InfoTech Spectrum: Iraqi Journal of Data Science (IJDS)
ISSN: 3007-5467, Vol 3(No.1), 58-73 (2026)
DOI: https://doi.org/10.51173/ijds.v3i1.63
https://ijds.mtu.edu.iq

EVALUATING TESTABILITY-DRIVEN DEVELOPMENT (TsDD) VERSUS TEST-DRIVEN DEVELOPMENT (TDD)IN SOFTWARE QUALITY

Sahar A. Hussein Altaee [1], Saeed Parsa [1*]

[1] *Department of Computer Science, Iran University of Science and Technology, Tehran, Iran*

** Corresponding author E-mail:* parsa@iust.ac.ir (Saeed Parsa)

RESEARCH ARTICLE

ARTICLE INFORMATION

SUBMISSION HISTORY:
Received: 12 September 2025
Revised: 14 October 2025
Accepted: 20 October 2025
Published: 30 January 2026

KEYWORDS:

Testability-Driven Development;
Test-Driven Development;
Software Quality;
Software Engineering;

ABSTRACT

Testability, as an essential property, plays a crucial role in software quality and testing techniques. In this work, we are conducting a comparative study of TDD and TsDD. Whereas TDD focuses on generating unit tests before implementing the code, TsDD aims to evaluate the testability of code before running static analysis techniques and changing the design before carrying out test execution. This study is quantitative and focuses on collecting and analysing numerical data, using code coverage (CC) and testing timeframe as two important criteria for evaluating software development and testing frameworks. The impact of this approach on software quality is measured using advanced data testing instruments, monitoring code coverage, and testing time trade-offs across three open-source software projects. Results show that code coverage is significantly improved and that the software becomes more testable, with TseDD outperforming TDDe by 14.20% on testability and 12.54% on code coverage. This underscores the significance of this technique as a successful approach to improving software quality and streamlining development processes.

1. INTRODUCTION

The last 50 years have introduced many changes to software development methodologies, making it challenging to select the right method. Being aware of empirical evidence is important for the pragmatic assessment of Agile practice. The rise and promulgation of Agile development over the past 15 years have highlighted the importance of understanding its elements, relationships, and empirical support[1], [2]. The complexity of software engineering is not without hazard; there may be delays in delivery and computer virus-ridden releases due to a shortage of time, which can lead to client loss. For successful outcomes, rigid IT solutions might not be enough, as shelf products may not be flexible either. Software quality and application testing are vital to the success of software projects. TDD enhances code quality by requiring tests before coding. But with the growing complexity of testing and maintenance, the conditions for testing performance are even more demanding. To conquer these challenges, consider ability-driven development (TsDD) as a practice that specialises in comparing and enhancing code testability, advancing testing strategies that are greener and more effective, thereby improving software quality and maintainability [3], [4]. Software testing is vital to software program improvement, ensuring quality and overall performance by detecting defects. Agile methodologies, such as Scrum, Kanban, XP , and Lean, focus on iterative development, collaboration, and quick feedback. These methods also promote close cooperation among builders, testers, and stakeholders in a dynamic, responsive growth process [5], [6]. The main goal of this paper is to compare test-driven development (TDD) with test-driven development (TDD) in terms of software quality evaluation.

DOI: https://doi.org/10.51173/ijds.v3i1.63

2. LITERATURE REVIEW

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle. Such a process, iteratively refined and developed, is intended to deliver high-quality software. TDD helps communicate intent, provide quick feedback, write modular and decoupled code, and preserve existing functionality, thereby affecting software engineering practices[7]. Testability-driven development (TsDD) is a methodology to address the complexity and challenges of software testing. Recognising that programming is not just about writing code but also involves a significant investment in setting up and checking out, especially in areas such as cyber-physical and clinical software, TsDD argues for a prevention-focused testing approach. By focusing on early code testability, this approach aims to address the challenges posed by expensive, labour-intensive testing.

Testability-driven development (TsDD) is a paradigm that aims to address the limitations and issues in software testing. The approach is based on the observation that software testing accounts for a significant portion of development effort, particularly when developing cyber-physical and medical software. It acknowledges that testing can be costly and labour-intensive. To address these issues, this approach promotes proactive checking out by ensuring code testability long before checkout begins [8], [9]. In TsDD, engineering magnificence and problem diagrams are translated immediately into executable code, so that test training can be generated for each level of gear use, similar to JUnit. Code is created, checked, and prepared even before complete execution, which implies continuous evaluation and improvement of code testability throughout development. Test suites are prepared by hand or by a robot, and successful tests result in small code additions that meet the gadget's needs. TsDD style consists of adding a test, fast, watching it fails, and refactoring to make the code testable. The TsDD slogan focuses on efficient test creation, execution, and validation by introducing testability measures at some stage of the software development process [10]. TsDD encourages value and effort trading in code reviews, which promotes refactoring for testability. The computerisation of size and improvements in testability may be a major win in fostering more green and effective testing practices. Code screenings are essential to improve code quality by identifying potential issues early. It is important to evaluate their effectiveness as a wear-out empirical study and using linguistic literatures are necessary. This generally involves evaluating development strategies based purely on quality attributes such as defect density, code coverage, maintainability and time/effort. Studying look at effects offers treasured insights into how code examination practices can decrease time, effort, and costs while at the same time improving code quality [11].

Shekhar [12] implemented Agile to enhance the software program, supporting improvement and testing. They highlighted its benefits in planning, time management, and aligning patron checkout of belongings with the agile crew. Agile testing, tested effectively across small and large industries, enables automated test improvement alongside code development on the same set of requirements. The agility to adapt to evolving customer needs is a key advantage, boosting morale, confidence, and satisfaction among developers and testers. Proper planning and alertness throughout the venture ensure productivity, coordination, and a high level of confidence in the final product. Test-driven and conduct-driven development strategies have become essential practices for improving code quality. However, their blessings can lead to slower implementation due to more trial-and-error attempts and demanding situations in adoption within companies.

