Information and Software Technology 174 (2024) 107511

Contents lists available at ScienceDirect

Information and Software Technology

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

Measuring and improving software testability at the design level

Morteza Zakeri-Nasrabadi [a], Saeed Parsa [b,*], Sadegh Jafari [b]

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

ARTICLE INFO

Keywords:
Software testability
Design for testability
Object-oriented design
Class diagram
Automated refactoring
Design patterns

ABSTRACT

Context: The quality of software systems is significantly influenced by design testability, an aspect often overlooked during the initial phases of software development. The implementation may deviate from its design, resulting in decreased testability at the integration and unit levels.
Objective: The objective of this study is to automatically identify low-testable parts in object-orientated design and enhance them by refactoring to design patterns. The impact of various design metrics mainly coupling (*e.g.*, fan-in and fan-out) and inheritance (*e.g.*, depth of inheritance tree and number of subclasses) metrics on design testability is measured to select the most appropriate refactoring candidates.
Method: The methodology involves creating a machine learning model for design testability prediction using a large dataset of Java classes, followed by developing an automated refactoring tool. The design classes are vectorized by ten design metrics and labeled with testability scores calculated from a mathematical model. The model computes testability based on code coverage and test suite size of classes that have already been tested via automatic tools. A voting regressor model is trained to predict the design testability of any class diagram based on these design metrics. The proposed refactoring tool for dependency injection and factory method is applied to various open-source Java projects, and its impact on design testability is assessed.
Results: The proposed design testability model demonstrates its effectiveness by satisfactorily predicting design testability, as indicated by a mean squared error of 0.04 and an R^2 score of 0.53. The automated refactoring tool has been successfully evaluated on six open-source Java projects, revealing an enhancement in design testability by up to 19.11 %.
Conclusion: The proposed automated approach offers software developers the means to continuously evaluate and enhance design testability throughout the entire software development life cycle, mitigating the risk of testability issues stemming from design-to-implementation discrepancies.

1. Introduction

Software design is a crucial and challenging activity affecting software products' quality and reliability. A good software design should not only meet the functional and non-functional requirements of the system but also support its testing in a given test context. Design testability is the degree to which a software design supports its testing [1]. A testable design facilitates early testing of program modules and their interactions, reduces testing effort and cost, increases test coverage and effectiveness, and improves the maintainability and evolvability of the system. On the other hand, a design that does not consider testability as an essential quality factor may result in increased complexity, coupling, ambiguity, and redundancy, which make testing more complex and less efficient [2].

Software testability can be measured and improved according to different testing levels, such as units, integration, and systems testing, each one is based on specific software artifacts [1]. Unit testing performance is mostly affected by the source code testability, while the performance of integration and system tests heavily depends on design testability. However, measuring and enhancing software design testability is not trivial due to various design artifacts, systems complexities, and the unavailability of design artifacts in most legacy projects. Most existing methods for design testability measurement rely on executing the design with test cases that are time and resource-consuming and require full implementation of the system [3]. Moreover, most of the existing methods for design testability improvement are based on

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

https://doi.org/10.1016/j.infsof.2024.107511
Received 15 October 2023; Received in revised form 6 May 2024; Accepted 7 June 2024
Available online 21 June 2024

manual refactoring that may introduce new errors or degrade other quality attributes [4]. Recent research on software testability improvement has focused on applying different methods, such as refactoring [4–6], design patterns [7,8], aspect-oriented programming [9], and model-driven engineering [10]. These methods aim to enhance the quality attributes of software design, such as cohesion, coupling, encapsulation, abstraction, modularity, and reusability, which often indirectly affect design testability.

Few works have been dedicated to studying software testability at the design level directly, despite remarkable attempts to measure and improve the testability of source code. This paper presents an approach to measuring design testability based on the standard definition of software testability in the ISO/IEC 25010 standard [11]. Then, methods based on refactoring to design patterns are introduced to automatically improve the testability of a given design. Testability measurement is a prerequisite for testability improvement since it is difficult to guide the improvement process in the right direction without an appropriate measurement approach.

ISO/IEC 25010 standard emphasizes two factors of test effectiveness and test efficiency in the definition of software testability. In our recent work [5], we measured the source code testability in terms of test effectiveness and test efficiency of the unit tests. This paper extends our work to measure design testability in terms of the effectiveness and efficiency of the integration tests for a given design described by the UML class diagram. Integration testing aims at testing interaction between units, best described by the class diagram as a well-known static view of the object-oriented design in UML. However, it cannot describe the exact runtime interactions between components. Moreover, class diagrams are often ambiguous and incomplete and may lead to false interpretations at implementation time. On the other hand, other UML diagrams, such as collaboration and sequence diagrams, only display snapshots of some possible behaviors of the system, not an exhaustive design view [2]. In addition, collaboration and sequence diagrams are often not available for a given system, nor can they be extracted accurately from the source code. The class diagram with the runtime information obtained from testing can be best used to measure design testability by mapping the design metrics to test information. Class diagrams can be extracted from the source code. Therefore, it is possible to collect a large set of class diagrams to create the mapping between design metrics and test information for design testability prediction.

This paper proposes an automated approach to measure and improve software design testability based on class diagram testability prediction and refactoring to design patterns. Identifying testability issues during design prevents downstream problems. Throughout the software development life cycle, we can continuously evaluate and enhance design testability. To this aim, we adapt the proposed source code testability prediction approach by Zakeri-Nasrabadi and Parsa [5] to be used for design testability prediction by limiting the required metrics to the design-level metrics. The authors in [5] have proposed using machine learning to predict source code testability based on an extensive set of source code metrics. It should be noted that not all source code metrics are available during the design phase. While source code metrics focus on low-level constructs (*e.g.*, lines of code and cyclomatic complexity), design metrics consider higher-level abstractions (*e.g.*, class coupling and inheritance) appearing early in the design phase. Our approach maps these design-level metrics available at design time to test criteria during training machine learning models which consequently learn from these mappings to predict testability. These metrics serve as our compass in navigating the transition from source code to design, design to testable design, and design to source code, ultimately leading to more testable software.

The proposed approach consists of the following four steps. (1) An extended class diagram is extracted as a directed labeled graph from the production code of a given software as the primary artifact used for measuring and improving design testability, (2) ten design metrics are computed using the created graph to vectorize the different classes of the class diagram, (3) the created class vectors are used as samples to learn a model measuring the testability probability of each class in a class diagram, (4) two well-known patterns, the factory method [12] and dependency injection (DI) [13], mainly concerned with facilitating tests, are identified and applied to improve the design testability automatically. The program source code is analyzed, required design artifacts are extracted, refactoring opportunities are identified, and finally, the identified refactoring is applied directly to the source code. The main contributions of this paper are summarized as follows:

