tools. Their research emphasizes the critical role of automation in modern software testing, illustrating how automation tools not only reduce the manpower required but also minimize errors that manual testers might overlook. The study underscores the importance of adopting automation in testing processes, highlighting its contribution to the success and efficiency of software testing.

Bertolino acknowledged in their article the active research in the automatic generation of test inputs but noted the limited impact on industry practices, where test generation remains largely manual. Bertolino identified promising directions in model-based approaches, random generation techniques, and search-based methods for test data generation. The convergence of these approaches, along with advancements in underlying technologies and computational power, holds the potential to overcome current limitations and achieve more efficient test automation [7].

Pham et al. [41] described how AI/ML is changing the software testing process. Firstly, AI/ML greatly benefits the generation of test cases, making it faster and more efficient compared to manual creation. Techniques like natural language processing and computer vision are utilized to understand the System Under Test (SUT) and automatically generate test cases for various scenarios, including updates in product features. Moreover, AI/ML enhances data-driven testing by generating valid test data for verifying application functionalities. This can be based on project specifications or source code. Techniques include using possible combinations from datasets, search-based data generation, or heuristic approaches. This aspect is in its infancy but is expected to grow with further research. AI aids in automating the management, prioritization, and scheduling of test cases across different devices and environments, addressing the challenge of varying test conditions. It increases test coverage and execution speed while saving significant effort and resources. AI/ML helps address the instability of code-based test automation by dynamically updating test cases to keep up with application changes, known as the self-healing feature. It employs techniques like data analytics, visual hints, and natural language processing to identify and update objects in scripts, reducing the need for manual test script maintenance. Lastly, AI improves quality control by providing insights into what goes wrong and why, identifying impacted test cases due to feature changes, and tracing them to im-

pacted user stories and requirements. This automation helps Quality Assurance (QA) engineers avoid time wasted on false positive error reports [41].

Khaliq et al. [31] described the impact of AI on software testing in multiple ways. Firstly when it comes to the specification, AI facilitates the automation of test case generation from specifications, using techniques like Info-Fuzzy Networks (IFN) for inductive learning from execution data, which helps in recovering missing specifications, designing regression tests, and evaluating software outputs. Similar to what Pham et al. described, Khaliq et al. explained that AI automates the creation of test cases that meet adequacy criteria, significantly impacting this area. Various AI techniques, including genetic algorithms, ant colony optimization, and natural language processing, are applied to generate test cases, improve software coverage, and create efficient testing strategies. Also, AI approaches generate test inputs and data, improving testing coverage and effectiveness. Techniques like genetic algorithms and ML models are utilized to produce complex test data, such as images and sequences of actions for app testing. Zubair added the possibility for AI to aid in solving the test oracle problem by automating the verification of correct behavior in software testing. Techniques include using neural networks and ML algorithms to automatically generate test oracles and predict expected outputs. Another important impact is Test Case Prioritization when AI arranges test execution orders to maximize defect detection early in the testing process. Techniques include reinforcement learning, clustering, and ML models to prioritize based on factors like test history, failure rates, and test case characteristics. Lastly, AI techniques can provide predictions on testing efforts and costs, utilizing ML models to estimate efforts based on historical data and test suite characteristics [31].

Souri et al. [19] employed a Multi-Objective Genetic Algorithm (MOGA) in the context of web applications testing and found that MOGA can minimize test cases while maintaining coverage and reducing costs.

Straub and Huber [50] introduced the Artificial Intelligence Test Case Producer (AITCP), which optimizes human-generated scenarios using AI algorithms. Their findings indicate that AI methods like AITCP can effectively test AI systems, proving particularly efficient for both surface and airborne robots. Moreover, through the authors' analysis it is evident that humans perform slower when testing software compared to AI. Specifically, the AI-based testing technique (AITR) significantly outperforms manual human testing, especially in complex scenarios. The results show that while human performance degrades significantly with the complexity of the testing scenario, the AI maintains a consistent performance level, indicating a substantial efficiency advantage of AI over manual testing in software evaluation.

Yerram et al. [58] discusses how AI-driven software testing leverages machine learning and AI to automatically create test cases by analyzing software specifications and past testing data. This approach enhances code coverage and fault detection, reducing the manual effort required in test case design and allowing testers to focus on higher-level tasks.

Moreover, according to Porter et al. [42], a key application of machine learning and AI in quality assurance is the automation of test generation which replaces the labor-intensive and time-consuming manual creation of test cases. Traditional methods rely on requirements, specifications, and domain expertise, but often miss critical edge cases or scenarios. Machine learning addresses these limitations by automating test case creation.

Shajahan et al. [48] talk about how AI-based testing enhances productivity and allows testers to focus on more critical tasks. It improves test coverage and accuracy by identifying intricate

relationships within software systems, leading to more comprehensive test scenarios and better fault detection.

Serra et al. [47] conducted an empirical study comparing the effectiveness of manual and automatic unit test generation. The study found that while manually written tests tend to be more readable and contextually accurate, automatic unit test generation can significantly reduce the time and effort required for test creation. However, the automatically generated tests often suffer from readability and maintenance issues.

2.4 Challenges of AI in Software Testing

In this section, we discuss some challanges of AI in software testing including the integration of AI in software testing that shows promise in overcoming challenges like test oracle maintenance and data acquisition, but faces obstacles in data sufficiency, model adaptability, and computational demands.

Ghosh et al. [20] discusses in their research several challenges in applying AI to software testing. One major challenge is the need for vast amounts of high-quality training data, which can be difficult to obtain and time-consuming to label. Another challenge is ensuring the AI models can generalize well across different types of software, requiring sophisticated techniques to handle diverse and unpredictable test scenarios. The above findings are also supported by a research by Bajaj and Samal [6] which discussed that domain-specific knowledge is crucial for correct data type recognition in software testing. Generative AI models must be trained on domain-specific datasets to ensure that test cases and bug identifications are relevant, accurate, and meaningful within the particular context of the software system.

Bajaj and Samal [6] also mention in their article ethical considerations. Privacy and security concerns are paramount when dealing with sensitive data, such as source code of a software. AI test-case generation and bug identification must have proper safeguards to protect this data and ensure compliance with privacy regulations.

Khaliq et al. [31] believed that integrating AI in software testing results in a significant positive impact on it. However, they considered several challenges including the Test Oracle challenge when dynamic traits of SUTs make it difficult to maintain the effectiveness of test oracles. AI techniques show promise in addressing this challenge, especially in scenarios where documentation for generating test oracles is lacking.

Another challenge is that acquiring sufficient data for training and testing AI models in software testing is challenging. Unlike in fully automated fields, much testing in software remains manual, making it difficult to capture data. This scarcity of data poses a bottleneck to training effective AI models for software testing. Moreover, AI models heavily rely on the data they are trained on. Ensuring that models can adapt to changes in data distribution over time is crucial for their continued effectiveness. However, detecting the optimal time to readjust models and automating this process poses significant challenges.

Testing AI models in software testing requires a comprehensive understanding and selection of test data. Ensuring that the test data adequately covers the distribution of potential real-world scenarios is challenging and crucial for preventing biased model outcomes.

Lastly, AI techniques in software testing often demand significant computational resources, which can be incompatible with large-scale testing problems. While advancements in computing

infrastructure like Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) have been beneficial, further work is needed to fully exploit their potential while designing techniques that require less computation without sacrificing performance [31].

2.5 Multiple-Criteria Decision-Making

In this section, we delve into the method of Multiple-Criteria Decision-Making (MCDM), which is a method for exploring and optimizing decisions within complex scenarios involving multiple criteria. We discuss the process of assigning weights, calculating WSM scores, and the advantages and disadvantages of using MCDM for evaluating unit testing tools and methodologies.

Multiple-Criteria Decision-Making (MCDM) refers to a method used for evaluating, prioritizing, and selecting from among various alternatives that are characterized by multiple, often conflicting criteria[5]. MCDM methods facilitate the decision-making process by integrating different criteria, allowing for a structured and more systematic analysis of trade-offs to identify the most suitable alternative. This method is widely applied in various fields to address complex decision-making problems where traditional single-criterion approaches are insufficient[5]. Multiple-Criteria Decision-Making (MCDM) is a useful method that aims to optimize decisions within complex scenarios and needs to be evaluated in a systematic manner where multiple criteria are involved [2]. In most MCDM methods, the criteria are typically assigned weights of importance, which are normalized to add up to one. Moreover, an MCDM problem can be expressed in a matrix format [16]. There are multiple methods to solve an MCDM problem, however, Triantaphyllou [16] suggested in their book that the Weighted Sum Model (WSM) is the most commonly used approach.

The rationale for selecting this decision-making method is that MCDM is recognized as one of the most precise approaches to decision-making [54], since this method helps to reduce individual bias and subjectivity, leading to more objective and rational decision-making [36]. Employing this method enables us to effectively determine which unit testing tool or method is superior, considering 19 criteria. One advantage of the MCDM method is that the evaluations become more comprehensive and accurate since multiple options are considered in relation to the importance of each criterion. Additionally, using MCDM organizes the decision-making process making it systematic which leads to consistent evaluations. However, the disadvantages of MCDM include some degree of subjectivity in assigning importance weights to each criterion. Another drawback is that a large number of criteria result in a complex weighted decision matrix leading to a lot of time consumption [36].

The MCDM method comprises three main steps. The first step involves assigning weights of importance to our 19 criteria. The second step involves assigning score values to each criterion for each unit testing alternative. Specifically, it determines the extent to which each AITT and the MI method meet each criterion. The final step involves calculating the WSM score for each unit testing alternative using the formula described in Subsection 2.5.2. The unit testing alternative that achieves the highest WSM score is deemed the most effective method. These three steps of the MCDM method are in detail applied and explained in Section 3.4.

2.5.1 Assigning Weights to the Criteria

Assigning weights of importance to our 19 criteria is an essential step to calculate the WSM score for each alternative, which consists of 7 AITT and 1 MI method. In this regard, the weights need to be assigned effectively and fairly. Triantaphyllou described an approach known as *The Eigenvalue Approach*, which derives weights from pairwise comparison matrices [16]. The initial step involves constructing a 19x19 matrix A where each element a_{ij} represents the result of the pairwise comparison between criteria i and criteria j. The matrix must be reciprocal for the comparisons to be pairwise, meaning that $a_{ij} = \frac{1}{a_{ji}}$ and $a_{ii} = 1$. The second step is to assign values based on how much more important one element is than another. We use Saaty's scale [45] for the assignment of the values, where a value of 1 indicates equal importance, a value of 3 indicates moderately more importance, a value of 5 indicates strongly more importance, a value of 7 indicates very strongly more importance, and lastly a value of 9 indicates extremely more importance of one element over another. Reciprocal values are used for inverse comparisons. Subsequently, the principal right eigenvector (the eigenvector that corresponds to the largest eigenvalue of the matrix) w of matrix A is calculated. This eigenvector provides the raw weights for each criterion. Finally, once the principle eigenvector is obtained, we normalize it so that the sum of its components sums up to one. This normalization provides the relative weight for each criterion.

2.5.2 Formula for Computing the WSM Score

To compute the WSM score to select the best alternative based on multiple criteria the following formula has to be followed [16]:

$$A^*_{\text{WSM-score}} = \max_i \left(\sum_{j=1}^{n} a_{ij} \cdot w_j \right), \quad \text{for } i = 1, 2, 3, ..., m.$$

where:

- $A^*_{\text{WSM-score}}$ is the WSM score of the best alternative,
- m is the number of alternatives (7 AITT + 1 MI method),
- n is the number of decision criteria (19 criteria),
- a_{ij} is the score value of the i-th alternative in terms of the j-th criteria (the calculation method for the score values of each criteria is mentioned in Subsection 3.4.2),
- w_j is the weight of importance of the j-th criteria.

Chapter 3
Method

In this chapter, we describe the methodological steps in this thesis. We start by conducting a literature review of articles discussing unit testing, good practices in unit testing, the impact and challenges of unit testing, and the Multiple-Criteria Decision-Making approach. We search through peer-reviewed articles, books, and studies, utilizing multiple databases including IEEE Xplore and ACM Digital Library. Keywords such as "Unit testing," "AI tools," and "AI in software testing" are employed. Initially, we review the abstracts to assess their relevance to the targeted information. Subsequently, we read the selected articles and studies, documenting the needed information. For books, we focus on relevant sections by using their table of contents.

Then, we review 18 AITT and filter them down to 7 tools capable of generating executable unit tests. We summarize each tool's documentation, covering its functionalities, features, and security aspects. Following this, we analyze the code of ILA to familiarize ourselves with its functions. The next step involves condensing the literature study on best practices of unit testing into a list of 18 criteria. This list serves as the road map for the evaluation. Additionally, another criterion is also taken into consideration which is the quality of the generated test cases. We initialize the evaluation by assessing the existing MI unit tests, followed by evaluating the AI-generated unit tests for each selected AITT. Afterwards, an interview is held with the IKEA engineer to obtain feedback on the quality of the AI-generated test cases, based on the typical considerations IKEA engineers use for assessing test case quality. Thereafter, we compare the evaluation results using the Multi-Criteria Decision-Making approach to rank the AITT along with the MI unit testing. Finally, we interview an IKEA tech leader to discuss the challenges associated with adopting each AITT. In Figure 3.1, we provide an overview figure of the entire method to help readability.

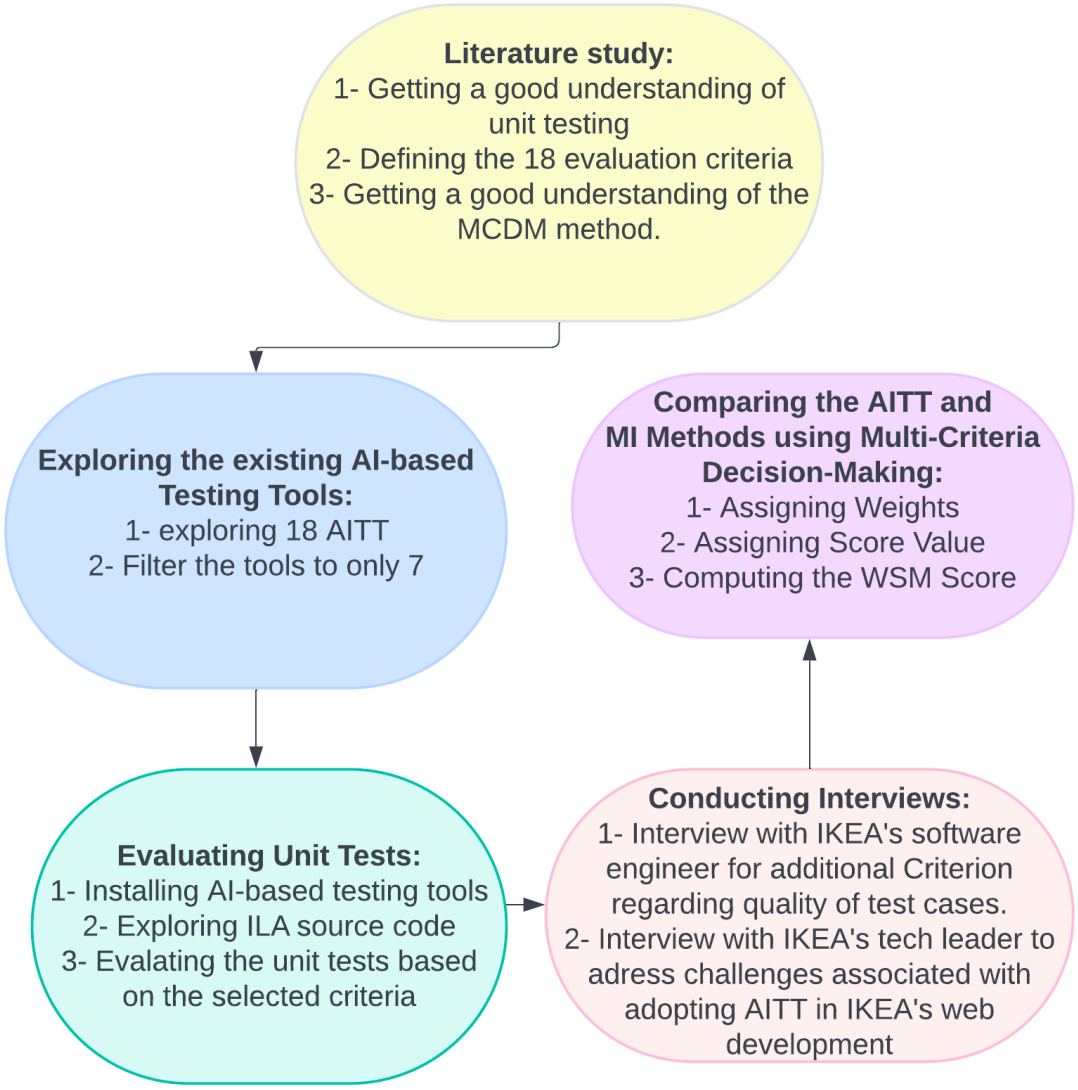

Figure 3.1: Overview figure of the method

3.1 Exploring the existing AITT

Given the vast array of AITT available, many may be non-functional or fail to deliver on their documented promises. This section of the thesis delves into the exploration of several existing AITT with a focus on those capable of generating unit tests. Beginning with a thorough review of their solutions. Following the identification phase, a primary evaluation is conducted on a small piece of code in ILA to filter the tools that can be installed, run, and generate unit tests. This evaluative phase is crucial for identifying the most promising tools for further analysis.

