Computer Standards & Interfaces 91 (2025) 103877

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi

Testability-driven development: An improvement to the TDD efficiency

Saeed Parsa[a,*], Morteza Zakeri-Nasrabadi[a,b], Burak Turhan[c]

[a] *School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran*
[b] *School of Computer Science, Institute for Research in Fundamental Sciences (IPM), P. O. Box 19395-5746, Tehran, Iran*
[c] *Faculty of Information Technology and Electrical Engineering, Oulu, Finland*

ARTICLE INFO

Keywords:
Software methodology
Test-first development
Software testability
Automated refactoring
Machine learning

ABSTRACT

Test-first development (TFD) is a software development approach involving automated tests before writing the actual code. TFD offers many benefits, such as improving code quality, reducing debugging time, and enabling easier refactoring. However, TFD also poses challenges and limitations, requiring more effort and time to write and maintain test cases, especially for large and complex projects. Refactoring for testability is improving the internal structure of source code to make it easier to test. Refactoring for testability can reduce the cost and complexity of software testing and speed up the test-first life cycle. However, measuring testability is a vital step before refactoring for testability, as it provides a baseline for evaluating the current state of the software and identifying the areas that need improvement. This paper proposes a mathematical model for calculating class testability based on test effectiveness and effort and a machine-learning regression model that predicts testability using source code metrics. It also introduces a testability-driven development (TsDD) method that conducts the TFD process toward developing testable code. TsDD focuses on improving testability and reducing testing costs by measuring testability frequently and refactoring to increase testability without running the program. Our testability prediction model has a mean squared error of 0.0311 and an R^2 score of 0.6285. We illustrate the usefulness of TsDD by applying it to 50 Java classes from three open-source projects. TsDD achieves an average of 77.81 % improvement in the testability of these classes. Experts' manual evaluation confirms the potential of TsDD in accelerating the TDD process.

1. Introduction

Test-first development (TFD), a software development approach proposed and embraced by numerous software developers, necessitates the creation of automated tests prior to the actual code writing [1]. TFD is an integral component of agile methodologies such as Extreme Programming (XP) [2] and Test-Driven Development (TDD) [3], which underscore the importance of iterative and incremental development, collaboration with customers, and continuous improvement [2,4],. Advocates of TFD assert that this approach can enhance code quality, diminish debugging time, provide expedited feedback, facilitate superior documentation, and enable more straightforward refactoring [5–7]. Nevertheless, TFD also presents challenges and limitations, demanding additional effort and time to write and maintain test cases, particularly for large-scale and complex projects [8–10]. The following are some of the factors that contribute to this challenge:

- The difficulty of writing good test cases that cover all the possible scenarios and edge cases, as well as the expected and unexpected behaviors of the system [11],
- The need to update and modify test cases frequently to reflect the changes in the system's requirements, design, and implementation [12],[13],
- The lack of adequate tools and frameworks that support TFD, especially for some application domains and technologies that are not well suited for automated testing [8],
- There is a lack of sufficient skills and experience among developers and testers in applying TFD effectively and efficiently, as well as a lack of resistance to change from traditional testing approaches [14].

These factors can exacerbate the cost and intricacy of software testing in TFD, potentially outweighing the benefits of this approach under certain circumstances. As per a study conducted by Jones and Bonsignour [15], the costs associated with software testing typically account for 15 to 25 % of the overall project cost. Indirect testing costs,

* Corresponding author at: School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran 16846-13114, Iran.
E-mail address: parsa@iust.ac.ir (S. Parsa).

https://doi.org/10.1016/j.csi.2024.103877
Received 4 October 2023; Received in revised form 1 March 2024; Accepted 26 May 2024
Available online 29 May 2024

encompassing defect rectification and code reworking, often surpass the direct costs considerably. Consequently, it is paramount to identify effective and efficient methods for conducting software testing and reducing its costs.

Refactoring for testability entails enhancing the internal structure of source code while maintaining its functionality to facilitate testing [16]. Refactoring for testability can curtail the cost of software testing and expedite the test-first life cycle in several ways, such as:

- Extracting code that handles input/output operations, such as database or API calls, into separate modules that can be mocked or stubbed during testing. This can reduce the dependency on external systems and resources and make the tests more reliable and faster.
- Splitting complex code into smaller, simpler pieces that can be tested individually and independently [17]. This can increase the production code's test coverage, readability, and maintainability, making locating and fixing defects easier.
- Getting rid of the global state and static variables that can cause side effects and interfere with testing. This can improve the code's modularity, reusability, and isolation and make the tests more predictable and consistent.
- Avoiding instantiating services or objects directly in the code and using dependency injection or inversion of control patterns instead. This can decouple the code from its dependencies and make it possible to inject mock or fake objects during testing [18].
- Removing or replacing any die/exit statements that can terminate the execution of the code prematurely and prevent testing. This can ensure that the code follows a normal flow and returns a valid output that tests can verify.

The application of refactoring techniques, along with others, enables developers to reshape their code into a design more amenable to testing, thereby supporting test-first development practices. This refactoring approach for testability can also uncover new requirements, design deficiencies, or potential enhancements within the code, providing feedback for further refactoring endeavors [16,18],. However, a crucial step precedes refactoring for testability: assessing the software's testability. This evaluation establishes a baseline for examining the current state of the software and identifying areas requiring improvement. By measuring testability, developers gain insights into the ease or difficulty of testing their software and identify factors that influence its testability, such as observability, controllability, isolateability, separation of concerns, and understandability [19] —Moreover, measuring testability aids in estimating testing effort and cost, facilitating the comparison of different testing approaches and techniques [20,21],. By clearly understanding the software's testability, developers can prioritize and plan their refactoring activities more effectively and efficiently, ensuring that changes positively impact software quality and performance [22].

This paper presents a mathematical model to calculate class testability, predicated on test effectiveness and effort. To estimate testability precisely, the model considers the test suite's parameters, such as test coverage, size, and budget. However, the application of this mathematical model necessitates the generation and execution of test data, a process that is both costly and time-consuming. Consequently, this paper introduces a machine-learning regression model that predicts testability utilizing source code metrics. This model can statistically measure the testability of classes within a given object-oriented program without the need for code execution. In order to construct our testability prediction model, we employ the mathematical model a single time to compute the testability of 23,800 Java classes within the SF110 dataset [23]. Concurrently, we calculate the source code metrics for these classes and input them, along with the testability values, into machine learning models. The models that result from this process are then capable of estimating testability without the need for program execution or the use of the parameters of the mathematical testability mode.

The testability-driven development (TsDD) method proposed in this paper conducts the TFD process toward developing testable code. The primary motivation behind TsDD is to avoid unnecessary tests while encouraging quality software development. TsDD focuses on improving testability and reducing testing costs. If the code is untested, it can be rewritten, modified, or refactored before being tested. Thus, TsDD helps programmers improve their code quality instantly before and after writing the code. The production code can be made ready for testing by measuring testability frequently and refactoring to increase testability without running the program. The main contributions of this paper are as follows:

1. An enhanced version of the TDD is introduced to support test-first, particularly TDD approaches by white box and automated test data generation tools and techniques. The proposed approach introduces testability as a quantifiable quality attribute to improve agility during a new software development approach called testability-driven development (TsDD).
2. A mathematical model to formulate and measure testability in terms of test effectiveness and test effort, mentioned in the ISO/IEC 25,010:2011 standard [24], is proposed and evaluated.
3. A machine-learning model is presented to measure testability regarding the test effectiveness and effort learned while testing a repository of 23,000 Java classes. The model is a voting regressor that may instantaneously predict the legacy and new code under-development testability.

Experimental results in this article demonstrate the applicability of TsDD as an agile process to reduce the execution time and improve the testability of 50 Java classes from three real-life open-source projects [25–27] by an average of 6.5 h and 77.81 %, respectively, in comparison with TDD. One may argue that these improvements resulted from refactorings and have nothing to do with the TsDD process. However, TsDD conducts refactoring toward testability by measuring the testability before and after refactoring to ensure the necessity and impact of the refactoring. The experimental results in Section 4.3 show that refactorings negatively affect testability in some cases. Overall, the experiments are conducted to answer the following research questions:

- **RQ1:** *What is the performance of machine learning regression models to predict testability?*
- **RQ2:** *On average, how much testability may improve when applying the TsDD approach?*
- **RQ3:** *On average, how much test effectiveness may improve when applying the TsDD approach?*
- **RQ4:** *On average, how much total testing time is reduced when applying the TsDD approach?*
- **RQ5:** *Which refactoring operations improve testability more than the others?*
- **RQ6:** *What is the impact of TsDD on the other software quality attributes?*

The remaining parts of this article are organized as follows: Section 2 introduces the testability-driven development approach and describes the concrete implementation of its enabler technique, the testability prediction method. Section 3 provides a working example to demonstrate the efficiency of testability-driven development. Section 4 explains the experimental setup used for evaluating TsDD. Section 5 evaluates the testability-driven development process and our proposed testability prediction software tool while practically responding to the research questions. The threats to validity are discussed in Section 6. Section 7 summarizes discussions about TDD and outlines the related works. Finally, a conclusion is drawn in Section 8.

2. Methodology

Testability-driven Development (TsDD) enhances Test-driven

Development (TDD) by mitigating blind refactoring and ineffective tests. It builds upon TDD by necessitating developers to prepare programs for effective tests before transferring to testers. During the code completion process, TsDD advocates for the assurance of the code's testability before any testing. The TsDD approach is delineated in Section 2.1. Sections 2.2 and 2.3 present a mathematical model to compute testability by actively generating test suites and measuring test efficiency and effectiveness. Section 2.4 employs the testability model to compute testability as the objective of a learning process to construct testability prediction models.

2.1. *Testability-driven development approach*

Software testing is frequently time-intensive, accounting for more than 50% of the total development time [28]. This is particularly applicable to scientific and engineering software [29], such as Scijava-common [26] and Weka [25],[30], which possess inherent complexities that render testing both labor-intensive and costly [29], [31–33]. Testability refers to the relative ease or difficulty associated with testing units and modules or developing test cases [19]. Testability-driven Development (TsDD) strives to alleviate the testing burden and associated costs by refining and rendering the code testable prior to testing. Test-driven Development (TDD) commences with a failing unit test, which is subsequently passed with the minimum amount of code before being refactored into production code. However, TDD does not address the preparation of the code for efficient and effective tests [5],[8],[29]. According to Nanthaamornphong and Carver [29], writing a proficient test can prove more challenging than implementing the actual code. They also discovered that testing without the aid of a tool or framework complicates the process of writing tests. This is where TsDD becomes instrumental. TsDD augments the TDD process by predicting and refactoring testability frequently to ensure the production of high-quality tests, which are the direct outcomes of testable code. Unlike conventional TDD approaches, test cases can also be generated and augmented automatically. Following a successful test, the code is modified to meet the subsequent system requirement, and the process is repeated until the code is complete. Consequently, improvements in testability guide the software design and development process in conjunction with the test results.

The TDD and TsDD processes are shown in Fig. 1 alongside each other. Fig. 1.**b** shows the TsDD process, in which an additional testability measurement step is performed while implementing a feature. This way, unnecessary program executions for testing are avoided. The testability measurement and improvement loop continues until testability reaches a specific threshold, τ, (defined by the developer based on the quality requirements), or refactoring does not improve testability any further. The threshold can be adjusted depending on the developer's preferences, application domain, system criticality, software standards, and priority of testability. The upper bound of the testability value in our measurement model is 1.

After a testable code implementing the given requirements is obtained, the manually generated test suite is amplified with the test data generated automatically to increase the quality of the test and maximize fault detection power. The rest of the TsDD process is similar to the traditional TDD process shown in Fig. 1.**a**. In summary, the rhythm of TsDD is as follows:

1. Quickly add a test (denoted by the read color),
2. Run all tests and see the new one fail,
3. Make a little change (denoted by the green color),
4. Make sure the code is testable,

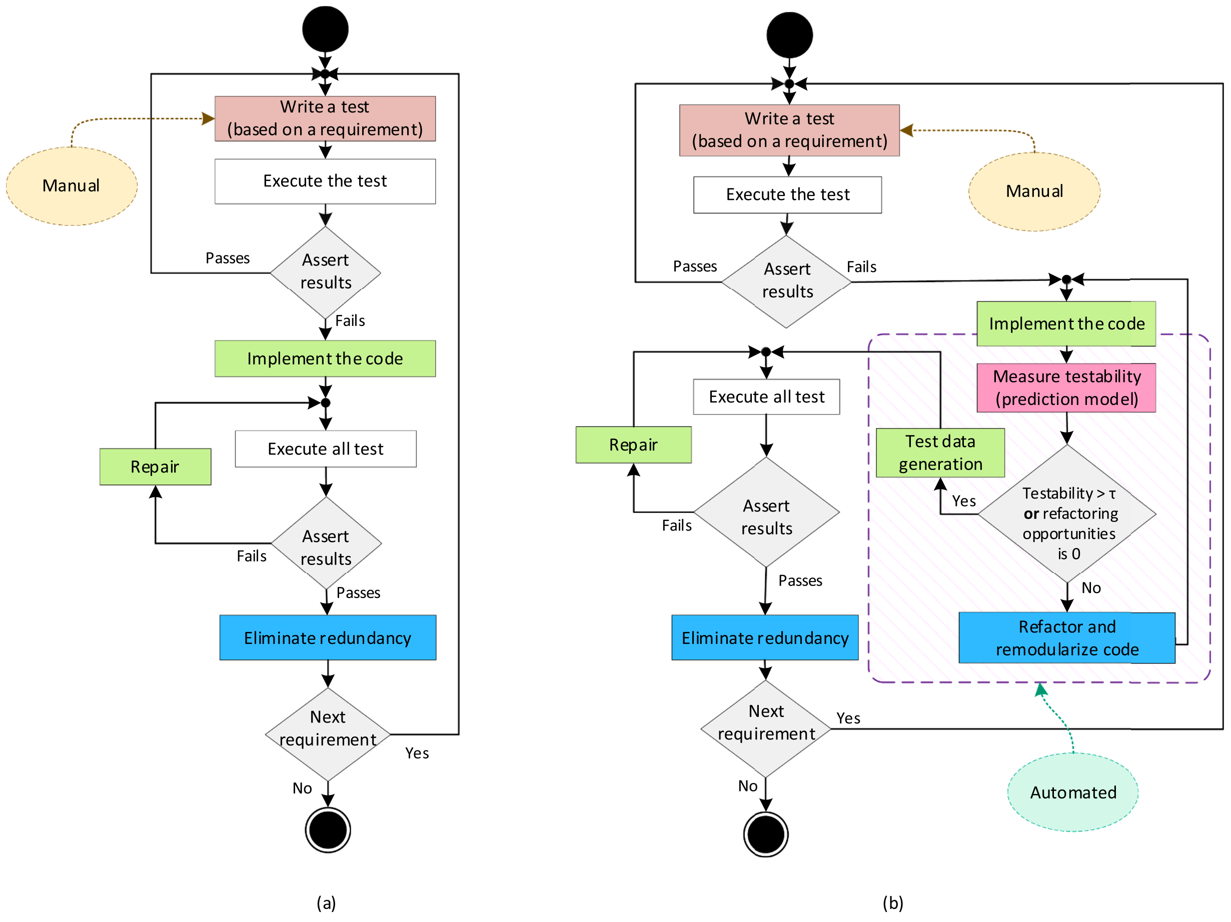

Fig. 1. The flowcharts of TDD and TsDD approaches.

   4.1. Measure (predict) testability (denoted by the pink color),
   4.2. While testability < a given threshold, τ, **or** there is a refactoring action to apply:
      4.2.1. Refactor for testability (denoted by the blue color),
      4.2.2. Implement new code (if required) or fix existing faults,
      4.2.3. Measure (predict) testability,
5. Complete the manual test suite with automatic test data generation (denoted by the blue color).
6. Run all tests and observe them all succeed.
7. Refactor to remove duplication (denoted by the blue color).

According to Fig. 1.**b**, TsDD uses frequent refactoring to improve testability instead of frequently running test codes, which are time and resource-consuming. Hence, TsDD enables quality software development by encouraging programmers to move towards automated refactoring to improve testability and other quality factors such as modularity, maintainability, and scalability.

The fascinating point in the kernel of the TsDD process is that the entire testability prediction and improvement loop, as well as test suite completion and amplification, can be performed automatically. Indeed, the rationale behind testability-driven development is effectively using the testability measurement information in the software development lifecycle (SDLC). It encourages the developers to minimize the test cost and effort by emphasizing refactoring for testability. The automation requires three categories of tools: testability measurement (proposed in this paper), automated refactoring, and test data generation, which already exist for most programming languages. TsDD allows for iteratively completing and amplifying the initial test suite (created manually at the beginning of the developing process) using automatic tools after the code is ready to test. Nevertheless, the entire TsDD process can also be performed manually. As shown in Fig. 2, the mantra of TsDD is:

1. Write a test that fails. This step is shown in red, meaning the expected behavior has not yet been implemented.
2. Make it run. This step is shown in green to indicate that code development proceeds while running the tests as far as the tests pass.
   - Make tests effective in a pink-blue cycle inside the main loop of TsDD.
3. Make it right. This step is shown in blue, in which the developer refactors the code to improve quality attributes such as readability and reusability without affecting its functionality or behavior or concerning testability.