Recent studies have sought to quantify the consequences of these strategies on code fine, implementation pace, and organisational adoption hurdles [13], [14], [15]. Testing software for complex systems, such as cyber-physical and clinical software, can be time-consuming and in-depth. The TsDD methodology addresses this by prioritising code refactoring for enhanced testability before writing tests. By deferring testing until the code is strong, TsDD enables more efficient testing, offering an iterative approach to continuous code development. TsDD aims to reduce the number of times it is tried and the cost by delaying it until the code is ready for optimal testability [16], [17] . Arya and Khan [18] conducted a systematic literature review of the testability of object-

oriented software structures, analysing 118 research papers. The assessment highlights elements such as layout patterns, code complexity, and the use of frameworks that affect testability. The authors introduce a testability assessment framework for evaluating software structures, providing a valuable tool for enhancing the testability of object-oriented systems. Gurbuz and Tekinerdogan [19] proposed a method to investigate the evolution of model-based testing on the integrity verification of software. A multi-stage approach to the selection process, from 751 research papers to 36 primary studies, is used to determine major trends and methodologies, as well as barriers. The study shows that model-based testing offers significant advantages for software verification. Trunzer et al., in [20] described the results and experiences from implementing telecommunication systems within a joint project of three Fraunhofer institutes: the Model-Driven Telecommunication Systems (MDTS) development project. The key objective of the project was to formalise a model-driven software development process to support integration between system development and testing. The realisation of a supporting tool-kit, which should be used to automate the described process was part of the project. The resulting concepts were implemented in the field of telecommunications as a prototype application. Exciting things are also being made possible by this application in mission-critical industrial environments such as aircraft maintenance or chemical manufacturing. As the amount of code considered increases, the learning costs increase significantly. Test-pushed improvement (TDD) and behavior-driven development (BDD) make a contribution to this push by the use of failed tests driving code layout and implementation choices.

Parsa [21] proposed testability-driven development (TsDD), which iteratively assesses and improves testability through refactoring. Test Driven Development with TsDD TsDD uses "test first" approach for coding and development. Zakeri in [16] presents a research on the relation between software testability and test coverage. They proposed a model based on machine learning to predict code coverages using source codes metrics, which may reduce additional testing costs. The results of the experiment presented an 81.94% precision in estimating and predicting testability of software. Zakeri and Parsa [22] introduced TsDD as an innovative software development method to overcome the faced obstacles of TDD. TsDD has testing at a few level of the development process as an agenda, additionally, unlike TDD which typically specializes on unit checking out, TsDD nonetheless recurrently advocates “take a look at all the issues”. This paper describes the strategy and its benefits and demonstrates its effectiveness via a case study. TsDD offers a valuable approach to supplement existing approaches, particularly TDD.

Test-Driven Development (TDD) leads development with automated testing, which increases code exceptional and supports refactoring. However test times are very difficult to keep, even in simple projects. Testability refactoring makes code easier to test. Nasrabadi et al. [23] ntroduced Testability-Driven Development (TsDD) that emphasizes testability driven coding. TsDD enhanced testability by 77.81% over 50 Java classes. 96 Experts validated that TsDD speeds up testdriven development. Ghafari et al. [24]reviewed TDD papers posted in top medical journals and identified 5 types of factors that directly affect the impacts of TDD studies. Their results can help inform researchers how to better conduct more dependable research and aid practitioners in understanding the risks when consulting TDD research. Ivo et al. [25] presented an approach to apply Test-Driven Development (TDD) to stochas-tic algorithm development, and revealed the Random Share Testing Framework (ReTest), which is a JUnit extension that makes such possible. Their trial- and developer study confirmed the method's practicability for stochastic software development with TDD, indicating the helpfulness/user-friendliness of the ReTest framework. Mäkinen and Münch [26] conducted a study indicating that test-driven development can reduce defects and lead to more maintainable code. Their research also suggests that some parts of the code may be smaller and less complex. Fucci et al [27] investigated the correlation between incoherence metrics and testability in object-orientated structures. Concentrating on unit testing, they empirically analyzed records from two Java structures with JUnit check instances. The check effects offer evidence of a hyperlink among incoherence and take a look at capacity. Table 1 represents the most important results of previous work on testing methodologies.

Table 1. Important result of previous work on testing methodologies

Methodology	Description	Key Features	Advantages	Challenges	Author
Test-Driven Development (TDD)	Writing tests before code, then developing code to pass tests, with continuous refinement of tests and code.	Clarifies expected behaviour, provides immediate feedback, supports modular, loosely coupled code, and ensures regression safety.	Improves code quality, simplifies refactoring, and ensures functionality.	It can slow down development and requires a culture change.	Shekhar et al.
Testability-Driven Development (TsDD)	Focus on making code testable from the start with continuous testability improvement and refactoring.	Code made test-ready early, continuous testability improvements, and manual and automated test suites.	Reduces testing effort and time, enhances automation, and lowers cost.	Complex implementation, fewer dedicated researchers.	Parsa et al.
Agile Testing	Testing is integrated into iterative Agile development cycles.	Continuous testing, adapts to changing requirements.	Aligns with rapidly changing needs, supports automation.	Requires skilled teams and coordination challenges.	Arya and Khan
Behaviour-Driven Development (BDD)	Extends TDD focusing on behavior and collaboration with stakeholders.	Uses natural-language specifications and bridges business and tech teams.	Improves communication and clear requirements.	Learning curve, depends on stakeholder involvement.	Parsa et al.

The exam of Test-Driven Development (TDD) and Testable-Driven Development (TsDD) is well-known and shows considerable benefits; however, quantifying those benefits is hard due to implementation complexities. Thorough research is essential to identify the strengths and weaknesses of these methodologies and develop methods to enhance TsDD. The shortage of dedicated researchers in TsDD underscores the need for additional research (Table 1. Comparing TDD and TsDD, code insurance metrics help verify the effectiveness of checks; higher insurance costs often indicate improved test coverage. Analysing open-source obligations using each methodology can offer valuable insights to developers and testers. The research method consists of evaluating TDD and TsDD across both the SF110 splendour set and three open-source projects using EvoSuite Test Builder. The goal of this test is to flex trying out expertise, address experimental evaluation of annoying scenarios, and control actual international facts acquisition. It particularly focuses on a comparative analysis of the effectiveness of Test-Driven Development (TDD) vs. Testability-Driven Development (TsDD) in terms of time, effort, and value discount. By leveraging a strict empirical comparative approach, the work aims to provide tangible evidence to help firms decide whether to adopt these technologies in line with their own objectives and constraints. This study fills an important research gap, presenting TsDD as a novel methodology and highlighting the need to work with the full dataset to validate the credibility and relevance of its outputs. The novelty of this work lies in the comprehensive comparison between the TsDD and TDD approaches.