We begin our exploration by examining the existing tool extensions in the Visual Studio Code IDE. We also investigate well-known AI-based tools available online, finding a web page called MSPowerUser to be particularly useful [39]. The tools we explore claim in their documentation the use of AI and are capable of generating unit tests. We test these tools by using them to generate unit tests for a small part of the source code. Of the 18 tools initially considered, only 7 are capable of generating executable unit tests.

List of the 18 tools that are originally considered, the first 7 tools that are written in bold text are the selected ones:

1. **Codiumate - by CodiumAI**
2. **UnitTestAI - by Paterson Anaccius**
3. **TabNine - by Codota**
4. **PaLM API Unit Test Generator - by Zazmic**
5. **GitHub Copilot - by GitHub**
6. **CodePal Unit-Tests Writer - by CodePal**
7. **IKEA GPT - by IKEA**
8. Cody AI - by Sourcegraph
9. AI Smart Coder - by Ainity
10. Unit Test Generator OpenAI - by Tadeu Tupinamba
11. SIM ChatGPT: Unit Test - by pj-sim-chatgpt-dev
12. XUnit Snippets - by Spencer Farley
13. Copilot for Software Testing - by GoCodeo
14. Software Testing AI - by Software Testing AI
15. Symflower - by Symflower
16. Qyrus Test Forge - by Qyrus AI
17. AI Coding - by GalilAI

18. JC Unit Test Generator - by Juan Calero Projects

A documentation study of these 7 promising AITT is conducted which includes their features, functionality, and security. Moving forward, the unit tests generated by these tools undergo further assessment based on the criteria outlined in Section 3.2

3.1.1 GPT Models

This subsection introduces Generative Pre-trained Transformers (GPTs) since some of the explored AITT leverage this technology such as Codiumate, TabNine, GitHub Copilot, and IKEA GPT. GPTs are a series of language models developed by OpenAI that use deep learning techniques to produce human-like text. These models are autoregressive, meaning they predict the next word in a sentence based on the words that came before. This is achieved through the use of Transformer architectures, which rely on a mechanism called self-attention to weigh the influence of different words in the sentence regardless of their position [17].

Evolution of GPT

The initial version of the GPT models was launched with 110 million parameters, establishing the groundwork for utilizing Transformers in language modeling tasks. Subsequently, an enhanced version, GPT-2, was released, featuring 1.5 billion parameters. This version proved more effective, and capable of generating more coherent and contextually appropriate text. However, OpenAI initially restricted its release due to concerns about its potential misuse in creating deceptive content. With an impressive 175 billion parameters, GPT-3 represented a substantial advance in the capacity of machines to produce human-like text. It is capable of performing a broad array of tasks, such as translation, summarization, and question-answering, often requiring minimal to no task-specific training [17]. GPT-3.5 was then introduced as an improved version that addressed some of GPT-3's limitations, enhancing its ability to understand and produce text for particular applications. Finally, GPT-4, the most recent iteration of OpenAI's language models, demonstrates significant progress over its predecessors. GPT-4 is adept at processing both text and images, marking a notable enhancement in generating computer code from visual inputs [46].

3.1.2 Codiumate - by CodiumAI

Codiumate is a plugin developed by CodiumAI, designed to enhance code quality, testing, and review processes within development environments like Visual Studio Code and IntelliJ IDEA. Leveraging advanced AI technologies, including OpenAI's GPT-3.5 and GPT-4, Codiumate stands out for its emphasis on the testing domain, offering features tailored to simplify the creation and management of test cases, code reviews, and overall code integrity [12] [14].

Key Features and Capabilities:

- Automated Test Generation: Codiumate automates the generation of test cases, significantly reducing the time and effort developers spend on testing. It covers a wide range of

scenarios to ensure comprehensive code coverage and helps identify potential issues early in the development cycle.

- Code Explanation and Suggestions: The plugin offers in-depth explanations of code snippets and provides suggestions for improvements. This feature aids developers in understanding complex code and implementing best practices, enhancing code quality and maintainability.

- Customizable Testing Options: Users have the flexibility to customize test names, objectives, and other parameters, making it easier to align tests with specific project requirements and standards.

- Integration with Development Environments: Codiumate seamlessly integrates with popular Integrated Development Environments (IDEs), allowing developers to access its functionalities directly within their coding workflow.

Codiumate builds upon OpenAI's GPT models and incorporates CodiumAI's proprietary algorithmic engine. This combination enables the plugin to understand context, generate relevant test cases, and offer actionable insights for code enhancement. The approach involves distributing and chaining multiple prompts to create a diverse set of tests, efficiently gathering a broad code context, and allowing interactive adjustments to the generated tests[14] [13]

For the paid version, the data including code that is stored in the servers for troubleshooting purposes if needed, will automatically be deleted within 48 hours [18].

3.1.3 UnitTestAI - by Paterson Anaccius

UnitTestAI is a Visual Studio Code extension created by Paterson Anaccius [55], aimed at simplifying the unit test generation process for developers by leveraging AI technology. This plugin supports a high number of programming languages, making it a versatile tool for developers working across various platforms and languages. Once installed, developers can easily generate unit tests for their code by simply selecting a code block and using the 'Generate Tests' option from the context menu. UnitTestAI uses the current file extension to identify the programming language and suggests related test frameworks to choose from, offering flexibility in adhering to developers' preferred testing methodologies [55]. UnitTestAI further emphasizes the plugin's capacity to make unit testing more straightforward and effective [55]. By recommending the breakdown of code into smaller chunks or single functions, this tool can focus on specific parts of the code, ensuring more accurate and helpful test results. UnitTestAI enhances the effectiveness of the generated unit tests, aiding developers in achieving better code coverage and identifying potential issues early in the development cycle.

3.1.4 TabNine - by Codota

TabNine is an AI-based coding assistant designed to enhance the efficiency and productivity of developers by providing real-time code completions, suggestions, and assistance directly within their IDE. It utilizes advanced AI and Machine Learning technologies to understand and predict

developers' needs, offering highly relevant code suggestions that significantly reduce the time spent on coding tasks. One of the key features of TabNine is its ability to generate unit tests for an implemented code by right-clicking on a specific function and then selecting the "generate unit tests" option. TabNine can even generate unit tests based on natural language comments, making it a powerful tool for speeding up the development process [51] [53].

TabNine employs a sophisticated approach by integrating three distinct pre-trained Machine Learning models that collaborate seamlessly. Initially, it leverages a vast dataset comprising over a billion lines of code sourced from open repositories. This primary model offers the flexibility to run either locally or on cloud infrastructure, with a default preference for local execution of code which can enhance the code privacy of the user.

TabNine ensures the privacy of user code by employing temporary processing, where user code is only retained on the server for the duration of computing the desired result and is never persisted. As users code or ask questions in chat, TabNine requests AI assistance from its cluster. Requests include some code from the local IDE workspace as context to provide relevant answers. This context is immediately deleted after generating the tests. TabNine creates an RAG index based on the code in the local workspace of the IDE for each user. The company develops its models based on open-source code with permissive licenses, and no third-party Application Programming Interfaces (APIs) are used [52]. All of this leads to high privacy of the processed code. These privacy conditions apply to the two versions (Tabnine Protected) and (Tabnine + Mistral), but not to the other versions that use GPT models, as they do not guarantee full privacy.

3.1.5 PaLM API Unit Test Generator - by Zazmic

The PaLM API Unit Test Generator is a Visual Studio Code extension that leverages the power of AI to automatically generate unit tests for selected code snippets. Compatible with any programming language, it aims to enhance code quality and reliability by simplifying the test creation process. Users can generate unit tests quickly, in a matter of seconds, without worrying about data security since the extension does not store user data. It requires no additional dependencies. The PaLM API Unit Test Generator utilizes the pre-trained PaLM Language Model. This advanced AI technology analyzes code snippets to generate high-quality unit tests, catering to any programming language and ensuring an efficient test creation process. The tool claims to not store any code in their servers [59].

3.1.6 GitHub Copilot - by GitHub

GitHub Copilot, developed in collaboration between GitHub and OpenAI, serves as an innovative AI-powered tool aimed at enhancing coding efficiency, code testing, and creativity for developers. Utilizing the Codex model, a sophisticated descendant of the GPT-3. It stands out by understanding a multitude of programming languages and frameworks, thus capable of generating code for new methods, completing parts of the code, or generating fully unit test suits [21] [25].

GitHub Copilot operates under a subscription model and integrates seamlessly with the IDE through a dedicated extension, facilitating an intuitive setup for its users. Upon the initiation of a function header, An innovative feature includes the code generation from natural language

comments [24]. Additionally, GitHub Copilot extends its innovation to testing, simplifying the creation of unit tests. Through Copilot Chat or right-clicking on a selected function, it offers the capability to generate test case snippets, drawing from the open code or highlighted snippets within the editor. This feature aids in swiftly crafting tests for specific functions, suggesting potential inputs and expected outputs, and even asserting the correctness of functions based on their context and semantics. Copilot Chat's ability to recommend tests for edge cases and boundary conditions—such as error handling and unexpected inputs—further bolsters code robustness [22].

GitHub Copilot operates predominantly on the user's local machine. It generates code suggestions based on the context of the current file and any related files without sending this sensitive information to external servers. This local processing ensures that private code remains private, safeguarding against unauthorized access or leakage [23].

3.1.7 CodePal Unit-Tests Writer - by CodePal

CodePal is an AI-based coding tool built with Machine Learning algorithms and designed to assist developers by generating code from plain language and generating unit tests, thus enhancing productivity [10] [39].

The tool's comprehensive platform includes a range of features, such as code generation, refactoring, documentation, generating unit tests, and bug detection [29]. Moreover, CodePal's unit test generator stands out by having the ability to generate unit tests based on the provided code on their page [1]. Regarding the information CodePal collects including code, the website's terms of use specify that users with premium plans are not obligated to grant CodePal a license to use, reproduce, distribute, or create derivative works from their code, except for generating the requested unit tests [11].

3.1.8 IKEA GPT - by IKEA

IKEA GPT is an internally developed AI tool designed to assist IKEA employees by leveraging the capabilities of GPT. Utilizing the GPT models provided through OpenAI's service, IKEA GPT is an innovative approach to secure, private, and efficient workplace assistance. Developed as a response to the risks associated with using external AI platforms like ChatGPT, which could potentially lead to data leaks [26].

IKEA GPT offers a wide array of functionalities aimed at enhancing the productivity and creative capabilities of IKEA's workforce including the the ability to generate code and unit tests [26].

Security is a critical aspect of IKEA GPT's implementation. The tool was specifically developed to maintain high data privacy standards, ensuring that no IKEA data could leak outside the organizational boundaries [26].

3.2 Evaluating Unit Tests

In this section, we begin by installing the selected AITT after reading the usage and set-up instructions in each of their documentation. For most AITT, this is done by installing the tools from the Visual Studio Marketplace where each tool has a page that has a clear description of how to set-up and use them.

A documentation study of ILA's source code is conducted by analyzing it to understand the functions that would be tested. The ILA source code used in this thesis is implemented in Typescript and consists of 4 files: Save-labels with 97 lines of code, Sort-labels with 28 lines of code, Update-DesignList-Validity with 83 lines of code, and FilterFunctions with 127 lines of code.

Following a discussion with software engineers at IKEA, a choice is made to proceed with all of ILA's TypeScript files that contain fully implemented label management functions. Specifically, these files include one for saving labels, one for sorting labels, one for validating the label list, and one for filtering labels. The rationale for selecting these files is due to the existing unit tests that were already created by IKEA developers for these four files. This allows for comparisons with AI-generated unit tests, providing a realistic benchmark since it resembles a real-world scenario. Additionally, we condense the criteria described in Section 2.2 to have a list of 18 criteria that is used later in the evaluation process of the unit tests. Later, we send the list to both IKEA's and Lund University's supervisors to get any additional feedback and approval. In addition to the list, one further criterion is introduced which is the quality of the test cases. These criteria are carefully chosen from the existing literature as their combination represents the state-of-the-art of unit testing.

List of criteria:

1. Includes the AAA (Arrange, Act, Assert) pattern
2. Avoids multiple cycles of arrange, act, and assert steps
3. Avoids if statements
4. Contains a one-line act section
5. Provides protection against regressions by minimizing false negative errors
6. Offers resistance to refactoring by minimizing false positive errors
7. Ensures high test accuracy
8. Guarantees fast execution time (no longer than a few minutes) to ensure quick feedback and facilitate a rapid development process.
9. Is easy to maintain with clear naming conventions, and a well-organized structure for the test suite.
10. Achieves sufficient code coverage, exceeding 60%.

11. Achieves sufficient statement coverage, exceeding 60%.
12. Achieves sufficient branch coverage, exceeding 60%.
13. Small test code size, and low number of assertions
14. Adheres to the single responsibility/single concept per test principle
15. Demonstrates reliability
16. Includes both Happy and Sad Tests
17. Ensures test independence / Test Isolation
18. Is self-validating

We generate unit tests using each AITT, according to their usage instructions. Most AITT offer the option to generate unit tests by right-clicking on the functions being tested. However, IKEA GPT and CodePal are exceptions. IKEA GPT uses a natural language method where we input the function to be tested and request unit tests, and the tool subsequently provides the unit test as a result. Additionally, CodePal allows for the generation of unit tests by inputting the tested functions on their website and selecting "generate". By using the Vitest Framework, we are able to see the test outcome, including number of test cases failed, the number of test cases passed, and the percentages for the code coverage, statement coverage, and branch coverage. We assess both the MI unit tests and the AI-generated unit tests using the criteria mentioned above. The results of these evaluations are detailed in Section 4.1 and Section 4.2 respectively. The quality of the test cases is evaluated together with an IKEA engineer through an interview. This step is crucial to intensify the evaluation process. Following this, the challenges associated with adopting each AITT are explored in discussions with an IKEA tech leader in another interview. The findings from these discussions are presented in Section 4.3.

The terms that are used for evaluation in the tables in Chapter 4 are:

- Pass: The criterion is fulfilled (=100% of the test cases meet the criteria).
- Predominantly pass: The criterion is mostly fulfilled (≥75% to <100% of the test cases meet the criteria).
- Partially pass: The criterion is Partially fulfilled (≥25% to <75% of the test cases meet the criteria).
- Fail: The criterion is not fulfilled (<25% of the test cases meet the criteria).