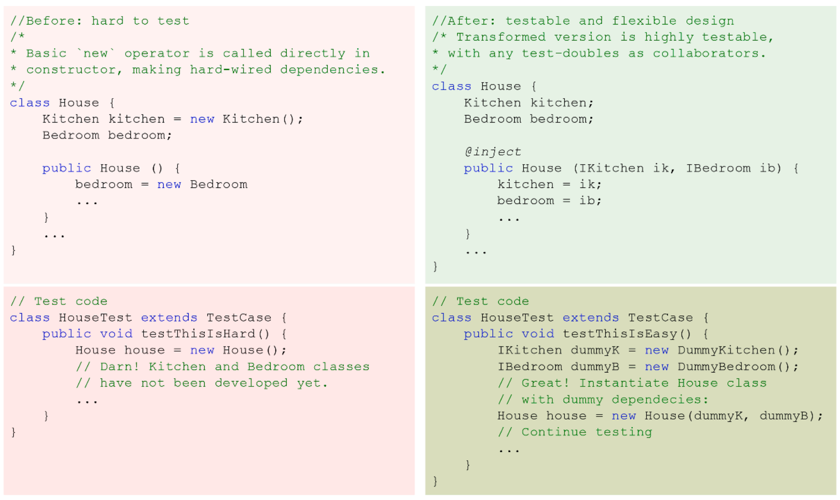
i. A new approach to measure software design testability based on machine learning is presented that does not require executing the design with test cases. Traditional testability assessment often relies on executing code, which can be time-consuming and resource-intensive. By leveraging machine learning, our new approach provides a faster and more efficient way to evaluate design testability. It also opens up possibilities for early testability assessment during the design phase, reducing downstream defects. As a result, researchers and practitioners can now focus on design improvements without waiting for code implementation.
ii. The most important design testability metrics are determined using the prediction model feature importance analysis. Understanding which metrics significantly impact testability allows targeted interventions. This knowledge contributes to a more informed and effective design process. Researchers can prioritize efforts to improve specific aspects of design (*e.g.*, modularity, coupling, cohesion, or inheritance). Moreover, practitioners gain insights into where to focus their attention during design reviews and refactoring.
iii. Methods based on refactoring to design patterns are proposed to automatically improve software design testability by changing the most influential design metrics. Refactoring is a powerful technique for improving code quality, but applying it at the design level is less explored. Design testability is enhanced by automatically transforming the existing design into established design patterns that focus on changing influential design metrics.
iv. The identification and application of factory methods and dependency injection patterns are automated to separate concerns of objects' creation and usage in the software design. Separating concerns (creation *vs.* usage) improves modularity and testability. A tool that supports our approach is implemented and evaluated on several open-source projects. The implemented tool, publicly available on GitHub [14], facilitates real-world adoption and evaluation of design testability on open-source projects.

These contributions collectively advance the state-of-the-art in software design testability by providing novel assessment methods, actionable insights, and practical tools for enhancing testability during the design phase.

The rest of this paper is organized as follows: In Section 2, we provide concrete examples of the code that is pretty testable and the code that is not as a motivating example. Section 3 outlines background information on software testability and discusses related work. Section 4 describes our approach to measuring software design testability based on machine learning and methods to improve software design testability based on refactoring to design patterns. Section 5 describes our experiments with the proposed approach and discusses the results. Section 6 discusses some threats to validity of the proposed method and its evaluation. Finally. Section 7 concludes the paper and outlines future work.

2. Motivating example

One of the common flaws in design destroying testability is performing real work in the constructor body. When the constructor has to instantiate and initialize its collaborators, the result tends to be an



```java
//Before: hard to test
/*
* Basic `new` operator is called directly in
* constructor, making hard-wired dependencies.
*/
class House {
    Kitchen kitchen = new Kitchen();
    Bedroom bedroom;

    public House () {
        bedroom = new Bedroom
        ...
    }
    ...
}
```

```java
// Test code
class HouseTest extends TestCase {
    public void testThisIsHard() {
        House house = new House();
        // Darn! Kitchen and Bedroom classes
        // have not been developed yet.
        ...
    }
}
```

(a) Untestable design

```java
//After: testable and flexible design
/* Transformed version is highly testable,
* with any test-doubles as collaborators.
*/
class House {
    Kitchen kitchen;
    Bedroom bedroom;

    @inject
    public House (IKitchen ik, IBedroom ib) {
        kitchen = ik;
        bedroom = ib;
        ...
    }
    ...
}
```

```java
// Test code
class HouseTest extends TestCase {
    public void testThisIsEasy() {
        IKitchen dummyK = new DummyKitchen();
        IBedroom dummyB = new DummyBedroom();
        // Great! Instantiate House class
        // with dummy dependecies:
        House house = new House(dummyK, dummyB);
        // Continue testing
        ...
    }
}
```

(b) Testable design

Fig. 1. Example of untestable design and its testable counterpart.

inflexible and prematurely coupled design. Such constructors shut off the ability to inject test objects when testing resulting in poor testability. Fig. 1.a shows the "House" class implemented in Java with hard-wired dependency to its collaborator classes, *i.e.*, "Kitchen" and "Bedroom". These collaborator classes must be implemented and compiled before any test can be performed on the House class. Using the dependency injection (DI) pattern makes it possible to test the House class with dummy objects from its collaborators. Fig. 1.b shows how refactoring to dependency injection patterns can solve the problem of hard-wired dependence, reduce concentrate coupling, and improve design testability. The constructor of the "House" class takes interfaces of "Kitchen" and "Bedroom" classes as its arguments which can be passed by the client with different types of objects at runtime. Fig. 4 in Section 4.4 shows the corresponding class diagrams for code snippets in Fig. 1 before and after refactoring to the dependency injection pattern. In this paper, we propose algorithms to automate the transformation from code snippets in the style of Fig. 1.a to code snippets in the style of Fig. 1.b to improve software design testability.

3. Related work

Testability is a critical quality attribute of software systems, as it can significantly impact the cost and effectiveness of testing [15]. A recent survey by Garousi et al. [1] indicates many challenges in measuring and improving testability, including the lack of a standardized definition and the difficulty of measuring testability in large, complex systems. Similarly, as reported by Bluemke and Malanowska [16], there are almost no reliable tools for testing effort estimation, and the industry does not use any testing effort estimation-specific tool support. Sharma et al. [17] have concluded that developers' opinions and perspectives about testability and testability smells are not supported by empirical software engineering data. While these surveys provide a comprehensive overview of the research on software testability, they do not suggest specific guidance on how to improve testability in practice, specifically design testability.

A highly testable design can reduce the time and effort required for testing, improve the accuracy and completeness of testing, and increase the reliability and maintainability of the software system [18–20], specifically object-oriented software. The object-oriented paradigm has facilitated software design and development through the key concepts of abstraction, inheritance, and information hiding. However, object-oriented software has been demonstrated to have lower testability than procedural implementations [21].

Design for testability (DFT) concerns early life-cycle activities that can promote the testability of the software systems at unit and integration testing levels [21–23]. An essential prerequisite for successful DFT in object-oriented software is a precise testability measurement mechanism for object-oriented design artifacts, such as the class diagram. Nevertheless, most of the existing approaches focus on measuring source code testability using software metrics [5,24–29]. These approaches require the full implementation of the system, *i.e.*, the source code, and hence do not support testability assessment at the early stages of development.

Few studies have attempted to measure software testability at design levels using dependency analysis of UML diagrams [2,23,30]. However, these studies have some limitations, such as manual extraction of information, lack of tool support, low efficiency, and incompleteness. For instance, Baldini and Prinetto [23] have proposed a DFT approach to highly reconfigurable component-based systems [31] by manually extracting the kinds of dependency trees from UML diagrams. The dependency trees represent the relationships between three levels, including requirements, logic components, and resources, enabling the identification of the tests necessary for each version. Their approach is not specific to a software design artifact such as the class diagram nor supported by any tool. Moreover, the complexity of the design is not considered when creating the dependency trees, and no additional information is added to the design to measure and improve its testability.

The software design can be augmented with additional information facilitating testing components. Rocha and Martins [30] have presented a strategy to improve component testability by including self-checking capabilities and tracking mechanisms in the byte codes of Java components. The proposed testable module contains the component under test, a Tracker component responsible for the tracking mechanisms, and a Tester component performing the assertions inclusion. The authors have not provided any experimental results for their approach. In addition, the required information obtained by instrumenting the program byte code might not be available at design time.

Software artifacts that are mostly created and available during the design phase of object-oriented software are the class diagrams. Baudry and Traon [2] have proposed an approach to measure the testability of the UML class diagram based on the number of interactions between classes in a given design. The authors have quantified the count of class interactions as an approximation for object interactions. This estimation relies on the hypothesis that during integration testing, for each class interaction in a given class diagram, either a test case is generated to demonstrate a corresponding object interaction or a report is produced indicating that this interaction is not feasible. Therefore, the number of interactions between classes estimated the difficulty of testing a system.

The class interactions between two classes are measured with a proposed 'complexity of interaction' metric.