3. METHODS AND MATERIALS

3.1. Methodology

In this paper, we compare two test-driven development (TDD) approaches: TDD with code insurance metrics to assess checkout effectiveness. Good code insurance is essential for making fine software even better. The report presents an assessment of 3 open-source tasks that investigated the adoption of JUS TDD and TSD, and offers some guidance on when to choose one method for specific tasks and aims. We have selected three open-source projects (commons-codec, Water Simulation, and JSON202011115) based on database size, test suite availability, development language, and compatibility with our research objectives. For TsDD, the focus is on:

- The static code analysis with a couple of fine metrics on testability.
- Automatic re-factorisation of the code, mostly according to evaluation to attain clarity and simplicity.
- With a predominantly manually developed automated test generator that focuses on elements with low testability.

This method differs from traditional TDD, where unit tests are written manually or semi-automatically before the code. The check additionally contrasts TDD (Test-Driven Development) with TsDD (Test-Driven Development) methodology, posing four questions that highlight their differences, advantages, and challenges in practical software development projects. The comparison arises absolutely on the queries:

- Q1- s the software program higher in Testability-Driven development (TsDD) than in Test-driven development (TDD)?
- Q2- What are the variations in software program quality, improvement time, and developer productivity between TsDD and TDD?
- Q3- What are the basic ideas and procedures of TDD, and TsDD, and how do they differ?
- Q4- What are the potential advantages and drawbacks of implementing TsDD in comparison to TDD in software initiatives?

By answering these questions, a comprehensive analysis of the TsDD methodology and its comparison with TDD is provided.

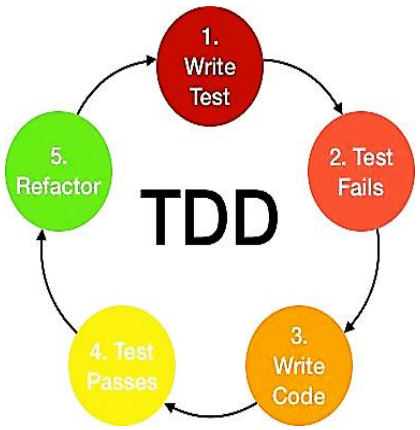

Figure 1. 5 steps to conduct a TDD test [22]

3.2. The TDD And TsDD Approach

Test-Driven Development (TDD) encourages continuous testing and refactoring in software development. TDD involves including test cases, running tests to verify failures, writing code to skip

tests, retesting, and repeating the process for each new capability (see Fig. 1). This iterative approach ensures thorough testing, reliable code, and continuous improvement, focusing on delivering high-quality software [28]. Test-Driven Development (TDD) encourages continuous testing and refactoring in software development. TDD involves adding test cases, writing code to skip tests, retesting, and repeating the process for each new capability. This iterative method guarantees thorough testing, reliable code, and continuous improvement, specialising in delivering terrific software.

In Test-Driven Development (TDD), developers start by writing a failing test that defines the expected behaviour, then write just enough code to pass the test. After passing the test, the code can be refactored to enhance structure while maintaining functionality. This iterative technique, acknowledged for its "purple-inexperienced-refactor" cycle, includes writing failing assessments, coding to bypass them, and optimising design via refactoring, see Fig.2. TDD aims to improve performance, scalability, and clarity by using an iterative approach to incrementally improve the solution while ensuring that programmers are assured in writing tests first and continuously improving code.

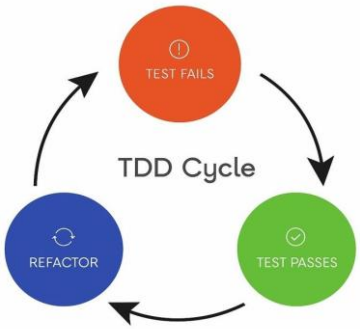

Figure 2. TDD Live cycle [22]

Testability-Driven Development (TsDD) specialises in code refactoring to enhance testability, improve testing practices, and deliver high-quality software. By prioritising testability, conducting iterative testing, and making critical changes, TsDD streamlines development, lowers costs, and enables continuous improvement. This technique delays testing until code is optimised and uses automated testing tools for quicker, simpler testing. In scientific software development, systematic testing is crucial for error detection and high confidence, especially in critical applications like climate modelling. TDD and TsDD are key in presenting feedback, improving communication, assisting choice-making, and improving programming practices, see Fig.3. These practices increase the importance of coding techniques in improving code quality, speeding up development, identifying issues as soon as possible, developing arounds-the-clock improvement and enhancing teamwork for very good software production and multiplied development productivity [29], [30].

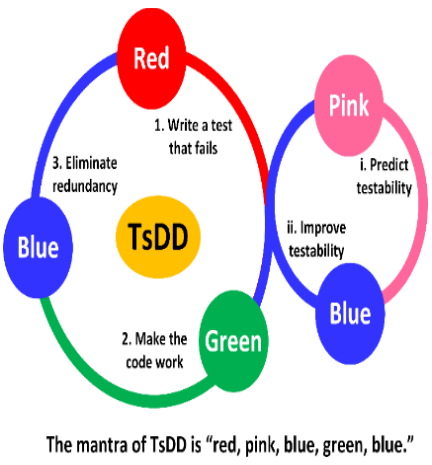

Figure 3. TSDD-based software development [22]

3.3. Theoretical Aspect

The objective of unit testing is to test small, cohesive units such as classes and functions, thereby minimising debugging time while also making your code more readable. More advanced techniques, like ATDD (Acceptance Test-Driven Development), facilitate automating customers' requirements through acceptance tests. The above is due to Test-Driven Development (TDD)'s insistence on important behaviour and disregard for all other trivial details. Unit tests are structured with a "when-then" syntax and must be of Kent Beck quality level. Such best practices include the following: writing small, clear, focused tests that are easy to maintain for effective/reliable testing [34]. Clean code necessitates clean tests. Just as an artisan will take time and care in sanding all parts of a physical product, so too must software engineers do the same with production and test code. A good test is readable, maintainable, and self-contained, revealing the most important properties without outside knowledge of the program (but not necessarily without preparation!). Having high-quality test standards is important; however, they will help avoid maintenance hurdles by identifying testing pitfalls written into production code. Clean tests should be fast, independent (one does not depend on another), repeatable (they return the same result from indifferent environments), self-validating (a test is not required its developer interference to pass or fail) and trustworthy; each unit test is concerned with one concept and has only one assertion [31], [32]. Testing Test-Driven Development (TDD) is vital to ensuring code stays great, and you never accumulate technical debt. It consists of improving code structure without changing behaviour, ensuring full code coverage, and gradually increasing maintainability over time [33]. In TDD and ATDD, checks are coded into test cases that aim to reach an accepting state. These examples build test suites for specific cases. Organised suites comprise meaningful units, where each and every test reflects an entirely distinct scenario for full coverage and clarity. Numerous tools facilitate Test-Driven Development (TDD) by simplifying test creation, execution, and management. Testing frameworks such as JUnit, pytest, RSpec, Jasmine, Selenium, Cypress, Playwright, and Robot Framework are popular choices that each offer specific functionality for efficient testing.