How the criteria are assessed :

- *C1*: We verify whether the test cases include the sections "Arrange", "Act", and "Assert". The evaluation terms above are used for this assessment.
- *C2*: We verify that there are no cycles of the "Arrange", "Act", and "Assert" sections. The evaluation terms above are used for this assessment.

- *C3*: We verify that there are no 'if' statements in the test cases. The evaluation terms above are used for this assessment.
- *C4*: We verify that there is only a one-line "Act" section(s) in the test cases. The evaluation terms above are used for this assessment.
- *C5*: We introduce a fault in the code and check the number of failing test cases out of the total number of test cases. The evaluation terms above are used for this assessment.
- *C6*: We calculate the number of passing test cases when the functionality of the code is correct. The evaluation terms above are used for this assessment.
- *C7*: We follow the formula in Figure 2.2.4 by dividing the sum of the true positive and true negative values by the sum of these values added to the false positive and false negative values. After calculating the percentage, we use the evaluation terms above for the assessment of this criterion.
- *C8*: This criterion passes if the execution time is less than 2 minutes, otherwise it fails. This ensures rapid feedback which does not hinder the developer from a fast continuous development process as described in Subsection 2.2.2.
- *C9*: We control and see if the test cases have clear naming conventions and a well-organized structure for the test suits. The evaluation terms above are used for this assessment.
- *C10*: We get a coverage percentage from the Vitest framework provided as an output in the IDE that we are using (Visual Studio Code). This criterion passes if the coverage is 60% or above, otherwise it fails. This ensures sufficient coverage as described in Subsection 2.2.3.
- *C11*: We get a coverage percentage from the Vitest framework provided as an output in the IDE that we are using (Visual Studio Code). This criterion passes if the coverage is 60% or above, otherwise it fails. This ensures sufficient coverage as described in Subsection 2.2.3.
- *C12*: We get a coverage percentage from the Vitest framework provided as an output in the IDE that we are using (Visual Studio Code). This criterion passes if the coverage is 60% or above, otherwise it fails. This ensures sufficient coverage as described in Subsection 2.2.3.
- *C13*: We control if the test cases have a small test code size (less than 50 lines of code), and a low number of assertions (2 or less). The evaluation terms above are used for this assessment.
- *C14*: We verify that the test cases have a single purpose. The evaluation terms above are used for this assessment.
- *C15*: We control that the output of the test cases remains the same each time the tests are run (without any modification on the source code). The evaluation terms above are used for this assessment.
- *C16*: We verify that there are Happy and Sad Tests included in the test suite. The evaluation terms above are used for this assessment.

- *C17*: We control that the results of the existing test cases do not depend on newly added test cases. In other words, no test is dependent on another test. The evaluation terms above are used for this assessment.

- *C18*: We verify that the test cases provide either a pass or a fail output, where no manual monitoring or interpretation of the test results is needed. The evaluation terms above are used for this assessment.

3.3 Interview with IKEA's Engineers

Subsequently, we organize a face-to-face six-hour-long interview with IKEA's senior software engineer in ILA's team to obtain professional feedback on assessing the quality of the generated unit test cases by AITT. The software engineer with over 6 years of experience in software development, having significantly influenced the implementation of the existing unit tests, is intimately familiar with the key considerations for assessing the quality of the test cases within ILA's team. The assessment of the test cases' quality hinges on three questions:

- How relevant is the test case's aim to the functionality of the tested function?

- How relevant is the test case's assertion to the aim of the test case?

- Are there enough test case variations to cover most of the possible scenarios?

The outcomes of the interview are presented for each AITT evaluation results in Subsection 4.2.1,Subsection 4.2.2, Subsection 4.2.3,Subsection 4.2.4,Subsection 4.2.5,Subsection 4.2.6, and Subsection 4.2.7. The outcome was translated to numbers by firstly using the evaluation method described in Section 3.2, and secondly assigning scores using the method mentioned in Subsection 3.4.2.

Likewise, a one hour long online interview is held with IKEA's tech leader to address the challenges associated with adopting AITT in IKEA's web development, after showing them the evaluation results. A tech leader/senior software engineer with 12 years of experience in the software engineering field is chosen for this interview. They are responsible for decision-making about the tools used in projects and have considerable expertise in incorporating new technologies into IKEA's systems. Correspondingly, several questions are prepared beforehand to steer the discussion outlined as the following:

- Would you trust the unit tests generated using the evaluated AITT? Furthermore, would you undertake any additional actions if you were to use them?

- Could the AITT be used while considering both the AITT privacy policy (including those with a policy of storing data for 48 hours or those that claim not to store any data) and IKEA's policy?

- Are there any other challenges of introducing the selected and evaluated AITT into IKEA's web development context?

The outcomes of the interview are presented in Section 4.3. During the interviews, detailed notes were taken to capture the key points and feedback provided by the engineers. These notes were then reviewed and analyzed. The analysis involved identifying key points relevant to the evaluation of the unit test cases and the associated challenges. These points were organized into categories based on their type of challenge and relevance to the evaluation criteria. This step facilitated an understanding of the engineers' perspectives on the quality of the test cases and the potential challenges of adopting AITT. These categorized points were then used in subsequent comparison steps.

3.4 Using Multi-Criteria Decision-Making to Compare the AITT and MI Methods

In this section we compare the evaluation results of the AITT and MI methods using Multi-Criteria Decision-Making, we first assign weights to the criteria through a pairwise comparison in a 19x19 matrix, calculated with a Python script. Next, we assign score values by converting evaluation terms into numerical scores and averaging these across all files. Finally, we compute the Weighted Sum Model (WSM) scores by summing the products of each criterion's score and its weight for each alternative method to determine the final ranking of the evaluated methods.

3.4.1 Assigning Weights

During this stage, we follow the steps mentioned in Subsection 2.5.1 to assign weights of importance for each criterion. We begin by constructing a 19x19 square matrix, with the rows and columns representing the criteria. For each element in the matrix, we perform a pairwise comparison of the criteria and then assign a value using Saaty's scale discussed in Subsection 2.5.1, where we sit together and jointly start assessing the importance of each criterion compared to other criteria. This is done by doing an oral discussion, examining each criterion and comparing its significance to that of other criteria, based on the importance of the criterion's impact on the final results of unit tests, and then agreeing on a value for each criterion. For instance, when comparing the criterion "includes the AAA pattern" with "ensures high test accuracy," the latter holds greater importance. This is because high accuracy is crucial; without it, a unit test might fail. On the other hand, including the AAA pattern primarily clarifies the unit tests and does not directly affect the outcome. Given that calculating the eigenvectors and eigenvalues for a 19x19 matrix would be exceedingly challenging, a simple Python script is developed to assist us with these calculations. The Python code is presented in Listing 3.1:

```
1 import numpy as np
2
3 # Define the 19x19 matrix
4 matrix =[
5     [1, 3, 3, 5, 1/9, 1/9, 1/9, 1/7, 1/3, 1/7, 1/7, 1/7, 1/3, 1/5, 1/9, 1/7, 1/5, 1/3,
      1/9],
6     [1/3, 1, 1, 1, 1/9, 1/9, 1/9, 1/7, 1, 1/7, 1/7, 1/7, 1, 1/5, 1/9, 1/7, 1/5, 1/3, 1/9],
7     [1/3, 1, 1, 1/3, 1/9, 1/9, 1/9, 1/7, 1/3, 1/7, 1/7, 1/7, 1/3, 1/5, 1/9, 1/7, 1/5, 1/5,
      1/9],
8     [1/5, 1, 3, 1, 1/9, 1/9, 1/9, 1/7, 1, 1/7, 1/7, 1/7, 1, 1, 1/9, 1/7, 1/5, 1/5, 1/9],
9     [9, 9, 9, 9, 1, 1, 1, 3, 5, 3, 3, 3, 5, 7, 1, 3, 5, 5, 1],
```

```
    [9, 9, 9, 9, 1, 1, 1, 3, 5, 3, 3, 3, 5, 7, 1, 3, 5, 5, 1],
    [9, 9, 9, 9, 1, 1, 1, 3, 5, 3, 3, 3, 5, 7, 1, 3, 5, 5, 1],
    [7, 7, 7, 7, 1/3, 1/3, 1/3, 1, 5, 1, 1, 1, 5, 5, 1, 3, 5, 5, 1],
    [3, 1, 3, 1, 1/5, 1/5, 1/5, 1/5, 1, 1/5, 1/5, 1/5, 1, 1, 1/7, 1/5, 1, 3, 1/9],
    [7, 7, 7, 7, 1/3, 1/3, 1/3, 1, 5, 1, 1, 1, 5, 5, 1/3, 1, 5, 7, 1/3],
    [7, 7, 7, 7, 1/3, 1/3, 1/3, 1, 5, 1, 1, 1, 5, 5, 1/3, 1, 5, 7, 1/3],
    [7, 7, 7, 7, 1/3, 1/3, 1/3, 1, 5, 1, 1, 1, 5, 5, 1/3, 1, 5, 7, 1/3],
    [3, 1, 3, 1, 1/5, 1/5, 1/5, 1/5, 1, 1/5, 1/5, 1/5, 1, 1, 1/7, 1/5, 3, 3, 1/9],
    [5, 5, 5, 1, 1/7, 1/7, 1/7, 1/5, 1, 1/5, 1/5, 1/5, 1, 1, 1/7, 1/7, 1, 3, 1/9],
    [9, 9, 9, 9, 1, 1, 1, 1, 7, 3, 3, 3, 7, 7, 1, 5, 7, 7, 1/3],
    [7, 7, 7, 7, 1/3, 1/3, 1/3, 1/3, 5, 1, 1, 1, 5, 7, 1/5, 1, 5, 5, 1],
    [5, 5, 5, 5, 1/5, 1/5, 1/5, 1/5, 1, 1/5, 1/5, 1/5, 1/3, 1, 1/7, 1/5, 1, 1, 1/9],
    [3, 3, 5, 5, 1/5, 1/5, 1/5, 1/5, 1/3, 1/7, 1/7, 1/7, 1/3, 1/3, 1/7, 1/5, 1, 1, 1/9],
    [9, 9, 9, 9, 1, 1, 1, 1, 9, 3, 3, 3, 9, 9, 3, 1, 9, 9, 1]
]


def calculate_weights(matrix):
    # Calculate the eigenvalues and eigenvectors
    eigen_vals, eigen_vecs = np.linalg.eig(matrix)

    # Find the index of the largest eigenvalue
    max_eigen_index = np.argmax(eigen_vals)

    # Get the Principal Eigenvector (the eigenvector corresponding to the largest
    eigenvalue)
    principal_eigenvector = eigen_vecs[:, max_eigen_index]

    # Normalize the weights
    normalized_weights = principal_eigenvector / principal_eigenvector.sum()

    return np.real(normalized_weights)


weights = calculate_weights(matrix)

print("Weights:", )
print(weights)
print("\nSum of weights:")
print(weights.sum())
```

Listing 3.1: Python script for calculating weights

With the use of the above code, we are able to calculate the eigenvalues and eigenvectors of our matrix. Next, we obtain the principal right eigenvector by choosing the eigenvector corresponding to the largest eigenvalue. Lastly, we normalize this eigenvector to obtain the normalized weights for our 19 criteria.

3.4.2 Assigning Score Value

To allocate a total score value for each criterion we follow these steps:

1. Conversion of evaluation terms into scores as follows:

    - Percentage range [75.1% - 100%]: Equivalent score value is 3.
    - Percentage range [50.1% - 75%]: Equivalent score value is 2.
    - Percentage range [25.1% - 50%]: Equivalent score value is 1.
    - Percentage range [0% - 25%]: Equivalent score value is 0.

2. The total score for each criterion in each tool table is calculated by averaging the scores for all files. This is done by adding the scores for each file and then dividing by the number of files (4 in this case). The average score is then recorded in the tables in the Chapter 4 as the total score for the criteria across all files.

3.4.3 Computing the WSM Score

In this step of the Method, we calculate 8 WSM scores for each of our alternatives, consisting of 7 AITT and 1 MI method. We use the formula which is presented in Subsection 2.5.2 where we calculate the sum of the products of each criteria's score and its corresponding weight. The results of the WSM calculations are shown in Subsection 4.4.2.

Chapter 4
Result

In this chapter, we present the results of the evaluations of the existing MI unit tests and generated unit tests for each AITT. Additionally, we include feedback from an interview with an IKEA engineer on assessing the quality of the test cases for the generated unit tests, presented for each AAIT. Subsequently, we show the outcomes of an interview with IKEA's tech leader regarding the challenges associated with adopting each AITT. Finally, we present the results from the Multi-Criteria Decision-Making method, with tables showing the normalized weights and the final WSM scores for each alternative.

4.1 Evaluation of The Existing MI Unit Tests

In this section, we present the evaluation of the existing unit tests for the files (save-labels, sort-label, update-designList-validity, and filterFunction), based on the criteria outlined in Section 3.2. In the Table 4.1 and Table 4.2, each criterion is evaluated with a final average score.

A document study is performed on the existing unit tests to have an idea of the test cases where the existing unit tests were divided across four files, Save-labels has 4 unit tests, Sort-labels has 2 unit tests, Update-DesignList-Validity has 9 unit tests, and lastly, FilterFunctions has 8 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview, the preparation time for the unit tests involves developing three main sections: the arrange section, the act section, and the assert section. Most significantly, it includes devising and planning a variety of test cases that cover all potential scenarios, which generally requires the most time. The duration needed to prepare the unit tests is influenced by two factors. First, the tester's knowledge and expertise in unit testing significantly affect the speed of the process. Second, the tester's familiarity with the source code, as well as the code's cleanliness and clarity, can either expedite the process or pose challenges if the code is not well-maintained and clear. In our

specific case, which involves 4 medium-sized TypeScript files, the engineer from IKEA reported that implementing the unit tests required two workdays (approximately 16 hours). Moreover, the performance of the MI unit tests averaged 11.9 seconds, as shown in Table 4.1 . This indicates that the tests are quick in terms of execution time, which sustains a high development velocity by allowing the developer to rapidly receive feedback and continue coding.

The test cases for the existing MI unit tests align closely with the functionality of the function being tested, incorporating relevant and essential assertions. The variation of test cases is covering all possible scenarios with no redundancy. The success of these test cases often relies on the tester's expertise. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 3.*

In Table 4.1 and Table 4.2 we see that the MI unit tests are implemented using the AAA approach without any repeated AAA cycles or 'if' statements. Each test is designed to fulfill a single responsibility, maintain test independence, and be self-validating. However, some test cases include multiple lines in the act section and multiple responsibilities. For instance, the test case titled "Should return a valid design List array for one selected item when the user selects one value, template, dataset or app Method" could be divided into three separate tests. The first would verify the return of a valid design list array upon selecting a template. The second would do so upon selecting a dataset, and the third upon selecting an app method. By doing so, we could eliminate multiple lines in the act section while maintaining a single responsibility per unit test. Overall, the structure is adequate, and the multiple lines in the act section do not compromise the test results.

Moreover, the Table 4.1 and Table 4.2 show that the MI unit tests are reliable, with no random values used, and test results remain consistent unless the source code is modified. Additionally, The MI unit tests demonstrate a high measurement accuracy in terms of protection against regression and resistance to refactoring. Consequently, they achieve a high accuracy rate. According to IKEA's software engineer, this outcome was anticipated because the tester carefully considers the appropriate test cases to implement, which directly influences accuracy. Moreover, the greater the time and effort invested in developing the test cases, the higher the accuracy percentage will be.

Furthermore, the coverage in the MI unit tests is sufficient for line, code, and branch coverage, exceeding 60%. The percentages achieved suffice to address the most critical test case scenarios. However, the coverage could be improved if the developer concentrated on including negative (sad path) tests where they were missing in some cases. Incorporating both positive (happy path) and negative tests along with a high percentage of coverage can ensure comprehensive testing of the tested function's full functionality.