To compute the complexity of interaction [2], first, the class diagram is transformed into an annotated graph, called the class dependency graph (CDG) where the nodes are classes or interfaces and the edges show interactions. After that, all possible paths between two nodes of the CDG are computed. If there are n classes in the descendent's path, which are not pure interfaces, the complexity of the sub-component is $n(n-1)$. The advantage of this approach is that the program source code is not required to compute the complexity of interaction. However, the time complexity of the proposed algorithm to compute the interaction complexity between all pairs of classes in a design with v classes equals $O(v!)$. It means considerable time is required to compute the proposed metric for large designs. On the other hand, the complexity of interaction only corresponds to the design test effort. Indeed, according to the standard definition [11], test effectiveness should be considered when measuring testability.

Test effectiveness is typically measured through code coverage metrics, such as statement and branch coverages [32]. Some researchers have used machine learning to predict code coverage of the class under test at unit testing as a proxy of test effectiveness based on source code metrics [33,34]. These approaches are unsuitable for measuring design testability since they do not consider test effort metrics, such as the complexity of interaction [2]. In our recent research, we have used the time required to generate test data along with the size of the minimal test suite as crucial metrics to include test effort in testability measurement [5]. However, the proposed machine learning model required source code metrics, which may not be available at the design level.

This paper uses a set of well-known design metrics based on the inheritance and dependency relationships computable from the class diagram to predict the testability of each class at the design phase. The low efficiency and incompleteness of the Baudry and Traon approach [2] to measuring design testability is addressed by introducing a design testability prediction model built upon actual test criteria. The prediction model enables efficient and frequent measuring of design testability after applying any transformation on the class diagram to indicate potential improvements.

Our approach also introduces automated refactoring to creational design patterns to improve class diagram testability. Baudry et al. [7] have computed the complexity of interaction for different design patterns and suggested using the "create" stereotype to guarantee that the implementation will not introduce actual objects' interactions and side effects. Creational patterns that we automatically introduce to an existing design use the "create" stereotype to reduce the complexity of interaction. The design patterns are introduced to the legacy design by automated refactoring algorithms working at the design level, *i.e.*, class diagram.

Manual refactoring to design patterns was initially discussed by Kerievsky [8]. To the best of our knowledge, limited attempts have been made regarding automated refactoring to design patterns [35–38] in software systems. Wei et al. [35] have proposed automated algorithms to identify and apply factory and strategy design patterns at the source code level. They used Eclipse JDT to extract the program abstract syntax tree (AST) and then performed the required analysis and transformation for each refactoring operation. However, their approach is not supported by a publicly available tool. In addition, the impact of refactoring on different quality attributes, specifically testability, has not been studied.

Zafeiris et al. [36] have introduced a code transformation for refactoring Call Super (calling a parent class method in the child class) instances to the template method design pattern [12]. They conduct refactoring to the template method by identifying before and after fragments in the Super Call method, extracting them as separate methods, and forming the template method in the parent class to remove the Super Call statements in the child class. The authors have shown the improvement of the Specialization Index (SIX) metric, which expresses the proportion of concrete overriding among all the methods of a given class multiplied by the depth of the class in the inheritance hierarchy. However, they do not consider the impact of the refactoring on external quality attributes such as testability. In addition, the information required to identify and apply this refactoring is mainly available at the source code level, *i.e.*, after implementing a given design, and is not suitable for refactoring the class diagram.

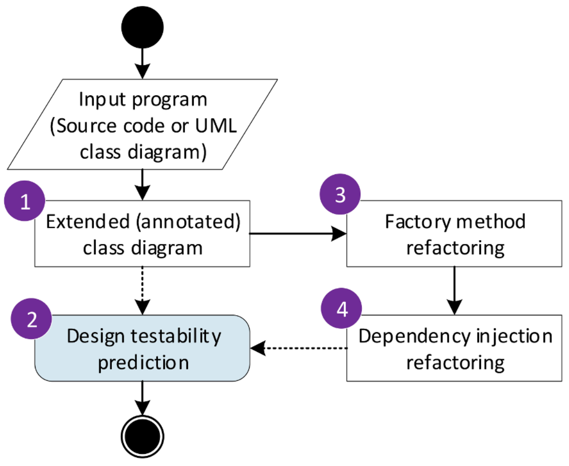

Input program
(Source code or UML class diagram)
1
Extended (annotated) class diagram
3
Factory method refactoring
2
Design testability prediction
4
Dependency injection refactoring

Fig. 2. An overview of the proposed testability measurement and improvement approach.

Ouni et al. [37] have proposed a search-based refactoring recommendation approach to introducing design patterns. They have introduced the refactoring to four design patterns, including the visitor, factory method, singleton, and strategy patterns, to improve design quality and remove smells. The refactoring opportunities are randomly selected and optimized during a many-object search with the NSGA-III algorithm [39]. Randomly initializing the parameters of the refactoring operations makes the process inefficient [40]. For example, the factory method must be parametrized by one creator class and a set of product classes, which forms a huge search space. Their approach also does not consider testability as an objective. Therefore, it is difficult for this approach to improve testability with refactoring. Moreover, their proposed tool, MORE [37], is not publicly available for additional usage and evaluation.

In this paper, we automate the identification and application of factory method and dependency injection patterns and study their impact on the design testability measured with the proposed testability prediction model. Motivated by related work, we propose algorithms to automate these refactoring which works at the design and source code levels. Our approach is different from the existing ones with respect to some flexibility factors that aim to improve the generalization of the approach. For the factory method, we introduce a heuristic to find nearly similar classes as candidates' product classes, making the algorithm flexible and applicable to various designs. For the dependency injection pattern, we first make an interface for classes whose objects are injected into the target class. Then, change the constructor(s) of the target class to accept the parameters of the newly created interfaces. This way the code will be ready for testing in isolation.

4. Methodology

This section presents our proposed approach for measuring and improving software testability at the design level. Our approach evaluates and improves testability at the design level which is mainly concerned with classes, interfaces, and their relationships. The main reason behind working at the design level is that the low code may not be developed or available in hand. On the other hand, the design artifacts such as class diagrams may not be available for the legacy software.

Finally, synching design and implementation is an important part of the improvement process in the maintenance phase. For these reasons, our approach works directly at the design level while it can extract the design from legacy code and then program change to the code. We begin with an overview of the approach and discuss each step in detail.

4.1. Overview

The proposed approach consists of four main steps, illustrated by a flowchart in Fig. 2. In Step 1, an annotated version of the class diagram is extracted from the input program. The input can be either a source code or a class diagram. The tool automatically extracts the class diagram if the user provides the source code. Design testability is measured in Step 2 using testability prediction. In Step 3 and Step 4, factory method and dependency injection opportunities are identified and applied at the design level and propagated to the source code level. Finally, the refactored system's testability is predicted to determine the improvement in design testability value. It is worth noting that parts of codes responsible for creating and manipulating objects are required for our approach to operating. For instance, constructors and fields of classes are attached to each node in the class diagram. However, full implementation, such as methods' bodies, is not necessary to be available.

We begin by searching for factory method refactoring opportunities because it yields greater benefits regarding design testability and at the same time understandability and reusability. Indeed, if we first refactor to the dependency injection pattern, we may miss those factory patterns that their products affected by the dependency injection. As described in Section 5.5, DI refactoring removes some objects created by a class (in its constructors), which may lead to the loss of a factory method refactoring opportunity. On the other hand, as shown in Table 7, the improvement of applying the dependency injection method without considering the factory is about 4.81 %, which is lower than that obtained by applying both the factory method and dependency injection refactoring operations. Hence, we prioritize refactoring to the factory method over dependency injection. It is worth noting that applying the two refactoring operations in parallel is only applicable to parts without any overlap and shared dependencies. In such a situation the parallel execution yields the same results with sequential execution.