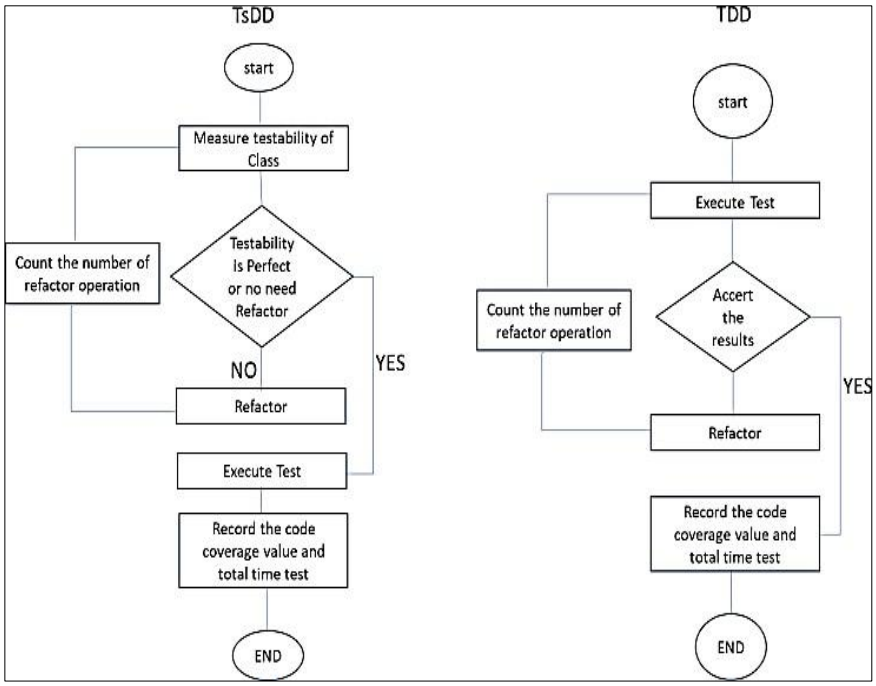

Figure 4. Practice the procedure for applying both methodologies.

For example: JUnit: Commonly used with Java, it provides annotations, assertions, and support for test suites and runners. NUnit: Designed for the NET atmosphere, supplying test fixtures, assertions, and test runners for languages like C#. These frameworks facilitate test execution, result tracking, and standard TDD implementation [34]. Automated testing tools like EvoSuite can

effectively gather data on software performance. By running computerised assessments on projects like commons-codec, Water Simulation, and JSON202011115, containing 141 Java training, we can compare methodologies. Initially, computerised selection with TDD was completed, followed by the TsDD methodology, in which testability was expected for the CodeArt project. Refactoring was then performed with a focus on different aspects, as depicted in Fig. 4.

In that work, only Java class refactoring was considered, an important software development activity aimed at restructuring code to enhance a system's design, readability, maintainability, and performance without affecting its functionality. Refactoring techniques served as tools to detect opportunities for code improvement, leading to changes that increased the application's quality. Refactoring tasks included method extraction, elimination of code duplication, better class organisation, and removal of dead code, as shown in Table 2.

Table 2: Code smells discovered from the SonarLint tool and refactoring operations suggestions.

Code Smell	Refactors Operation
Local variables should not be declared and then immediately returned or thrown.	Immediately return this expression instead of assigning it to the temporary variable "y."
Cognitive Complexity of methods should not be too high	Extract method Splitting a conditional statement Identify Repeated Code Introduce Variable
Sections of code should not be commented out	Remove Dead Code
Add a nested comment explaining why this method is empty, throw an exception, or complete the implementation.	Remove Dead Code
Methods should not return constants.	Extract constant
Move method	"Private" methods called only by inner Classes should be moved to those classes
Return of Boolean expressions should not be wrapped in an "if-then-else" statement.	Simplify the Boolean expression.
Methods should not have too many parameters	Extract method
Modifiers should be declared in the correct order	Reformat code
Return of Boolean expressions should not be wrapped in an "if-then-else" statement	Simplify the Boolean expression
Switch statements should have "default" clauses	Add Default Clause
Deprecated code should be removed.	Remove it
Class variable fields should not have Public Accessibility	Encapsulate Field
The Methods should not be empty.	Remove Or Replace Empty Method

TsDD aims to improve testability during software development by combining machine learning and software metrics to predict and improve testability. This practice reduces the amount of code our tests need to check before we've actually started testing, and it smooths out the overall testing flow. Being endorsed through capabilities, including test insurance and test case count relative to the testing budget, TsDD also improves testability for new and legacy codebases, resulting in higher software quality. The SF110 test set contains 23,886 Java statements from 110 projects and is also used to predict testability. Each elegance is converted into an attribute vector by extracting code metrics via static analysis that capture many aspects of code form and complexity. This corpus provides classes and metrics to produce the feature vectors. To predict software testability, regression models are trained and evaluated using these vectors, along with known testability values [29]. The testability estimation process has two main stages: learning and inference (Fig. 5).