As depicted in Fig. 2, the Testability-driven Development (TsDD) cycle modifies the Test-driven Development (TDD) cycle by predicting and enhancing testability during the green phase of TDD. The green phase, which is the second phase, is when the developer crafts code that

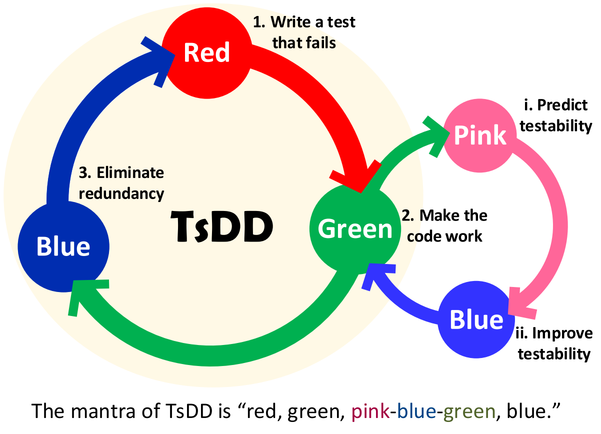

Fig. 2. Testability-driven development extends the TDD process.

satisfies the requirement and allows the test to pass. Upon running the code, it passes the test, thereby meeting the requirement and producing the expected output. The green phase intends to implement code that enables the test to pass and to verify that the code operates as anticipated and fulfills the requirements. However, should the tests fail during the green phase, this indicates that the code either does not meet the requirement or contains a bug. In such instances, the programmer should revert to the green phase and rectify the code until the test passes. The programmer should not proceed to the blue phase until the code has successfully passed the test. The TDD cycle ensures the code is continuously tested and verified, guaranteeing its high quality and readability. The pink phase, as proposed by TsDD, measures the testability of the developed code to ensure the effectiveness of the test. If the code is not testable during the blue phase, it is refactored for testability, and this subcycle is repeated until the code is prepared for an effective and efficient test.

The proposed TsDD approach can be applied to a vast range of legacy and developing codebases while maximizing machine utility and minimizing human effort. TsDD extends TDD by leveraging automated software engineering techniques, including automated test data generation, testability measurement, and automated refactoring to improve testability (and not solely refactoring for code duplication). It is shown in this paper that refactoring for testability improves testability and other software quality factors, such as modularity and maintainability. Section 2.2 describes the testability measurement method used in TsDD.

2.2. Testability mathematical model

Considering the standard definition offered by ISO/IEC 25,010:2011 [24], *test efforts*, $T^{effort}(X,C)$, and *test effectiveness*, $T^{effect}(X,C)$, are the two significant ingredients to determine the testability, $T(X)$, of a given class, X, against the test criterion, C, *e.g.*, statement coverage, branch coverage, or mutation score. The higher the test coverage value, the more effective the test is. In contrast, the more tests are required to satisfy the coverage criterion, C, the more effort is needed to test the component under test. Therefore, testability, $T(X)$, can be defined as the division of test effectiveness,$T^{effect}(X,C)$, and test effort, $T^{effort}(X,C)$:

$$T(X) = \frac{T^{effect}(X,\ C)}{T^{effort}(X,\ C)} \tag{1}$$

The effectiveness of the test depends on the percentage of the code coverage that can be achieved by automatically generating test data. Automatic test data generation tools such as EvoSuite [34] minimize test suites to keep only influential test cases without sacrificing coverage. We call a test case in the minimized test suite $\tau(X,\ C)$, an *influential test case, i. e.*, a test case that enhances the coverage. First, we define *Coverageability,* $C_\mu(X,C)$, as the mean coverage, $C(X)$, gained by an influential test case in a minimized test suite, $\tau(X,\ C)$, multiplied by the number of test cases, b_B, that can be generated considering the test budget, B:

$$C_\mu(X, C) = \frac{\overline{C}(X)}{|\tau(X,\ C)|} \times b_B \tag{2}$$

$$\overline{C}(X) = \sqrt[n]{(C_1(X) \times\ C_2(X) \times\ \dots\ \times\ C_n(X))} \tag{3}$$

In Eq. (2), $|\tau(X,\ C)|$ is the size of the minimized test suite, τ, required to satisfy the test criterion, C, on the class, X and $\overline{C}(X)$ denotes the mean of different test coverage criteria selected to be measured during the test. The value of $|\tau(X,\ C)|$ also determines the upper bound of b_B, *i.e.*, $b_B \leq |\tau(X,\ C)|$. The test effectiveness, $T^{effect}(X,C)$, of the class, X, can be computed as a maximum of Coveragebility, $C_\mu(X,C)$, achieved when b_B equals to $|\tau(X,\ C)|$:

$$T^{effect}(X,\ C) = C_\mu(X, C) \qquad s.t. \qquad b_B = |\tau(X, C)| \tag{4}$$

Indeed, we use the expected value of code coverage to quantify the test effectiveness. More precisely, statement coverage, branch coverage, and mutation score are used to calculate $\overline{C}(X)$ in our experiments. There is ongoing research on the impacts of different combinations of coverage metrics on the test and fault localization effectiveness and effort [35]. An important finding is that the combination of coverage criteria can be more effective than a single criterion [36]. Experiments with EvoSuite [34] suggest combining statement, branch, and mutation coverage can boost test effectiveness given a fixed test budget. It should be noted that only complete branch coverage subsumes statement coverage. However, often, test suites do not provide complete branch coverage. As a result, a test suite with a relatively high branch coverage may only cover a few code statements due to the imbalanced distribution of source code statements in branches. Therefore, combining different coverage criteria may lead to a more precise test effectiveness model. In this paper, we compute the geometric mean of the coverage criteria rather than the arithmetic mean when computing $\overline{C}(X)$ since the metrics may depend on each other while they are complexly separate concepts. It is worth noting that the arithmetic mean is not appropriate in such a context.

The number of required tests to achieve a certain coverage within a given time budget measures the test effort. While generating test data, we observed that the growth rate of code coverage (Eq. (3)) decreases as time proceeds. In fact, as test data generation proceeds, generating an influential test case becomes more challenging since the probability that a newly generated test case covers the same parts as the previous test cases increases. Therefore, the required test effort increases while proceeding with test data generation. On the other hand, if for a given time budget (the test suite generation time), T, the number of test cases generated for a class, X_1, is more than a class, X_2, then it can be induced that the class X_1 is relatively more testable than the class X_2. Indeed, the lower average time for generating a test case for class X_1 indicate that generating tests for class X_1 is simpler than class X_2. Hence, the testability equation must consider the average time for generating one influential test case as the factor that its incrementation negatively affects the class testability. To measure the impact of this factor, we define the parameter, ω(X), as the average time expected to generate an influential test case for a given class, X, as follows:

$$\omega(\mathrm{X}) = \frac{\mathrm{T}}{|\tau(X,\ C)|} \tag{5}$$

where T is the total time taken to generate the test suite, $\tau(X, C)$, for the class, X, in the software under test. As a result, the test effort, $T^{effort}(X,C)$, for a class, X, can be best measured as follows:

$$T^{effort}(X,\ C) = (1+\omega(\mathrm{X}))^{\left\lceil \frac{|\tau(X,\ C)|-1}{TNM(X)} \right\rceil} \tag{6}$$

In Eq. (6), $TNM(X) > 0$ is the total number of methods in the class, X. Finally, testability, $T(X)$, of a class, X, is computed by substituting the relations for test effectiveness,$T^{effect}(X,C)$, and test effort,$T^{effort}(X,C)$ in Eq. (1) as follows:

$$T(X) = \frac{T^{effect}(X,\ C)}{T^{effort}(X,\ C)} = \frac{max\left(C_\mu(X,\ C)\right)}{(1+\omega(\mathrm{X}))^{\left\lceil \frac{|\tau(X,\ C)|-1}{TNM(X)} \right\rceil}} = \frac{\overline{C}(X)}{(1+\omega(\mathrm{X}))^{\left\lceil \frac{|\tau(X,\ C)|-1}{TNM(X)} \right\rceil}} \tag{7}$$

As shown in Eq. (7), testability is not dependent on the test budget, B. In Section 2.3, we show that it is also time-independent.

2.3. The stability of the proposed testability model

Testability is an inherent feature of source code that should not change over test time. We prove that our proposed model for testability measurement is independent of the time budget dedicated to the test data generation, and it completely depends on the source code features. To this aim, we begin with the empirical evidence about code coverage changes obtained in the experiments with automatic test data generation. Our experiments with EvoSuite [34] show that code coverage, $C(X)$, increases logarithmically as the time budget increases. When testing a class, X, several times with different time budgets, T, we observed that the growth rate of $C(X)$, decreases logarithmically with time.

Fig. 3.**a**, shows the variations in code coverage when testing, with different time budgets, 22 Java classes of the JSON-java project [37]. The tests have been generated with EvoSuite using a time budget of 1 to 8 min for each class. It is observed that the growth rate of the coverage level reduces as test data generation time increases. Therefore, as shown in Fig. 3.**b**, the coverage growth rate reduction can be estimated with the $\ln(t)$ function, where the parameter t indicates the time elapsed since the test data generation started.

On the other hand, due to the increased time lag between generating two successive influential test cases, the test effort, $T^{effort}(X,C)$, grows exponentially with time. The change in the denominator of the testability formula (Eq. (7)) can be approximated with e^t since as the number of test cases increases, the denominator of Eq. (7) approaches to e^t:

$$\lim_{|\tau(X,C)|\to\infty}\left(1+\frac{T}{|\tau(X,C)|}\right)^{\frac{|\tau(X,C)|-1}{TNM(X)}} = e^{(T/TNM(x))} \qquad TNM\ (X)\ >\ 0 \tag{8}$$

The total time dedicated to the test data generation, T, and the number of methods, $TNM(X)$, is constant during the unit test of a class, X. Therefore, the momentary changes, $\frac{\partial}{\partial t}T(X;t)$, in testability, $T(X)$, of class X over the (test) time, t, allocated to generate test data, is computed as follows:

$$\frac{\partial}{\partial t}T(X;t) \approx \frac{\partial}{\partial t}\left(\frac{\ln(t)}{e^t}\right) = \frac{\frac{e^t}{t} - e^t\ \times \ln(\mathrm{t})}{e^{2t}} \tag{9}$$

Fig. 4 shows the plot of testability, $T(X,\ t)$, of class, X, *versus* the elapsed time, t. It is observed that after a while, about *six* minutes, the testability reaches a steady state. The inspiration is that time does not affect our proposed model to compute testability as an inherent source code feature.

2.4. Testability prediction model

The mathematical testability model discussed in Section 2.2 measures source code testability based on the definition of testability according to the ISO/IEC 25,010 standard [24]. However, the model requires the class under test to be executed to generate the test suite and measure test adequacy criteria, including code coverage and mutation score. In other words, the elements in Eq. (7), including line coverage, branch coverage, mutation score, and test suite size, are determined by generating and executing test cases. As discussed in the introduction, the main motivation behind the testability-driven development approach is to avoid unnecessary tests while encouraging quality software development. To this aim, we propose a machine learning regression model that estimates the value of testability based on source code metrics. This way, the code can be prepared for testing by frequent predictions of testability followed by refactoring to improve testability without any need to execute the program.

Fig. 5 depicts our process for predicting testability, which encompasses two primary phases: (1) model training and validation and (2) inference. Phase 1 entails the preparation of learning samples and the training of testability prediction models. Initially, a dataset of software projects is assembled, each containing diverse Java classes. Subsequently, each class is translated into a vector of source code metrics acquired through static analysis. Concurrently, automated test data generation is employed to test each class within a specified time frame. Computation of the essential test code metrics for evaluating class testability (*e.g.*, statement coverage (St Cov.), branch coverage (Br Cov.), mutation coverage (μ Cov.), and test suite size (τ)) is conducted based on

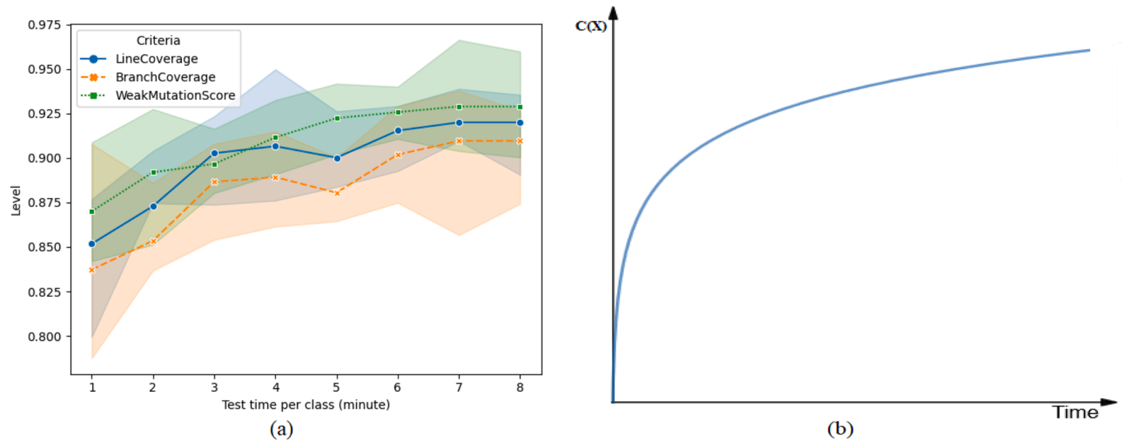

Fig. 3. Code coverage level per testing time.

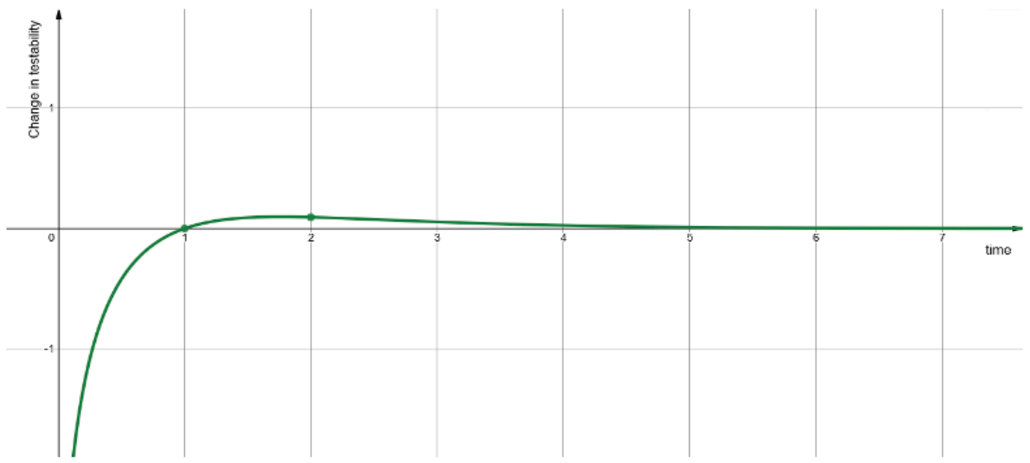

Fig. 4. Testability variations in the order of the elapsed time.

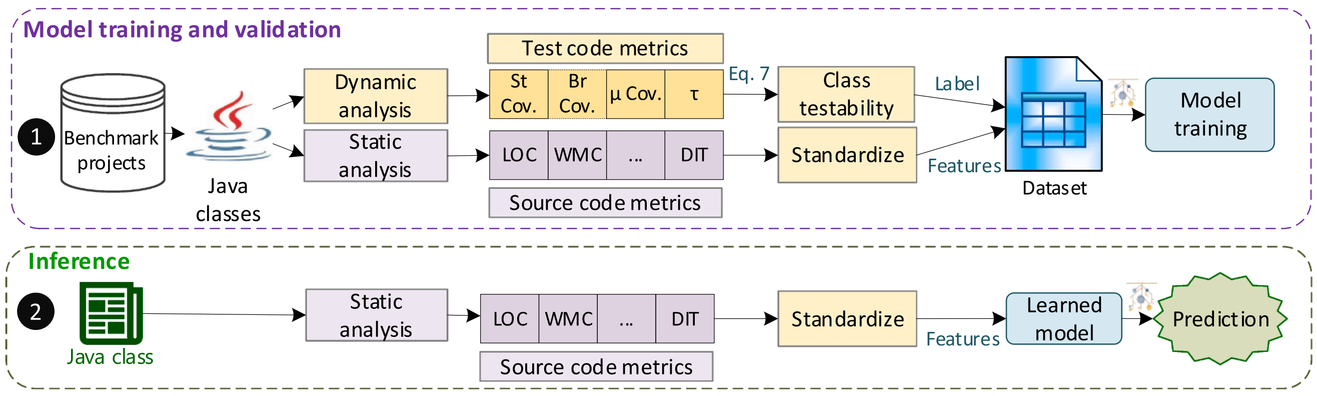

Fig. 5. Testability prediction process.

the outcomes of the test data generation tool.

Furthermore, the vectors of class metrics are standardized for application in machine learning algorithms. Each vector is assigned a label corresponding to its class testability, which is determined by the mathematical model expounded in Eq. (7). This procedure culminates in creating learning samples for the testability prediction models. Step 1 concludes with the training and validation of the regression models.

In Fig. 5, Phase 2 illustrates the process of predicting testability during the inference time. As depicted in the figure, solely the static analysis of the given class is mandated at the time of inference, obviating the necessity to assess the input class with an automatic test data generation tool. Thus, no testing cost is incurred in computing class testability. In Phase 2, the initial step involves extracting source code metrics for the input class, which are subsequently standardized using the same procedure in Phase 1. Following this, the prepared vector is input into the learned model from Phase 1 to estimate the testability of the provided class. The remaining sections of this segment (Sections 2.4.1 to 2.4.4) elaborate on the intricacies of dynamic analysis, static analysis, regression models, and model training for implementing the necessary software for the TsDD approach.

2.4.1. *Dynamic analysis*

Dynamic analysis is performed once to provide the parameters required by Eq. (7) and label data samples with testability values. Generating test cases and executing programs for a large corpus of source codes is performed automatically by using a test data generator tool. We used EvoSuite [34] to generate test cases for all projects in the SF110 corpus [23]. Using an automatic test data generator in this phase provides several benefits. First, it can generate a large amount of data required for proper training machine learning models. One can imagine how lengthy and difficult it is to write unit tests for more than 23,000 Java classes manually. Second, if test data is created manually, human expertise should also be considered a factor in testability prediction models. Third, it is possible to configure the test budget with a test data generator in terms of time. Indeed, there is no standard and systematic approach to manually quantify the effort and time taken to generate test suites.