Lastly, the test code for the MI unit tests has variables with clear names and well-defined linked functions' components. Additionally, the code includes a clear description of the test cases that correspond to the purpose of the functionality. There are no redundant assertions that would degrade the quality of the test cases. However, the size of the tests could be reduced by creating shared test data, rather than initiating it in each test case. The degree of maintainability can vary depending on an individual's expertise. In our case, the maintainability is very good, which positively influences the quality of the tests.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Predominantly pass	Pass	Fail	Fail	1.25
Provides protection against regressions	4/4=100% (Pass)	2/2=100% (Pass)	8/9=88.8% (Predominantly)	7/8= 87.5% (Predominantly)	3
Offers resistance to refactoring	4/4=100% (Pass)	2/2=100% (Pass)	9/9=100% (Pass)	8/8=100% (Pass)	3
Ensures high test accuracy	(4+4)/ (4+4+0+0) =100% (Pass)	(2+2) / (2+2+0+0) =100% (Pass)	(9+8)/ (9+8+1+0)= 94.4% (Predominantly)	(8+7)/ (8+7+1+0)= 93.75% (Predominantly)	3
Guarantees fast execution time	11.34s (Pass)	11.19s (Pass)	12.18s (Pass)	12.81s (Pass)	3
Is easy to maintain	Predominantly pass (due to the arrange line in the assertion section)	Pass	Pass	Pass	2.75

Table 4.1: Evaluation of ILA Unit Tests Across Different Classes for MI unit tests (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	95.75% (Pass)	100% (Pass)	100% (Pass)	75.6% (Pass)	3 (92.8%)
Achieves sufficient code coverage	95.75% (Pass)	100% (Pass)	100% (Pass)	75.6% (Pass)	3 (92.8%)
Achieves sufficient branch coverage	70.37% (Pass)	90% (Pass)	100% (Pass)	91.6% (Pass)	2.75 (87.9%)
Small test code size, and low number of assertions	Partially pass. No redundant assertions: Pass	Good code size: Partially pass. No redundant assertions: Pass	Good code size: Partially pass. No redundant assertions: Pass	Good code size: Partially pass. No redundant assertions: Pass	2
Adheres to the single responsibility	Predominantly pass	Pass	Predominantly pass	Pass	2.5
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Fail (Missing Sad tests)	Partially pass (Partially missing Sad tests)	Partially pass (Partially missing Sad tests)	1.25
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.2: Evaluation of ILA Unit Tests Across Different Classes for MI unit tests (Part 2)

4.2 Evaluation of the Selected AITT

In this section, we start by learning and installing the selected AITT on our devices. The process of setting up the tools and learning about them does not take much time. This is due to the existing documentation on each tool's page, which clearly shows how to set up each tool and how to use each feature. Moreover, this section presents the evaluation of the unit tests generated by each of the 7 selected AITT. The evaluation is conducted based on the criteria outlined in Section 3.2. The outcome of the interview with the IKEA engineer regarding the evaluation of the quality of the test cases is presented for each AITT.

The preparation of the unit tests generally involves merely right-clicking on the code segment to be tested and selecting "generate tests". Some tools, such as Codiumate, offer options to generate a greater or fewer number of test cases, or to choose the type of test cases to generate (edge, happy, or sad test cases). Furthermore, some tools feature AI interaction options like GitHub Copilot Chat, IKEA GPT, and TabNine, which help manage the type of test cases generated as in the previous example. However, on average, preparing the unit tests took about 3 minutes. All unit tests generated by AITT, as well as the MI tests, were executed on the same computer for a fair comparison. The average execution time for the unit tests generated by AITT, as shown in Table 4.3, Table 4.5, Table 4.7, Table 4.9, Table 4.11, Table 4.13, and Table 4.15 is as follows: UnitTestsAI = 7.5s, PaLM = 8.2s, TabNine = 8.3s, Codiumate = 6.1s, GitHub Copilot = 6s, CodePal = 5.3s, IKEA GPT = 7.1s respectively, with an overall average of 6.9s. This demonstrates that the unit tests were performed swiftly, which helps maintain high development velocity since this execution time does not hinder the developer from getting fast feedback. Moreover, this execution time is considered fast compared to the average execution time of 11.9s for the MI unit tests.

4.2.1 UnitTestsAI

Table 4.3 and Table 4.4 display the results of the evaluation of the unit tests generated by UnitTestAI for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 4 unit tests, Sort-labels has 7 unit tests, Update-DesignList-Validity has 5 unit tests, and lastly, FilterFunctions has 7 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that each test case demonstrates relevance between the test case's aim and the functionality of the tested function, as well as relevance between the assertion and the test case's aim. However, the test cases lack variety in the "arrange" section (inputs for the assertions), as they explore only two scenarios: either all inputs are correct or they are not. Nonetheless, these issues should not prevent users from utilizing the tools, as they can generally be fixed manually, ensuring excellent quality of the test cases without requiring substantial time. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 2.*

In Table 4.3 and Table 4.4, we see that UnitTestAI implements the AAA approach, maintains a single responsibility with a one-line act section, is self-validating, ensures test indepen-

dence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with proper structural consideration. Therefore, the user does not need to concern or recheck the structure after the generation process is complete. Moreover, the mentioned tables indicate that UnitTestsAI demonstrates reliability, with no random values used, and test results remain consistent unless the source code is modified. Additionally, the level of protection against regression and resistance to refactoring scored an average of 2.5/3 for both criteria, resulting in a high accuracy level, which in turn indicates their safety for use in real-life projects. This level of test accuracy is sufficient to demonstrate whether the function is correctly implemented in terms of functionality.

Furthermore, according to the measurements in the Table 4.3 and Table 4.4, UnitTestsAI implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by using various types of test cases and covering both happy and sad tests.

Additionally, Table 4.3 and Table 4.4 show that the tool features tests that are easy to maintain and understand in most cases where clear variable names are used with an organized code structure, and the components of linked functions are well-defined. Moreover, there are no redundant assertions, which could decrease the quality of the test cases. Moreover, UnitTestsAI generally has a very appropriate code size where not a lot of shortening is needed.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	3/4 =75% (Predominantely pass)	6/7 =85.7% (Predominantely pass)	3/5 =60% (Partially pass)	7/7=100% (Pass)	2.5
Offers resistance to refactoring	4/4=100% (Pass)	4/7=57.1% (Partially pass)	5/5=100% (Pass)	5/7=71.4% (Partially pass)	2.5
Ensures high test accuracy	(3+4)/ ((3+4+1+0) =87.5% (Predominantely pass)	(4+6)/ (3+6+3+1) = 71.4% (Partially pass)	(5+3)/ (5+3+2+0) = 80% (Predominantly pass)	(5+7) /(5+7+2+0) = 85.7% (Predominantly pass)	3
Guarantees fast execution time	6.61s (Pass)	6.20s (Pass)	7.49s (Pass)	9.70s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Pass	3

Table 4.3: Evaluation of ILA unit tests across different files using UnitTestsAI tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	96.22% (Pass)	94.64% (Pass)	99.44% (Pass)	94.27% (Pass)	3 (96.1425%)
Achieves sufficient code coverage	96.22% (Pass)	94.64% (Pass)	99.44% (Pass)	94.27% (Pass)	3 (96.1425%)
Achieves sufficient branch coverage	68% (Pass)	60% (Pass)	94.44% (Pass)	84.61% (Pass)	2.5 (76.7625%)
Small test code size, and low number of assertions	Predominantly pass. No redundant assertions: Pass	Good code size: Pass. No redundant assertions: Pass	Good code size: Pass. No redundant assertions: Pass	Good code size: Pass. No redundant assertions: Pass	2.875
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Pass	Pass	Pass	3
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.4: Evaluation of ILA unit tests across different files using UnitTestsAI tool (Part 2)

4.2.2 PaLM API

Table 4.5 and Table 4.6 display the results of the evaluation of the unit tests generated by PaLM for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 2 unit tests, Sort-labels has 1 unit test, Update-DesignList-Validity has 5 unit tests, and lastly, FilterFunctions has 6 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that the majority of the test cases have relevant aims and suitable assertions related to the functionality. However, one file includes a few test cases with unclear aims. Meanwhile, all files display a sufficient variety in test cases. Despite these issues, they should not prevent users from utilizing the tools, as generally, these can be manually corrected, ensuring high-quality test cases without demanding significant time. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 2.*

In Table 4.5 and Table 4.6, we observe that PaLM primarily adopts the AAA approach with a one-line act section and generally avoids multiple cycles of AAA, where only a few tests fail to meet this criterion. However, the tool maintains a single responsibility, is self-validating, ensures test independence, and avoids 'if' statements. These results show the tool's effectiveness in generating tests with a great structure. Therefore, the user does not need to be concerned or recheck the structure after the generation process is complete. Moreover, the mentioned tables indicate that the tool demonstrates reliability, using no random values. Additionally, PaLM provides excellent protection against regression with an average score of 3/3 and good resistance to refactoring with an average score of 2/3, resulting in a satisfactory average accuracy score of 2.25/3. This score signifies their sufficient safety for use in real-life projects, where some monitoring might be necessary.

Furthermore, according to the measurements in Table 4.5 and Table 4.6, PaLM implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by using various types of test cases. However, PaLM does not include some sad test cases, which impacts their coverage negatively. Integrating sad test cases could enhance the diversity of the test cases, thereby improving the quality and coverage of the tests. Nonetheless, these are not a fundamental requirement for unit tests. In an ideal scenario, where inputs are always valid (for example, entering a number in a digit-only input box), happy test cases may suffice. The absence of some sad test cases does not make the tools ineffective, but it is a factor to consider. Perhaps, the developers' implementation of some missed sad tests could fulfill the aim of the AITT.

Additionally, Table 4.5 and Table 4.6 reveal that the tool features tests that are easy to maintain and understand in most test cases, with clear variable names, organized code, and well-defined linked functions' components. Furthermore, there are no redundant assertions, which could decrease the quality of the test cases. PaLM generally maintains appropriate code sizes, although they could be further reduced by creating shared data.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Predominantly pass	2.75
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Predominantly pass	2.75
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Predominantly pass	2.75
Provides protection against regressions	2/2=100% (Pass)	1/1 =100% (Pass)	4/5=80% (Predominantly pass)	5/6 =83.3% (Predominantly pass)	3
Offers resistance to refactoring	1/2=50% (Partially pass)	1/1=100% (Pass)	4/5 =80% (Predominantly pass)	3/6 =50% (Partially pass)	2
Ensures high test accuracy	(1+2)/(1+2+1+0) = 75% (Predominantly pass)	(1+1)/(1+1+0+0) = 100% (Pass)	(4+4)/(4+4+1+1) = 80% (Predominantly pass)	(3+5)/(3+5+3+1) = 50% (Partially pass)	2.25
Guarantees fast execution time	12.65s (Pass)	5.89s (Pass)	6.33s (Pass)	8.03s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Predominantly pass	2.75

Table 4.5: Evaluation of ILA unit tests across different files using PAL tool(Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	83.49% (Pass)	98.27% (Pass)	94.66% (Pass)	74.88% (Pass)	2.75 (87.825%)
Achieves sufficient code coverage	83.49% (Pass)	98.27% (Pass)	94.66% (Pass)	74.88% (Pass)	2.75 (87.825%)
Achieves sufficient branch coverage	57.14% (Fail)	50% (Fail)	81.25% (Pass)	87.5% (Pass)	2.25 (68.9725%)
Small test code size, and low number of assertions	Pass. No redundant assertions: Pass	Good code size: Pass. No redundant assertions: Pass	Good code size: Predominantly pass. No redundant assertions: Pass	Good code size: Predominantly pass. No redundant assertions: Pass	2.25
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Partially pass (Missing Sad tests)	Partially pass (Missing Sad tests)	Pass	Partially pass (Missing Sad tests)	1.5
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.6: Evaluation of ILA unit tests across different files using PAL tool (Part 2)

4.2.3 TabNine

Table 4.7 and Table 4.8 display the results of the evaluation of the unit tests generated by TabNine for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 6 unit tests, Sort-labels has 3 unit tests, Update-DesignList-Validity has 4 unit tests, and lastly, FilterFunctions has 6 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that the unit tests generated for TabNine include a significant number of irrelevant test cases. Consequently, the assertions in these test cases are also irrelevant to their objectives. Some test cases fail to implement the arrange section with the correct types of values (such as template names and template categories' names), which results in some failed test cases. Additionally, one file lacks test cases for certain functions that should have been tested. The unit tests also do not offer sufficient variation in test scenarios. These numerous issues cannot be resolved quickly manually, making the test cases unreliable. The quality of test cases should not be overlooked; higher-quality tests more effectively fulfill the goals of unit testing. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 0.*

In Table 4.7 and Table 4.8, we observe that TabNine does not successfully implement the AAA approach in two files. This failure arises because the tool merges the act and assert sections, which draws away from the clear structure of the unit test. Nevertheless, the tool maintains a single responsibility with a one-line act section, is self-validating, ensures test independence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with good structural consideration but with a partial lack of implementation of the AAA pattern. Therefore, the user must recheck the AAA pattern after the generation process is complete. Moreover, the mentioned tables indicate that TabNine demonstrates reliability, with no random values used. The level of resistance to refactoring is very high, however, the protection against regression is very low, with an average score of 1.25/3. This score indicates the tool's unreliability for use in real-life projects. Nevertheless, minor manual adjustments to the generated tests—such as correcting the data in the arrange section or modifying the assert section—could enhance these protection and resistance rates, leading to improved accuracy to a sufficient level. Overall, the accuracy evaluation of the tool suggests that it might be enough in certain situations—for instance, if a developer lacks the time to implement unit tests, or the tests are not a top priority, yet the developer seeks to roughly gauge the quality of the source code. In such cases, this level of test accuracy would be improved with some modifications to the tests.

Furthermore, according to the data in Table 4.7 and Table 4.8, TabNine implements unit tests with sufficient coverage in terms of lines, code, and branches, exceeding 60%, by generating various types of test cases. Regarding the coverage and diversity of test cases for TabNine, it is the same analysis applied to the PaLM's coverage and test case diversity analysis. On the other hand, one file had missing test cases for some functions that should have been tested.

Additionally, Table 4.7 and Table 4.8 show that TabNine features tests that are easy to maintain and understand in most cases, with clear variable names, organized code, and well-defined components of linked functions. Moreover, there are no redundant assertions, which could de-

crease the quality of the test cases. Lastly, TabNine generally has good code sizes, although they could be further reduced by creating shared data.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Fail	Fail	Pass	1.5
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	2/6=33.3% (Partially pass)	1/3 =33.3% (Partially pass)	0/4=0% (Fail)	6/6 =100% (Pass)	1.25
Offers resistance to refactoring	6/6=100% (Pass)	3/3=100% (Pass)	4/4 =100% (Pass)	6/6=100% (Pass)	3
Ensures high test accuracy	(6+2)/ (6+2+4+0) = 66.6% (Partially pass)	(3+1)/ (3+1+2+0) = 66.6% (Partially pass)	(4+0)/ (4+0+4+0) = 50% (Partially pass)	(6+6)/ (6+6+0+0) = 100% (Pass)	2
Guarantees fast execution time	9.76s (Pass)	7.17s (Pass)	8.57s (Pass)	7.75s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Pass	3

Table 4.7: Evaluation of ILA unit tests across different files using TabNine tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	98.42% (Pass)	100% (Pass)	77.33% (Pass)	100% (Pass)	3 (93.9375%)
Achieves sufficient code coverage	98.42% (Pass)	100% (Pass)	77.33% (Pass)	100% (Pass)	3 (93.9375%)
Achieves sufficient branch coverage	71.42% (Pass)	90% (Pass)	100% (Pass)	100% (Pass)	2.75 (90.355%)
Small test code size, and low number of assertions	Predominantly pass. No redundant assertions: Pass	Code size: Pass. No redundant assertions: Pass	Code size: Predominantly pass. No redundant assertions: Pass	Code size: Pass. No redundant assertions: Pass	2.75
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Partially pass (Partially Missing Sad tests)	Pass	Partially pass (Partially Missing Sad tests)	2
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.8: Evaluation of ILA unit tests across different files using TabNine tool (Part 2)

4.2.4 Codiumate

Table 4.9 and Table 4.10 display the results of the evaluation of the unit tests generated by Codiumate for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 29 unit tests, Sort-labels has 26 unit tests, Update-DesignList-Validity has 67 unit tests, and lastly, FilterFunctions has 62 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that all the unit tests generated by Codiumate have relevant aims and proper assertions. On the one hand, the unit tests offer sufficient variation to cover all test scenarios, but on the other hand, the tool generates a large number of redundant test cases, including some that are unnecessary. However, it is important to mention that it is the user that decides which test cases to generate and how many. Nonetheless, these issues should not prevent users from utilizing the tools, as they can generally be fixed manually, ensuring excellent quality of the test cases without requiring substantial time. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 2.*

In Table 4.9 and Table 4.10 , we see that Codiumate implements the AAA approach, maintains a single responsibility with a one-line act section, is self-validating, ensures test independence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with proper structural consideration. Therefore, the user does not need to concern or recheck the structure after the generation process is complete. Moreover, the mentioned tables show that Codiumate demonstrates reliability, with no random values used. The tool provides excellent protection against regression with an average score of 2.75/3 and below-average resistance to the refactoring score of 1.75/3. These scores indicate good accuracy but are insufficient for real-life scenarios. However, minor manual adjustments to the generated tests—such as fixing the data in the arrange section or modifying the assert section—could enhance these protection and resistance percentages, leading to sufficiently improved accuracy. Overall, this level of accuracy might be enough in certain situations, for instance, if a developer has limited time for implementing unit tests or the tests are not a top priority, yet the developer still wants to roughly assess the quality of the source code. In such cases, this level of test accuracy would suffice with some manual modifications.