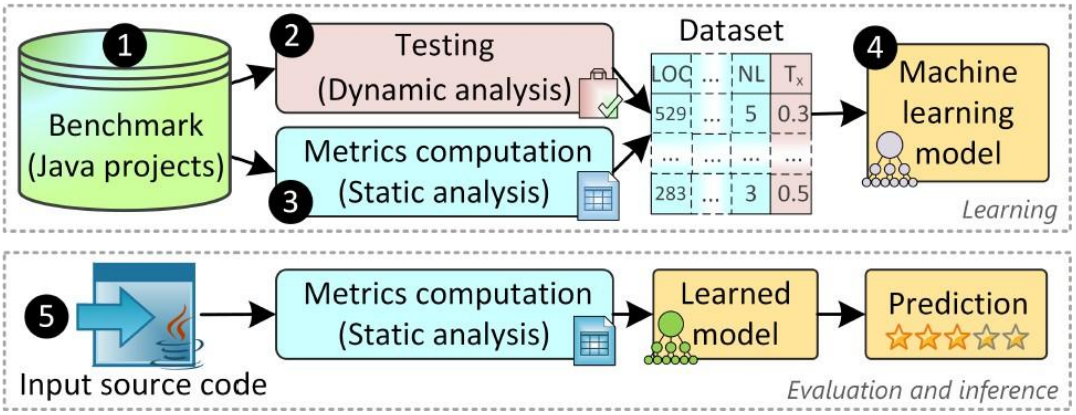

Figure 5. Testability in prediction [23]

3.4. Experiments

Software testing plays an important role not only in checking the quality of software, ensuring it runs error-free, meeting requirements, and rating performance, but also in improving efficiency, accuracy, and usability. The choice was to download three projects with good general coverage of an extensive code base from both Test-Driven Development (TDD) and Testability-driven Development (TsDD). Evaluation criteria were code quality (code coverage), development time, and virus detection and resolution effectiveness. Automated testing tools, including EvoSuite (whose integration with test automation in the IntelliJ IDE was used above), were used to increase code coverage. SonarLint was great to have, as it pointed out code errors and could help with refactoring. Both approaches were adjusted using the same refactorings to be equated. In this testing environment, software testability is defined as the ease with which its elements can be tested using structures, units, and documentation to demonstrate the testability of their own tests effectively. Error detection within the system is simplified when software is inherently testable, thereby making testing easier. Three complex programs—commons-codec, water-simulator, and JSON202011115 of the sf110 package — were assessed using EvoSuite, a library for producing effective test suites centred on relevant test cases. TsDD recommends reducing the cost and effort of checking out by focusing on refactoring, e.g., to improve testability. An important task in software testing is achieving extensive code coverage, meaning test cases cover a significant portion of the software code to discover bugs and provide high-quality assurance. Coverage is the standard metric for various coverage dimensions, such as statements, branches, and cases, that we monitored throughout the comparison between the two approaches. How far to hide (While larger coverage probabilities make sense, a goal of 100% is fine but difficult to achieve in complex packages [23]).

In the chosen tasks, the Testability-driven Development (TsDD) approach was implemented with CodART's help. CodART was chosen because it automates testability evaluation and, therefore, is an appropriate tool to support TsDD adoption. It provides core functions for analysing code and computing testability; it is seamlessly integrated into the TsDD process and helps improve the quality and architecture of a project. CodART evaluates testability qualitatively using source code metrics, but it produces quantitative ratings for the entire code base. CodART (Source Code Automated Refactoring Toolkit) is a powerful refactoring engine that performs multi-objective software transformation and optimisation. It uses supply code metrics to characterise software program houses, including coverage metrics that analyse test suite execution. Testability CodART uses these metrics in a set of testable prediction models to calculate testability, which includes three classes of measures: code complexity, code coupling , and code coverage. While reading these metrics, CodART provides information and reporting to increase codebase testability, supporting refactoring measures that reduce complexity, decrease coupling, and build insurance, thereby improving software productivity. SonarLint is a sophisticated static code analysis tool that provides live feedback and guidance on how to improve your code, best via green refactoring. It identifies and resolves code-related issues through thorough code analysis, pinpointing bugs, code smells,

security vulnerabilities, and deviations from coding requirements. SonarLint provides instant comments in the Integrated Development Environment (IDE), enabling the detection and resolution of problems and prioritising them by severity.

3.5. Validation

In a water simulation mission, we applied CodeArt to calculate testability improvements resulting from refactoring 32 Java classes. Our results showed an average improvement of 14.21% in testability. To verify our results, we compared them with a previous study of 42 Java classes in which the authors) stated in [16] and [21] a mean of 15.57% improvement in testability. This discrepancy implied that the results were consistent, thereby increasing the reliability of our findings.

4. RESULTS

In this work, we compared the influence of Test-Driven Development (TDD) and Testability-driven Development (TsDD) on the testability of three mature software programs (Commons-Codec, Water-simulator, JSON202011115). We studied the influence on code coverage and overall testing time. Specific results of applying TDD and TsDD on different project types are reported in our GitHub repository [1].

4.1. Code Coverage Effect

The refactoring efforts have resulted in significant improvements in the testability of the code base. The testability of the water simulator increased by 14.20847% after reconstruction using the TsDD methodology. In addition, the TsDD approach had an average increase of 12.56919% in unit test code coverage compared to the TDD approach. It follows that TsDD dominates TDD, thereby further improving software quality. When investigating the 10-Water-Simulator project, we found significant differences between TDD with and without Travis regarding software testability. We have shown that, in both the magnitude of change and the proportional difference in testability, TsDD has made a substantial improvement, followed by refactoring (average testability increased from 0.57 to 0.66). TsDD also confirmed better common refactoring results, with higher scores, showing an increase in testability and software quality. Moreover, line locking, section locking, and mutation point coverage metrics comparisons between TsDD and TDD reported enhancements across all coverage types. More precisely, line coverage exceeded 10.88%, branch coverage improved by over 12.95%, and mutation points increased by at least 13.85%. These results suggest that TsDD gives a significant improvement in code coverage over TDD, as is evident from Fig. 6. That result answers question Q1 directly and makes a strong claim that TsDD outperforms Test-Driven Development in delivering higher software quality.

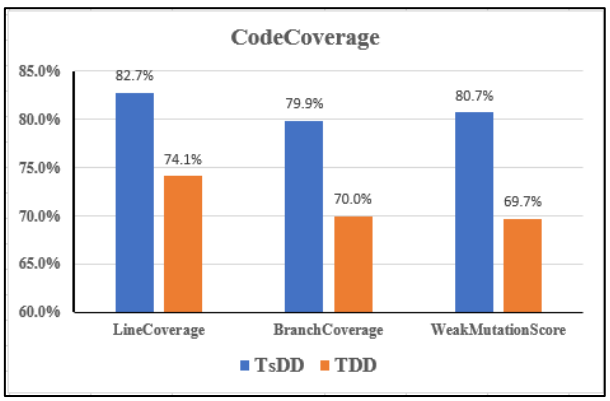

Figure 6. Comparison between methodologies of TDD and TsDD based on Code Coverage.