4.2. Extended (annotated) class diagram

According to Baudry and Traon [2] the main views on which testability must be analyzed are class diagrams and statecharts, and the other views only display snapshots of some possible behaviors. Moreover, the full implementation of the system may not be available in all situations. Hence, our work focuses on the testability weaknesses of UML class diagrams, rarely covered by the literature. The extended class diagram proposed in this section contains all relationships between classes including those created by method interactions. It captures all information required for refactoring to the factory method and dependency injection. We use static analysis and abstract syntax tree processing at the source code level mentioned whenever possible, *i.e.*, when the course code is available to generate the extended class diagram. However, our method also works in the absence of program source code, *e.g.*, at the development phase.

Technically, the class diagram can be modeled as a directed labeled graph in which vertices represent classes and interfaces of an object-oriented system and edges class associations. In the proposed extended class diagram the nodes contain basic information about classes' fields and their types, methods, parameters, and constructors required to identify and perform refactoring to factory method and dependency injection. These data are saved as node attributes in the corresponding directed graph. Furthermore, edges in the extended form are labeled with different stereotypes that clarify the precise type of relationships between the two classes. The complete list of the stereotypes is as follows:

1. *Realization*: An edge that connects the implementing class to its implemented interface.
2. *Extend*: An edge that connects the child's class to its parent.
3. *Create*: An edge that connects class A to class B whenever class A creates an instance of class B.
4. *Use-consult*: An edge that connects class A to class B whenever class A uses an object of class B and does not change the status of the object.
5. *Use-def*: An edge that connects class A to class B whenever class A uses an object of class B and also changes the internal state of the object.

The reason for separating the creation and use dependencies from the use-def dependencies is that they do not need to be tested during integration testing. In other words, these relationships do not incur any test costs. Moreover, the object creation subgraph could be entirely replaced by a fake subgraph that uses test doubles to realize the test in isolation of each component. If one type of the above relationships occurred more than once between two classes, a weight indicating the frequency of occurrence is assigned to the corresponding edge.

We developed a custom software tool aligned with our requirements to extract the annotated class diagram from the program source code aligned with the described requirements. The proposed tool is publicly available in our GitHub repository [14] to be used and improved by other researchers. The parse trees of the production source files are created and traversed to extract the extended class diagram from the source code. The nodes, edges, and annotation data are extracted while traversing the parse trees in a depth-first manner. The algorithm's time complexity is $O(n)$, where n is the number of nodes in the parse tree [41].

To create and traverse the parse tree for Java programs, we employed the ANTLR parser generator [42]. ANTLR generates the parser code for the Java language from a formal description of Java grammar using the recursive decenet parsing technique [43]. A listener interface containing methods called when visiting the parse tree nodes is also generated along with the parser. For instance, the 'enterClassDeclaration' method is invoked when the parser enters a class definition within the Java source file. We override this method such that it creates a new node corresponding to the entered class and adds it to the extended class diagram. Similarly, other methods of the ANTLR listener interface were implemented to extract the required information for building the annotated classed diagram as a directed graph modeling a Java project.

4.3. Design testability prediction model

According to ISO/IEC 25010:2011 standard [11], testability is subject to the effectiveness and efficiency of tests. The testability of a given design can be best estimated based on the testability of its building blocks. We have recently proposed a mathematical model to compute the component testability regarding test effectiveness and efficiency [5]. In this model, the testability, $T(X)$, of the class, X, is calculated based on the parameters obtained by the actual testing of class X using Eq. (1):

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

The parameters constituting Eq. (1) are available after complete testing of a class in a system with an automatically or a manually generated test suite and are defined formally as follows:

- $\tau(X,\ C)$: The test suite generated for testing the class under test, X, regarding the test adequacy criteria, C. The test adequacy criteria considered in this paper are widely used code coverage metrics, including statement coverage, branch coverage, and mutation coverage [44]. These criteria are calculated and reported by many

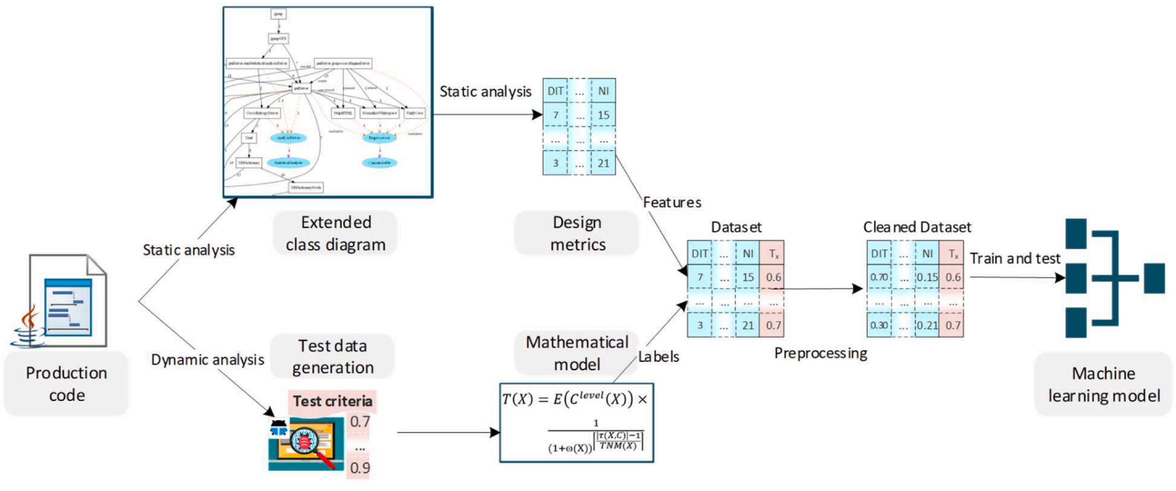


Fig. 3. Testability prediction pipeline.

test data generation tools, specifically the EvoSuite tool [45] for Java programs, during and after running the tool on the program under test. $|\tau(X,\ C)|$ denotes the size of the test suite, *i.e.*, the number of test data in the test suite which should be greater than zero on a successfully completed test. A test suite with at least one test data is required to compute the next parameter, $E\left(C^{level}(X)\right)$. The larger test suite results in a greater test effort reducing the testability of class, X.

- $E\left(C^{level}(X)\right)$: Expected coverage is defined as the mean value of the coverage regarding mutation, branch, and statement coverages, C, achieved when executing X with the test suite $\tau(X,\ C)$. The value of each code coverage metric is essentially in the range of $[0,\ 1]$ where zero means no coverage and one denotes the full coverage regarding that metric. Therefore, the average of these three code coverage metrics is also in the interval of $[0,\ 1]$. The higher $E\left(C^{level}(X)\right)$ results in a more effective test which signs for high testability of the class under test.
- ω(X): The average time of generating one test case which is equal to $\frac{t}{|\tau(X,\ C)|}$, where t is the overall execution time of the test cases $\tau(X,\ C)$. The testability equation considers the average time for generating one test case as the factor that its incrementation negatively affects the class testability.
- $TNM(X)$: The number of methods in the class, X. As the size of the class grows in terms of its methods the test suite size naturally increases. Therefore, Eq. (1) normalized the test suite size with the number of methods in the class to avoid basing the testability value on the class size.

Two main ingredients of Eq. (1) are $E\left(C^{level}(X)\right)$, specifying the test effectiveness, and $(1+\omega(\mathrm{X}))^{\left\lceil\frac{|\tau(X,\ C)|-1}{TNM(X)}\right\rceil}$, estimating the test effort. Hence, the proposed formulation conforms to the standard definition of testability proposed by ISO/IEC [11]. After computing the testability of all the classes that serve as building blocks of the program under test, the testability T(X) of each class component X in the program under test can be estimated. The design testability T ¯(D) can then be calculated as the average testability of all the classes using Eq. (2):

$$\overline{T}(D) = \frac{1}{n}\sum_{i=1}^{n} T_i(X) \tag{2}$$

Eq. (2) measures the program design testability precisely. However, each class in the design should be executed several times with a given test data set to establish the degree of its testability by Eq. (1), which is very time-consuming. As described by Fraser and Arcuri [46], to ensure that their experiments with testing 23,886 Java class in the SF110 benchmark of projects finish within a finite amount of time, a general timeout of six minutes per class was assigned to test each class in the project. In total, it took them about $(23{,}886 \times 10 \times 6)/(60 \times 24) = 995$ days to test all classes in SF110. In other words, it took their automatic test data generation tool, EvoSuite [45], approximately 9 days to test a Java project on average. EvoSuite [45] uses an evolutionary process, executing the software under test several times to generate test data and compute its coverage. In the case of scientific code, it may take several minutes, hours, or even days to run a program that, for instance, uses an iterative algorithm to solve a partial differential equation [47]. This situation becomes even worse in our case where the testability of the program is to be checked after every refactoring. Moreover, Eq. (2) requires the testability of all classes in the program, while at the design phase, even the program's source code is not available.