Furthermore, according to the measurements in Table 4.9 and Table 4.10, Codiumate implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by employing various types of test cases covering both positive and negative scenarios. Codiumate offers the option to continue generating test cases. This capability is beneficial as it allows for broader coverage but also carries the risk of increasing test case redundancy, which can make the test code less clear and harder to maintain for other developers.

Additionally, Table 4.9 and Table 4.10 indicate a good level of maintainability, where Codiumate features tests that are easy to maintain and understand in most cases, with clear variable names, organized code, and well-defined linked functions' components. There are no redundant assertions, which could otherwise decrease the quality of the test cases. However, Codiumate struggles with excessive code length and redundancy, including unnecessary test cases. While this issue does not affect the accuracy of results, it does reduce the quality of the tests and complicates future modifications.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	22/29 =75.8% (Predominantly pass)	24/26 =92.3% (Predominantly pass)	42/67 =62.6% (Partially pass)	52/62 =83.8% (Predominantly pass)	2.75
Offers resistance to refactoring	15/29 =51.7% (Partially pass)	3/26 =11.5% (Fail)	54/67 =80.5% (Predominantly pass)	45/62 =72.5% (Partially pass)	1.75
Ensures high test accuracy	(15+22)/ (15+22+7+14) =64% (Partially pass)	(3+24)/ (3+24+ 2+23) =52% (Partially pass)	(54+42)/ (54+42+ 13+25) =72% (Partially pass)	(45+52)/ (45+52+ 17+10) =78% (Predominantly pass)	2
Guarantees fast execution time	7.15s (Pass)	5.65s (Pass)	5.38s (Pass)	6.11s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Pass	3

Table 4.9: Evaluation of ILA unit tests across different files using Codiumate tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	94.79% (Pass)	94.44% (Pass)	100% (Pass)	88.67% (Pass)	3 (94.475%)
Achieves sufficient code coverage	94.79% (Pass)	94.44% (Pass)	100% (Pass)	88.67% (Pass)	3 (94.475%)
Achieves sufficient branch coverage	78.12% (Pass)	60% (Pass)	100% (Pass)	100% (Pass)	2.75 (84.53%)
Small test code size, and low number of assertions	Fail. No redundant assertions: Fail	Code size: Fail. No redundant assertions: Fail	Code size: Fail. No redundant assertions: Fail	Code size: Fail. No redundant assertions: Fail	0
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Pass	Pass	Pass	3
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.10: Evaluation of ILA unit tests across different files using Codiumate tool (Part 2)

4.2.5 GitHub Copilot

Table 4.11 and Table 4.12 display the results of the evaluation of the unit tests generated by GitHub Copilot for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 3 unit tests, Sort-labels has 7 unit tests, Update-DesignList-Validity has 17 unit tests, and lastly, FilterFunctions has 17 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that the majority of the unit tests generated by GitHub Copilot have appropriate aims and relevant assertions. Only a few test cases exhibit issues with some of their assertions, such as infeasible inputs in the assertion section or assertions that are unrelated to the test case's goal. These infeasible inputs result in some test failures, which consequently reduce the percentage of coverage. Furthermore, the unit tests lack sufficient variation in test cases to encompass all testing scenarios. These numerous issues can not be quickly resolved manually, making them unreliable. The quality of test cases should not be overlooked; higher-quality test cases more effectively fulfill the goals of unit testing. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 1.*

InTable 4.11 and Table 4.12, we see that Copilot implements the AAA approach, maintains a single responsibility with a one-line act section, is self-validating, ensures test independence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with proper structural consideration. Therefore, the user does not need to concern or recheck the structure after the generation process is complete. Moreover, the mentioned tables show that GitHub Copilot demonstrates reliability, with no random values used. The level of protection against regression is good enough with a 2.5/3 average score, and resistance to refactoring is low with a 1.5/3 average score, leading to an average accuracy, which is insufficient for real-life scenarios. In addition, correcting the unit tests generated by GitHub Copilot is time-consuming due to multiple issues, such as infeasible input for assertions or assertions unrelated to the aim of the tested function.

Furthermore, according to the measurements in Table 4.11 and Table 4.12, GitHub Copilot implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by using various types of test cases covering both happy and sad tests. GitHub Copilot provides interactive features allowing the developer to request additional test cases. This capability is beneficial as it allows for broader coverage but carries the risk of increasing test case redundancy, which can make the test code less clear and harder to maintain for other developers.

Additionally, Table 4.11 and Table 4.12 show that GitHub Copilot features tests that are easy to maintain and understand in most test cases, with clear variable names, organized code, and well-defined linked functions' components. Additionally, there are no redundant assertions, which could decrease the quality of the test cases. However, GitHub Copilot generally has appropriate code sizes, although they could be further shortened by eliminating a few redundant test cases.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	3/3 =100% (Pass)	4/7 =57.1% (Partially pass)	8/17 =47% (Partially pass)	14/17 =82.3% (Predominantly pass)	2.5
Offers resistance to refactoring	1/3 =33.3% (Partially pass)	3/7 =42.8% (Partially pass)	12/17 =70.5% (Partially pass)	10/17 =58.8% (Partially pass)	1.5
Ensures high test accuracy	(1+3)/ (1+3+0+2) =67% (Partially pass)	(3+4)/ (3+4+3+4) =50% (Partially pass)	(12+8)/ (12+8+5+9) =59% (Partially pass)	(10+14)/ (10+14+3+7) =71% (Partially pass)	1.75
Guarantees fast execution time	5.36s (Pass)	7.97s (Pass)	5.06s (Pass)	5.56s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Pass	3

Table 4.11: Evaluation of ILA unit tests across different files using GitHub Copilot tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	65.05% (Pass)	94.64% (Pass)	94.66% (Pass)	83.72% (Pass)	2.75 (84.5175%)
Achieves sufficient code coverage	65.05% (Pass)	94.64% (Pass)	94.66% (Pass)	83.72% (Pass)	2.75 (84.5175%)
Achieves sufficient branch coverage	50% (Fail)	60% (Pass)	81.25% (Pass)	91.66% (Pass)	2.25 (70.7275%)
Small test code size, and low number of assertions	Pass. No redundant assertions: Pass	Code size: Pass. No redundant assertions: Pass	Code size: Predominantly pass. No redundant assertions: Pass	Code size: Predominantly pass. No redundant assertions: Pass	2.75
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Pass	Pass	Pass	3
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.12: Evaluation of ILA unit tests across different files using GitHub Copilot tool (Part 2)

4.2.6 CodePal

Table 4.13 and Table 4.14 display the results of the evaluation of the unit tests generated by CodePal for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 15 unit tests, Sort-labels has 12 unit tests, Update-DesignList-Validity has 15 unit tests, and lastly, FilterFunctions has 39 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that the majority of the unit tests generated by CodePal are well-aligned with the intended aim and feature relevant assertions. Additionally, these unit tests exhibit sufficient variation to encompass all test scenarios, though there are a few redundant test cases. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 3.*

In Table 4.13 and Table 4.14, we see that CodePal implements the AAA approach, maintains a single responsibility with a one-line act section, is self-validating, ensures test independence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with proper structural consideration. Therefore, the user does not need to concern or recheck the structure after the generation process is complete. Moreover, the mentioned tables indicate that CodePal demonstrates reliability, using no random values. CodePal provides protection with a 1.5/3 average score and resistance with a 2.5/3 average score. These scores lead to satisfactory accuracy but are inadequate for real-life applications. However, minor manual adjustments to the generated tests—such as refining the data in the arrange section or modifying the assert section—could improve these protection and resistance percentages, thereby enhancing accuracy to a sufficient level. Overall, this accuracy level may suffice in certain circumstances—for instance, if a developer lacks the time to implement unit tests or if these are not a high priority, yet the developer still wishes to measure the quality of the source code roughly. In such instances, this level of test accuracy would be sufficient to demonstrate whether the function is correctly implemented in terms of functionality with some manual modifications.

Furthermore, according to the measurements in Table 4.13 and Table 4.14, CodePal implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by employing various types of test cases that cover both positive and negative scenarios.

Additionally, Table 4.13 and Table 4.14 reveal that the tool features tests that are easy to maintain and understand in most cases, with clear variable names, organized code, and linked functions' components being well defined. Moreover, there are no redundant assertions, which could diminish the quality of the test cases. However, CodePal struggles with excessive code length and redundancy or unnecessary test cases.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	3/15 =20% (Fail)	6/12=50% (Partially pass)	8/15=53.3% (Partially pass)	33/39 =84.6% (Predominantly Pass)	1.5
Offers resistance to refactoring	14/15 =93.3% (Predominantly Pass)	10/12 =83.3% (Predominantly Pass)	13/15 =86.6% (Predominantly Pass)	16/39 =41% (Partially pass)	2.5
Ensures high test accuracy	(14+3)/ (14+3+1+12) =57% (Partially pass)	(10+6)/ (10+6+2+6) =67% (Partially pass)	(13+8)/ (13+8+2+7) =70% (Partially pass)	(16+33)/ (16+33+ 23+6) =63% (Partially pass)	2
Guarantees fast execution time	5.41s (Pass)	4.98s (Pass)	5.75s (Pass)	5.16s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Predominantly pass	2.75

Table 4.13: Evaluation of ILA unit tests across different files using CodePal tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	95.69% (Pass)	100% (Pass)	100% (Pass)	88.32% (Pass)	3 (96.0025%)
Achieves sufficient code coverage	95.69% (Pass)	100% (Pass)	100% (Pass)	88.32% (Pass)	3 (96.0025%)
Achieves sufficient branch coverage	75% (Pass)	80% (Pass)	100% (Pass)	100% (Pass)	2.75 (88.75%)
Small test code size, and low number of assertions	Partially pass. No redundant assertions: Pass	Code size: Partially pass . No redundant assertions: Pass	Code size: Predom-inantly pass. No redundant assertions: Pass	Code size Fail. No redundant assertions: Pass	2
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Pass	Pass	Pass	3
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.14: Evaluation of ILA unit tests across different files using CodePal tool (Part 2)

4.2.7 IKEA GPT

Table 4.15 and Table 4.16 display the results of the evaluation of the unit tests generated by IKEA GPT for each criterion, with a final average score. A detailed analysis of the results is presented in this section.

The generated unit tests were divided across four files, Save-labels has 10 unit tests, Sort-labels has 9 unit tests, Update-DesignList-Validity has 18 unit tests, and lastly, FilterFunctions has 18 unit tests.

Based on the thoughts of IKEA's software engineers obtained during our interview to assess the quality of the test cases, we found that the majority of the unit tests generated by IKEA GPT demonstrate appropriate aims and relevant assertions. However, certain test cases are incorrectly implemented in the arrange section, resulting in some failures. Additionally, the unit tests provide sufficient variation in test cases to encompass all testing scenarios. Nonetheless, these issues should not prevent users from utilizing the tools, as they can generally be fixed manually, ensuring excellent quality of the test cases without requiring substantial time. Based on the approach we are using to assign the scores in Subsection 3.4.2, *the score for quality of test cases: 2.*

In Table 4.15 and Table 4.16, we see that IKEA GPT implements the AAA approach, maintains a single responsibility with a one-line act section, is self-validating, ensures test independence, and avoids multiple cycles of AAA and 'if' statements in most of the generated unit tests. These results demonstrate the tool's ability to generate tests with proper structural consideration. Therefore, the user does not need to concern or recheck the structure after the generation process is complete. Moreover, the mentioned tables indicate that IKEA GPT demonstrates reliability, without using any random values. IKEA GPT provides comparable levels of protection and resistance with an average score of 2/3, leading to satisfactory accuracy but proving insufficient for real-life applications. Nonetheless, minor manual adjustments to the generated tests—such as correcting data in the arrange section or revising the assert section—could improve these percentages of protection and resistance, resulting in enhanced accuracy to a sufficient level. Overall, this level of accuracy suggests that it might suffice in certain situations—for example, when the developer lacks time to implement unit tests or they are not a top priority, yet the developer wishes to roughly measure the quality of the source code. In such instances, this degree of test accuracy would be sufficient to demonstrate whether the function is implemented correctly in terms of functionality with some manual modifications.

Furthermore, according to the measurements in Table 4.15 and Table 4.16, IKEA GPT implements unit tests with sufficient coverage in terms of line, code, and branch coverage, exceeding 60%, by utilizing various types of test cases covering both happy and sad scenarios. IKEA GPT offers interactive features that allow the developer to request additional test cases. This capability is advantageous as it allows for more extensive coverage but carries the risk of increasing test case redundancy, which can make the test code less clear and more challenging to maintain for other developers.

Additionally, Table 4.15 and Table 4.16 show that the tool features tests that are easy to maintain and understand in most cases, with clear variable names, organized code, and well-defined linked functions' components. Moreover, there are no redundant assertions, which could decrease the quality of the test cases. Finally, IKEA GPT generally maintains appropriate code sizes, although they could be further reduced by creating shared data.