From the comparative analysis shown in Fig. 7, the code coverage provided by the TsDD technique is, on average, better than that of the TDD technique. The data reveal a 12.54% increase

1- https://github.com/saharamf/DataTDDTsDD

in average code coverage when employing the TsDD approach. This outcome strongly implies that TsDD provides a higher level of code coverage than TDD. Consequently, TsDD enables more thorough testing and has the potential to improve the overall quality of the software.

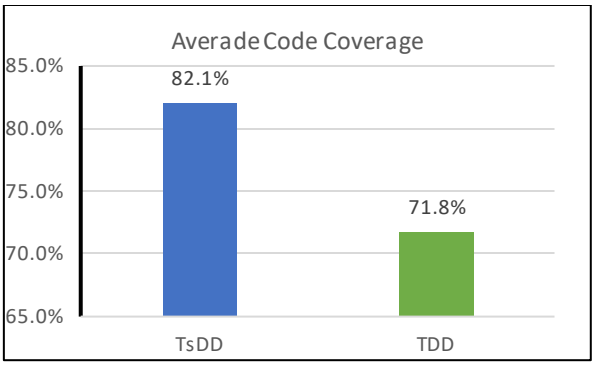

Figure 7. Comparison between the TsDD and TDD methods based on average Code Coverage.

Figs. 8-10, indicate the results of the comparison between the two methodologies based on code coverage metrics for the three projects, respectively.

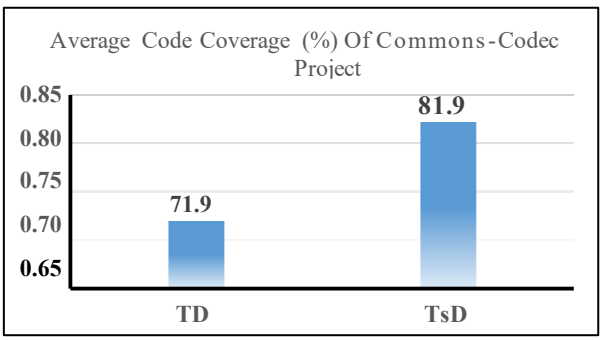

Figure 8. Comparison between TSDD and TDD based on code coverage in the commons-codec project

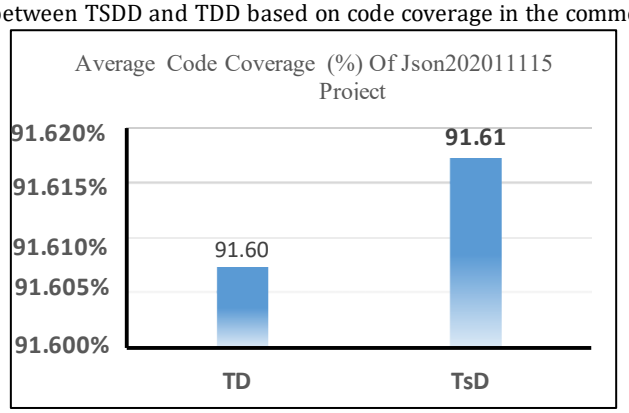

Figure 9. Comparison between TsDD and TDD based on code coverage in the json202011115 project.

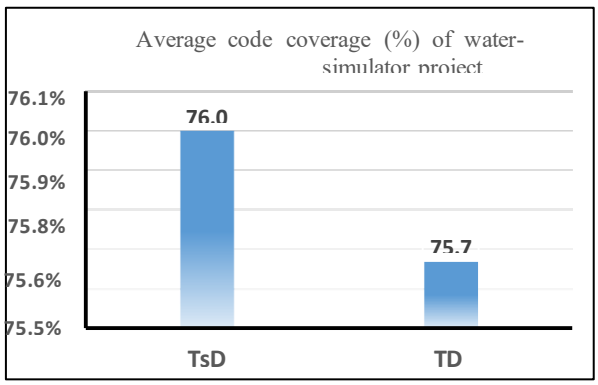

Figure 10. Comparison between TSDD and TDD based on code coverage in the water-simulator

4.2. Total Time Effect

In Figure 11, a comparison is depicted between the TsDD and TDD methodologies concerning total testing time. The Figure shows that the TsDD approach reduces overall testing time compared to TDD. It is worth noting that there is a decrease of about 8.96% in the total ageing time. This observation provides evidence that, thanks to TsDD, we can optimise and shorten time expenditures in our software development work.

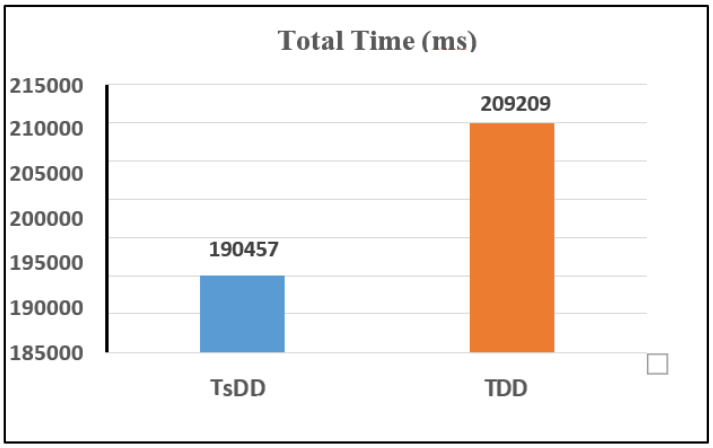

Figure 11. Comparison between TsDD and TDD methods based on total time.

A two-sample t-test was conducted to evaluate the behaviour of the two agencies. The t-value of 7.383464405 indicates a significant difference between the two agencies, supported by a p-value of 2.67585, indicating that this difference is not due to chance. Therefore, there are statistically significant differences between the two groups. To compare total testing time, the t-value is 0.39583951, indicating a slight difference between the two groups' means. The high p-value of 0.712979943 suggests that this difference is unlikely to be due to chance. Therefore, there is no statistically significant difference between the two groups.

4.3. Comparison Side

Test-Driven Development (TDD) vs. Testability-Driven Development (TsDD) Both Test-Driven Development (TDD) and Testability-Driven Development (TsDD) are methodologies that emphasise testing to enhance software quality. Here are the similarities and differences between these procedures:

- Similarities:
  - Both TDD and TsDD advocate writing checks earlier than or along with the improvement of code, guiding the development procedure.
  - Both methodologies prioritise catching errors early within the improvement cycle, reducing the fee and effort of solving problems later within the process.
  - Both TDD and TsDD sell writing code that is maintainable and effortlessly modifiable over time, improving the software program's robustness.
- Differences:
  - TDD concentrates on writing assessments to validate person units of code (unit testing), making sure the correctness of small code additions.
  - TsDD, on the other hand, stresses that all of the machines should be testable. Architecting the system and interfaces with testing in mind leads to testability throughout the device.
  - More sophisticated efforts in designing inherently testable device architecture, involving the overall testability of the devices, are needed for TSDD.
  - More detailed TDD can be executed with more freedom on individual Code Devices without a big architectural overhead.