Once EvoSuite execution on a class is finished, the statement coverage, branch coverage, mutation score, and number of generated test data (the size of minimized test suite) are calculated and fed to Eq. (7) to compute the testability of the class. EvoSuite [34] follows an evolutionary algorithm to generate test data. Such non-deterministic algorithms may generate different test suites in different algorithm runs. We repeated the test data generation process and averaged the results to reduce the randomness effects caused by EvoSuite in the calculated test statistics, including coverage levels and test suite size. More precisely, we repeated the test data generation process five times and averaged each parameter's output to minimize the results' randomness. As discussed in Table 3, five-time repetition highly reduces the average variance between the results of different runs for different classes. As a result, our testability labels are reliable for creating a testability prediction model with regression learning.

2.4.2. *Static analysis*

Source code metrics are used to quantify software characteristics. We use 20 well-known metrics to map a class source code into a vector of real numbers, each indicating a different source code metric. As shown in Table 1, the source code metrics are of six different categories according to their subject. Therefore, using these metrics, different aspects of source code are considered for training the testability prediction model. In contrast, the efficiency of the prediction process is preserved due to a relatively low number of metrics. Previous studies have shown the effectiveness of source code metrics in predicting test effectiveness [38–41]. Source code metrics are of different ranges, causing difficulties in the learning process. The difficulty is resolved by scaling the metrics into a standard range [42].

Table 1
Source code metrics used in testability prediction.

Subject	Metric	Full name
Size (volume)	LOC	Line of code
	TLOC	Total lines of code
	LLOC	Logical line of code
	TLLOC	Total logical line of code
	TNOS	Total number of statements
	TNM	Total number of methods
	NG	Number of accessor methods
	TNG	Total number of accessor methods
	TNA	Total number of attributes
Complexity	WMC	Weighted methods per class
	NL	Nesting level of control constructs
	NLE	Nesting level else-if
Cohesion and Coupling	LCOM5	Lack of cohesion in methods 5
	CBO	Coupling between objects
	CBOI	Coupling between object classes inverse
	RFC	Response set for a class
	NII	Number of incoming invocations
	NOI	Number of outgoing invocations
Visibility	TNPM	Total number of public methods
Inheritance	DIT	Depth of inheritance tree

2.4.3. *Regression learning models*

Section 2.2 introduces a mathematical model that computes testability in a continuous space, specifically within the interval [0, 1]. This restriction is imposed to enhance interpretability. Given that our target variable (*i.e.*, testability) is numerical and confined within a closed interval, regression models are the most suitable choice for prediction. In our case, the independent variables, or predictors, are source code metrics, and the target variable is testability, as our mathematical model quantifies. The regression model is trained to map a real-valued vector, V, which represents a class source code based on its metrics, to the class testability, $T(X)$, a continuous number in the interval [0, 1].

To achieve this, we construct an ensemble of three well-known regression models: a deep feed-forward neural network (DNN) [43], a random forest regressor (RFR) [44], and a histogram gradient boosting regressor (HGBR) [45]. This ensemble model, also known as a voting regressor (VoR), combines the predictions of these three regressors and returns the weighted average of their individual predictions as the final predicted value. The selection of these three models is based on their performance, as they all have the least mean squared error.

The VoR model is beneficial when dealing with a set of equally well-performing models, as it can balance out their individual weaknesses when predicting class testability. Depending on its algorithm and the provided data, each base regressor in the VoR model may excel in a specific part of the learning space. To support this hypothesis, we have chosen the base regressors from different families of learning algorithms, including function-based (DNN), tree-based (RFR), and boosting-based (HGBR) methods.

Each of these regressors has unique characteristics. The DNN [43] is an artificial neural network with multiple layers of neurons connected by weighted links. It learns from the data by adjusting the weights using a backpropagation algorithm, allowing it to approximate complex nonlinear functions and capture high-level features from the data. The RFR [44] combines multiple decision trees on different data subsets and features, with the final prediction obtained by averaging the predictions of the individual trees. This model can handle heterogeneous and noisy data and reduce the variance and overfitting of single trees. The HGBR [45] is a type of gradient boosting method that uses histograms to bin the continuous features and reduce the number of split points. It builds a series of weak learners sequentially fitted on the negative gradient of the loss function, improving the speed and accuracy of gradient boosting while handling missing values and outliers.

2.4.4. *Implementation details*

The complete implementation of the testability prediction model and the best hyperparameters for each model (base regressors and VoR regressor) are available on our GitHub repository[1] [46]. The testability dataset is publicly available at Zenodo [47]. Source code metrics for each project are extracted using Open Static Analyzer, a free static analysis tool [48]. It scans the source code directory and produces the source code metrics for each class as a CSV file. The testability prediction model has been implemented in Python using the Scikit-Learn library [42], an open-source library that provides different machine-learning algorithms.

To enhance the generalizability of the testability prediction model and mitigate the risk of model overfitting, we employed cross-validation and model selection techniques. The learning dataset was partitioned into two subsets: a training set and a testing set, comprising 80 % and 20 % of the data, respectively. We used a grid search strategy [49] with

[1] https://github.com/m-zakeri/TsDD

five-fold cross-validation [42] on the training set to ascertain the optimal hyperparameters for each of the three selected base regressors. During each fold of the cross-validation process, a randomly selected 20 % subset of the training set served as the validation set. The R^2 score was employed as a scoring metric to compare regressor instances trained with varying hyperparameters. The scores reported by the *k*-fold cross-validation were averaged for each hyperparameter configuration during the grid-search iteration, culminating in selecting the best set of hyperparameters. Subsequently, the models were retrained on the entirety of the training set using the optimal hyperparameters identified during the grid search with a cross-validation process. This approach was adopted because the validation set becomes redundant upon completing the cross-validation process, allowing all data to be utilized for training a robust model. Finally, the regression models were evaluated on the testing set, and a range of regression analysis performance metrics were computed to assess the effectiveness of the model. It is important to note that the testing set was not utilized during the model selection and training processes to circumvent methodological errors. The results of the hyperparameter tuning and model selection processes are accessible on the TsDD GitHub repository [46].

3. Running example

This Section offers a working example to demonstrate the applicability of TsDD as a supporting process to remove some deficiencies of TDD. Refactorings do not necessarily improve testability. A particular refactoring may improve a project's testability while degrading another project's testability due to different contexts and code structures. The proposed testability prediction models provide the developers with immediate measurement of refactoring effects on testability. As a result, any refactoring with a negative impact on the testability could be rolled back immediately.

3.1. Step-wise testability improvement

We aim to iteratively improve the testability of the GridGenerator class in the *Water Simulator* project [50] (also available in the SF110 corpus of the Java projects [23]) using testability measurement and refactoring. Water Simulator is an open-source agent-based urban water demand simulator that contains non-trivial source codes written in Java. Table 2 lists applied refactoring operations and test results at each step. The last column indicates the testability values in the interval [0, 1], computed and predicted by the proposed testability mathematical and machine learning models, discussed in Sections 2.1 and 2.4 of the methodology. The testability prediction model is not 100 % accurate. It predicts the testability value for a given Java class with some errors. Therefore, the results of the predicted testability and the one computed by the mathematical model differ. The first row of Table 2 represents the test results obtained for the class, GridGenerator, by an automated test data generator [51] in six minutes. We observed that the class testability is very low before applying any refactoring operation automatically, 0.1118. It means that the GridGenerator class is almost untestable.

We initiate the enhancement of class testability, as outlined in the proposed TsDD approach in Section 2, by automatically identifying opportunities for refactoring. We then select one refactoring operation and apply it to the class. After that, we measure class testability to find whether the change in testability is useful. The refactoring is selected if it improves testability. Otherwise, it is not applied in practice. The getNeighbours method, shown in Fig. 6, suffers from the *long method* [12] smell detected by the JDeodorant [52] refactoring tool. In step 1, we fix this smell by applying the *extract method* refactoring [53] to the getNeihbours method. The method is refactored by extracting the lines that check whether two given grid cells are adjacent to a new method. We name the new method as isNeighbour, highlighted in Fig. 6. The refactoring reduces the complexity of the getNeighbours method and creates a new method that can be called whenever we need to check the adjacency of two grid cells.

As a result of refactoring, the actual testability of the GridGenerator class, computed after testing the class utilizing the mathematical testability model, increases from 0.1118 to 0.1890 (a relative improvement of 69.05 %). Our testability prediction model also shows an improvement from 0.3118 to 0.3890 (a relative improvement of 24.76 %). Based on the changes in source code metrics, particularly the average weighted methods per class (WMC) in this step, the prediction model estimates the testability of the refactored code. Extracting the highlighted region (lines 8 to 14) in Fig. 6 reduces the original method's cyclomatic complexity from 4 to 3. Moreover, the average cyclomatic complexity of the class reduces from 2.80 to 2.33. The reason is that two nested 'if' blocks are removed from the original method after applying the extracted method. Therefore, our model predicts relatively higher testability after applying this refactoring. The advantage of measurement by prediction is that it does not require the program to be tested, which significantly improves the efficiency of the process. In our example, it took about 6 min to recompute testability. However, the testability prediction was executed in less than one second. The reason for changes in the model prediction results after refactoring is the changes in source code metrics used in the feature vector of the testability prediction model.

We observed that the isNeighbour method suffers from *feature envy* [12] code smell. This method has six accesses to attributes in the ConsumerAttributes class, while it does not access any member of the GridGenerator class. Therefore, in step 2, we apply a *move method* refactoring [54] to move the isNeighbour method from the GridGenerator class to the ConsumerAttributes data class. As a result, the testability is improved from 0.1890 to 0.2026 (a relative improvement of 7.20 %). The move method refactoring removes both the feature envy and data class code smells simultaneously. It also eliminates the first parameter of the isNeighbour method, an instance of the class ConsumerAttributes to which the method is moved. This way, the attained coverage for the ConsumerAttributes class increases to 100 %. The class had no methods before, so adding a method with its own tests and low dependency increased the class coverage. The improvement in coupling metrics allows our prediction model to predict higher testability than step 1 at this step.

In step 3.a, we pick and apply the next refactoring operation, which is an extract method candidate (lines 9 to 14 of Fig. 7.a) identified by JDeodorant [52] to the positionAgentsOnGrid method. In this case, the actual testability of the GridGenerator class reduces from 0.2026 to 0.1688 (a relative decrease of 16.68 %), leading to a test with relatively lower coverage compared to step 2. Our testability prediction model also

Table 2
Test results and testability measures computed for the GridGenerator class.

Step	Refactoring	Coverage				Test suite		Testability	
		Branch	Statement	Mutation	Geometric mean	Size	Time	Predicted	Computed
# 0	None (initial version)	12.50 %	29.73 %	6.10 %	13.14 %	2	6 min	0.3118	0.1118
# 1	Extract method	11.54 %	33.33 %	8.93 %	15.09 %	2	6 min	0.3890	0.1890
# 2	Move method	15.00 %	34.38 %	12.35 %	18.54 %	2	6 min	0.4026	0.2027
# 3.a	Extract method	15.15 %	30.05 %	12.20 %	17.70 %	2	6 min	0.3420	0.1688
# 3.b	Simplifying conditional logic	72.20 %	93.33 %	65.33 %	76.07 %	7	6 min	0.4239	0.4639

```java
Vector getNeighbours(ConsumerAttributes a, int sightLimit){
int x0,y0;
int x,y;
Vector myNeighbours = new Vector();
x0 = a.getX();
y0 = a.getY();
for (int j=0; j <consumers.size(); j++){
  if (! (a.getName().equals(((ConsumerAttributes)consumers.elementAt(j)).getName()))){
     x = ((ConsumerAttributes)consumers.elementAt(j)).getX();
      y = ((ConsumerAttributes)consumers.elementAt(j)).getY();
      if ((Math.abs(x0-x) <= sightLimit) && (Math.abs(y0-y) <= sightLimit)){
         myNeighbours.addElement((ConsumerAttributes)consumers.elementAt(j));
      }
  }
 }
 return myNeighbours;
}
```

Fig. 6. The getNeighbours method in GridGenerator with an extract method refactoring opportunity.

```java
private void positionAgentsOnGrid(){
// help variables
 boolean done = false;
 int x,y;
 String agentName = "consumer_";
 Random gen = new Random();  // number Generator
 while (!done){
   if ((( x = gen.nextInt(NxN)) != 0) && (( y = gen.nextInt(NxN)) != 0)){
     agentName += x + "," + y;
      ConsumerAttributes a = new ConsumerAttributes(agentName,x,y);
      if (!checkList(a)){
         consumers.addElement(a);
         if (consumers.size() == population){done = true;}
      }
      agentName = "consumer_";
    }
  }
 }
```

(**a**) before refactoring.

```java
private void positionAgentsOnGrid(){
 int x,y;
 String agentName = "consumer_";
 Random gen = new Random();  // number Generator
  while (consumers.size() < population){
    if ((( x = gen.nextInt(NxN) ) != 0) && (( y = gen.nextInt(NxN) ) != 0)){
      agentName += x + "," + y;
      ConsumerAttributes a = new ConsumerAttributes(agentName,x,y);
      if (!checkList(a)){
         consumers.addElement(a);
       }
       agentName = "consumer_";
    }
  }
 }
```

(**b**) after refactoring.

Fig. 7. The body of positionAgentsOnGrid method before and after the application of simplifying conditional logic.

predicts lower testability for the refactored class. As a result, we discard the refactoring in step 3 and undo the refactoring to turn back the source code to step 2. The red color in the third row of Table 2 shows that the refactoring applied in this row should be discarded. Therefore, one must select and apply the next possible refactoring as suggested by the TsDD approach.

In step 3.b, after applying the *simplifying conditional logic* refactoring [12] to the positionAgentsOnGrid method, as shown in Table 2, the testability increases from 0.2026 to 0.4239 (a relative improvement of 109.23 %), and the number of test data generated in 6 min increases from two to seven. The higher number of influential test data implies higher productivity [55]. In Fig. 7.b, the nested structure caused by the "if" statement is flattened using a "while" construct. As a result, it is observed that the mean code coverage is improved from 18.54 to 76.07 %.

After step 3.b, there is no more refactoring opportunity to apply to the GridGenerator class. Therefore, the testability improvement process is finished. We ran the test data generator on the GridGenerator class five times during this process, which took about 30 min. However, using the testability prediction model helps one achieve the same results with a significantly lower time, in our example, less than five seconds.

3.2. Discussion on the motivating example

As the size and complexity of the software under test grow, the number of execution paths and, thereby, the cost of the test increases exponentially [56]. The test data generation could be time-consuming for projects with hundreds of complex classes [23]. In addition, a developing code that is not executable yet or depends on code that is not completed cannot be tested. A recent study by Khanam and Ahsan [57] on the advantages and pitfalls of TDD reveals the incapability of TDD in the rapid and cost-effective design of influential test cases for the projects under test. As a result, iterative testability improvement is impossible without an appropriate means to measure testability efficiently. Our proposed testability prediction model improves the efficiency of the TDD process by preventing unnecessary execution required to evaluate the testability during the improvement process with interactive refactoring. Moreover, the prediction model can be used even if the unit under test is not compilable since source code metrics are computed statically.

One may argue why measuring or predicting testability before and after any refactoring is necessary to improve the testability. The answer is that, as shown in the working example of Section 3.1, refactoring does not necessarily improve testability. A specific refactoring may improve a given source code's testability while degrading another's testability. The reason is that some source code metrics may not be improved or even degraded after refactoring. For instance, extract method refactoring may lead to the long-parameter list smell in some situations, depending on the code under refactoring. We observed this situation in step 3.a of the refactoring list in Table 2. As discussed in Section 5.2, not all refactorings enhance the testability of a given class in real-world programs. The order of refactoring operations also matters, especially when applying batch refactoring [58]. Hence, testability measurement is required to achieve the appropriate batch of refactoring operations, even after applying each refactoring, maximizing the testability. When testability measurement is used, in the cases that refactoring reduces testability, developers may roll back and discard them rapidly. Despite other quality attributes affecting software maintainability, testability improvement is not considered in automated refactoring studies [59] [60],.

In general, refactorings may improve any software quality. In TsDD, refactorings are performed for testability improvement before verifying and validating the code. Refactorings may be performed to improve the other quality attributes once the code passes all the tests. We need to measure testability to determine whether refactoring is required or has improved the quality. In TDD, refactoring preliminary emphasizes removing duplicates and redundancies [3]. However, refactoring is not dedicated to duplicate eliminations. TsDD improves TDD by refactoring for testability in the red phase. Basically, blind refactoring does not necessarily improve all source code quality attributes. By blind refactoring, we mean any refactoring without evaluating the impacts on the quality attributes such as testability. According to Mkaouer et al. [59], refactoring operations might improve or deteriorate the software quality, and testability is no exception.

4. Experimental setup

This section describes our testability prediction dataset, refactoring operations, and case study projects used to evaluate the TsDD approach. All experiments have been conducted on a Windows 10 computer machine with an Intel Core i7 CPU and 16 GB of RAM. We repeated each experiment five times and averaged the results to ensure their reliability.

4.1. Dataset

We used the SF110 corpus of the Java projects [23] to train and evaluate our testability prediction model. It consists of 110 open-source Java projects with 23,885 classes. The use of open-source projects guarantees the reproducibility of our study. EvoSuite was executed five times to generate tests for each class in SF110. Apparently, as the number of iterations increases, the variance of the results reduces. We noticed that when repeating the test data generation five times, the mean-standard deviation of the coverage provided by the generated test data reduces to 0.024, which is acceptable for use in training testability prediction models.