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Includes the AAA (Arrange, Act, Assert) pattern	Pass	Pass	Pass	Pass	3
Avoids multiple cycles of arrange, act, and assert steps	Pass	Pass	Pass	Pass	3
Avoids if statements	Pass	Pass	Pass	Pass	3
Contains a one-line act section	Pass	Pass	Pass	Pass	3
Provides protection against regressions	5/10 =50% (Partially pass)	7/9 =77.8% (Predominantly pass)	12/18 =66.7% (Partially pass)	11/18 =61.1% (Partially pass)	2
Offers resistance to refactoring	9/10 =90% (Predominantly pass)	4/9 =44.4% (Partially pass)	13/18 =72.2% (Partially pass)	16/18 =88.9% (Predominantly pass)	2.25
Ensures high test accuracy	(9+5)/(9+5+1+5) =70% (Partially pass)	(4+7)/(4+7+5+2) =61% (Partially pass)	(13+12)/(13+12+5+6) =69% (Partially pass)	(11+16)/(11+16+2+7) =75% (Predominantly pass)	2
Guarantees fast execution time	10.04s (Pass)	5.36s (Pass)	7.37s (Pass)	5.67s (Pass)	3
Is easy to maintain	Pass	Pass	Pass	Pass	3

Table 4.15: Evaluation of ILA unit tests across different files using IKEA GPT tool (Part 1)

Criteria	Save-Labels	Sort-Labels	Update-DesignList-Validity	Filter Functions	Score Value
Achieves sufficient statement coverage	98.38% (Pass)	100% (Pass)	100% (Pass)	100% (Pass)	3 (99.595%)
Achieves sufficient code coverage	98.38% (Pass)	100% (Pass)	100% (Pass)	100% (Pass)	3 (99.595%)
Achieves sufficient branch coverage	100% (Pass)	90% (Pass)	100% (Pass)	100% (Pass)	3 (97.5%)
Small test code size, and low number of assertions	Pass. No redundant assertions: Pass	Code size: Pass. No redundant assertions: Pass	Code size: Predom-inantly pass. No redundant assertions: Pass	Code size: Predom-inantly pass. No redundant assertions: Pass	2.75
Adheres to the single responsibility	Pass	Pass	Pass	Pass	3
Demonstrates reliability	Pass	Pass	Pass	Pass	3
Includes both Happy and Sad Tests	Pass	Pass	Pass	Pass	3
Ensures test independence / Test Isolation	Pass	Pass	Pass	Pass	3
Is self-validating	Pass	Pass	Pass	Pass	3

Table 4.16: Evaluation of ILA unit tests across different files using IKEA GPT tool (Part 2)

4.3 Interview with IKEA's Tech Leader

In this section, we present the findings of the interview held with IKEA's tech leader aimed at exploring the challenges associated with adopting the AITT.

1. ***Would you trust the unit tests generated using the evaluated AITT? Furthermore, would you undertake any additional actions if you were to use them?*** The tech leader would rely on the unit tests generated by the AITT, provided they consistently show sufficient coverage. If this condition is met, no further verification of these unit tests would be necessary. This approach applies solely to unit tests, unlike integration tests, which require additional verification steps.

2. ***Could the AITT be used while considering both the AITT privacy policy (including those with a policy of storing data for 48 hours or those that claim not to store any data) and IKEA's policy?*** The tech leader noted that sensitive data typically does not reside in the front-end of web applications. This absence of sensitive data permits the use of tools like Codiumate and CodePal in front-end components. Codiumate retains data for 48 hours in its paid version for troubleshooting purposes, while CodePal's premium users are not required to provide CodePal with a license to use their code policy. However, it is not clearly stated whether CodePal stores users' code, either temporarily or permanently. Other tools, such as UnitTestAI, are also deemed acceptable despite lacking detailed data storage privacy information as long as there is no sensitive data in the front-end part. Nevertheless, direct communication with tool providers is necessary to obtain specific policy details. Furthermore, tools like PaLM, TabNine (free version), GitHub Copilot, and IKEA GPT are considered safe for use in both front-end and back-end development at IKEA, as they do not store data during use. However, verification of privacy policies is required from the tool providers, except for IKEA GPT, which was developed internally.

3. ***Are there any other challenges of introducing the selected and evaluated AITT into IKEA's web development context?*** In response to the third question, the tech leader identified additional complications, such as the need for multiple team approvals, including the architecture team, which may consume a lot of time. Another challenge arises when changes to the source code occur, corresponding adjustments in unit tests would then be necessary. Modifying generated unit tests can be time-consuming, as developers must understand and revise tests not originally written by them. Nonetheless, even in regular cases, it is sometimes a different developer who modifies the MI unit tests. This amounts to the same time consumption of modifying AI-generated unit tests. Lastly, developers would not find it challenging to learn to use these tools because of the available documentation and tutorial videos on the tools pages.

4.4 Results of the Multi-Criteria Decision-Making Method

In this section, we provide the results of both the normalized weights for each of the 19 criteria and the final WSM scores for each of the 7 AITT and the MI methods. In Section 5.3 we discuss the results of the WSM scores and lastly, in Chapter 6 we conclude which unit testing approach is superior based on the WSM scores.

4.4.1 Normalized Weights

The comparison matrix displayed in Table 4.18 serves as input for the Python script described in Listing 3.1. This matrix shows the relative importance of a specific criterion compared to other criteria. For example, on the left side, we see that the criterion *Includes the AAA pattern* holds equal importance to itself, hence the value is 1. In contrast, this criterion is moderately more important than the subsequent criterion *Avoids multiple cycles of AAA steps*, thus the value is 3. Meanwhile, the criterion is significantly less important than the criterion named *Provides protection against regressions*, which explains why the value is $\frac{1}{9}$. The entire matrix is filled with values using Saaty's scale [45], which serves as a method to eventually determine the importance of each criterion.

Running the Python script with our matrix results in an array of 19 normalized weights where the sum of those weights adds up to 1. Table 4.17 lists the 19 criteria with their corresponding weight of importance. These weights resemble the relative importance of each criterion with all 19 criteria taken into consideration. From the Table, we can see for example the criterion *Includes the AAA pattern* has a weight of 0.01140775 while the criterion *Test case quality* has a weight of 0.1232828. This means that the latter criterion carries more influence in the decision-making process, impacting the overall WSM score more strongly.

Criteria	Weight
Includes the AAA pattern	0.01140775
Avoids multiple cycles of AAA steps	0.0087644
Avoids if statements	0.00722379
Contains a one-line act section	0.00999873
Provides protection against regressions	0.1096437
Offers resistance to refactoring	0.1096437
Ensures high test accuracy	0.1096437
Time to prepare and perform tests	0.07050637
Is easy to maintain	0.01683098
Achieves sufficient code coverage	0.0589628
Achieves sufficient statement coverage	0.0589628
Achieves sufficient branch coverage	0.0589628
Small test code size, and low number of assertions	0.01872291
Adheres to the single responsibility/	0.01920049
Demonstrates reliability	0.11159307
Includes both Happy and Sad Tests	0.06018157
Ensures test independence	0.01999545
Is self-validating	0.01647219
Test case quality	0.1232828
	Total: 1.0

Table 4.17: The evaluation criteria with corresponding weight

Criteria	Includes the AAA (Arrange, Act, Assert) pattern	Avoids multiple cycles of AAA steps	Avoids if statements	Contains a one-line act section	Provides protection against regressions by minimizing false negative errors	Offers resistance to refactoring by minimizing false positive errors	Ensures high test accuracy	Time to prepare and perform tests	Is easy to maintain (simple to understand and modify)	Achieves sufficient code coverage	Achieves sufficient statement coverage	Achieves sufficient branch coverage	Small test code size, and low number of assertions of assertions, and number of redundant assertions)	Adheres to the single responsibility / single concept per test principle	Demonstrates reliability	Includes both Happy and Sad Tests	Ensures test independence / Test isolation	Is self-validating	Test case quality
Includes the AAA (Arrange, Act, Assert) pattern	1	3	3	5	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{3}$	$\frac{1}{5}$	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{5}$	$\frac{1}{3}$	$\frac{1}{9}$
Avoids multiple cycles of AAA steps	$\frac{1}{3}$	1	1	1	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{7}$	1	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	1	$\frac{1}{5}$	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{5}$	$\frac{1}{3}$	$\frac{1}{9}$
Avoids if statements	$\frac{1}{3}$	1	1	$\frac{1}{3}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{3}$	$\frac{1}{5}$	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{9}$
Contains a one-line act section	$\frac{1}{5}$	1	3	1	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{9}$	$\frac{1}{7}$	1	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	1	1	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{9}$
Provides protection against regressions	9	9	9	9	1	1	1	3	5	3	3	3	5	7	1	3	5	5	1
Offers resistance to refactoring	9	9	9	9	1	1	1	3	5	3	3	3	5	7	1	3	5	5	1
Ensures high test accuracy	9	9	9	9	1	1	1	3	5	3	3	3	5	7	1	3	5	5	1
Time to prepare and perform tests	7	7	7	7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	1	5	1	1	1	5	5	1	3	5	5	1
Is easy to maintain (simple to understand and modify)	3	1	3	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	1	$\frac{1}{7}$	$\frac{1}{5}$	1	3	$\frac{1}{9}$
Achieves sufficient code coverage	7	7	7	7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	1	5	1	1	1	5	5	$\frac{1}{3}$	1	5	7	$\frac{1}{3}$
Achieves sufficient statement coverage	7	7	7	7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	1	5	1	1	1	5	5	$\frac{1}{3}$	1	5	7	$\frac{1}{3}$
Achieves sufficient branch coverage	7	7	7	7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	1	5	1	1	1	5	5	$\frac{1}{3}$	1	5	7	$\frac{1}{3}$
Small test code size, and low number of assertions	3	1	3	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	1	$\frac{1}{7}$	$\frac{1}{5}$	3	3	$\frac{1}{9}$
Adheres to the single responsibility	5	5	5	1	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{5}$	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	1	$\frac{1}{7}$	$\frac{1}{7}$	1	3	$\frac{1}{9}$
Demonstrates reliability	9	9	9	9	1	1	1	1	7	3	3	3	7	7	1	5	7	7	$\frac{1}{3}$
Includes both Happy and Sad Tests	7	7	7	7	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$	5	1	1	1	5	7	$\frac{1}{5}$	1	5	5	1
Ensures test independence / Test isolation	5	5	5	5	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	1	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{3}$	1	$\frac{1}{7}$	$\frac{1}{5}$	1	1	$\frac{1}{9}$
Is self-validating	3	3	5	5	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{7}$	$\frac{1}{5}$	1	1	$\frac{1}{9}$
Test case quality	9	9	9	9	1	1	1	1	9	3	3	3	9	9	3	1	9	9	1

Table 4.18: Comparison Matrix of the criteria

4.4.2 WSM Scores

Figure 4.1 illustrates the WSM score for each unit testing alternative, sorted in ascending order, where the MI method scored the highest and TabNine scored the lowest.
Table 4.19, Table 4.20, Table 4.21, and Table 4.22 present the WSM calculations for each of the 8 alternatives. The first row in these tables represents the 19 evaluation criteria, while the second

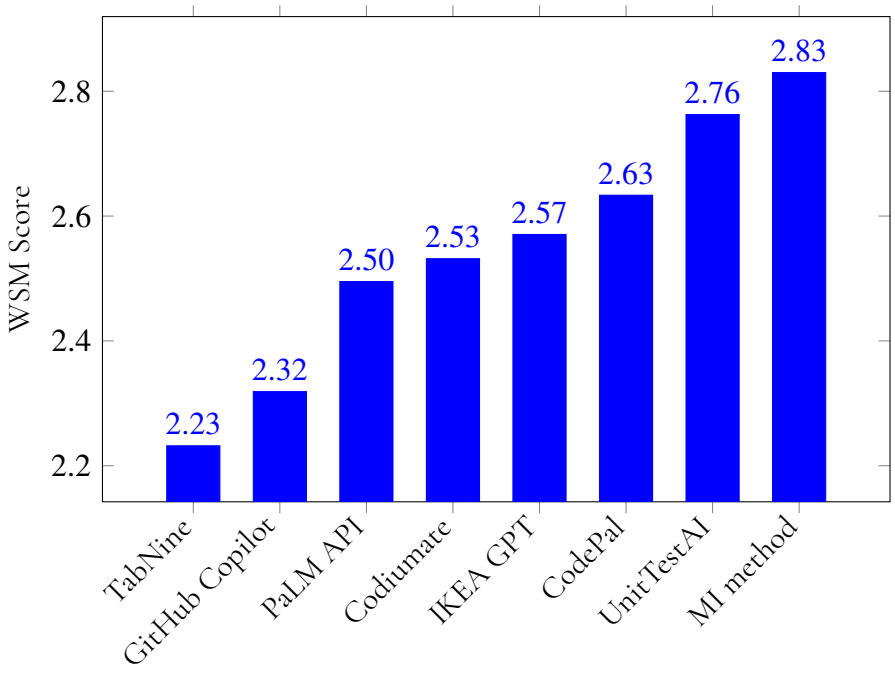

Figure 4.1: Various unit testing methods ranked by the WSM score

row illustrates the normalized weights corresponding to these criteria. Rows 3-10 show the 8 unit testing alternatives to be assessed and assigned a final WSM score.

For each unit testing alternative row, multiplication occurs between two numbers under every criterion column. The first number is the normalized weight of a specific criterion, and the second number is the score value that the unit testing alternative received for that specific criterion (final column in the evaluation tables). For example, TabNine has a score of 1.5 for the criterion *Includes the AAA pattern* (since it failed to completely implement the AAA approach, see Subsection 4.2.3), and this criterion's weight is 0.01140775, therefore "0.01140775∗1.5". PaLM API has a score of 2.75 for this criterion since it implemented the AAA approach relatively better than TabNine, thus "0.01140775∗2.75". The rest of the unit testing alternatives have the highest score of 3 since they were able to fulfill this criterion flawlessly "0.01140775∗3", indicating that these alternatives are equally good in this aspect.

These multiplications are calculated for each criterion and then summed together. In other words, it is the total sum of multiplications, whose formula is introduced in Subsection 2.5.2. This sum represents the WSM score. The overall WSM score for each alternative is displayed in the column labeled 'WSM Score'.

Finally, the last column indicates the rank of each alternative, with rank 1 denoting the most superior unit testing approach.

Criteria	Includes the AAA pattern	Avoids multiple cycles of AAA steps	Avoids if statements	Contains a one-line act section	Protection against regressions
Weights	0.01140775	0.0087644	0.00722379	0.00999873	0.1096437
MI method	0.03422325	0.0262932	0.02167137	0.0124984125	0.3289311
Codiumate	0.03422325	0.0262932	0.02167137	0.02999619	0.301520175
UnitTestAI	0.03422325	0.0262932	0.02167137	0.02999619	0.301520175
TabNine	0.017111625	0.0262932	0.02167137	0.02999619	0.137054625
PaLM API	0.0313713125	0.0241021	0.02167137	0.0274965075	0.3289311
GitHub Copilot	0.03422325	0.0262932	0.02167137	0.02999619	0.27410925
CodePal	0.03422325	0.0262932	0.02167137	0.02999619	0.16446555
IKEA GPT	0.03422325	0.0262932	0.02167137	0.02999619	0.2192874

Table 4.19: WSM Scores for the AITT and MI Method Across 19 Evaluation Criteria (Part 1/4)

Criteria	Offers resistance to refactoring	Ensures high test accuracy	Time to prepare and perform tests	Is easy to maintain	Sufficient code coverage
Weights	0.1096437	0.1096437	0.07050637	0.01683098	0.0589628
MI method	0.3289311	0.3289311	0.21151911	0.046285195	0.1768884
Codiumate	0.191876475	0.2192874	0.21151911	0.05049294	0.1768884
UnitTestAI	0.27410925	0.3289311	0.21151911	0.05049294	0.1768884
TabNine	0.3289311	0.2192874	0.21151911	0.05049294	0.1768884
PaLM API	0.2192874	0.246698325	0.21151911	0.046285195	0.1621477
GitHub Copilot	0.16446555	0.191876475	0.21151911	0.05049294	0.1621477
CodePal	0.27410925	0.2192874	0.21151911	0.046285195	0.1768884
IKEA GPT	0.246698325	0.2192874	0.21151911	0.05049294	0.1768884

Table 4.20: WSM Scores for the AITT and MI Method Across 19 Evaluation Criteria (Part 2/4)

Criteria	Sufficient statement coverage	Sufficient branch coverage	Small test code size, and low number of assertions	Adheres to the single responsibility	Demonstrates reliability
Weights	0.0589628	0.0589628	0.01872291	0.01920049	0.11159307
MI method	0.1768884	0.1621477	0.03744582	0.048001225	0.33477921
Codiumate	0.1768884	0.1621477	0	0.05760147	0.33477921
UnitTestAI	0.1768884	0.147407	0.05382836625	0.05760147	0.33477921
TabNine	0.1768884	0.1621477	0.0514880025	0.05760147	0.33477921
PaLM API	0.1621477	0.1326663	0.0421265475	0.05760147	0.33477921
GitHub Copilot	0.1621477	0.13266635	0.0514880025	0.05760147	0.33477921
CodePal	0.1768884	0.1621477	0.03744582	0.05760147	0.33477921
IKEA GPT	0.1768884	0.1768884	0.0514880025	0.05760147	0.33477921

Table 4.21: WSM Scores for the AITT and MI Method Across 19 Evaluation Criteria (Part 3/4)

Criteria	Includes both Happy and Sad Tests	Ensures test independence	Is self-validating	Test case quality	WSM Score	Rank
Weights	0.06018157	0.01999545	0.01647219	0.1232828	-	-
MI method	0.0752269625	0.05998635	0.04941657	0.3698484	2.8299128749999998	1
Codiumate	0.18054471	0.05998635	0.04941657	0.2465656	2.53169852	5
UnitTestAI	0.18054471	0.05998635	0.04941657	0.2465656	2.76266266125	2
TabNine	0.12036314	0.05998635	0.04941657	0	2.2319168024999994	8
PaLM API	0.090272355	0.05998635	0.04941657	0.2465656	2.4950722225	6
GitHub Copilot	0.18054471	0.05998635	0.04941657	0.1232828	2.3187081475	7
CodePal	0.18054471	0.05998635	0.04941657	0.3698484	2.633397545	3
IKEA GPT	0.18054471	0.05998635	0.04941657	0.2465656	2.5705162974999993	4

Table 4.22: WSM Scores for the AITT and MI Method Across 19 Evaluation Criteria (Part 4/4)

Chapter 5
Discussion

In this chapter, we discuss the findings from our evaluation of MI and AITT unit testing approaches, focusing on performance, quality, and challenges. Our comparative analysis highlights the strengths and weaknesses of each method, illustrating how advanced AI tools are narrowing the gap with traditional manual testing. Moreover, we discuss the comparison of the evaluation results, which reveal that while manual inspection (MI) testing remains the gold standard due to its thoroughness and developer insight, several AI tools show promising potential in automating and enhancing the testing process. Lastly, challenges such as data type recognition, test redundancy, and privacy concerns need to be addressed to fully integrate AITT into mainstream software development practices.