In sum, TDD emphasises unit-level testing, while TSDD stresses device-level testability through architectural design. Both methods aim to improve software quality by focusing on specific testing levels and providing tailored outcomes for system development. This response addresses

Q3. Ultimately, both TDD and TsDD are efficient methodologies for developing high-quality code that performs well, is robust, maintainable, and scalable. The best method of approach will depend on the specific project requirements and your team's comfort and proficiency level. The advantages of TDD and TSDD are: better code quality, faster development, improved communication, greater trust (a more salmon-pink colour), and easier maintainability. These techniques enable the generation of top-notch code with sufficient performance and sanity, which is important in today's software development.

The shortcomings of TDD and TSDD were problems with understanding, the challenge of integrating legacy code, and the difficulty of adding more and more checks. The successful application of those approaches requires careful planning and the addressing of challenges. Developers need to explore its pros and cons before considering it for use in projects. TDD and TsDD have a significant impact on high-quality by improving code quality, reducing malfunctions, and enhancing developer productivity. TDD Provides Clean, Maintainable Code and Expertise through early writing of tests. TsDD design prioritises testability, resulting in complete test suites with improved fault tolerance and more predictable behaviour. With these methods, we end up with faster development, more efficient code, and "greener" developers. TDD has the potential to shine under changing requirements or ambiguity, while TSDD may be more appropriate for complex systems with high testing expectations (or high reliability thresholds). The distinction between these two approaches will be task-specific and depend on the specifics of Q2. Test-Driven Development (TDD) and Testability-Driven Development (TsDD) could slow down the development process: too much time is spent creating and maintaining tests; it’s hard to learn, and it doesn’t work with legacy code. Managing multiple tests and potential oversights can also arise, leading to a false sense of security. Successful implementation requires thoughtful consideration of these challenges and the advantages they offer, which is the solution to Q4.

5. CONCLUSION

As compared the effectiveness and efficiency of Test-Driven Development (TDD) and Testability-driven Development with refactoring (TsDD) methodologies in open-source software program tasks. Three tasks with acceptable test insurance and strong codebases have been cautiously selected. Evaluation standards, such as code insurance, improvement time, insects, and fixes, were established. EvoSuite was used to generate optimised check suites, while SonarLint played a vital role in code detection and refactoring. The TsDD approach, specifically targeted at improving testability through refactoring, achieved a staggering 14.20% increase in testability for the water simulation routine. There has also been a 12.54% increase in class-A average code coverage over TDD. These outcomes show that TsDD provides additional code coverage and expertise, enabling more thorough testing and enhancing software quality. Furthermore, TsDD also demonstrated better check time results than TDD. As a static code analysis tool, SonarLint was very helpful in identifying and fixing significant code issues throughout refactoring. The experiment demonstrated the desirable effects of both TDD and TsDD on code insurance quantity, as well as testing time, confirming the criticality of robust software testing and the benefits these practices bring to open-source projects, emphasising coverage metrics, testability, and quality to enhance overall performance and reliability. This can be achieved to a significant extent with the help of automated checkout and code analysis tools. Utilising evaluation tools such as EvoSuite and SonarLint can significantly contribute to achieving the dream of software excellence. Nevertheless, conducting additional research with a larger number of study subjects may be useful for validating and expanding the data obtained in this study. Answers to Q1, Q2, Q3, and Q4 were answered comprehensively. This is a complete version. The present research examined the two approaches, Test-Driven Development (TDD) and Testable-Driven Development (TsDD). It explored each approach’s pros and cons, as well as their impacts on software quality. Finally, the benefits and difficulties of each approach, along with their influence on software quality, were discussed, and the findings were presented.

CONFLICT OF INTEREST

The authors declare that there is *no* conflict *of interest* regarding the publication of this paper.

REFERENCES

[1] D. Strode, T. Dingsøyr, and Y. Lindsjorn, "A teamwork effectiveness model for agile software development," *Empirical Software Engineering 2022 27:2*, vol. 27, no. 2, pp. 56-, Mar. 2022, doi: 10.1007/S10664-021-10115-0.

[2] M. Moniruzzaman, D. Syed, and A. Hossain, "Comparative Study on Agile software development methodologies," Jul. 2013, Accessed: Jan. 10, 2026. [Online]. Available: https://arxiv.org/pdf/1307.3356

[3] G. Fraser and A. Arcuri, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, Dec. 2014, doi: 10.1145/2685612.

[4] A. Mohammad, B. Chirchir, A. Mohammad, and B. Chirchir, "Challenges of Integrating Artificial Intelligence in Software Project Planning: A Systematic Literature Review," *Digital 2024, Vol. 4, Pages 555-571*, vol. 4, no. 3, pp. 555–571, Jun. 2024, doi: 10.3390/DIGITAL4030028.

[5] A. I. Vlasov, B. V. Artemiev, and L. V. Juravleva, "Quality estimation method in advanced software systems," *AIP Conf Proc*, vol. 2467, no. 1, Jun. 2022, doi: 10.1063/5.0093003/2826315.

[6] A. Nguyen-Duc *et al.*, "Generative Artificial Intelligence for Software Engineering—A Research Agenda," *Softw Pract Exp*, vol. 55, no. 11, pp. 1806–1843, Nov. 2025, doi: 10.1002/SPE.70005;JOURNAL:JOURNAL:1097024X;WGROUP:STRING:PUBLICATION.

[7] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 4, Aug. 2015, doi: 10.1145/2699688;PAGE:STRING:ARTICLE/CHAPTER.

[8] Kent. Beck, "Extreme programming eXplained : embrace change," p. 190, 2000, Accessed: Jan. 10, 2026. [Online]. Available: https://books.google.com/books/about/Extreme_Programming_Explained.html?id=G8EL4H4vf7UC

[9] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," Jul. 2016, Accessed: Jan. 10, 2026. [Online]. Available: https://arxiv.org/pdf/1607.04347

[10] S. Parsa, M. Zakeri-Nasrabadi, and B. Turhan, "Testability-driven development: An improvement to the TDD efficiency," *Comput Stand Interfaces*, vol. 91, p. 103877, Jan. 2025, doi: 10.1016/J.CSI.2024.103877.