Table 3 summarizes the descriptive statistics of the testability values. The last column of Table 3 shows the average of the testability standard deviation values calculated over five different runs of each class. The low standard deviation of 0.024 in Table 3 indicates that the properties of the generated test suites have not been highly affected by the random nature of the evolutionary test data generation tool. In our experiments, 374,445 test cases were generated for nearly 23,000 Java classes. The testability of each class was computed using Eq. (7) based on the generated test data by EvoSuite. We removed Java classes for which EvoSuite failed to generate test data, which were a relatively low number (< 800). It should be noted that other test data generation tools can be used to generate tests for such classes. Fig. 8 shows the distribution of testability values in the obtained dataset. The testability value of most classes (2500 of 23,000) is observed to be between 0.5 and 0.6. At the same time, the data distribution is almost equal in all bins, making this dataset suitable for machine learning.

The question is, how good is a testability value? As shown in Table 3, the mean testability of the 23,000 Java classes is 0.5084. The value in Table 3 could be used to fuzzify the testability values. We used a descriptive statistical analysis approach to illustrate the locality and spread of testability values. Fig. 9 shows the box-whisker plot for testability values in our dataset. The quartile values, shown in Fig. 9, assist the developers in choosing the suitable threshold for testability when using the TsDD approach. For instance, choosing a threshold value, τ, greater than the third quarter (*i.e.*, 0.74) means that the testability of the given project should be greater than 75 % of all benchmark projects, indicating relatively high testability.

4.2. Refactoring operations

We selected seven refactoring operations for our experiments with TsDD. Based on available statistics [52], we chose the refactoring that developers have applied frequently, supported by automatic refactoring tools, and highly affected the source code metrics in Table 1. Other refactoring operations could be considered in future studies. Table 4 shows the selected refactorings together with the tools used to identify and apply each refactoring operation. The first column shows the selected refactoring operations, and the second column shows the software tools we used to perform the refactorings automatically.

4.3. Case studies

Three real-world Java projects containing scientific computations were chosen to experiment with the TsDD approach. Table 5 shows the Java projects, the application domain, and the selected classes. The comprehensive test of these projects is very time-consuming since they contain many classes with pretty long execution times due to scientific computations. For instance, Weka [25,30], various optimizations and statistical analyses on large datasets. EvoSuite [17] failed to generate test cases for some classes in these projects.

On the other hand, automated refactoring tools such as JDeodorant [52] support certain and limited types of refactoring operations. Therefore, only a subset of classes in these projects that are amenable to bad testing practices has been selected for our purposes in this study. For

Table 3
Descriptive statistics of testability values in the learning dataset.

Minimum	Maximum	Mean	Standard deviation (S.D.)	Mean SD
0.0	1.0	0.5084	0.1031	0.0240

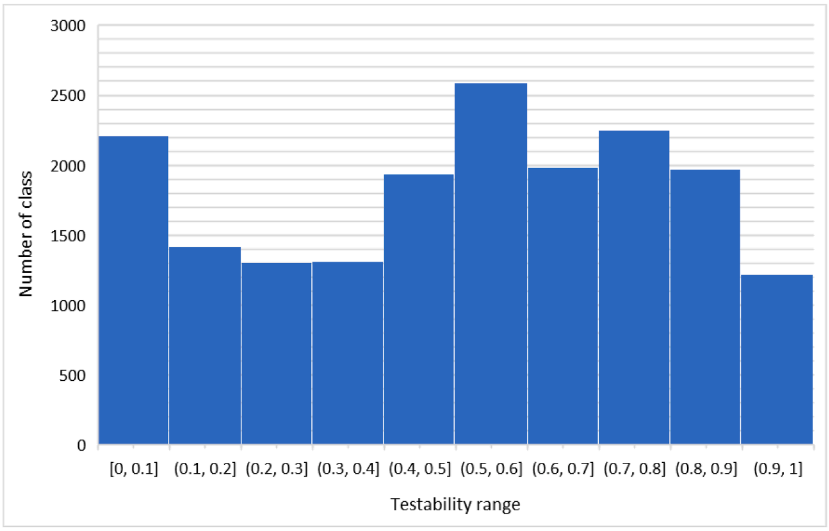

Fig. 8. Distribution of class testability in our dataset.

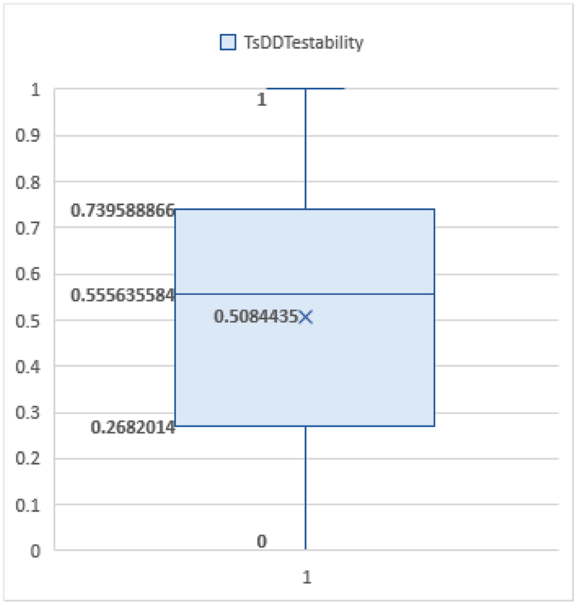

Fig. 9. The box-whisker plot of testability data from our experiment with 23,000 Java classes.

Table 4
Refactoring operations and tools used in the experiments.

Refactoring	Tool
Extract method	JDeodorant [52]
Move method	JDeodorant [52]
Extract class	JDeodorant [52]
Simplifying conditional logic	IntelliJ IDEA (C.E.) [61]
Make field final	IntelliJ IDEA (C.E.) [61]
Make field static	IntelliJ IDEA (C.E.) [61]
Remove dead code	IntelliJ IDEA (C.E.) [61]

these reasons, we selected classes satisfying the two conditions to be used in our experiments:

1. *At least one refactoring opportunity (code smell) is found in the class by either JDeodorant* [52] *or IntelliJ IDEA* [61] (*as listed in* Table 4),

Table 5
Java project used in our experiments.

Project	Application domain	Public classes	Selected classes
Weka [25]	Data mining software in Java	857	31
Scijava-common [26]	Java libraries for scientific computing	319	11
FreeMind [27]	*Mind* mapping application in Java	421	8
Total		**1597**	**50**

2. *At least one test case is automatically generated for that class before and after refactoring.*

In Scijava-common [26] and FreeMind [27] all classes with the above conditions were selected. However, for Weka [25], we randomly selected 31 classes with the mentioned conditions. The rationale behind 31 classes is that a minimum number of 30 samples is required for any data to follow the normal distribution [62]. As shown in Table 5, 50 Java classes were selected and used in our experiments with the TsDD approach. Addressing the limitations of automatic test data generation tools, automatic refactoring tools, and other existing tools enables the possibility of applying TsDD to more legacy code automatically.

5. Experimental results

This section elucidates the responses to the research questions delineated in Section 1. Section 5.1 substantiates the viability of employing the Testability-driven Development (TsDD) approach for the prediction of testability, as evidenced by the mean absolute error, mean square error, and R^2 score of each regression model. Section 5.2, which pertains to the second research question, RQ2, serves as a cautionary note to Test-Driven Development (TDD) practitioners that not all refactorings inherently enhance testability. Indiscriminate refactoring may inadvertently deteriorate specific quality attributes, including testability. This observation underscores the necessity for tools capable of quantifying testability before and after each refactoring operation.

In response to Research Question 3, Section 5.3 illustrates a notable enhancement in test effectiveness upon implementing TsDD. The authenticity of these results is substantiated by employing a box-whisker plot, which showcases progress in several test criteria - statement coverage, branch coverage, mutation score, test suite size, and test effectiveness - following the refactoring of specific classes across three projects.

Tests can indeed contribute significantly to the overall costs of software development. In response to RQ5, Section 5.4 illustrates the substantial reduction in test execution time achieved when applying the TsDD approach. This highlights a non-negligible increase in productivity and underscores TsDD as a superior method, particularly when aligned with the agility objective.

Research Questions 5 and 6 pertain to the influence of refactoring operations on testability and other quality attributes. The fundamental query to be addressed is: for what objective should refactoring be undertaken? Our empirical findings reveal the dichotomous effects of various refactoring operations on testability and specific other quality attributes. Consequently, we advocate for implementing a distinct refactoring phase subsequent to rectifying the code. Furthermore, we propose that refactoring techniques should be categorized according to their respective positive and negative impacts on a range of quality attributes.

5.1. Testability prediction performance analysis

This section addresses the first research question (RQ1): What is the performance of machine learning regression models to predict testability? The objective is to evaluate the accuracy and effectiveness of our machine learning non-linear regression models in predicting the testability of source code.

To evaluate the correctness of our testability prediction model, we computed the mean absolute error (MAE), mean square error (MSE), and R^2 score for each regression model on the test set, including 20 % of samples in our dataset. The metrics are computed by Eqs. (10) to (12). In these equations, $\widehat{y}_i$ is the predicted value of the ith sample, y_i is the corresponding true value for $\widehat{y}_i$ (computed by Eq. (7)), $\overline{y}$ is the mean value of y, and finally, n is the number of samples. MAE and MSE measure the model's prediction error for the data samples in the test set. The lower the error, the better the prediction model. The R^2 score indicates how well the model can predict unseen samples. Therefore, a model with a higher R^2 score has better performance. According to Eq. (12), the best possible value for the R^2 score is 1.0, and it can even be a negative number since the model can be arbitrarily worse. A constant model that always predicts the expected value of y, disregarding the input features, would get an R^2 score of 0.

$$\text{MAE}\ (y,\widehat{y}) = \frac{1}{n}\sum_{i=0}^{n-1}|y_i - \widehat{y}_i| \tag{10}$$

$$MSE\ (y,\ \widehat{y}) = \ \frac{1}{n}\sum_{i=1}^{n-1}(y_i - \widehat{y}_i)^2 \tag{11}$$

$$R^2\ (y,\widehat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2} \tag{12}$$

In conjunction with the primary dataset comprising 20 source code metrics, we employed the ‘SelectKBest’ feature selection method from the scikit-learn library [42]. This method utilizes ‘f_regression’ as its scoring function to select the top 10 features from our dataset. The ‘f_regression’ function is a scoring mechanism used in the ‘SelectKBest’ class of the scikit-learn library. It conducts univariate linear regression tests for each feature, yielding the F-statistic and the p-value. This function quantifies the linear relationship between a continuous target variable and each feature, selecting the most significant features based on the highest F-statistic or the lowest p-value. It should be noted that ‘f_regression’ operates under the assumption that the features and the target are normally distributed, exhibit a linear relationship, and that the variance of the target remains constant for each feature value.

Table 6 presents the evaluation results of the machine learning models on the test set and the optimal hyperparameters for each model, as determined during the model and feature selection processes. The most favorable result for each evaluation metric is highlighted in bold within each column. It is observed that all the models have been trained appropriately. The VoR model, trained on the primary dataset, demonstrates superior performance characterized by minimal error and maximal R^2 score.

Table 6
The result of evaluating regression models.

Model	Features	Best hyperparameters (in form of the Python statements)	MAE	MSE	R^2 score
DNN	All	'activation': 'tanh', 'hidden_layer_sizes': (256, 100), 'max_iter': 100, 'solver': 'adam'	0.1417	0.0381	0.5443
	10 best	'activation': 'tanh', 'hidden_layer_sizes': (512, 256, 100), 'max_iter': 100, 'solver': 'adam'	0.1507	0.0419	0.4999
RFR	All	'max_depth': 33, 'min_samples_split': 2, 'n_estimators': 400	0.1252	0.0324	0.6123
	10 best	'max_depth': 18, 'min_samples_split': 2, 'n_estimators': 400	0.1398	0.0386	0.5393
HGBR	All	'max_depth': 13, 'min_samples_leaf': 22	0.1268	0.0325	0.6107
	10 best	'max_depth': 8, 'min_samples_leaf': 12	0.1451	0.0394	0.5291
VoR	All	'dnn_weight': 0.17, 'rfr_weight': 0.33, 'hgbr_weight': 0.50	**0.1239**	**0.0311**	**0.6285**
	10 best	'dnn_weight': 0.17, 'rfr_weight': 0.33, 'hgbr_weight': 0.50	0.1407	0.0376	0.5521

Furthermore, it is noted that applying the feature selection technique does not enhance the performance of testability prediction. In fact, it appears to diminish the performance of the models. We conducted a paired *t*-test to ascertain whether the models' performance differences were statistically significant. Table 7 displays the p-value of the paired *t*-test concerning the root mean squared error of the models trained on the datasets with all features and selected features during the cross-validation process. The null hypothesis posits that the mean difference between two similar models is zero, while the alternative hypothesis asserts that the mean difference between those pairs of models is not zero. Based on the resultant p-values, the difference is statistically significant at a confidence level of 0.05 in all cases. Consequently, all selected source code metrics are deemed essential in predicting class testability.

We compared our proposed model with a branch coverage prediction model proposed in [39] and a linear regression model already used for testability prediction [63]. Fig. 10 presents a comparative analysis of our testability prediction model's mean absolute error (MAE) and R^2 score with the branch coverage prediction model [39] and a linear regression model [63] employed for predicting testability. It is discerned that the linear model exhibits inferior performance relative to the testability prediction model and the branch coverage prediction model. Our model surpasses both the linear and branch coverage prediction models in performance.

Table 7
Statistical test results on feature selection for testability prediction models.

Comparing root mean squared error (all *vs.* 10-best features)	DNN	RFR	HGBR	VoR
Paired *t*-test p-value	1.520×10^{-2}	6.4319×10^{-122}	6.1986×10^{-92}	1.277710^{-36}

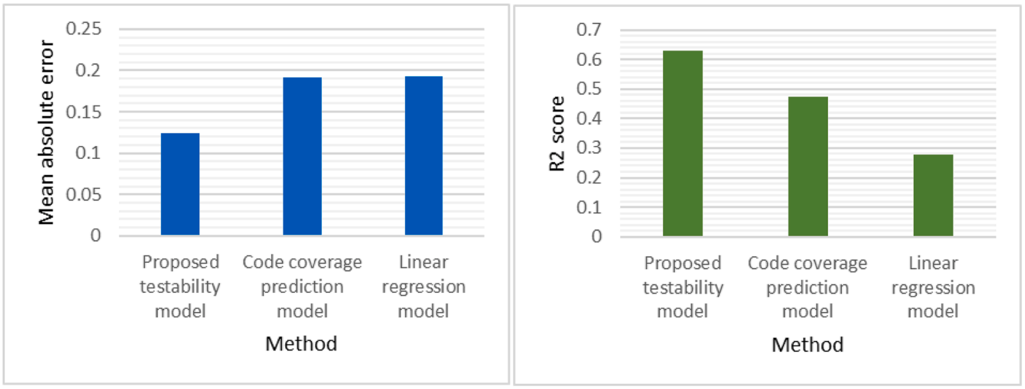

Fig. 10. Comparison of the prediction testability model with code coverage and linear regression prediction models.

Two salient observations emerge from this experiment. Firstly, a linear model cannot learn a sophisticated, non-linear relationship between source code metrics and testability. Secondly, the normalization of test quality with test effort culminates in a superior prediction model for testability. Models that concentrate solely on code coverage [39] underperform relative to ours, as two classes may exhibit identical code coverage with a disparate number of test cases. The relatively low prediction error furnishes a robust indication of accuracy, thereby affirming the applicability of our proposed prediction model.

5.2. TsDD effectiveness analysis

This section addresses Research Question 2 (RQ2) by comprehensively exploring the process of measuring testability prior to refactoring. The discussion revolves around how the structure of the refactored code can influence testability, potentially leading to its enhancement or degradation. Concurrently, when implementing Testability-Driven Development (TsDD) on selected classes, as referenced in Section 4.3, it is posited that using the precision model is indispensable for measuring testability after applying each refactoring. Subsequently, findings about improvements in testability are presented.

5.2.1. Testability improvement process

The question arises as to why we should measure testability before refactoring. The rationale behind this is that refactoring can potentially negatively impact testability in some instances. At the same time, in other cases, it can enhance testability. The extent of the impact relies on the composition of the refactored code. For example, as demonstrated in the first row of Table 8, applying the extract method refactoring to the findSplitNumericNumeric method of the weka.classifiers.trees.DecisionStump class in the Weka project [25] reduces its testability by approximately 0.023 (corresponding to a relative reduction of 10.47 %). This is because the extracted method includes eight parameters, leading to a long parameter list smell [12], which makes testing challenging. Additionally, in this specific case, the extract method refactoring generates some duplicate code that adds to the complexity of the DecisionStump class. However, as illustrated in the second row of Table 8, applying the same refactoring type, namely, the extract method, to the getNPointPrecision method of the weka.classifiers.evaluation package results in improved testability.

The ThresholdCurve class in the Weka project [25] experiences an improvement in testability by 0.1229 (representing a relative improvement of 166.53 %). In this scenario, the extract method refactoring diminishes the size and complexity of the refactored method. Similar trends are observed when implementing other types of refactoring operations. For example, as depicted in the third row of Table 8, the move method refactoring reduces the testability of the weka.classifiers.evaluation.ThresholdCurve class. Nevertheless, the move method enhances the testability of another method within the same class.

The remaining entries in Table 8 illustrate instances of refactoring operations that either diminish or leave the testability unchanged. Within the confines of Table 7, the symbol “↑” indicates improvements in testability, while the symbol “↓” represents a decrease. Furthermore, the symbol “≈” denotes that the refactoring operation has not changed testability. It is paramount to measure testability after applying each refactoring function to a class to determine whether it augments testability. Should a refactoring process reduce testability, it can be reversed, paving the way for exploring an alternative refactoring operation.

Applying the Testability-Driven Development (TsDD) methodology facilitates the identification of the optimal set of automated refactoring operations, thereby maximizing the testability of the given project. As elaborated in the subsequent section, this procedure was employed to identify and execute the refactoring operations that augment the testability of selected classes within three comprehensive Java projects.