5.1 Performance of MI Unit Testing (RQ1)

In this section, we discuss the findings of the evaluation of the performance of MI unit testing. **Our findings show that setup and learning are relatively quick for experienced developers, but preparation and execution times significantly vary based on code clarity and tester expertise. Despite these variations, MI unit tests demonstrate high accuracy, quality, and maintainability with an organized structure and fair coverage, emphasizing their essential role in ensuring robust software quality.**

Unit testing is crucial for every developer as it greatly influences code quality. Our evaluation of MI unit tests (RQ1) shows that MI testing enhances code quality, this can be seen throughout the high score of multiple evaluation criteria in Table 4.1 and Table 4.2, including the most important ones such as "Ensures high test accuracy", "Quality of the test cases", coverage, and more. Despite that, according to IKEA's software engineer, the time it takes to learn unit testing varies among individuals, depending on their academic backgrounds or prior experience. Typically, setting up a test environment on a computer involves preparing the necessary test data,

downloading the required test framework, managing dependencies, and accessing the source code. This process generally does not take much time. However, preparing the test data may take longer if the developer is unfamiliar with the source code and its functionality.

Moreover, setting up the test environment is quick according to IKEA's software engineer, but implementing the unit tests depends on familiarity with the source code and the developer's expertise. Beyond the immediate results, it is important to consider the long-term impact of MI unit testing on developer productivity. Based on what IKEA reported regarding the time to prepare the tests, MI unit tests can be time-consuming, particularly in terms of setup and learning. Frequent switching between writing code and tests can lead to cognitive fatigue, potentially reducing overall productivity.

Based on the MI unit test evaluation results in Table 4.1 and Table 4.2, we find that tests are structured and accurate where they adhere to the AAA pattern and have relative test case aim to the tested function. The coverage is considered good since it exceeds 60%, but could be improved with more negative tests. Test code is maintainable and well-organized. Overall, the high-quality test cases effectively cover relevant scenarios, with a fast average execution time enabling rapid feedback and sustained development, where the average time is 11.9 seconds. However, we cannot regard this evaluation of MI unit testing as a standard for all MI unit tests, as it varies among developers as mentioned earlier. Therefore, there can be significant differences in the evaluation results with other MI unit tests implemented by different developers with different years of experience. This suggests a need for good knowledge-sharing and documentation practices within development teams. Creating a culture where documentation is prioritized and peer reviews are a regular practice can ensure that knowledge about the codebase and best testing practices is scattered across the team. This can be particularly beneficial for new team members, who can quickly get up to speed with the project's testing standards and methodologies.

5.2 Performance of the AITT in Unit Testing (RQ2)

In this section, we discuss our findings from the evaluation regarding the performance of the AITT. **Our findings show that AITT demonstrate a simplified setup and learning process, supported by extensive documentation and user-friendly interfaces. The AITT significantly reduce the time needed to prepare unit tests with and 6.9 average execution time. While the structure, quality, and accuracy of the tests differ among tools, some, such as CodePal and UnitTestsAI, nearly achieve the effectiveness of the MI tests, suggesting they are a feasible supplement to conventional testing methods.**

Our analysis of the unit tests generated by the AITT (RQ2) offers several key insights. Initially, learning to use AITT was straightforward, thanks to the thorough documentation provided on their respective websites. These resources typically feature step-by-step guides or brief videos that explain how to use the tools and their commands. Such instructions and videos contribute to a quick and effortless learning experience for new users. Furthermore, most tools allow users to select only a specific segment of the source code for testing; then, with a simple right-click, the unit testing option is available to select. It is worth noting that users do not require expertise in unit testing to utilize these tools and generate tests, as reading the provided doc-

umentation is generally sufficient. However, the setup process involves the same requirements mentioned in Section 5.1, along with the installation of the tool itself. Typically, an additional step includes registering and logging in, which is a brief procedure.

Moreover, AITT significantly reduce the time required to prepare and conduct unit tests, with most tests generated and executed within minutes, supporting high development velocity as they provide developers with rapid feedback without impeding progress.

Based on the evaluation of AITT, most tools adhere to good unit test practices, although some tools require attention. On the one hand, the accuracy of the generated tests varies significantly, but on the other hand, most tools achieve sufficient coverage. Maintaining the tests appears straightforward due to organized code and clear variable naming, although code size and redundancy issues with some tools could complicate future maintenance efforts. The quality of the test cases varies; some tools produce sufficient and well-aligned test cases with proper assertions such as Codepal and UnitTestAI where they could generate high-quality tests as seen in Table 4.3, Table 4.4,Table 4.13, and Table 4.14. However, other AITT have issues making them unreliable, such as Codiumate which generated a high number of redundant test cases. Lastly, while there are areas for improvement, particularly in enhancing test accuracy, coverage, and quality, AITT offer significant advantages for developers looking to streamline unit testing processes. With minor manual adjustments and skillful use of tool capabilities, these tools can greatly aid in improving the quality and efficiency of software testing. Furthermore, the findings revealed that AITT show great promise but sometimes lack the depth of understanding seen in MI tests, specifically TabNine which had a 0 score value regarding the quality of test cases based on the evaluation from IKEA's software engineer. Customizing and configuring AI tools to better fit specific project needs can help bridge this gap. This could involve incorporating domain-specific knowledge into the AI models. For example, configuring AI tools to recognize common patterns in the codebase or specific business logic can improve the relevance and accuracy of the generated tests.

5.3 Comparison between AITT and MI approaches (RQ1 & RQ2)

In this section, we explore an analysis of the findings from the comparison between AITT and MI unit tests. **Our findings show that while MI unit tests remain the gold standard, certain AITT demonstrate significant potential. As AI technology continues to develop, tools such as UnitTestAI and CodePal are likely to soon equal or surpass the quality of MI unit tests. Moreover, they could replace MI unit testing with satisfactory quality. Middle-ranked tools offer a valuable mix of automation and manual intervention, optimizing the testing workflow. However, lower-ranked tools need further development to become usable additions, as their current implementation might lead to increased effort rather than efficiency.**

The results shown in Table 4.22 provide a comparison between various AITT and MI unit tests. As expected, MI unit tests scored the highest. This result is anticipated since developers have a thorough understanding of the system architecture and test scenarios, enabling them to address numerous edge cases. Additionally, developers spend considerable time accurately verifying the implementation details and ensuring the quality of these unit tests. However, the

effectiveness of MI unit tests depends on the developer's expertise and knowledge in software testing, introducing variability based on individual skill levels.

Notably, the tools UnitTestAI and CodePal, which secured second and third places respectively, are not far behind MI unit tests in terms of their WSM scores. This closeness suggests that these tools use advanced technologies that nearly match the effectiveness of MI unit testing or serve as a strong supplement to it. With rapid advancements in AI, it is expected that these tools will soon match the quality of MI unit tests. For teams seeking an optimal level of quality for unit tests, these AI tools can produce tests that, with minor modifications, can compete with the quality of MI unit tests, thus saving significant time and resources. In scenarios where satisfactory unit tests are needed quickly, these tools can effectively replace MI unit tests, providing a reasonable solution in a shorter timeframe with less effort.

Moreover, the tools IKEA GPT, Codiumate, and PaLM, which ranked fourth, fifth, and sixth respectively, also display competitive WSM scores. These tools can serve as valuable supplements to MI unit tests by generating an initial suite of unit tests, which developers can then enhance by adding any missing test cases. This hybrid approach can potentially reduce the overall time and resources required for unit test implementation, offering a balanced strategy for achieving quality and efficiency.

Conversely, tools like GitHub Copilot and TabNine, which ranked seventh and eighth respectively, pose challenges as effective supplements for developers. Initially using these tools might require extensive modifications or issue resolutions, potentially resulting in longer development times compared to MI unit tests. Therefore, these tools may not currently offer the same level of utility as the higher-ranked AI tools or MI unit tests.

Furthermore, a relevant study mentioned in Chapter 2 by Straub and Huber [50] found that AI-based testing techniques, like the AITCP, are significantly faster than manual testing. Their analysis revealed that AI maintains a consistent performance level even in complex scenarios, whereas human testers experience a significant slowdown. This demonstrates the superior efficiency and speed of AI in software testing compared to manual methods. This aligns with our findings, where the AITT were extremely fast in preparing and generating unit tests, in contrast to the MI method, which takes more time and depends on the individual's expertise and familiarity with the source code.

Last but not least, a relevant study mentioned in Chapter 2 by Serra et al. [47] found that automatic testing tools achieve line coverage comparable to or even higher than manual tests. This high coverage is due to the tools' primary goal of optimizing test coverage on production code. This demonstrates that automatic tools can surpass manual testing in ensuring thorough code coverage. This also aligns with our findings, where some AITT were able to achieve a high line coverage and in some cases even higher than manual testing.

5.4 Challenges of Using AITT (RQ3)

In this section, we discuss the challenges of using AITT based on the outcome of the interview with IKEA's tech leader as well as our own experience of using AITT. **Our findings show that these challenges include issues with data type recognition, redundancy in generated tests, and privacy concerns. While these tools streamline the unit testing process, developers must still address these challenges to ensure accuracy and high test quality. Strategies such as manual**

test review, monitoring coverage, and ensuring data privacy are crucial to overcoming these obstacles and effectively integrating AITT into software development workflows.

There are some challenges encountered when using the AITT. Firstly, some AITT occasionally generate unit tests with incorrect data types or exclude some inputs. This can be seen with tools like GitHub Copilot and Codiumate where a significant number of their generated test cases did not pass partly due to issues with the data types. With that being said, this issue arises because the tools cannot recognize all data types that are MI and not built-in in the programming language itself. Although infrequent, it still poses a challenge. A research mentioned in Chapter 2 by Ghosh et al. [20] discusses the importance of domain-specific knowledge in the context of AI applications. The research mentions that AI models, particularly deep neural networks, often require large amounts of data for effective analysis and classification. The merging of diverse datasets is emphasized as a key concept to improve model performance, indicating the necessity for domain-specific knowledge and data to enhance AI capabilities. This is backed by an article by Bajaj and Samal [6] where they discuss how AI models need to be trained on datasets specific to the domain to ensure that test cases and bug identifications are relevant, precise, and meaningful within the specific context of the software system. A potential solution for this problem involves modifying the data type or importing data types from external files. This solution is not time-consuming but needs to be kept in mind when using the AITT tools.

Another challenge involves tools like Codiumate, which have a click option to continue generating unit tests, and chat interaction tools like IKEA GPT, TabNine, or Copilot. The primary issue is determining when to stop generating new test cases, which sometimes leads to redundancy. The solution involves continually monitoring coverage while generating unit tests, which, although time-consuming, helps reduce redundancy. However, this could be complex as coverage is not the sole quality indicator of test coverage; variation in test cases is crucial as it encompasses possible testing scenarios (edge, sad, and happy scenarios). Therefore, an additional measure is required which involves manually reviewing the generated tests to ensure they cover all potential scenarios and maintain sufficient coverage (line, statement, and branch coverage). We encountered no challenges regarding the learning process or setting up the tools since all tools feature clear documentation, including usage guides. All tools are free, although some, like Codiumate, offer paid upgrades for more secure versions.

Moreover, as per the tech leader we interviewed, privacy is a challenge with tools that lack privacy information, posing a risk of data leakage. Tools that retain data for up to 48 hours could be restricted to front-end web application usage. This aligns with the article by Bajaj and Samal [6] which is mentioned in Chapter 2 regarding the ethical considerations in AI-based technology, particularly AI-generated test cases. The authors highlight significant privacy and security concerns, which gives the need for proper safeguards and compliance with privacy regulations to protect sensitive data, such as source code, from potential breaches and misuse. However, tools deemed secure could be utilized in both front-end and back-end parts of the web application. There are no challenges in starting to use these tools, except for the time required to receive approval from IKEA's architect team. Another challenge arises when the source code gets modified, requiring modifications to the generated unit tests. The developer would need to understand the existing unit tests before making changes. However, this could be swiftly resolved by generating new unit tests for the updated source code, although it is still required to check the coverage and scenario coverage, which is faster than altering MI unit tests.

Additionally, another related work is mentioned in Chapter 2 regarding the challenges of

AI in software testing. While Khaliq et al. [31] believe that AI techniques in software testing often require significant computational resources, we have not encountered such an issue while testing any of ILA's files. In fact, we experienced the opposite, since the majority of the AITT were simple extensions in the Visual Studio IDE which did not require significant computational resources.

5.5 Threats to Validity

5.5.1 Subjectivity in the Pairwise Comparison Process of MCDM

In the initial step of the MCDM method, specifically during the pairwise comparison process, we focused on one aspect while completing the comparison matrix with numerical values. We evaluated which criterion, among the two being compared, would have a greater influence on the outcomes of the unit tests. Consequently, the criterion "Ensures high test accuracy" received a higher value, typically 9, especially when compared with "Avoids 'if' statements," as the former has a more significant impact on the final results of the unit tests, while the latter does not. However, the assignment of these values was subjective, which could pose a threat to the validity of our results. We performed the pairwise comparison while sat together and discussing the values to be put, however in order to enhance the validity, we could have filled out the matrix individually and then discussed our findings to incorporate diverse perspectives. Moreover, an additional step for lowering the threat to the validity would be involving a software engineer from IKEA to validate our values which we used to fill out the matrix and provide their insights on the relative importance of each criterion.

5.5.2 Generalizability of Results

The findings from this study are based on a specific set of AI-powered tools and a particular web application (IKEA's ILA). There may be limitations in generalizing these results to other contexts, tools, or applications. Different AI tools or different types of software projects might yield different results.

5.5.3 Influence of Subjective Evaluations

It is important to consider the subjective nature of evaluating test case quality based on interviews with IKEA's software engineers. While these insights provide valuable perspectives, the evaluation's scope could be broadened by gathering input from a diverse range of software engineers. This approach would offer a more comprehensive assessment, potentially leading to refined evaluations and insights. Similarly, addressing challenges identified by team leaders through interviews with multiple tech leaders would provide a broader perspective, enhancing the credibility and robustness of our findings. Thus, incorporating diverse viewpoints strengthens the validity and relevance of our research outcomes.