To address the above-mentioned challenges, we propose a testability prediction model using machine learning regression techniques. The testability prediction model is trained to map a set of design metrics extracted from classes in a given class diagram to the testability value of many classes in different designs. The testability of the classes in the training set is calculated with Eq. (1) only once to build the design testability prediction model. Afterward, the learned model is used to predict design testability. The resultant model would predict design testability without any need to execute the program under test. Another important promising point is that once the machine learning model to measure the class testability is built, there will be no need to access the source code.

Fig. 3 shows the machine learning pipeline to construct and evaluate the design testability prediction model. First, the design metrics for a large corpus of the projects are computed from their extended class diagram. After that, the proposed mathematical model calculates the testability of each class in the class diagram. The design metrics and testability values form the dataset on which a machine learning model is trained to predict design testability. Each class is mapped into a vector of design metrics, measured while analyzing the class statically. At the same time, each class is tested using an automatic test data generation with a specific time budget. The required test code metrics for computing class testability, including statement coverage (*St* Cov.), branch coverage (*Br* Cov.), mutation coverage (μ Cov.), and test suite size (τ), are computed according to the results of a test data generation tool, *e.g.*, EvoSuite [45]. The vectors of class metrics are preprocessed and standardized to be fed into the machine learning regression algorithms. As mentioned, each fixed-length vector representing a class in the design is labeled with its corresponding class testability, computed by the mathematical model in Eq. (1). As a result, learning samples for training and evaluating the testability prediction models in a supervised

Table 1
Design metrics used in testability prediction.

Relationship type	Metric name	Description	Level
Dependency	CBO	Coupling between objects	Class
	FANIN	Number of incoming invocations	Class
	FANOUT	Number of outgoing invocations	Class
Inheritance	DIT	Depth of inheritance tree	Class
	NOC	Number of children	Class
	NOP	Number of parent classes	Class
	NIM	Number of inherited methods	Class
	NMO	Number of methods overridden	Class
	NOII	Number of implemented interfaces	Class
	NOI	Number of interfaces	Package
Total	10		

manner are prepared. At prediction time, only static analysis of the given class is required, and there is no need to test the input class with an automatic test data generation tool. This way, no testing cost is implied when computing class testability.

4.3.1. Design metrics

The input of machine learning models typically is fixed-length feature vectors. Therefore, classes must be converted into fixed-length vectors representing their features. We employ a comprehensive set of design metrics that can be extracted from the proposed extended class diagram to vectorize each class in the design, modeled as class diagrams. A total of 10 metrics, shown in the second column of Table 1, are used to build the feature vector for the machine learning pipeline. The only criterion we considered in choosing these metrics is their availability in the design phase. As discussed in Section 1, the majority of software metrics can be only computed based on the information available at the source code level. Our approach extracts class diagrams from the source code of the legacy system when it is provided or accepts the class diagram produced as design phase artifacts.

As shown in the first column of Table 1, each metric in the feature vector is categorized into either dependency or inheritance class relationships. Therefore, it is possible to study the impact of each type of these coupling relationships on the design testability. The fourth column, "level," indicates whether each metric concerns a 'class' or a 'package' entity. The only package-level metric shows the number of interfaces defined in the package enclosing the class to be vectorized. The metric is used to calculate the impact of the design abstraction on the design testability.

Initially, we do not know how each design metric would affect the testability and to which extent this impact is important. The impact of each metric on testability is learned while building the machine learning regressor models. Post analysis the learned model indicates the most important design metrics affecting testability. Section 5.4 discusses the results of feature importance analysis on our testability prediction.

4.3.2. Test data generation

The vectors representing classes in a design are labeled with the testability value computed by Eq. (1) to form the design testability prediction dataset. To obtain the test suite information required by Eq. (1), we used the EvoSuite test data generation tool [48] on a large corpus of Java projects previously collected by the EvoSuite developers. EvoSuite generates test data for each Java class with the genetic algorithm, executes the class with the generated test data, and reports comprehensive information and metrics about the test suite, including the test coverage, test time, and the number of test data. These data are fed into Eq. (1) to compute the accurate testability of each class in the given projects. The advantage of using automatic test data generation instead of manually created test data is that the overall testing process is not biased towards the human tester skill. Moreover, the computational resources and test budget used to test all classes in all projects are the same. This way, we can ensure that the computed testability values are only affected by the inherent features of the design, which are described with design metrics. It is essential to control the tester's skills and test budget when measuring testability since testability is an intrinsic feature of the software artifact.

Manual test data generation on the other hand is subject to the expertise of the human tester. Indeed, we cannot ignore the variability and differing skill levels of human testers. A human tester with limited skill or knowledge could produce an extensive test suite that attains low coverage, while an experienced tester might produce a smaller suite that attains higher coverage of the same class. Therefore, we should consider the variability and differing skill levels of human testers as a significant factor affecting the test budget. The testing budget could vary depending on the human tester's expertise, mood, and decisions. However, standards do not make any considerations for the tester's expertise since testability is an inherent characteristic of software design or source code affected only by software engineering principles.

Another important advantage of using test data generation tools is the ability to generate tests for a large corpus of software projects. Such a corpus is required to create a large and quality dataset for the testability prediction task. Creating manual tests for a high number of projects, 110 projects, in our case, is not practical. It not only takes considerable time but also requires the human tester to understand the structure and logic of the code before designing any tests.

4.3.3. Learning algorithms

Testability computed by the proposed model is a real number in the interval $[0,1]$. Therefore, we used different algorithms from the family of regression models to find the best algorithm for the testability prediction task. As a final result, we found that the combination of three regressors, including multi-layer perceptron (MLP) [49], random forest (RF) [50], and histogram gradient boosting (HGB) [51], could be used to predict the design testability with a maximum R^2 score. Our hybrid model combines conceptually different machine learning regressors and returns the weighted average of predicted values by each base regressor to balance out the individual regressors' weaknesses. The final prediction value is computed based on Eq. (3) below:

$$T_{predicted}(X) = \frac{1}{\sum_{m\in\{MLP,\ RF,\ HGB\}} R_m^2} \sum_{m\in\{MLP,\ RF,\ HGB\}} R_m^2 \cdot T_m(X) \tag{3}$$

In Eq. (3), X is the input class, m is one of the base regressors, R_m^2 is R^2 score corresponding to model m, and $T_m(X)$ is the prediction result for class X based on model m. To compute $T_{predicted}(X)$ with voting regressor, first, each base regressor model, m, is called to predict the testability value, $T_m(X)$. This value is multiplied by the R^2 score corresponding to model m. The results of all individual models are then summed up and divided by the total value of base regressors R^2 scores. The weighted average in this way allows the model with a higher R^2 score contributes to the final output more than the model with a lower R^2 score. The overall results of the voting regressor are more accurate than the based regressor, since most likely for each input sample, one base regressor operates better than the other. Indeed, the prediction error of any base regressor for different samples is different which is reduced by averaging the results of different regressors on that sample.

The grid search strategy with the cross-validation technique was used to optimize base regressors' hyperparameters and avoid overfitting. The feature vectors were standardized with the QuantileTransformer preprocessing API of the Scikit-learn library [52]. It transforms all features into the same desired distribution, the uniform distribution in our application, to minimize the impact of outlier values.