5.2.2. Testability improvement analysis

To address Research Question 2 (RQ2), we conducted a comparative analysis of the testability of selected Java classes, both pre and post-refactoring. For this purpose, test data was automatically generated for each class listed in Table 5, with a time budget of six minutes allocated per class. The testability of each class was subsequently calculated using Eq. (7). The test data generation process was repeated five times for each experiment, and the results were averaged to mitigate the impact of randomness on the resulting test suite,

RQ1: ***What is the performance of machine learning regression models to predict testability?***

Answer to RQ1: *The machine learning non-linear regression models predict the source code testability with a mean absolute error of 0.1239 and an R^2 score of 0.6285. The results confirm the possibility of using testability prediction in the TsDD approach. Our models outperform code coverage prediction models with a higher R^2 score and lower prediction error. This improvement is mainly because of combining various test criteria to measure test effectiveness, normalizing the test effectiveness by the test effort, and using a larger dataset.*

Table 8
Impact of refactoring on testability value in different situations.

Class under test	Refactoring	Refactored entity	Testability	
			Before	After
weka.classifiers.trees.DecisionStump	Extract method	findSplitNumericNumeric	0.2168	0.1941 ↓
weka.classifiers.evaluation.ThresholdCurve	Extract method	sourceClass	0.0738	0.1967 ↑
weka.classifiers.evaluation.ThresholdCurve	Move method	makeInstance	0.0738	0.0688 ↓
weka.classifiers.evaluation.ThresholdCurve	Move method	getProbabilities	0.0738	0.2114 ↑
weka.classifiers.pmml.consumer.TreeModel	Make field final	m_classLabel	0.1164	0.1075 ↓
weka.classifiers.pmml.consumer.Regression	Simplifying conditional logic	determineNormalization	0.0134	0.0123 ≈
weka.classifiers.trees.DecisionStump	Move method	printDist	0.2168	0.1924 ↓
weka.core.pmml.Apply	Extract method	toString	0.0221	0.0125 ↓
weka.classifiers.pmml.consumer.RuleSetModel	Extract method	score	0.0608	0.0552 ↓

Test data generation was performed before and after refactoring to assess changes in testability. Following applying a refactoring operation, we employed testability prediction to ascertain whether there had been an improvement in testability. If a refactoring operation decreased source code testability, as exemplified in Table 8, it was disregarded, and the refactoring would be reversed during the pink-blue phase of TsDD. The practice of measuring testability before and after each individual or sequence of refactorings, as suggested by [59], assists developers in verifying the efficacy of the refactoring. In instances where testability is reduced, the developer has the option to reverse or roll back the refactoring.

We continued the refactoring process until no more refactoring opportunities were found with the tools mentioned in Table 4. As discussed earlier, the testability prediction model avoids repeating test data generation after applying each refactoring for testability and highly increases the efficiency of the process. It should be noted that refactoring to improve other quality attributes is still applied in the refactoring (Blue) phase of TsDD, just like the original TDD approach. Moreover, as discussed in Section 5.5, refactoring for testability also improves the functionality and reusability of software components in addition to testability.

Tables 9 to 11 represent the testability improvements for three real-world, large-scale software, Weka [25], Scijava-common [26], and Free-mind [27], over the TsDD life cycle. The good news for the developers of these software systems with an active development community is that our refactorings have made it much easier to test their code, which is continuously being improved and updated. The testability of Weka [25], Scijava-common [26], and Free-mind [27] are respectively improved by 0.1335 (94.75 %), 0.2594 (95.05 %), and 0.0554 (43.62 %). The average improvement in testability of evaluated classes is 0.1494 (77.81 %).

The number of refactorings applied to each class and testability values in our case study projects are also reported in Tables 9, 10, and 11. It shows that legacy code units must be refactored multiple times to prepare for testing. The average improvement per refactoring operation is 0.0090 (6.37 %), 0.0587 (21.51 %), and 0.0048 (3.70 %), respectively, for the Weka [25], Scijava-common [26], and Free-mind [27] projects. Even though we did not experiment with different sequences of

Table 9
The result of applying TsDD to the Weka project.

I.D.	Package	Class	Number of refactorings	Testability	
				Before	After
1	weka.classifiers.pmml.consumer	Regression	14	0.0135	0.2167
2	weka.classifiers.pmml.consumer	GeneralRegression	27	0.0231	0.0561
3	weka.core	ClassCache	4	0.0255	0.0588
4	weka.classifiers.evaluation	MarginCurve	3	0.0457	0.60
5	weka.classifiers.pmml.consumer	RuleSetModel	12	0.0608	0.0734
6	weka.classifiers.evaluation	ThresholdCurve	8	0.0738	0.2902
7	weka.classifiers.pmml.consumer	NeuralNetwork	31	0.0824	0.1655
8	weka.classifiers.meta	MultiClassClassifierUpdateable	3	0.1013	0.3897
9	weka.core	Optimization	14	0.1086	0.1575
10	weka.classifiers.bayes.net	BIFReader	15	0.1149	0.1975
11	weka.classifiers.evaluation	EvaluationUtils	6	0.1156	0.1702
12	weka.classifiers.pmml.consumer	TreeModel	26	0.1164	0.1539
13	weka.core.neighboursearch.kdtrees	KMeansInpiredMethod	12	0.1365	0.1891
14	weka.classifiers.lazy.kstar	KStarNominalAttribute	15	0.1430	0.5201
15	weka.classifiers.bayes.net.estimate	SimpleEstimator	11	0.1447	0.5841
16	weka.associations	AprioriItemSet	18	0.1452	0.1796
17	weka.classifiers.rules.part	MakeDecList	6	0.1475	0.3564
18	weka.filters.supervised.attribute	NominalToBinary	10	0.1564	0.1605
19	weka.classifiers.rules	OneR	14	0.1675	0.1784
20	weka.core.neighboursearch.kdtrees	SlidingMidPointOfWidestSide	6	0.1709	0.3676
21	weka.classifiers.bayes.net.search	SearchAlgorithm	10	0.1727	0.4401
22	weka.clusterers	EM	23	0.1867	0.3329
23	weka.attributeSelection	PrincipalComponents	17	0.1876	0.1955
24	weka.classifiers.functions	Logistic	11	0.1987	0.2516
25	weka.attributeSelection	ReliefFAttributeEval	15	0.2043	0.2570
26	weka.classifiers.trees.j48	C45Split	16	0.2074	0.3075
27	weka.classifiers.rules.part	ClassifierDecList	14	0.2121	0.3863
28	weka.classifiers.bayes.net	EditableBayesNet	31	0.2145	0.2268
29	weka.classifiers.functions	MultilayerPerceptron	49	0.2180	0.2907
30	weka.classifiers.trees.j48	ClassifierTree	8	0.2255	0.3015
31	weka.classifiers	Evaluation	27	0.2473	0.4518
Average			**14.88**	**0.1409**	**0.2744**

Table 10
The result of applying TsDD to the Scijava-common project.

I.D.	Package	Class	Number of refactorings	Testability	
				Before	After
1	org.scijava.script	ScriptFinder	1	0	0.7220
2	org.scijava.annotations.legacy	LegacyReader	8	0.1495	0.1546
3	org.scijava.util	MirrorWebsite	6	0.1626	0.5428
4	org.scijava.script	ScriptREPL	2	0.1835	0.4846
5	org.scijava.script	ScriptModule	6	0.2057	0.2744
6	org.scijava.script	ScriptLanguageIndex	3	0.2134	0.3359
7	org.scijava.script.process	ParameterScriptProcessor	3	0.2332	0.6846
8	org.scijava.util	TunePlayer	5	0.2558	0.3760
9	org.scijava.menu	ShadowMenu	9	0.3098	0.7282
10	org.scijava.tool	DefaultToolService	8	0.5833	0.6989
11	org.scijava.script	ScriptInfo	2	0.7048	0.8529
Average			**4.42**	**0.2729**	**0.5323**

Table 11
The result of applying TsDD to the Free-mind project.

I. D.	Package	Class	Number of refactorings	Testability	
				Before	After
1	freemind.main	FreeMindCommon	3	0	0.0186
2	freemind. modes. mindmapmode	MindMapMapModel	23	0	0.0305
3	freemind.view. mindmapview	NodeView	29	0.0232	0.0389
4	freemind. controller. filter	FilterController	4	0.0254	0.1568
5	accessories. plugins.time	TimeManagement	9	0.0627	0.0161
6	accessories. plugins	ClonePlugin	18	0.1709	0.2656
7	freemind. modes	MapRegistry	7	0.3185	0.3781
8	freemind. controller	MapModuleManager	13	0.4155	0.5546
Average			**11.78**	**0.1270**	**0.1824**

refactorings [59] and only used a limited set of refactoring operations, it is observed that automated refactoring of classes with low testability could highly improve their testability. In the future, we should consider composition [64] and batch-refactoring [58,65], to find the best sequence of refactoring operations that improve testability. In Tables 10 and 11, the testability of some classes is zero before refactoring despite at least one test case. According to Eq. (3), if one of the coverage criteria is zero, then $C(X)$ would be zero, which results in zero testability. Therefore, it is observed that TsDD improves different coverage criteria. The precise impact of testability improvement on the program code coverage and test effectiveness is discussed in Section 5.3. We observed that code coverage of evaluated classes increases after refactoring, increasing the probability of finding faults in these classes.

The prediction model enables us to analyze the root causes of testability improvement with automated refactoring. The main reason for the testability improvement has been the changes in the values of source code metrics made by the refactorings. The changes in each of the 20 object-oriented metrics used for testability prediction are illustrated by point plots in Fig. 11. The main observation is that the testability values are sensitive to these metrics. Therefore, any improvement in these metrics made by refactorings improves the testability. As a result, our testability prediction model can be used to guide the refactoring process toward maximizing testability. We used our testability prediction model to measure class testability before and after each refactoring.

Fig. 12 shows a relatively high correlation between the predicted result and the actual values computed by our mathematical model, confirming the applicability of the prediction model used in the TsDD approach. The correlation coefficient between measured and predicted testability is 0.4540 and 0.5476, respectively, before and after refactoring. Our testability prediction model predicts higher testability values more accurately than lower ones.

5.3. *Code coverage analysis*

In response to Research Question 3, which seeks to understand the average improvement in test effectiveness when applying the TsDD approach, Fig. 13 presents a box-whisker plot of various test criteria, including statement coverage, branch coverage, mutation score, test suite size, and test effectiveness before and after refactoring studied classes of the three projects listed in Table 5. The test effectiveness is calculated using Eq. (4). The plot reveals that the average value of all the criteria improves after the refactorings. Despite the constant test time budget for all experiments, this improvement is deemed significant. This is because a smaller test suite size, which includes only influential test cases, is also considered an enhancement.

It is worth noting that a box-whisker plot, also known as a boxplot, is a type of data visualization representing information in the form of a five-number summary: the minimum, first quartile, median, third quartile, and maximum. It provides a clear and concise summary of the data, making it easier to understand the distribution and detect any outliers. In this context, the box-whisker plot effectively illustrates the changes in the test criteria before and after refactoring, thereby providing a quick and easy comparison of the impact of the refactoring process on different metrics.

As shown in Fig. 13, the statement coverage for unit testing of Weka [25], Scijava-common [26], and Free-mind [27] are enhanced respectively by an average of 0.1585 (72.44 %), 0.6228 (72.14 %), and 0.0973 (52.97 %). Accordingly, the branch coverage of these projects is respectively enhanced by an average of 0.1619 (105.06 %), 0.3517 (113.29), and 0.0762 (40.54 %). All evaluated classes' statements and branch coverage are improved by 0.1716 (70.01 %) and 0.1899 (97.95 %) after refactoring for testability. Finally, our experiments' average improvement of mutation score and test effectiveness is 0.1849 (106.98 %) and 0.1821 (97.71 %).

The Wilcoxon rank-sum test, a nonparametric statistical test that compares two paired groups, was utilized with a 95 % confidence level ($\alpha = 0.05$) to determine whether the differences in the mean of different coverage criteria and test effectiveness are statistically significant. The null hypothesis (H0) was defined such that the mean of a given test criterion on classes in our experiments is equal before and after refactoring for testability. The alternative hypothesis (H1) was defined as the mean of a given test criterion on classes in our experiments after refactoring for testability being greater than the mean of the test criterion on the same classes before refactoring. The p-value of the Wilcoxon rank-sum test corresponds to the probability of rejecting the null

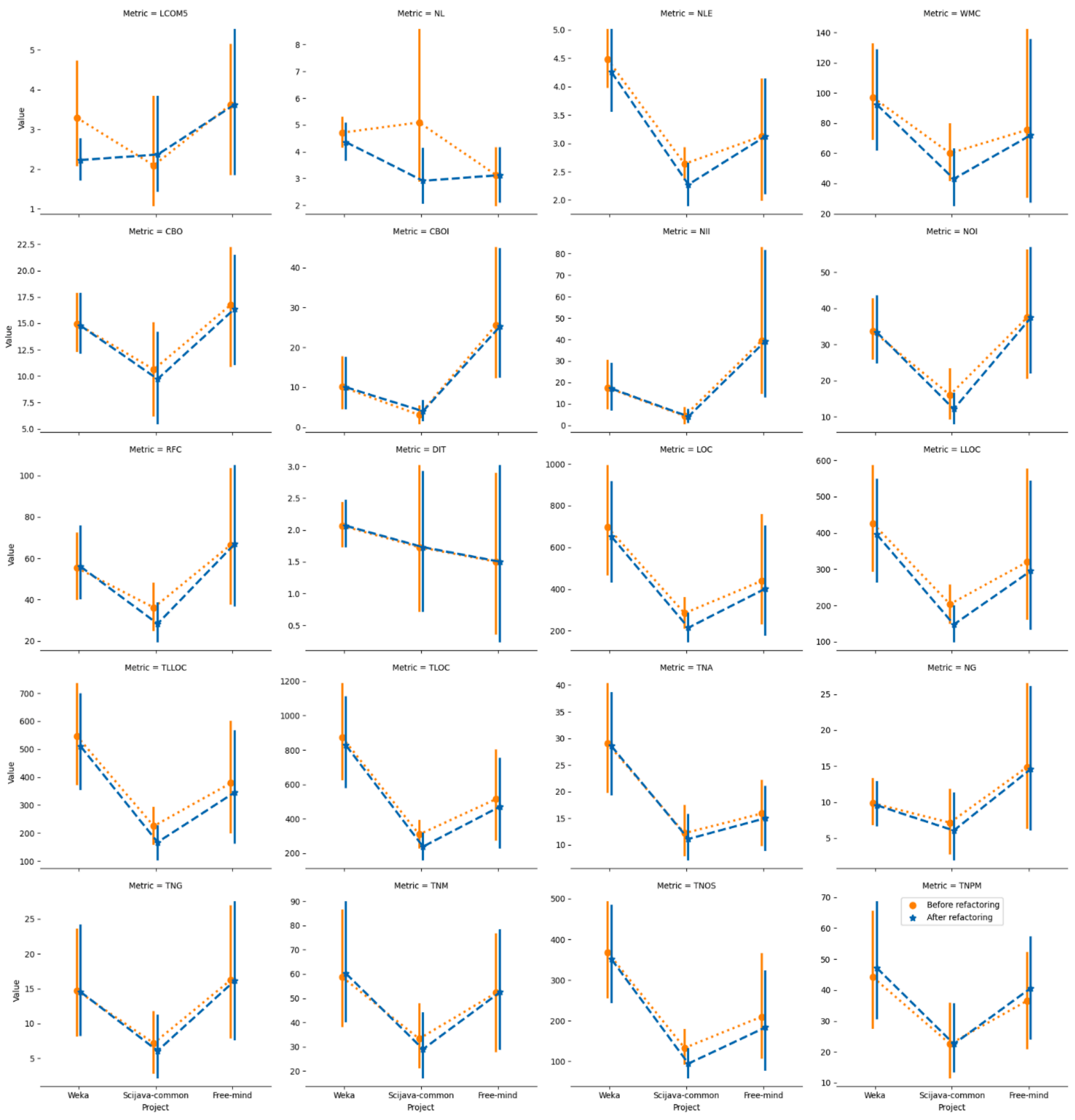

Fig. 11. Value of source code metrics before and after refactorings.

hypothesis H0 while it is true (type I error)—a p-value less than or equal to α (≤0.05) means H0 is rejected concerning H1.

It is worth mentioning that the Wilcoxon rank-sum test is a nonparametric test of the null hypothesis that, for randomly selected values X and Y from two populations, the probability of X being greater than Y is equal to the probability of Y being greater than X. The null hypothesis (H0) posits that there is no difference or relationship between the two groups being compared, while the alternative hypothesis (H1) proposes that there is a difference or relationship. The null hypothesis is rejected if the p-value is less than or equal to the significance level (α), indicating a statistically significant difference between the two groups. The p-value in statistics measures the strength of evidence against the null hypothesis.

Table 12 presents the p-values, representing the likelihood of observing a result as extreme as the one obtained, assuming the null hypothesis is true. These p-values are derived from the Wilcoxon rank-sum test, a non-parametric method for comparing two related samples or repeated measurements on a single sample. This test assesses whether the population mean ranks of these samples differ.