[11] S. Parsa, "Software Testing Automation: Testability Evaluation, Refactoring, Test Data Generation and Fault Localization," *Software Testing Automation: Testability Evaluation, Refactoring, Test Data Generation and Fault Localization*, pp. 1–580, Jan. 2023, doi: 10.1007/978-3-031-22057-9/COVER.

[12] P. C. Shekhar, "Accelerating Agile Quality Assurance with AI-Powered Testing Strategies," 2022. doi: 10.55041/IJSREM15369.

[13] F. Cammaerts, "Teaching Model-Driven Engineering from a Model-Testing Perspective," *Proceedings - 2024 IEEE Conference on Software Testing, Verification and Validation, ICST 2024*, pp. 454–456, 2024, doi: 10.1109/ICST60714.2024.00053.

[14] J. J. Gutiérrez, M. J. Escalona, and M. Mejías, "A Model-Driven approach for functional test case generation," *Journal of Systems and Software*, vol. 109, pp. 214–228, Nov. 2015, doi: 10.1016/J.JSS.2015.08.001.

[15] B. Marín, C. Gallardo, D. Quiroga, G. Giachetti, and E. Serral, "Testing of model-driven development applications," *Software Quality Journal 2016 25:2*, vol. 25, no. 2, pp. 407–435, Feb. 2016, doi: 10.1007/S11219-016-9308-8.

[16] M. Zakeri-Nasrabadi and S. Parsa, "An ensemble meta-estimator to predict source code testability," *Appl Soft Comput*, vol. 129, p. 109562, Nov. 2022, doi: 10.1016/J.ASOC.2022.109562.

[17] V. Garousi, M. Felderer, and F. N. Kılıçaslan, "A survey on software testability," *Inf Softw Technol*, vol. 108, pp. 35–64, Apr. 2019, doi: 10.1016/J.INFSOF.2018.12.003.

[18] M. Huda, Y. D. S. Arya, and M. H. Khan, "Measuring Testability of Object Oriented Design: A Systematic Review," 2014.

[19] H. G. Gurbuz and B. Tekinerdogan, "Model-based testing for software safety: a systematic mapping study," *Software Quality Journal 2017 26:4*, vol. 26, no. 4, pp. 1327–1372, Sep. 2017, doi: 10.1007/S11219-017-9386-2.

[20] E. Trunzer *et al.*, "Model-Driven Approach for Realization of Data Collection Architectures for Cyber-Physical Systems of Systems to Lower Manual Implementation Efforts," *Sensors 2021, Vol. 21,* vol. 21, no. 3, pp. 1–20, Jan. 2021, doi: 10.3390/S21030745.

[21] saeed parsa, M. Zakeri-Nasrabadi, and B. Turhan, "Testability-Driven Development: An Improvement to the Tdd Efficiency," 2023, doi: 10.2139/SSRN.4598484.

[22] S. Parsa, M. Zakeri-Nasrabadi, and B. Turhan, "Testability-driven development: An improvement to the TDD efficiency," *Comput Stand Interfaces*, vol. 91, p. 103877, Jan. 2025, doi: 10.1016/J.CSI.2024.103877.

[23] M. Z. Nasrabadi and S. Parsa, "Learning to Predict Software Testability," *26th International Computer Conference, Computer Society of Iran, CSICC 2021*, Mar. 2021, doi: 10.1109/CSICC52343.2021.9420548.

[24] M. Ghafari, T. Gross, D. Fucci, and M. Felderer, "Why research on test-driven development is inconclusive?," *International Symposium on Empirical Software Engineering and Measurement*, Oct. 2020, doi: 10.1145/3382494.3410687;CSUBTYPE:STRING:CONFERENCE.

[25] A. A. S. Ivo, E. M. Guerra, S. M. Porto, J. Choma, and M. G. Quiles, "An approach for applying Test-Driven Development (TDD) in the development of randomized algorithms," *Journal of Software Engineering Research and Development 2018 6:1*, vol. 6, no. 1, pp. 9-, Sep. 2018, doi: 10.1186/S40411-018-0053-5.

[26] S. Mäkinen and J. Münch, "Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies," *Lecture Notes in Business Information Processing*, vol. 166 LNBIP, pp. 155–169, 2014, doi: 10.1007/978-3-319-03602-1_10.

[27] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo, "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 597–614, Jul. 2017, doi: 10.1109/TSE.2016.2616877.

[28] M. M. Alam, S. I. Priti, K. Fatema, M. Hasan, and S. Alam, "Ensuring Excellence: A Review of Software Quality Assurance and Continuous Improvement in Software Product Development," *Studies in Big Data*, vol. 163, pp. 331–346, 2024, doi: 10.1007/978-3-031-73632-2_28.

[29] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative AI," *Automated Software Engineering 2024 31:1*, vol. 31, no. 1, pp. 26-, Mar. 2024, doi: 10.1007/S10515-024-00426-Z.

[30] N. L. Hyer and U. Wemmerlöv, "Group Technology and Productivity," *Revitalizing Manufacturing: Text and Cases*, pp. 387–399, Jan. 2024, doi: 10.4324/9781003571872-32/GROUP-TECHNOLOGY-PRODUCTIVITY-NANCY-HYER-URBAN-WEMMERL.

[31] H. A. Ramzan, S. Ramzan, and T. Kalsum, "Test-Driven Development (TDD) in Small Software Development Teams: Advantages and Challenges," *2024 5th International Conference on*

Advancements in Computational Sciences, ICACS 2024, 2024, doi: 10.1109/ICACS60934.2024.10473291.

[32] J. Cui, "A Comparative Study on the Impact of Test-Driven Development (TDD) and Behavior-Driven Development (BDD) on Enterprise Software Delivery Effectiveness," Nov. 2024, Accessed: Jan. 10, 2026. [Online]. Available: https://arxiv.org/pdf/2411.04141

[33] S. Parsa, "Testability Driven Development (TsDD)," *Software Testing Automation*, pp. 159–189, 2023, doi: 10.1007/978-3-031-22057-9_4.

[34] E. Farchi and S. Route, "Quality Engineering for Agile and DevOps on the Cloud and Edge," Feb. 2023, Accessed: Jan. 10, 2026. [Online]. Available: https://arxiv.org/pdf/2302.03651