4.4. Improving design testability

Separating the object creation responsibilities from object use can be best performed using the factory method [12] and dependency injection [13] design patterns. They mainly help achieve loosely coupled, flexible,

Algorithm 1
Factory method refactoring.

```
Input: Digraph class_diagram, /* The program extended the class diagram as a directed labeled graph */float sensitivity. /* A threshold for considering dependent classes as products */
Output: Digraph refactred_class_diagram
1. foreach Node creator in class_diagram do /* creator is a candidate for Factory class */
// Step 1: Factory candidate detection
2. neighbor_classes ← creator.output_edges().destinations() /* neighbor_classes are candidates for Product classes */
3. if length(neighbor_classes) < 1:
4. continue
5. endif
6. neighbors_methods_dict ← create_neighbors_methods() /* Map each class to its methods, keys are class names, values are method names*/
7. result_dict ← find_products(creator, neighbors_methods_dict, sensitivity) /* A dictionary with two keys of factory creator and products: {'factory': creator_name, 'products':
   {'classes': [], 'methods': []}}*/
8. if length(result_dict.products) < 1: /* The number of products depends on the value of sensitivity */
9. continue
10. endif
// Step 2: Factory application
11. interface ← create_interface(result_dict.products_common_methods, class_diagram) /* Create an interface including the common methods of the products */
12. class_diagram ← update_class_diagram(creator, interface, result_dict.products, class_diagram) /* Add factory methods to creator class and replace instantiation statements with factory
   methods and update class diagram */
13. endfor
14. return class_diagram
```

and testable designs [53–55]. The identification and application of these refactoring are proposed to improve design testability. The factory method is a creational pattern that uses factory methods to deal with the problem of creating objects without specifying the concrete class of object aimed to be instantiated. According to Kerievsky [8], placing creational responsibilities in classes that should not play any role in an object's creation results in the 'creation sprawl' smell. In such a design, knowledge for creating an object is spread across numerous classes, making integration testing difficult. Factory pattern fixes this problem by using one class to encapsulate both creation logic and a client's instantiation/configuration preferences.

4.4.1. *Factory method refactoring*

Factory method design pattern [12] consists of a set of similar classes, named products, which are instantiated throughout another class, called the creator class. Factory method refactoring aims to refactor a class diagram by introducing factory methods to improve the code structure. The main idea of identifying a factory method opportunity is to find the products and creator candidate classes. Our proposed approach for automating factory method refactoring begins by iterating through each node in the class diagram, considering them as potential factory classes. For each candidate, the proposed algorithm evaluates neighboring classes to identify potential product classes based on a specified sensitivity threshold. If the number of potential products meets the threshold, it creates an interface with common methods among the products. Then, it updates the class diagram by adding factory methods to the creator class and replacing instantiation statements with factory method calls. To identify potential products, the algorithm examines pairs of neighboring classes, considering those with a significant overlap in methods as candidates. The most promising product class is selected based on a quality metric that considers the commonality of methods. Finally, the factory method refactoring algorithm returns the modified class diagram reflecting the introduced factory methods and updated instantiation patterns.

After detecting a factory method candidate, the responsibility of creating the instance of the products class is delegated to the creator class by introducing the corresponding creator methods in that class. It is important to find the right candidate classes from the viewpoint of the software engineer. Otherwise, the applied refactoring might not be meaningful and logical even if it improves testability. In other words, a tradeoff between testability and other design quality attributes, such as understandability, must be considered when applying each refactoring. Therefore, two main preconditions are used to filter out factory method refactoring candidates and select the best ones.

The first precondition is the existence of a creator class in design. Existing approaches [37] introduce an explicit creator class when applying factory method refactoring. This increases the design size with classes that may exist in the initial design. Our approach selects a class with at least one outgoing dependency as a creator candidate. This way, the creator class would be the class that is already used to instantiate its products but by using concrete dependencies. The second precondition checks the similarity between candidate products to ensure they are classes of the same supertype. If the similarity between a set of classes that the creator class depends on is lower than a given threshold, they are not eligible to form the factory method's product classes.

The sensitivity threshold parameter serves as the key criterion for determining which classes are suitable for refactoring and which ones are not. It defines the minimum level of method commonality required among neighboring classes for a class to be considered a potential product of a factory. When evaluating pairs of neighboring classes, the algorithm calculates the ratio of common methods to the total number of methods. If this ratio exceeds the sensitivity threshold, indicating a significant overlap in functionality, the second class in the pair is considered a potential product. Therefore, the sensitivity threshold acts as a comprehensive criterion that encompasses method commonality, ensuring that only classes with a substantial level of shared functionality are selected for refactoring. By adjusting the sensitivity threshold, developers can control the granularity of the refactoring process, balancing between broader inclusion of classes and more stringent criteria for selection.

Three additional preconditions are also considered when refactoring to factory methods to ensure the correctness and meaningfulness of the applied refactoring. These preconditions are as follows:

1. The products' entities and the creator entity should only be concrete classes not interfaces or enums.
2. The stereotype between products' classes and the creator class can only be 'create', 'use_consult', and 'use_def'.
3. The number of product candidates corresponding to a creator class should be greater than one.

Algorithm 1 shows the pseudocode of factory method refactoring, which performs both the identification and application of all refactoring candidates on a given design. It receives as input an extended class diagram and a sensitivity threshold. The main loop (lines 1 to 13) considers each class with more than one outgoing dependency as a creator class and sends its neighbor classes along with the sensitivity threshold to the find_products function (line 7). If more than one product class is found, the factory method refactoring corresponding to the current creator class is applied (lines 11 and 12). First, in line 11, an interface

Algorithm 2
function find_products(*class*: Node, *neighbors_methods*: Dict<Node, List<Method>>, *sensitivity*: float).

```
1. factory_info ← {}
2. factory_info.creator ← class
3. candidate_product_classes ← neighbor_methods.get_classes()
4. factory_qualitities ← {}
5. forech class1 in candidate_product_classes:
6. products ← set()
7. method_list ← class1.methods /* Initialize the method list with the methods of the first class */
8. forech class2 in candidate_product_classes:
9. common_methods ← intersect(method_list, class2.methods)
10.  if length(common_methods) / length(union(class1.methods, class2.methods)) >= sensitivity then
11.  method_list ← common_methods.copy()
12.  products.add(class2)
13.  endif
14.  endfor
15.  factory_quality ← products.number_of_common_methods / Max(prodcuts.number_of_methods)
16.  factory_qualitities.update({products: factory_quality})
17.  endfor
18.  products ← factory_qualitities.argmax()
19.  factory_info.products ← products
20.  factory_info.products_common_methods ← products.common_methods
21.  return factory_info
```

Algorithm 3
function update_class_diagram(*creator*: Node, *interface*: Node, *products*: Dict, *class_diagram*: Digraph).

```
1. foreach Class product in products do
2. foreach Constructor c in product do
3. m ← add_method(creator, "public static create" + product.name)
4. m.set_return_type(interface.name)
5. m.set_param_list(c.param_list)
6. m.set_body("return new " + product.name + "(" c.param_list + ")")
7. endfor
8. class_diagram ← replace_ instantiations(product, interface.name, class_diagram) /* Replace all new statements with factory methods */
9. endfor
10.return class_diagram
```

containing the common methods of product classes is created using the create_interface function. Then, the update_class_diagram function is called to create factory methods in the creator class and update all instantiation of the product classes with the creator methods. Finally, the refactored version of the extended class diagram is returned as the algorithm's output. The pseudocode of the find_products and update_class_diagram functions are respectively shown in Algorithms 2 and 3.