We observed that for Weka and Scijava-common, all code coverage criteria and the test effectiveness improved significantly after refactoring. However, the branch and mutation coverage did not improve significantly for the Free-mind project. We recommend applying new refactoring operations in such cases, which should be explored in future studies. Overall, the improvement of all criteria for all projects was statistically significant, which means that refactoring for testability

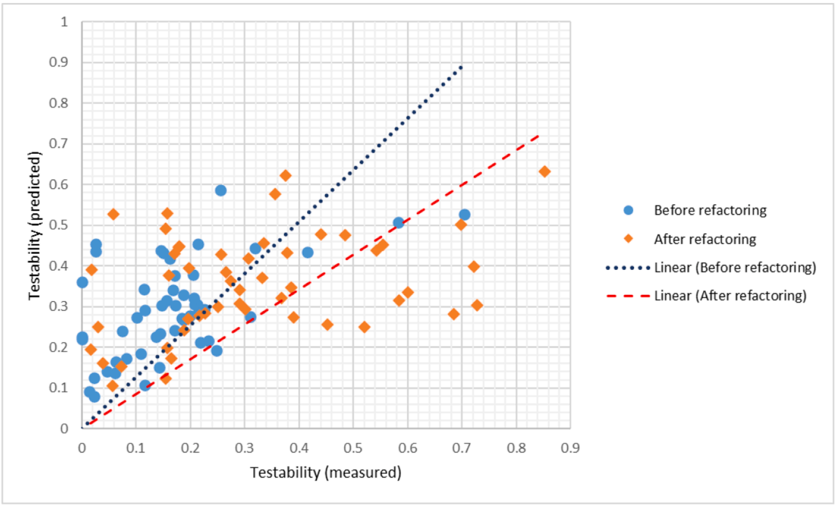

Fig. 12. Correlation between measured and predicted testability values before and after refactoring for all the classes.

RQ2: ***On average, how much testability may improve when applying the TsDD approach?***

Answer to RQ2: *A limited set of automatically applied refactorings improves the testability of 50 Java classes with an average of 0.1494 (77.81 %). According to our experiments, improving testability with automated refactoring is promising. The testability prediction model conducts the improvement process by statically predicting testability even before the code is runnable and without runtime overheads.*

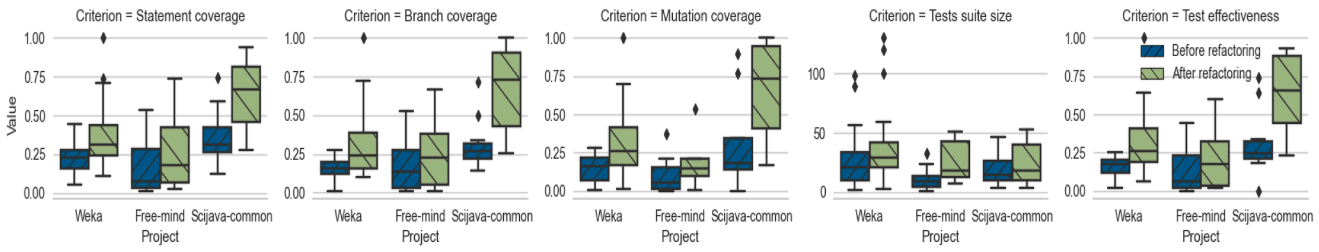

Fig. 13. Test coverage, size, and effectiveness before and after the refactorings.

Table 12
Results of the Wilcoxon rank-sum test (p-values) for various test criteria on selected classes.

Project	Statement coverage	Branch coverage	Mutation coverage	Test effectiveness
Weka (+ + + +)	2.5987×10^{-4}	8.2936×10^{-5}	5.5865×10^{-4}	3.4866×10^{-5}
Scijava-common (+ + + +)	3.9136×10^{-3}	2.1422×10^{-4}	9.8738×10^{-3}	2.6292×10^{-3}
Free-mind (+ − − +)	2.0041×10^{-2}	2.3112×10^{-1}	1.0379×10^{-1}	2.0041×10^{-2}
All (+ + + +)	1.2923×10^{-4}	3.7940×10^{-5}	9.3318×10^{-5}	1.7126×10^{-5}

enhanced the test effectiveness. It should be noted that the results in the last row of Table 12 were significant, with p-values less than the Bonferroni correction threshold, *i.e.*, $\frac{\alpha}{16} = 0.003$.

5.3.1. *TsDD process efficiency analysis*

TsDD recommends conducting tests after each refactoring operation. Each test may necessitate a unique test suite, contingent on the modifications introduced by the refactoring into the source code. For instance, applying a move method refactoring would require modification of the corresponding unit test class. Both manual and automatic testing become labor-intensive in such situations, particularly in extensive projects [18]. TsDD mitigates this issue by predicting testability and measuring the improvement in testability after each refactoring operation. Prediction is significantly faster than actual testing and does not demand all sections of the source code to be executable. This section contrasts the efficiencies of TsDD and TDD methodologies in legacy code bases, encompassing the projects listed in Table 5 and newly introduced features by expert developers in industrial projects.

5.3.2. *Test efficiency evaluation in legacy codebases*

As depicted in Figs. 141516, the TsDD approach, in comparison to the Traditional Test-Driven Development (TDD) method, significantly reduces the cumulative time required for automatic test data generation for Weka, Scijava-common, and Free-mind by 884.95, 83.84, and 195.76 min, respectively. In the TDD approach, test data are generated and executed after each refactoring operation, whereas in the TsDD approach, test data are only generated and executed once refactoring for testability has been completed. The substantial decrease in development time is primarily attributed to the fact that, in the TsDD approach, we

RQ3: ***On average, how much test effectiveness may improve when applying the TsDD approach?***

Response to RQ3: *The application of the TsDD approach leads to a significant improvement in the test effectiveness of 50 Java classes. Specifically, the average gain is 0.1821 (95.71 %). The primary factor contributing to this enhancement is the generation of more influential test cases within a fixed time budget, especially after refactorings. The Wilcoxon rank-sum test supports the statistical significance of this overall improvement in code coverage criteria across the experimental scenarios, including statement coverage, branch coverage, and mutation coverage.*

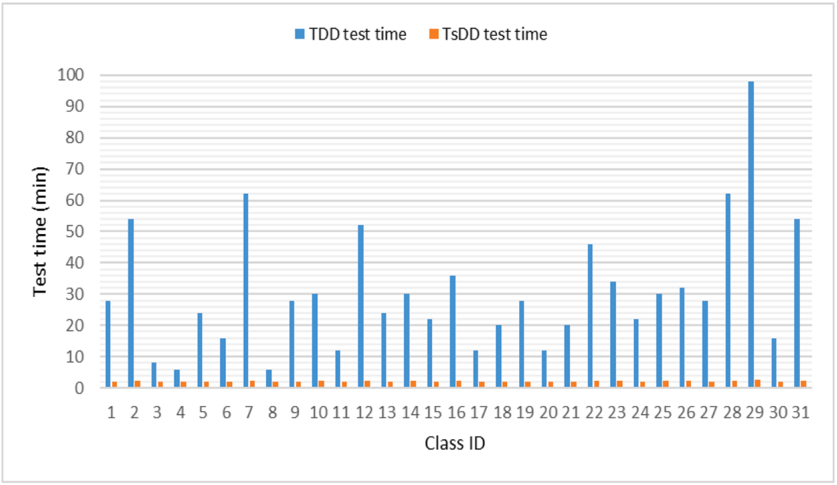

Fig. 14. Test time for TDD and TsDD in the Weka project.

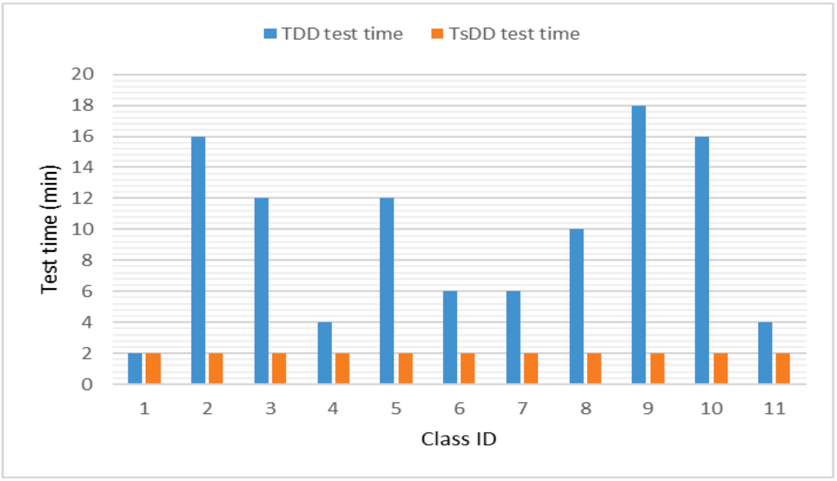

Fig. 15. Test time for TDD and TsDD in the Scijava-common project.

avoid generating and executing test data for the program after applying each refactoring, instead opting to predict testability statically.

5.3.3. *Manual evaluation of TSDD efficiency*

We rely on evaluations from experts' perspectives to substantiate our findings regarding the efficiency of TsDD on newly developed features. We engaged with three seasoned Java developers possessing experience in TDD to assess the development time when utilizing both TDD and TsDD methodologies in their regular workflows. The task assigned to these developers was to incorporate a new feature into one of the systems they were working on, employing both TDD and TsDD approaches, and subsequently answering a series of questions:

- **Q1:** *What is the number of changes in the test codes (refactoring test codes)?*
- **Q2:** *What is the number of execution of tests to complete the feature?*
- **Q3:** *What is the number of applied refactorings to achieve quality code?*

We undertook a training session to explain the TsDD approach to all developers. We requested them to utilize our prediction model, automated test data generation, and refactoring tools to accomplish their objectives and complete their tasks. Table 13 displays the results reported by the participating developers for both TDD and TsDD methodologies. For instance, it is observed that Developer #2 made seven alterations in test codes during the TDD phase, whereas he made only four modifications when using TsDD. Regarding questions 1 and 2, the number of test code changes and executions has significantly reduced.

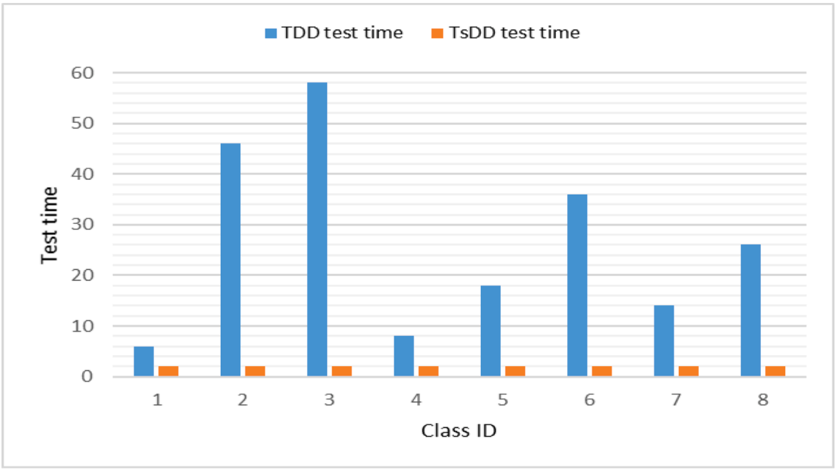

Fig. 16. Test time for TDD and TsDD in the Free-mind project.

Table 13
TDD and TsDD evaluations by experts.

Developer ID	TDD			TsDD		
	Q1	Q2	Q3	Q1	Q2	Q3
Dev #1	2	3	1	1 ↓	1 ↓	2 ↑
Dev #2	7	11	3	4 ↓	2 ↓	4 ↑
Dev #3	4	6	2	1 ↓	2 ↓	2

When tests are generated automatically, only the oracles of the test need to be altered.

Moreover, the testability results of refactoring can be evaluated using the prediction model, eliminating the need to generate and execute all pertinent tests. Contrarily, Developers 1 and 2 have conducted more refactoring operations in the TsDD method due to their knowledge of testability, as provided by our learned model. Expert opinion affirms that TsDD can circumvent unnecessary test executions after each refactoring, thereby enhancing testability and accelerating the process of developing quality code.

5.4. Refactoring analysis

To explore which types of refactoring can improve testability more than the others, we computed Spearman's rank correlation coefficient between each refactoring operation and the change in the testability values. To this aim, we counted the number of occurrences of each refactoring type for each class in Tables 8, 9, and 10. After that, we computed the testability improvement for each class by subtracting testability after refactoring from the testability value before refactoring. Finally, we calculated Spearman's rank correlation coefficient to find the monotonicity of the relationship between a refactoring type and testability improvement. The Spearman's rank correlation coefficient, ρ, is calculated by Eq. (13):

$$\rho = 1 - \frac{6\sum_{i=1}^{n} d_i^2}{n(n^2 - 1)} \tag{13}$$

where n is the number of samples and d_i is the difference between the two ranks of each sample.

Table 14 shows the contribution of each refactoring to testability improvement. The Spearman coefficient describes the relationship's strength and direction, and the p-value with a 95 % confidence level (α=0.05) reveals whether the computed correlation is statistically significant. The "+ " sign beside each refactoring operation reveals a statistically significant correlation coefficient. The "− " sign denotes the opposite.

It is observed that all selected refactoring operations except the *make filed static* can improve testability. However, the improvement is statistically significant (p-value < 0.05) for only three out of seven refactoring operations, including the extract class, remove dead code, and move method. Therefore, our experiments cannot confirm the statistical significance of the impact of the remaining refactoring on class testability due to the associated p-values. It implies that more experimental samples are required to discuss the implications of these refactorings, which should be considered in future studies on TsDD.

The low positive correlation coefficient (ρ < 0.5) [66] indicates that

Table 14
Spearman correlation coefficient with an associated p-value for applied refactoring.

Refactoring	Spearman's rank correlation	Associated p-value
Extract class (+)	0.31583	1.370×10^{-3}
Remove dead code (+)	0.20593	3.983×10^{-2}
Move method (+)	0.19326	5.404×10^{-3}
Extract method (−)	0.19105	5.691×10^{-2}
Simplifying conditional logic (−)	0.06594	5.145×10^{-1}
Make field final (−)	0.03597	7.223×10^{-1}
Make field static (−)	− 0.11943	2.145×10^{-1}

RQ4: ***On average, how much total testing time is reduced when applying the TsDD approach?***

Answer to RQ4: *Compared to the TDD approach, the proposed TsDD approach significantly reduces the test duration of 50 Java classes during development, averaging 388.18 min (approximately 6.5 h) per class. These findings underscore a significant increase in productivity, thereby establishing TsDD as a superior method to other agile software development approaches, especially the traditional TDD approach. Our expert developers also support these findings by adding new features to existing software systems.*

although refactoring with testability improvement as a goal tends to increase testability in general, the effect of a specific refactoring operation on class testability is not equal for all classes. Software refactoring may improve a class's testability while not changing or decreasing the testability of other classes. We conclude that a refactoring operation's impact on a class's testability highly depends on that class. It is worth noting that in all measurements, the testability changes have been only computed for the refactored class to minimize the impacts of refactoring orders on Table 14 results.

5.5. *TsDD impact on other quality attributes*

TDD suggests writing the minimum amount of code to make the test pass. In the "Green" phase, the aim is to write enough code to just cover the test. This is where unit testing comes into play. The code written at this stage is often a unit of functionality, which is what unit testing is designed to test. Unit testing suggests testing in isolation, which is supported by mocks and stubs. TsDD supports TDD in the green phase by ensuring the source code testability for effective and efficient tests. In the green stage, refactoring aims to improve the testability of the unit of the code under development to its test coverage. However, when performing refactoring iteratively, the order of the refactorings can highly affect the improvement in testability. However, searching for an appropriate order of refactoring is not a part of the TsDD methodology.

In the green phase, the primary concern of TsDD is to ensure test effectiveness and efficiency while refactoring for testability. Once all the unit tests succeed in the green stage, similar to TDD in the blue phase, TsDD suggests refactoring to improve desired quality attributes, including testability for other tests such as regression, integrated, system, and acceptance tests. A promising point about the TsDD approach is that it can simultaneously improve some other quality attributes with testability while refactoring. TsDD attempts to enhance testability to an acceptable level by frequently using its testability prediction model to measure testability, followed by refactoring. The model measures testability based on the value of source code metrics computed for the class under test. Therefore, the testability is enhanced as the influential metrics are improved. Any improvement in the metrics may also affect the other quality attributes.

In response to Research Question 6 (RQ6), we assessed the effects of our chosen refactorings on four additional quality attributes, reusability, functionality, extensibility, and effectiveness, in conjunction with testability. These quality attributes were quantified using the Quality Model for Object-Oriented Design (QMOOD) method [67], a hierarchical measurement technique grounded in object-oriented metrics. Definitions of these quality attributes are presented in Table 15.

The QMOOD quality attributes [67] have been previously employed in other software engineering research, specifically in the domain of automated refactoring [59],[68],[69], owing to their critical role in the software development lifecycle. However, to the best of our understanding, the interrelationship between these quality attributes and testability has not been explored in prior studies. Consequently, this section investigates the impact of the Test-Driven Development (TsDD) approach on quality attributes beyond testability.

Table 15
QMOOD quality attributes definitions [67].

Quality attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that incorporates new requirements in the design.
Effectiveness	Refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

We computed each quality attribute before and after applying Test-Driven Development (TsDD) on selected classes within three Java projects. Fig. 17 delineates the alterations in each quality attribute for the Weka [25], Scijava-common [26], and Free-mind [27] projects. The functionality of all projects was enhanced during the TsDD process. The reusability and extensibility were either augmented or remained unchanged. However, the effectiveness of the two projects diminished, indicating a potential conflict between testability improvement and program effectiveness. The conflict among QMOOD quality attributes has been addressed in [59] and [70], albeit without considering testability.

Based on our experimental results shown in Fig. 17, testability improvement in the TsDD approach improves the functionality and reusability of software components. However, it tends to reduce design effectiveness concerned with object-oriented design concepts and techniques. The impact of testability on the effectiveness of object-oriented design is a complex issue. Effectiveness is an important quality factor of object-oriented software, especially during the design phase for a high-quality product. It supports developers in achieving the specified functionalities, characteristics, better design quality, and behavior using appropriate object-oriented design (OOD) concepts and procedures. However, as our results suggest, efforts to improve testability might sometimes come at the expense of design effectiveness. This could be due to the inherent complexities and trade-offs in software design and testing [71]. For example, Encapsulation, a metric in object-oriented design, positively influences effectiveness but negatively affects testability. This is because as the quantity of a class's public attributes and methods increases, its Encapsulation, and consequently its effectiveness, diminishes. Conversely, the testability of the code is enhanced. Our observation suggests that the enhancement of testability in Java classes with at least one refactoring opportunity may adversely affect other quality attributes, specifically program effectiveness, as per QMOOD [67].