5.5.4 Dependency on Tool Evolution

AI-based testing tools are subject to frequent updates and improvements. The capabilities and performance of these tools can change, potentially altering the effectiveness of the results reported in this thesis over time.

5.5.5 Lack of underlying data

Other limitations impact the validity of our results. Firstly, due to time constraints, we were unable to provide a detailed spreadsheet that includes an evaluation of each test case separately. This limits the depth of our analysis in this specific aspect. Secondly, the confidentiality of IKEA's code necessitated the exclusion of the test case code from the report. Although this restricts external validation to some extent, we have ensured that all other aspects of our method are documented and explained. However, these limitations still pose a risk to the validity of our results, as external validation is hindered. Without access to the test cases' code or the underlying data, it is challenging for others to replicate our study or validate the accuracy and reliability of our evaluations. Addressing this limitation in future research would enhance the robustness of the conclusions drawn.

5.6 Ethical and Social Aspects

5.6.1 Privacy of Proprietary Code

After reading the privacy policies of each AITT and interviewing IKEA's tech leader, we found a privacy challenge with certain tools. Using tools that store data for up to 48 hours, such as the Codiumate tool, poses a risk of data leakage. Conversely, tools that do not store data, such as the GitHub Gopilot tool, do not pose such risks. To minimize the risk of data leakage, tools that store data can be employed in the front-end, while secure tools can be utilized in the back-end. Additionally, it may still be necessary to contact the tool companies to obtain confirmation regarding data privacy to further mitigate the risk of data leakage.

5.6.2 Ethical Use of AI in Software Development

Developers of AI-powered testing tools should adhere to ethical guidelines that consider the impact of these technologies on all stakeholders, including software developers, firms, and end-users. Moreover, informed consent should be obtained from all parties involved before these tools are deployed, particularly in contexts where they access and interact with private codebases.

5.6.3 Impact on Employment

In our thesis, we explore the social implications of AI-based testing tools, particularly focusing on the idea that these tools will not replace developers but rather enhance their capabilities.

By reducing the time required to implement unit tests, AI-based tools enable developers to concentrate more on core development tasks or allocate time to acquiring new knowledge and skills. This shift potentially fosters a more dynamic learning environment where developers can improve their testing proficiency by studying and refining the generated unit tests. Thus, AI-driven testing tools not only streamline the development process but also promote continuous learning and skill enhancement among developers.

Chapter 6
Conclusion

AITT advances software development by generating unit tests with a lower learning curve and faster preparation, though test quality varies. While MI unit testing currently produces higher-quality tests, it requires more effort. A hybrid approach combining AI-generated tests with manual adjustments is recommended for optimal results. The research highlights AITT's potential and suggests further investigation into hybrid methods and extending evaluations to various languages and frameworks.

In software development, there is a wide range of existing AITT to help developers generate unit tests. Since there are not enough studies on the AI-based tools' performance, this lack of knowledge might make companies hesitant to use the tools, leading them to MI unit tests instead.

Our evaluation of MI unit tests (RQ1) shows they are of high quality, with quick setup but variable implementation times depending on developers' familiarity and expertise. The tests are structured, accurate, and maintainable, with good coverage that could benefit from more sad tests. Overall, the tests effectively cover relevant scenarios and offer rapid feedback for sustained development.

Our evaluation of unit tests generated by AITT (RQ2) reveals several key insights. The tools have a low learning curve, since they are supported by extensive documentation, and are efficient in test preparation and execution. Most tools follow good unit test practices, though some need improvement. While the accuracy of generated tests varies, coverage is generally sufficient. Maintenance is facilitated by organized code and clear naming, despite potential issues with code size and redundancy. Test case quality varies, with some tools producing reliable tests and others less so. Overall, AITT significantly aid developers by simplifying and expediting unit testing, especially with minor manual adjustments and skillful use of the tools to enhance test accuracy, coverage, and quality.

The comparison study comparing AITT and MI methods found that MI unit testing is currently superior. Tools like UnitTestAI and CodePal are close to matching MI unit tests and can

provide a comprehensive unit test suite with few modifications. Conversely, IKEA GPT, Codiumate, and PaLM, while not matching MI quality, can accelerate the testing process by generating initial test suites that need further refinement. GitHub Copilot and TabNine require improvements to produce reliable tests.

We addressed challenges in using AITT (RQ3), such as data type issues and redundancy in generated unit tests with some tools, and the complexity of obtaining organizational approval and privacy checks for proper AITT usage.

We recommend a hybrid approach, combining AI-generated unit tests with manual modifications and monitoring, to ensure efficiency, effectiveness, and high quality. For front-end development without sensitive data, we recommend using UnitTestsAI and CodePal with manual modifications. For back-end development with sensitive data, we recommend using IKEA GPT and Codiumate with manual modifications. This enhances testing efficiency, quality, and potentially reduces costs, and accelerates release cycles.

Our findings suggest future research in evaluating the selected AITT across different programming languages to assess their versatility and adaptability to diverse development contexts. Another future research is investigating hybrid testing approaches using the selected AITT. This could include a detailed analysis and evaluation of scenarios where a hybrid approach yields optimal testing efficiency and effectiveness.

Overall, this thesis explores the impact of the AITT on unit testing methods in software development.

References

[1] Unit-tests writer. `https://codepal.ai/unit-tests-writer#qna_how_can_machine_learning_be_used_to_generate_unit_tests`. Accessed: 2024-03-23.

[2] Pavel Anselmo Alvarez, Alessio Ishizaka, and Luis Martínez. Multiple-criteria decision-making sorting methods: A survey. *Expert Systems with Applications*, 183:115368, 2021.

[3] Carina Andersson. Exploring the software verification and validation process with focus on efficient fault detection. 2003.

[4] Cyrille Artho and Armin Biere. Advanced unit testing: How to scale up a unit test framework. In *Proceedings of the 2006 international workshop on Automation of software test*, pages 92–98, 2006.

[5] Martin Aruldoss, T Miranda Lakshmi, and V Prasanna Venkatesan. A survey on multi criteria decision making methods and its applications. *American Journal of Information Systems*, 1(1):31–43, 2013.

[6] Yatin Bajaj and Manoj Kumar Samal. Accelerating software quality: Unleashing the power of generative ai for automated test-case generation and bug identification. *International Journal for Research in Applied Science and Engineering Technology*, 11(7), 2023.

[7] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103, 2007.

[8] David Bowes, Tracy Hall, Jean Petric, Thomas Shippey, and Burak Turhan. How good are my tests? In *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 9–14. IEEE, 2017.

[9] Kevin Buffardi, Pedro Valdivia, and Destiny Rogers. Measuring unit test accuracy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 578–584, 2019.

[10] CodePal. Codepal – ai code generator review. https://toolseeker.ai/codepal-ai-code-generator-review/. Accessed: 2024-03-23.

[11] CodePal. Codepal - terms of use. https://codepal.ai/terms, 2023. Accessed: 2024-05-18.

[12] CodiumAI. Codiumate - code, test and review with confidence - by codiumai. https://plugins.jetbrains.com/plugin/21206-codiumate--code-test-and-review-with-confidence--by-codiumai. Accessed: 2024-03-19.

[13] CodiumAI Team. Elevating machine learning code quality: The codium ai advantage. https://www.codium.ai/blog/elevating-machine-learning-code-quality-the-codium-ai-advantage/. Accessed: 2024-03-19.

[14] Codiumate. Codiumate. https://www.codium.ai/. Accessed: 2024-03-19.

[15] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211. IEEE, 2014.

[16] Triantaphyllou Evangelos. *Multi-criteria decision-making methods*. Springer, 2000.

[17] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[18] Gadi Zimerman. Codiumai security: Our commitment to data privacy and security. https://www.codium.ai/blog/codiumai-security-our-commitment-to-data-privacy-and-security/, 2023. Accessed: 2024-03-19.

[19] Jerry Gao, Chuanqi Tao, Dou Jie, and Shengqiang Lu. What is ai software testing? and why. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 27–2709. IEEE, 2019.

[20] Sourodip Ghosh, Ahana Bandyopadhyay, Shreya Sahay, Richik Ghosh, Ishita Kundu, and K.C. Santosh. Colorectal histology tumor detection using ensemble deep neural network. *Engineering Applications of Artificial Intelligence*, 100:104202, 2021.

[21] GitHub. About github copilot. https://docs.github.com/en/copilot/about-github-copilot, 2023. Accessed: 2024-04-02.

[22] GitHub. About github copilot chat. https://docs.github.com/en/copilot/github-copilot-chat/about-github-copilot-chat, 2023. Accessed: 2024-04-02.

[23] GitHub. About github copilot individua. https://docs.github.com/en/copilot/copilot-individual/about-github-copilot-individual, 2023. Accessed: 2024-04-02.

[24] GitHub. Getting started with github copilot. https://docs.github.com/en/copilot/using-github-copilot/getting-started-with-github-copilot, 2023. Accessed: 2024-04-02.

[25] GitHub. Github copilot introduction. https://docs.github.com/en/copilot/quickstart, 2023. Accessed: 2024-04-02.

[26] IKEA. Ikea gpt. Internal website, accessible only to IKEA employees, 2024.

[27] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. Code coverage at google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[28] Sukaina Izzat and Nada N Saleem. Software testing techniques and tools: A review. *Journal of Education and Science*, 32(2):30–44, 2023.

[29] Janine Heinrichs . Codepal review: Can i instantly generate code with ai? https://www.unite.ai/codepal-review/. Accessed: 2024-03-23.

[30] Paul C Jorgensen. *Software testing: a craftsman's approach, Third Edition*. Auerbach Publications, pages 3–12, 2013.

[31] Zubair Khaliq, Sheikh Umar Farooq, and Dawood Ashraf Khan. Artificial intelligence in software testing: Impact, problems, challenges and prospect. *arXiv preprint arXiv:2201.05371*, 2022.

[32] Z Khaliqa and S Farooqa. Artificial intelligence in software testing: Impact, problems, challenges and prospect. 10.48550. *arXiv preprint arxiv.2201.05371*, 2022.

[33] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[34] Mengyun Liu and K. Chakrabarty. Adaptive methods for machine learning-based testing of integrated circuits and boards. *2021 IEEE International Test Conference (ITC)*, pages 153–162, 2021.

[35] Pan Liu, Zhenning Xu, and Jun Ai. An approach to automatic test case generation for unit testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 545–552, 2018.

[36] Lucidspark. An introduction to the weighted decision matrix. https://lucidspark.com/blog/weighted-decision-matrix#:~:text=In%20a%20weighted%20decision%20matrix,criteria%20are%20of%20equal%20importance. Accessed: 2024-05-13.

[37] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[38] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[39] mspoweruser . Codepal review: Is it the best all-in-one ai coding solution? https://mspoweruser.com/codepal-review/. Accessed: 2024-05-19.

[40] Robert E Noonan and Richard H Prosl. Unit testing frameworks. *ACM SIGCSE Bulletin*, 34(1):232–236, 2002.

[41] Phuoc Pham, Vu Nguyen, and Tien Nguyen. A review of ai-augmented end-to-end test automation tools. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[42] Adam Porter, Cemal Yilmaz, Atif M Memon, Douglas C Schmidt, and Bala Natarajan. Skoll: A process and infrastructure for distributed continuous quality assurance. *IEEE Transactions on Software Engineering*, 33(8):510–525, 2007.

[43] Ramona Schwering, Jecelyn Yeen. Four common types of code coverage. https://web.dev/articles/ta-code-coverage. Accessed: 2024-05-07.

[44] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006.

[45] Thomas Saaty, Luis Vargas, and Cahyono St. *The Analytic Hierarchy Process*. 07 2022.

[46] Katharine Sanderson. Gpt-4 is here: what scientists think. *Nature*, 615(7954):773, 2023.

[47] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE, 2019.

[48] Mohamed Ali Shajahan. Fault tolerance and reliability in autosar stack development: Redundancy and error handling strategies. *Technology & Management Review*, 3(1):27–45, 2018.

[49] Karuturi Sneha and Gowda M Malle. Research on software testing techniques and software automation testing tools. In *2017 international conference on energy, communication, data analytics and soft computing (ICECDS)*, pages 77–81. IEEE, 2017.

[50] Jeremy Straub and Justin Huber. A characterization of the utility of using artificial intelligence to test two artificial intelligence systems. *Computers*, 2(2):67–87, 2013.

[51] Tabnine. The ai coding assistant that you control. https://www.tabnine.com/. Accessed: 2024-03-19.

[52] Tabnine. How tabnine protects your code privacy. https://www.tabnine.com/code-privacy. Accessed: 2024-03-19.

[53] Tabnine. Tabnine: AI Code Completion & Chat in Java JS/TS Python & More. https://plugins.jetbrains.com/plugin/12798-tabnine-ai-code-completion--chat-in-java-js-ts-python--more. Accessed: 2024-03-19.

[54] Hamed Taherdoost and Mitra Madanchian. Multi-criteria decision making (mcdm) methods and concepts. *Encyclopedia*, 3(1):77–87, 2023.

[55] UnitTestAI. Unittestai - ai powered unit tests generator. `https://marketplace.visualstudio.com/items?itemName=PatersonAnaccius.unittestai`. Accessed: 2024-03-19.

[56] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*, pages 194–212, 2012.

[57] Tao Xie, Nikolai Tillmann, and Pratap Lakshman. Advances in unit testing: theory and practice. In *Proceedings of the 38th international conference on software engineering companion*, pages 904–905, 2016.

[58] Sridhar Reddy Yerram, Suman Reddy Mallipeddi, Aleena Varghese, and Arun Kumar Sandu. Human-centered software development: Integrating user experience (ux) design and agile methodologies for enhanced product quality. *Asian Journal of Humanity, Art and Literature*, 6(2):203–218, 2019.

[59] Zazmic. Palm api unit test generator. `https://marketplace.visualstudio.com/items?itemName=Zazmic.palm-api-test-generator`. Accessed: 2024-03-19.

Appendices

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2024-06-13

EXAMENSARBETE

STUDENTER Emad Aldeen Issawi, Osama Hajjouz
HANDLEDARE Qunying Song (LTH)
EXAMINATOR Elizabeth Bjarnason (LTH)

Unit Testing in the Age of AI

POPULÄRVETENSKAPLIG SAMMANFATTNING **Emad Aldeen Issawi, Osama Hajjouz**

As web development grows increasingly complex, ensuring software quality has become a key challenge. This thesis explores how AI-driven tools can revolutionize unit testing—a vital step in software development—by comparing these tools with manually implemented testing method within the context of IKEA's ILA web application.

The shift toward AI solutions in companies like IKEA requires that software not only functions correctly but also efficiently. Manual testing methods, while reliable, are often time-consuming. AI-powered tools promise faster and potentially more thorough testing processes. In this study, seven different AI-based unit testing tools were evaluated and compared against the manually implemented testing method based on 19 criteria, including efficiency, quality, accuracy, and test coverage. We found that AI-based testing tools offer the potential to speed up the unit testing process by automating the generation of unit tests. While they excel in covering scenarios, some do not always match the precision or depth provided by tests created manually by developers, who can leverage intimate system knowledge. Surprisingly, one of the AI tools tested was able to generate unit tests of nearly the same quality as those manually implemented.

The evaluation insights from this research could guide software development teams on how to integrate AI-based testing tools effectively with existing testing protocols. This could lead to a hybrid testing strategy, combining the speed of AI tools with the meticulousness of manual testing. In the box below, you can find the tools that were evaluated. Maybe you want to try one? This thesis not only benchmarks the current capabilities of AI-based testing tools in enhancing software testing but also highlights the nuanced balance needed between automated efficiency and human expertise, paving the way for faster software development.