The most important and core part in automating refactoring to factory method design pattern is the find_products function shown in Algorithm 2. This algorithm aims to find a subgroup of classes from the neighbor classes of the creator class such that factory quality is maximized. The similarity between the two classes is computed as the ratio of their common methods to all methods. The first loop in the find_products function iterates on all candidate product classes. The second loop adds each class with a similarity greater than the *sensitivity* threshold to a temporary product set. The factory quality for this set of selected classes is considered as the ratio of their common method to the maximum number of methods in product classes (line 15). The set of selected products satisfying the sensitivity threshold (line 10), along with the computed factory quality (line 15), are added to the *factory_qualitities* dictionary (line 16). After finding all satisfying sets of product classes, a set with the maximum factory quality is considered the final product and returned by the find_products function (lines 18 to 21). This way, we can ensure that the product classes are semantically similar together and should realize the same interface.

The greater sensitivity value denotes that a pair of classes must be highly similar to form the product classes. Therefore, the number of identified factory refactoring opportunities depends on the sensitivity threshold value. In our experiments to improve design testability with factory refactoring, described in Section 4, we determine the sensitivity threshold to maximize the predicted testability using a different value for this threshold. The proposed sensitivity parameter is beneficial in regulating product selection quality for factory method refactoring and avoiding overly strong preconditions [56] to access useful transformations. The only automated refactoring approach for the factory method refactoring [37] has not considered such regularization.

The update_class_diagram function shown in Algorithm 3 is responsible for applying the factory method refactoring parametrized with the creator class, products classes, and interface of the products. First, for each constructor in the product classes, this algorithm creates a 'public static method' in the creator class (lines 1 to 7). The parameters of this method are the parameters of the corresponding constructor in the product class, set if line 5. However, its return type, set in line 4, is the interface generated for the product class in line 11 of Algorithm 1. After adding all creator methods to the creator class, the replace_instantiations function is called to replace all direct object instantiation by the creator methods of the creator class entire the design (line 8). This way, all dependencies of type 'create' are removed between the product classes and other concrete classes in the design. Moreover, adding a new product to the product class only requires defining a creator method in the creator class.

4.4.2. *Dependency injection refactoring*

Dependency injection is a design pattern in software engineering where an object or function receives its dependencies from an external source rather than creating them internally. In factory method refactoring, we seek a set of classes with similar functionality, *a.k.a.* products, to establish a creator class (with a create method) for the products. The creator class is selected as a class that has at least one dependency on all the products. However, a class may depend on different classes with different functionalities to perform its responsibilities that are not similar anyway. At the same time, instantiating and parametrizing all required dependencies is not the main responsibility of the class. In such cases, refactoring to the factory method is not a logically meaningful

Algorithm 4
Constructor injection refactoring.

```
Input: Digraph class_diagram, /* The program extended the class diagram as a directed labeled graph */
Class x, /* A class for which constructor injection is applied */
Output: Digraph refactored_class_diagram
1.  neighbor_classes ← x.output_edges("create").destinations() /* Retrieves all classes for which x creates instances */
2.  if len (neighbor_classes) < 1 then
3.      return class_diagram
4.  endif
5.  if len(x.constructors) < 1 then /* Create explicit constructor for class, x, if it does not have explicit constructor */
6.      m ← add_method("public " + x.name())
7.      x.add_constructor(m)
8.  else
9.      m ← x. constructors[0]
10. endif
11. foreach class in neighbor_classes do
12.     interface ← extract_interface(class, "I" + class.name()) /* Create an interface of the neighbor_class to be injected */
13.     class_diagram.add_node(interface)
14.     class_diagram.add_edge(class, interface, "realization")
15.     instances ← x.get_instances_of(class)
16.     foreach instance in instances do
17.         m.add_parameter(interface.name(), instance.name())
18.         x.replace_object_creations(class.name(), "I" + class.name())
19.         class_diagram.remove_edge(x, class, "create") /* Remove dependency between concrete classes */
20.         class_diagram.add_edge(x, interface, "use-consult")
21.     endfor
22. endfor
23. neighbor_classes ← x.input_edges("create").sources()
24. update(class_diagram, neighbor_classes)
25. return class_diagram
```

solution. Instead, a kind of dependency injection (DI) pattern, namely the constructor injection, can be applied to create and pass different dependencies to the constructors of a target class from the client class.

Constructor injection refactoring enhances a given class diagram by injecting the required objects into an identified dependent class via constructor parameters. It promotes loose coupling and enhances the modularity and testability of a system by allowing for greater flexibility in the configuration of its components. Our approach to automated constructor injection refactoring first identifies classes for which the chosen class creates instances. If there are such classes, it ensures the chosen class has explicit constructors. Otherwise, no modification is applied to the class diagram. For each identified class that does not already have an interface, the constructor injection refactoring algorithm creates an interface and establishes the 'realization' relationship between the class and its interface. Thereafter, it modifies the constructor of the chosen class to accept instances of these interfaces, replacing direct object creations with interface instances. Finally, it updates the diagram to reflect these changes and returns the modified class diagram. The following three preconditions are checked to filter out constructor injection refactoring candidates and select the best one:

1. *Existence of reference type attribute(s) or variable(s) in a dependee class.* The selected candidate class for constructor injection refactoring must have at least one instance attribute or constructor variable with a user-defined reference type. Otherwise, no constructor injection is required if all member attributes and constructor variables are primitive.
2. *Existence of object instantiation statement(s) in a dependee class.* At least one object instantiation must be performed inside the dependee class or its constructor(s). Using static methods and variables does not lead to class instantiation.
3. *Existence of a dependent class.* The dependee class must be used by at least one dependent class in the program. Constructor injection is typically applied to classes that are frequently instantiated by other classes within the system. Without dependent classes, there is no need for constructor injection.

Algorithm 4 shows the identification and application of constructor injection refactoring. It receives the extended class diagram and a candidate class as input and performs the constructor injection where possible. First, in line 1, all classes having a relationship of type 'create' with the input class, x, are retrieved as neighbor_classes. If no class exists, the algorithm is terminated without applying any transformation to the class diagram (lines 2 and 3). If the input class, x, does not have any explicit constructor, a contractor is added to the class (lines 5 to 7). Afterward, an interface is created for each neighbor_classes and added to

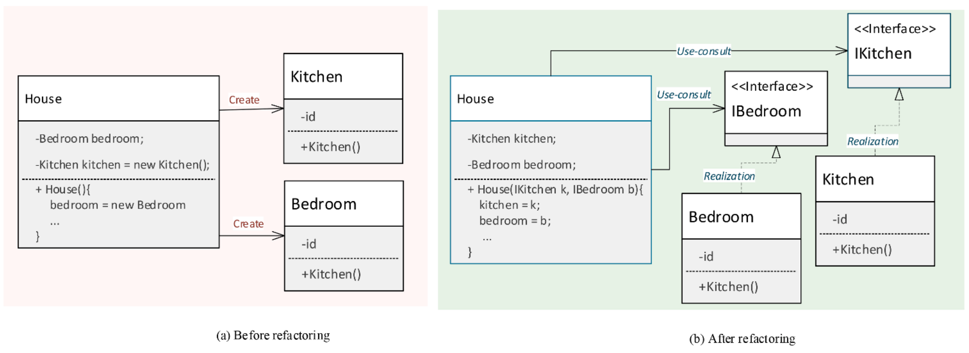


Fig. 4. Illustration of the proposed constructor injection refactoring algorithm on a sample class diagram.

the class diagram (lines 12 to 14). After that, in line 17, for each instantiation of a neighbor class in the input class, *x*, a parameter with the type of the generated interface is added to the first constructor of the class, *x*. The replace_object_creations function, called in line 18, moves all object creations of the neighbor class in the class, *x*, to the constructor and replaces direct object creation with the interface type. The class diagram is then modified by removing the 'create' edge between the two concrete classes, the input class, *x*, and the neighbor class, and adding a new edge from the input class, *x*, to the generated interface corresponding to the neighbor class. After updating the input class, all classes that instantiate this class are retrieved as neighbor_classes (line 23) and updated to pass the required dependencies in the creation time using the update function in line 24.