6. Threats to validity

Several factors discussed in this section may threaten the validity of our proposed approach and evaluations. The main threat to construction validity is that we used three test adequacy criteria and a limited set of source code metrics to build our testability model. Indeed, we used test

RQ5: ***Which refactoring operations improve testability more than the others?***

Answer to RQ5: *Three out of seven investigated refactoring operations, including extract class, remove dead code, and move method, significantly increase class testability, while the "make field static" refactoring seems to reduce testability. The impact of a specific refactoring operation on software testability is not the same for all classes. The sequence in which refactorings are implemented undoubtedly influences their cumulative effects on testability. However, identifying an even near-optimal sequence of refactorings could prove to be a labor-intensive task, particularly during the green phase when the software is in a state of ongoing development and continuous modification.*

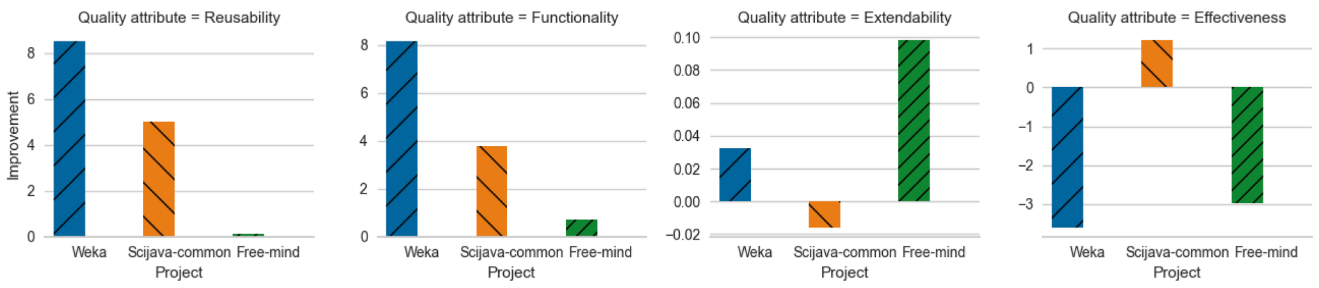

Fig. 17. The impact of TsDD on other quality attributes.

RQ6: ***What is the impact of TsDD on the other software quality attributes?***

Response to Research Question 6: *Refactoring for testability in the green phase positively influences specific quality metrics used to measure testability, akin to other quality attributes such as reusability, extendability, and functionality. However, it adversely affects other quality attributes, notably design effectiveness. This observation is not contingent upon the scope of our experiments and is directly related to the metrics employed to quantify other software quality attributes that overlap with the metrics utilized to calculate testability.*

criteria that mainly affect the testability of the software under test according to the existing research [21,72], [73], and can be accurately collected. A recent study also uses source code metrics to build the prediction model for software reusability [74]. The advantage of these metrics is that they can be computed efficiently with existing source code static analyzers such as [48]. Other test adequacy criteria and source code metrics can be used to increase the accuracy of both the proposed mathematical model and prediction model for testability. Another threat to construct validity is using only three machine learning regression algorithms to learn testability protection. We combined these individual regressors to build a robust voting regressor. Nevertheless, other machine learning algorithms and their combination can be considered to construct a more precise testability prediction model.

The most crucial threat to internal validity is using the automatic test data generation tool to generate test data and measure the actual testability of each class. Generating manual test data for a large number of source code classes is very time-consuming. However, we observed that the automatic test data generation tool [51] could not generate tests for simple classes that deal with files. For instance, a Java class with a method that reads the contents of a file and then performs a simple action depending on the contents may look simple in terms of complexity metrics. At the same time, EvoSuite [34] would be unlikely to produce the appropriate file content in the first place. We carefully preprocess data and remove simple classes with low test results to mitigate these behaviors and increase the model's reliability. Meanwhile, testability prediction seems to be a challenging machine learning task, and more data samples are required to improve the performance of prediction models. In addition, the evaluation by experts' opinions needs to be extended by a family of experimental development scenarios to reveal reliable results regarding the usefulness of TsDD in practice.

Another threat to the internal validity of our experiments is that we applied refactoring operations in a random order without considering the order of refactoring. The order of refactoring operations can imply quality improvement or worsening. For example, a "local" worsening can be an opportunity for "global" improvement and *vice versa*. However, the order of applying refactoring operations is irrelevant when they target different and non-overlapping entities, as used in our experiments. Indeed, our experiments consider only local improvement of the class testability. In the future, search-based refactoring methods can be used to find the best sequence of refactoring operations.

Concerning external validity, the main threat is the generality of results when used in other projects, specifically with programming languages other than Java. We demonstrate the applicability of TsDD on 50 classes in three real-world Java projects with non-trivial size and complexity. Therefore, the result is expected to be accessible in similar projects. Our testability prediction models can be used to determine the testability of programs written in other programming languages, especially those in the Java family, *i.e.*, C# and C++. Nevertheless, it is required to evaluate the performance of learned models in different programming languages and even various Java programs to ensure their applicability. Finally, the validity of our conclusions may be threatened in some cases where statistical tests do not show a significant difference. We reported a percentage of significant results and highlighted significant and non-significant results in all cases. Further experiments with the TsDD approach on new projects are suggested to generalize the conclusion.

7. Related works

According to ISO/IEC 25,010:2011 [24], testability is the degree of *effectiveness* and *efficiency* of test criteria establishment for a system, product, or component and performing tests to determine whether those criteria have been met. This definition inspires these simple questions:

- *How the effectiveness and efficiency of tests can be measured?*
- *What makes a test effective and efficient?*
- *How the effectiveness and efficiency of tests can be proved?*

The proposed TsDD approach in this paper aims to provide a practical means to answer these questions by integrating an efficient testability measurement and improvement cycle into the TDD process. In test-driven development (TDD), short coding cycles are interspersed with testing [4]. Therefore, TDD performance highly depends on code testability [75]. However, TDD has not made any specific precautions to enhance testability. In what follows, Section 7.1 describes the importance of testability in the TDD process, and Section 7.2 discusses the state-of-the-art methods for measuring source code testability.

7.1. Test-driven development and software testability

Test-driven development (TDD) aims to provide clean and fault-free code [3] by enforcing frequent testing cycles. Frequent testing could be time-consuming and costly if the unit under test is not testable. Despite the importance of testability, TDD does not provide any guidelines for testable software development. Therefore, it is observed that the use of TTD for large software projects is not suggested [5,10],. Empirical studies have reported a 16 percent decrease in programmers' productivity due to applying TDD, which shows that applying TDD sometimes took longer. [6]. In addition, it has been demonstrated that the TDD novices achieve a slightly higher code quality with test-last development than the TDD [14].

Romano et al. [76] have evaluated the different impacts of TDD by studying 20 participants to work on the implementation of a new feature for an existing software written in Java. They have concluded that developers write quick-and-dirty production code to pass the tests, do not update their tests, and often ignore refactoring. The tests are updated regularly in the proposed TsDD approach using automatic test data generation after making a testable unit.

Due to its test-first nature, TDD rarely benefits from automatic test data generation tools. Tosun et al. [11] have investigated the impact of test-driven development on the effectiveness of unit test cases compared to an incremental test last development (ITLD) in an industrial context. Their findings show that test cases written for the test-last development task cover more methods than those written for the TDD. Employing automated test data generation can solve this problem by enriching and completing the test cases provided manually in the test-first approach.

Borle et al. [77] have conducted a comparative analysis of GitHub repositories that adopt TDD to determine how TDD affects software development productivity and quality. They have used seven metrics, including the number of test files, average commit velocity, number of bug-referencing commits, number of issues recorded, usage of continuous integration, number of pull requests, and distribution of commits per author to compare TDD and Non-TDD projects. Their results reveal no observable benefits from using TDD in real-world projects.

The TDD process works well for projects where collocated teams develop small to medium-scale software systems. TDD has been challenging for large-scale projects regarding various quality and agility aspects [10,75],. Another issue with TDD is that it is unclear and complicated to adapt to legacy code. Remarkably, the automated regression test suites on the unit level, which are natural consequences of long-term TDD development, are often missing for legacy code [5,78], . The TsDD approach proposed in this paper facilitates applying TDD to legacy systems by employing automatic test data generation tools after refactoring each unit for testability and ensuring good design. A good design and testability are synergistic [79]. When looking for testability, developers end up with a good design. They end up with a testable class when looking for a good design.

It has been suggested that the benefits of TDD are not due to its distinctive test-first mechanism, but rather they are due to its fine-grained, steady steps that improve focus, flow, and quality [4]. TsDD facilitates making testable units and designs by frequent refactoring for testability before testing. The question is which refactorings improve unit testability. The point is that the impact of refactoring depends on the source code attributes measured in terms of software metrics. Therefore, there is a need for a software tool to measure source code testability.

7.2. Testability measurement and improvement approaches

Testability is proportional to the inverse of the effort required to test [80,81],. Indeed, a class may have a relatively low test effort because the associated tests are of poor quality and not because the class is easier to test. The industrial case studies about TDD suggest a reduced defect rate [82] and increased code quality [83] for the price of increased effort [82, 83],. For instance, a Microsoft case study projected that the TDD approach has almost double code quality but consumed 15 % more time writing tests. The IBM case study stated 40 % fewer defects, whereas there was a negligible effect on the team's productivity [83]. The effort could be significantly reduced by simply measuring the code testability before running it as far as the code is ready for the test. TsDD alleviates the test effort by evaluating the impacts of refactoring on the code testability, even before the code is executable.

Test effectiveness and efficiency are the two main ingredients of testability [24]. Gupta and Jalote [84] used mutation analysis to evaluate the effectiveness and efficiency of test coverage. They found that the predicate coverage criterion demonstrated the best effectiveness but at a higher cost. However, predicate coverage does not subsume statement coverage if the coverage is not 100 %. The more statements a test executes, the higher the probability of finding a faulty execution.

The efficiency of tests depends on the effort taken to establish the test criteria. The less effort it takes to test a class thoroughly, the more testable it is. However, it is assumed that a direct relation exists between testability and test effort [85]. Therefore, test coverage could be an appropriate indicator of test effectiveness, provided the coverage metrics were selected cautiously. Test coverage describes how the production codes are executed with a particular test case. It has been primarily used in software engineering to determine test effectiveness [84–86]. Inozemtseva and Holmes [87] suggest that while helpful in identifying under-tested parts of a program, code coverage is not a good indicator of test suite effectiveness. However, unlike what their experiments show, Gopinath et al. [88] claim that branch coverage is the most related to test-case effectiveness. We partially agree with Inozemtseva and Holmes [87] and Gopinath et al. [88], as we propose a combination of test adequacy criteria to measure test effectiveness, not considering them in isolation. To describe the reason, we suppose a test suite, S1, covers 90 % of branches, including only 10 % of the program statements. Furthermore, another test suite, S2, covers the remaining 10 % of branches, covering 90 % of the program statements. Apparently, the probability of detecting faulty executions when using S2 is greater than S1. Therefore, branch coverage on its own could not provide an appropriate criterion for evaluating test effectiveness. To resolve this branch coverage shortcoming, we use the branch and statement coverage average. Gay [36] found that multi-criteria test suites can be up to 31.15 % more effective at detecting faults than test suites generated to satisfy a single criterion.

Terragni et al. [21] have normalized test effort metrics with test quality metrics, including statement and branch coverage. They have shown that this action boosts test effort and source metrics correlation. Their approach can only estimate test effort concerning 28 metrics. Terragni's idea of normalization of test effort by test quality [21] is exactly the inverse of what we believe and is shown in our paper. We believe that the less effort required to test a program, the more testable the program under test is. Thus, the value of their proposed relation reduces testability as the test quality increases. Indeed, their work does not clarify the border between the test effort and testability. More importantly, Terragni et al. [22] use existing human-written tests as benchmarks, similar to other related works. We believe that such benchmarks are biased toward human testers' expertise, and the effort and budget behind them are not identical, resulting in unfair evaluations. Indeed, testability is an inherent feature of software under test, and human factors should not affect its measurement. Therefore, we prefer to use automatic test data generators. As a result, the term testability is used to measure the easiness of automatic testing. Albeit, our evaluation by experts confirms that it also helps manual testing due to using coarse-grain refactoring operations focused on smelly parts of the program. Zakeri and Parsa have proposed prediction models, respectively, for source code Coverageability [40,41], and testability measurement [89]. However, they do not propose any testability improvement approach based on the prediction models. This paper integrates a lightweight testability prediction model into the TDD process to enable testability improvement while coding. Our approach uses refactoring for testability to make a software version suitable for production. It differs from testability transformation techniques [90,91], which do not preserve the program behavior. Our work is a realization of the "refactoring as testability transformation" paradigm dreamed up by Harman [16], enabling the use of artificial intelligence in software engineering methodologies [92].

Harman [16] suggests using refactoring as a testability transformation [91], in which, instead of traditional testability transformation techniques, software refactoring techniques are employed to improve testability. To the best of our knowledge, his ideas have not been considered practically in previous research. Cinnéide et al. [93] have conducted a controlled experiment in which a small Java application with serious cohesion problems was refactored to improve testability. The improvement results obtained by several developers were different, and the authors have stated that the hypothesis that automated refactoring can improve testability still appears likely to be valid but requires further evaluation. Indeed, they have used the LSCC metric (Low-level design Similarity-based Class Cohesion [94]) to measure testability, which does not follow the standard definition of testability [24]. In this paper, we propose a testability measurement approach that considers the effectiveness and efficiency of the test. We guide the refactoring process to improve this measure at each development iteration.

8. Conclusion

The relatively high cost of tests, especially in large projects, can be significantly reduced by conscious testability improvement. The consciousness is supported by measuring testability before and after refactoring. TsDD encourages the development of testable software. TsDD fits well in TDD as the core of agile methodologies. TDD advises frequent cycles of testing, coding, and refactoring. However, frequent testing may oppose agility. Therefore, TDD does not fit well in the agile software development process despite being considered a key agile development technique. The main essence of the TsDD process is to alleviate the high cost of testing by postponing it as long as the code is ready for an effective and efficient test. This process brings about not only testable but also quality software. The TsDD process is applicable to both the legacy and the developing code that is incomplete or not ready to execute. Our experiments reveal that reusability and functionality attributes are also improved when refactoring for testability during the TsDD process. However, our experiments with three large-scale real-world Java projects reveal that certain quality attributes, such as effectiveness and extendability, may deteriorate when refactoring for testability and *vice versa*.

TsDD is supplied with a machine learning model to predict testability statically without running the code. We offer a mathematical model to compute the test efficiency and effectiveness regarding the test coverage and budget. The mathematical model uses the actual coverage obtained by a test suite to calculate testability. The computed testability for each class is used to label the class represented as a vector of source code metrics. The labeled vectors provide appropriate samples to predict testability. Regressor machine learning models fit the testability prediction task well because testability values computed by the mathematical model are continuous. The performance of the regressor models is optimized by applying a meta-ensemble model, voting the results provided by the regressors. We suggest using a feed-forward neural network (DNN), random forest regressor (RFR), and histogram gradient boosting regressor (HGBR) as base regressors.

In future work, we plan to experiment with search-based refactoring to find optimal sequences of refactorings that maximize the testability of the software under test. To this aim, the testability prediction model is used to determine the fitness of each sequence of refactorings. A multi-objective optimization algorithm can also be applied to improve other quality attributes such as readability, maintainability, and reusability simultaneously testability. We plan to compare the performance and effectiveness of different search algorithms and operators in generating high-quality refactoring sequences, such as genetic algorithms, simulated annealing, and hill-climbing. The testability prediction model can be constructed to work at the method level instead of the class level. Method-level testability prediction enables applying TsDD in non-object-oriented frameworks. Moreover, it allows measuring and improving the testability of individual methods or functions, which are the basic units of code in procedural or functional programming paradigms.

Ethical approval

This article does not contain any studies with human participants or animals performed by any of the authors.

CRediT authorship contribution statement

Saeed Parsa: Writing – original draft, Project administration, Methodology, Conceptualization. **Morteza Zakeri-Nasrabadi:** Writing – original draft, Visualization, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Burak Turhan:** Writing – review & editing, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The datasets generated and analyzed during the current study are available in Zenodo, https://doi.org/10.5281/zenodo.4650228.

Funding

This study has received no funding from any organization.

References

[1] D. Fucci, B. Turhan, A replicated experiment on the effectiveness of test-first development, in: 2013 ACM /IEEE International symposium on empirical software engineering and measurement, IEEE, 2013, pp. 103–112, https://doi.org/10.1109/ESEM.2013.15. Oct.

[2] K. Beck, *Extreme Programming explained: Embrace change.* in An Alan R. Apt Book Series, Addison-Wesley, 2000 [Online]Available, https://books.google.com/books?id=G8EL4H4vf7UC.

[3] K. Beck, *Test-driven development: By example.* in Addison-Wesley signature series, Addison-Wesley, 2003 [Online]Available, https://books.google.com/books?id=gFgnde_vwMAC.

[4] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, N. Juristo, A dissection of the test-driven development process: does it really matter to test-first or to test-last? IEEE Transact. Software Eng. 43 (7) (2017) 597–614, https://doi.org/10.1109/TSE.2016.2616877. Jul.

[5] A. Causevic, D. Sundmark, S. Punnekkat, Factors limiting industrial adoption of test-driven development: a systematic review, in: 2011 Fourth IEEE International conference on software testing, verification and validation, IEEE, 2011, pp. 337–346, https://doi.org/10.1109/ICST.2011.19. Mar.

[6] D. Janzen, H. Saiedian, Test-driven development concepts, taxonomy, and future direction, Computer. (Long. Beach. Calif) 38 (9) (2005) 43–50, https://doi.org/10.1109/MC.2005.314. Sep.

[7] L. Madeyski, The impact of test-first programming on branch coverage and mutation score indicator of unit tests: an experiment, Inf. Softw. Technol. 52 (2) (2010) 169–184, https://doi.org/10.1016/j.infsof.2009.08.007. Feb.

[8] J. Buchan, L. Li, S.G. MacDonell, Causal factors, benefits and challenges of test-driven development: practitioner perceptions, in: 2011 18th Asia-Pacific Software Engineering Conference, IEEE, 2011, pp. 405–413, https://doi.org/10.1109/APSEC.2011.44. Dec.

[9] M. Ghafari, T. Gross, D. Fucci, M. Felderer, Why research on test-driven development is inconclusive?, in: Proceedings of the 14th ACM /IEEE International symposium on empirical software engineering and measurement (ESEM) ACM, New York, NY, USA, 2020, pp. 1–10, https://doi.org/10.1145/3382494.3410687. Oct.

[10] R.S. Sangwan, P.A. Laplante, Test-driven development in large projects, IT. Prof. 8 (5) (2006) 25–29, https://doi.org/10.1109/MITP.2006.122. Sep.

[11] A. Tosun, M. Ahmed, B. Turhan, N. Juristo, On the effectiveness of unit tests in test-driven development, in: Proceedings of the 2018 International conference on software and system process, ACM, New York, NY, USA, 2018, pp. 113–122, https://doi.org/10.1145/3202710.3203153. May.

[12] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, 2nd Edi, Addison-Wesley, 2018 [Online]Available, https://refactoring.com/.

[13] G. Meszaros, *Xunit test patterns: refactoring test code*. in addison-wesley signature series (Fowler), Pearson Education, 2007 [Online]Available, https://books.google.com/books?id=-izOiCEIABQC.

[14] A. Santos, et al., A family of experiments on test-driven development, Empir. Softw. Eng. 26 (3) (2021) 42, https://doi.org/10.1007/s10664-020-09895-8. May.

[15] C. Jones, O. Bonsignour, The economics of software quality, Addison-Wesley, 2012 [Online]Available, https://books.google.com/books?id=_t5l5Cn0NBEC.

[16] M. Harman, Refactoring as testability transformation, in: 2011 IEEE Fourth International conference on software testing, verification and validation workshops, 2011, pp. 414–421.

[17] R.C. Martin, *Clean code: a handbook of agile software craftsmanship*. in Robert C. Martin series. Prentice Hall, 2009. [Online]. Available: https://books.google.com/books?id=dwSfGQAACAAJ.

[18] R.C. Martin. *Clean architecture: a craftsman's guide to software structure and design*, Pearson, 2017.

[19] V. Garousi, M. Felderer, F.N. Kılıçaslan, A survey on software testability, Inf. Softw. Technol. 108 (2019) 35–64, https://doi.org/10.1016/j.infsof.2018.12.003. Apr.

[20] J.M. Voas, K.W. Miller, Software testability: the new verification, IEEe Softw. 12 (3) (1995) 17–28, https://doi.org/10.1109/52.382180. May.

[21] V. Terragni, P. Salza, M. Pezzè, Measuring software testability modulo test quality, in: Proceedings of the 28th International Conference on Program Comprehension, ACM, New York, NY, USA, 2020, pp. 241–251, https://doi.org/10.1145/3387904.3389273. Jul.

[22] M. Ghafari, M. Eggiman, O. Nierstrasz, Testability first!, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–6, https://doi.org/10.1109/ESEM.2019.8870170. Sep.

[23] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM Transact. Software Eng. Methodo. 24 (2) (2014) 1–42, https://doi.org/10.1145/2685612. Dec.

[24] ISO and IEC, ISO/IEC 25010:2011 systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models, ISO, 2011, p. 34 [Online]Available, https://www.iso.org/standard/35733.html.

[25] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, "Weka 3: machine learning software in Java." Accessed: Jun. 23, 2021. [Online]. Available: https://github.com/Waikato/weka-3.8.

[26] C. Rueden and J. Koenig-Schindelin, "Scijava-common." Accessed: Jun. 23, 2021. [Online]. Available: https://github.com/scijava/scijava-common.

[27] J. Müller, "FreeMind - free mind mapping software." Accessed: Apr. 28, 2021. [Online]. Available: http://freemind.sourceforge.net/wiki/index.php/Main_Page.

[28] J. Edvardsson, "A survey on automatic test data generation," *Proceedings of the 2nd Conference on computer science and engineering*, no. x, pp. 21–28, 1999, doi: 10.1.1.20.963.

[29] A. Nanthaamornphong, J.C. Carver, Test-driven development in scientific software: a survey, Software Qual. J. 25 (2) (2017) 343–372, https://doi.org/10.1007/s11219-015-9292-4. Jun.

[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software, ACM SIGKDD Explorat. Newslett. 11 (1) (2009) 10, https://doi.org/10.1145/1656274.1656278. Nov.

[31] J.J. Li, "Prioritize code for testing to improve code coverage of complex software," in *16th IEEE International symposium on software reliability engineering (ISSRE'05)*, IEEE, pp. 75–84. doi: 10.1109/ISSRE.2005.33.

[32] T.L. Clune, R.B. Rood, Software testing and verification in climate model development, IEEe Softw. 28 (6) (2011) 49–55, https://doi.org/10.1109/MS.2011.117. Nov.

[33] L. Briand, S. Nejati, M. Sabetzadeh, D. Bianculli, Testing the untestable, in: Proceedings of the 38th International conference on software engineering companion, ACM, New York, NY, USA, 2016, pp. 789–792, https://doi.org/10.1145/2889160.2889212. May.

[34] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11, ACM Press, New York, New York, USA, 2011, p. 416, https://doi.org/10.1145/2025113.2025179.

[35] R. Santelices, J.A. Jones, Yanbing Yu, M.J. Harrold, Lightweight fault-localization using multiple coverage types, in: 2009 IEEE 31st International Conference on Software Engineering, IEEE, 2009, pp. 56–66, https://doi.org/10.1109/ICSE.2009.5070508.

[36] G. Gay, Generating effective test suites by combining coverage criteria, in: T. Menzies, J. Petke (Eds.), International symposium on search based software engineering, Springer International Publishing, 2017, pp. 65–82, https://doi.org/10.1007/978-3-319-66299-2_5.

[37] S. Leary and D. Crockford, "JSON-java." Accessed: Dec. 04, 2021. [Online]. Available: https://github.com/stleary/JSON-java.

[38] G. Grano, F. Palomba, H.C. Gall, Lightweight assessment of test-case effectiveness using source-code-quality indicators, IEEE Transact. Software Eng. 47 (4) (2021) 758–774, https://doi.org/10.1109/TSE.2019.2903057. Apr.

[39] G. Grano, T.V. Titov, S. Panichella, H.C. Gall, Branch coverage prediction in automated testing, Evolut. Process 31 (9) (2019), https://doi.org/10.1002/smr.2158. Sep.

[40] M. Zakeri-Nasrabadi, S. Parsa, Learning to predict test effectiveness, Int. J. Intell. Syst. (2021), https://doi.org/10.1002/int.22722. Oct.

[41] M.Zakeri- Nasrabadi, S. Parsa, Learning to predict software testability, in: 2021 26th International Computer Conference, Computer Society of Iran (CSICC), IEEE, Tehran, 2021, pp. 1–5, https://doi.org/10.1109/CSICC52343.2021.9420548. Mar.

[42] F. Pedregosa, et al., Scikit-learn: machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830 [Online]Available, http://jmlr.org/papers/v12/pedregosa11a.html.

[43] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT Press, 2016 [Online] Available, http://www.deeplearningbook.org/.

[44] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32, https://doi.org/10.1023/A:1010933404324.

[45] A. Guryanov, "Histogram-based algorithm for building gradient boosting ensembles of piecewise linear decision trees," 2019, pp. 39–50. doi: 10.1007/978-3-030-37334-4_4.

[46] M. Zakeri-Nasrabadi, S. Parsa, M.Wiem Mkaouer, and B. Turhan, "Testability-driven development (TsDD): an improvement to the TDD efficiency." May 2023. doi: https://doi.org/10.5281/zenodo.7916398.

[47] M. Zakeri-Nasrabadi and S. Parsa, "Testability prediction dataset." [Online]. Available: https://zenodo.org/record/4650228.

[48] R. FERENC, P. Siket, M. Schneider, OpenStaticAnalyzer, University of Szeged. Accessed, 2021. Jun. 23[Online]Available, https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.

[49] J. Bergstra, Y. Bengio, Random search for hyper-parameter optimization, J. Mach. Learn. Res. 13 (2012) 281–305, nullFeb.

[50] "DAWN water simulator." Accessed: May 22, 2022. [Online]. Available: https://sourceforge.net/projects/water-simulator/.

[51] A. Panichella, J. Campos, G. Fraser, EvoSuite at the SBST 2020 tool competition, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ACM, New York, NY, USA, 2020, pp. 549–552, https://doi.org/10.1145/3387940.3392266. Jun.

[52] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Ten years of JDeodorant: lessons learned from the hunt for smells, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 4–14, https://doi.org/10.1109/SANER.2018.8330192. Mar.

[53] N. Tsantalis, A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, J. Syst. Software 84 (10) (2011) 1757–1782, https://doi.org/10.1016/j.jss.2011.05.016. Oct.

[54] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Transact. Software Eng. 35 (3) (2009) 347–367, https://doi.org/10.1109/TSE.2009.1. May.

[55] D. Fucci, B. Turhan, On the role of tests in test-driven development: a differentiated and partial replication, Empir. Softw. Eng. 19 (2) (2014) 277–302, https://doi.org/10.1007/s10664-013-9259-7. Apr.

[56] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, cambridge, 2016, https://doi.org/10.1017/9781316771273.

[57] Z. Khanam, M.N. Ahsan, Evaluating the effectiveness of test driven development: advantages and pitfalls, Internat. J. Appl. Eng. Res. 12 (18) (2017) 7705–7716.

[58] A.C. Bibiano, et al., A quantitative study on characteristics and effect of batch refactoring on code smells, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, 2019, pp. 1–11, https://doi.org/10.1109/ESEM.2019.8870183. Sep.

[59] M.W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, K. Deb, On the use of many quality attributes for software refactoring: a many-objective search-based software engineering approach, Empir. Softw. Eng. 21 (6) (2016) 2503–2545, https://doi.org/10.1007/s10664-015-9414-4. Dec.

[60] M. Mohan, D. Greer, Using a many-objective approach to investigate automated refactoring, Inf. Softw. Technol. 112 (2019) 83–101, https://doi.org/10.1016/j.infsof.2019.04.009. Aug.

[61] JetBrains, "IntelliJ IDEA." Accessed: Apr. 09, 2022. [Online]. Available: https://www.jetbrains.com/idea/.

[62] J. Krithikadatta, Normal distribution, J. Conservat. Dent. 17 (1) (2014) 96, https://doi.org/10.4103/0972-0707.124171.

[63] R.A. Khan, K. Mustafa, Metric based testability model for object oriented design (MTMOOD), ACM SIGSOFT Software Eng. Notes 34 (2) (2009) 1, https://doi.org/10.1145/1507195.1507204. Feb.

[64] J. Kerievsky, Refactoring to Patterns, Addison-Wesley Professional, 2005.

[65] M. Shahidi, M. Ashtiani, M. Zakeri-Nasrabadi, An automated extract method refactoring approach to correct the long method code smell, J. Syst. Software 187 (2022) 111221, https://doi.org/10.1016/J.JSS.2022.111221. May.

[66] D.E. Hinkle, W. Wiersma, S.G. Jurs, Applied statistics for the behavioral sciences, 663, Houghton Mifflin College Division, 2003.

[67] J. Bansiya, C.G. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Transact. Software Eng. 28 (1) (2002) 4–17, https://doi.org/10.1109/32.979986.

[68] A. Almogahed, et al., A refactoring classification framework for efficient software maintenance, IEEe Access. 11 (2023) 78904–78917, https://doi.org/10.1109/ACCESS.2023.3298678.

[69] Y. Zhang, K. Guan, L. Fang, MIRROR: multi-objective refactoring recommendation via correlation analysis, Autom. Softw. Eng. 31 (1) (2024) 2, https://doi.org/10.1007/s10515-023-00400-1. Jun.

[70] R. Shatnawi, W. Li, An empirical assessment of refactoring impact on software quality using a hierarchical quality model, Internat. J. Software Eng. Applicat. (2011) 127–149.

[71] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, G. Antoniol, Finding the best compromise between design quality and testing effort during refactoring, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and

Reengineering (SANER), IEEE, 2016, pp. 24–35, https://doi.org/10.1109/SANER.2016.23. Mar.
[72] A. Salahirad, H. Almulla, G. Gay, Choosing the fitness function for the job: automated generation of test suites that detect real faults, Software Test., Verificat. Reliabil. 29 (4–5) (2019), https://doi.org/10.1002/stvr.1701. Jun.
[73] M. Badri, F. Toure, Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes, J. Software Eng. Applicat. 05 (07) (2012) 513–526, https://doi.org/10.4236/jsea.2012.57060.
[74] M.D. Papamichail, T. Diamantopoulos, A.L. Symeonidis, Measuring the reusability of software components using static analysis metrics and reuse rate information, J. Syst. Software 158 (2019) 110423, https://doi.org/10.1016/j.jss.2019.110423. Dec.
[75] B. George, L. Williams, A structured experiment of test-driven development, Inf. Softw. Technol. 46 (5) (2004) 337–342, https://doi.org/10.1016/j.infsof.2003.09.011. Apr.
[76] S. Romano, D. Fucci, G. Scanniello, B. Turhan, N. Juristo, Findings from a multi-method study on test-driven development, Inf. Softw. Technol. 89 (2017) 64–77, https://doi.org/10.1016/j.infsof.2017.03.010. Sep.
[77] N.C. Borle, M. Feghhi, E. Stroulia, R. Greiner, A. Hindle, Analyzing the effects of test driven development in GitHub, in: Proceedings of the 40th International conference on software engineering, ACM, New York, NY, USA, 2018, p. 1062, https://doi.org/10.1145/3180155.3182535. May–1062.
[78] E. Shihab, Z.M. Jiang, B. Adams, A.E. Hassan, R. Bowerman, Prioritizing unit test creation for test-driven maintenance of legacy systems, in: 2010 10th International conference on quality software, IEEE, 2010, pp. 132–141, https://doi.org/10.1109/QSIC.2010.74. Jul.
[79] M. Feathers, "The deep synergy between testability and good design." Accessed: Jun. 09, 2021. [Online]. Available: https://michaelfeathers.typepad.com/michael_feathers_blog/2007/09/the-deep-synerg.html.
[80] F. Toure, M. Badri, L. Lamontagne, Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software, Innov. Syst. Softw. Eng. 14 (1) (2018) 15–46, https://doi.org/10.1007/s11334-017-0306-1. Mar.
[81] M. Bruntink, A. van Deursen, An empirical study into class testability, J. Syst. Softw. 79 (9) (2006) 1219–1232, https://doi.org/10.1016/j.jss.2006.02.036. Sep.
[82] T. Bhat, N. Nagappan, Evaluating the efficacy of test-driven development: industrial case studies, in: Proceedings of the 2006 ACM/IEEE international symposium on International symposium on empirical software engineering - ISESE '06, ACM Press, New York, New York, USA, 2006, p. 356, https://doi.org/10.1145/1159733.1159787.
[83] E.M. Maximilien, L. Williams, Assessing test-driven development at IBM, in: 25th International Conference on Software Engineering, 2003. Proceedings, IEEE, 2003, pp. 564–569, https://doi.org/10.1109/ICSE.2003.1201238.
[84] A. Gupta, P. Jalote, An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing, Internat. J. Software Tools Techn. Transfer 10 (2) (2008) 145–160, https://doi.org/10.1007/s10009-007-0059-5. Mar.
[85] A.S. Namin, J.H. Andrews, The influence of size and coverage on test suite effectiveness, in: Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09, ACM Press, New York, New York, USA, 2009, p. 57, https://doi.org/10.1145/1572272.1572280.
[86] R. Ramler, T. Wetzlmaier, C. Klammer, An empirical study on the application of mutation testing for a safety-critical industrial software system, in: Proceedings of the Symposium on Applied Computing, ACM, New York, NY, USA, 2017, pp. 1401–1408, https://doi.org/10.1145/3019612.3019830. Apr.
[87] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014, ACM Press, New York, New York, USA, 2014, pp. 435–445, https://doi.org/10.1145/2568225.2568271.
[88] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, 2014, pp. 72–82, https://doi.org/10.1145/2568225.2568278. May.
[89] M. Zakeri-Nasrabadi, S. Parsa, An ensemble meta-estimator to predict source code testability, Appl. Soft. Comput. 129 (2022) 109562, https://doi.org/10.1016/j.asoc.2022.109562. Nov.
[90] D.W. Binkley, M. Harman, K. Lakhotia, FlagRemover: a testability transformation for transforming loop-assigned flags, ACM Transact. Software Eng. Methodol. (2011) 1–33, https://doi.org/10.1145/2000791.2000796. Aug.
[91] M. Harman, et al., Testability transformation, IEEE Transact. Software Eng. 30 (1) (2004) 3–16, https://doi.org/10.1109/TSE.2004.1265732. Jan.
[92] M. Harman, The role of artificial intelligence in software engineering, in: 2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE), IEEE, 2012, pp. 1–6, https://doi.org/10.1109/RAISE.2012.6227961. Jun.
[93] M.Ó. Cinnéide, D. Boyle, I.H. Moghadam, Automated refactoring for testability, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, 2011, pp. 437–443, https://doi.org/10.1109/ICSTW.2011.23. Mar.
[94] J. Al Dallal, L.C. Briand, A precise method-method interaction-based cohesion metric for object-oriented classes, ACM Transact. Software Eng. Methodol. 21 (2) (2012) 1–34, https://doi.org/10.1145/2089116.2089118. Mar.