Fig. 4 shows a sample class diagram before and after applying the constructor injection refactoring to demonstrate the application of Algorithm 4. Fig. 4.**a** shows a class diagram containing three concrete classes. The “House” class directly creates instances of the “Kitchen” and “Bedroom” classes, which makes testing the class hard due to tight coupling. In Fig. 4.**b,** the object creation codes have been removed from the “House” class after applying the constructor injection automatically. The new design allows testing in isolation of the “House” class by injecting test doubles [57] to the class at runtime. Moreover, the refactoring algorithm ensures that dependent classes can be injected via interfaces, forming loose coupling and improving testability. The relationships between the “House” class and the generated interfaces do not require testing. Since the interfaces are stateless, the dependencies between the concrete class and interfaces are of type 'use-consult.'

It should be noted that dependency injection is performed after factory method refactoring in our approach to prioritize the objection creation in the theme of factory design pattern where all products have a common interface. If the objects created by the creator class do not form a suitable set of product classes, they are injected individually into the creator class.

4.4.3. Adding stereotypes

The proposed factory method and constructor injection refactoring operations add appropriate stereotypes to newly created relationships during the refactoring process. More precisely, the new object creation relationships are labeled with the 'create' stereotypes, and the relationship between concrete classes and interfaces in cases of dependency relationship is annotated with the 'use-consult' stereotypes. An example of adding annotations has been shown in Fig. 4. According to Baudry and Traon [2], these additional specifications help programmers avoid implementing extra object interactions by removing ambiguities from the class diagram. They also help prioritize testing of the dependencies in integration testing. Automated verifications can then be used to check whether the code is in conformance with stereotypes' constraints.

5. Evaluation

The proposed design testability measurement and improvement have been evaluated on an extensive set of real-world Java software. This section describes the research questions, setups, and evaluation results of our experiments.

5.1. Research questions

The experiments in this section have been conducted to answer the following important questions about design testability measurement and improvement:

- **RQ1:** *How is the performance of regression models trained to predict design testability?* By assessing model performance, we can determine the reliability of design testability predictions in different contexts. Knowing the baseline performance helps find the best machine learning model for the task of testability prediction and compares new approaches or enhancements.
- **RQ2:** *What are the most influential design metrics affecting software design testability prediction?* Answering this question helps developers focus on improving critical metrics by selecting appropriate refactoring operations work at design and source code levels.
- **RQ3:** *To what extent does refactoring to the factory method and dependency injection patterns improve the software design testability?* Refactoring is the most common practice to enhance software quality, but its impact on design testability needs systematic investigation discussed in this research question. If refactoring improves testability, developers can apply these patterns intentionally. Moreover, understanding trade-offs (*e.g.*, increased design complexity *vs.* improved testability) informs decision-making.

To answer RQ1, we train four regression models discussed in Section 3.3.3 on 110 class diagrams extracted from 110 Java projects containing 23,000 Java classes, vectorized with ten metrics in Table 1. The performance of our testability prediction models is measured with typical regression evaluation metrics, mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), median absolute error (MdAE), and coefficient of determination (R^2) score. These metrics are calculated by Eq. (4) to (7). In all equations, $\hat{y}_i$ is the predicted value of the i^{th} sample, y_i is the corresponding true value for $\hat{y}_i$, $\overline{y}$ is the expected value of y, and n is the number of samples.

These are common and reliable metrics for evaluating the performance of regression models, such as our testability prediction model for calculating the testability score of classes in a given design. All metrics, except the last one, (Eq. (7)), evaluate the model error on samples. Hence, a lower value of them means a better model. Mean absolute error measures the average magnitude of the errors in the predictions, and mean square error measures the average squared magnitude of the errors in the predictions. On the other hand, the R^2 score indicates the goodness of fit and how well the model can predict unseen samples. Therefore, a model with a higher R^2 score has better performance. The best possible R^2 score is 1.0, and it can be negative since the model can be arbitrarily worse. A constant model that predicts the expected value of y, disregarding the input features, would get an R^2 score of 0. These metrics allow us to evaluate the accuracy and robustness of our testability prediction model and demonstrate its potential for practical use.

$$\mathrm{MAE}(y,\hat{y}) = \frac{1}{n}\sum_{i=0}^{n-1} |y_i - \hat{y}_i| \tag{4}$$

$$MSE\big((y,\ \hat{y}\big) \ = \ \frac{1}{n}\sum_{i=1}^{n-1}(y_i - \hat{y}_i)^2 \tag{5}$$

$$\mathrm{MdAE}(y,\hat{y}) = median(|y_1 - \widehat{y_1}|,\cdots,|y_n - \widehat{y_n}|) \tag{6}$$

$$R^2(y,\ \hat{y}) \ = \ 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2} \tag{7}$$

To answer RQ2, the feature importance analysis based on the permutation importance technique [50] is performed to automatically determine the importance of each design metric. The importance is measured in terms of the model R^2 score degradation when the metric's values in the test set are shuffled.

To answer RQ3, the proposed automated refactoring to the factory method and dependency injection patterns are applied to the extended class diagram of six Java open-source projects, and the change in testability is reported before and after applying all feasible refactoring. The best design testability prediction model (the model with the highest R^2 score) is used to measure the testability before and after refactoring the selected projects. The relative changes in testability are computed by Eq. (8):

Table 2
Learning dataset statistics.

Statistic	Nodes (classes + interfaces)	Edges (relationships)	Test data
Mean	275.73	1133.13	15.67
Minimum	3	2	0
Maximum	7955	41,611	414.4
Sum	29,503	121,245	377,444.9

$$\Delta T(X)_{relative} = \frac{\left|T_{after}(X) - T_{before}(X)\right|}{T_{before}(X)} \times 100 \tag{8}$$

5.2. Experiments setup

All experiments have been performed on a computer system with an Intel® Core i7 and 16 GB of RAM. Test data generation was repeated five times for each project, and the average of the results was used to compute testability values. This way, the uncertainty caused by the stochastic nature of the EvoSuite test data generation tool is minimized.

5.2.1. Software systems used for testability learning

The dataset used to create and evaluate the proposed testability prediction approach contains the class diagrams of the SF110 corpus [46]. SF110 was collected by EvoSuite developers as a benchmark to evaluate their tool. It contains 110 open-source software projects and over 23,000 concrete Java classes. More specifically, the SF110 dataset comprises a carefully chosen set of 100 Java projects sourced from SourceForge, a well-established open-source repository. Additionally, it includes the ten most popular Java applications, which have recently garnered millions of downloads and active users [46]. The resulting SF110 corpus encompasses a substantial collection of classes, spanning 110 projects and totaling 23,886 classes. Furthermore, it contains over 800,000 bytecode-level branches and an impressive 6.6 million lines of code. The primary strength of this corpus lies in its random selection from an open-source repository, ensuring that the distribution of software types—such as numerical applications, business software, and video games—is statistically representative of the broader open-source ecosystem.

We computed the design metrics and testability values for all available classes in SF110. Table 2 shows the descriptive statistics of the SF110 class diagrams, indicating that it is large enough to be used for appropriate machine learning. The design testability dataset was split into training and testing sets such that 75 % of samples were used for training, and the remaining 25 % of samples were used to test the models. The design for the testability dataset prepared and used in our experiments is publicly available on Zenodo [58].

5.2.2. Software systems used for refactoring

We selected six software projects from the SF110 corpus [46] to examine the proposed refactoring operations on them and evaluate the changes in design testability. Table 3 shows the projects, domains, and size of each project. As shown in the table, the selected projects are of different sizes and application domains, making it suitable to evaluate the proposed testability improvement approach. In total, 1507 Java classes with more than 73,000 lines of code are used in our experiments with refactoring Java classes.

5.3. RQ1: design testability prediction performance

Table 4 shows the performance of different design testability

Table 3
Benchmark projects, used to evaluate the proposed approach in this paper.

Project	Link to the source code	Short description	Number of classes	Lines of codes
Tullibee	https://sourceforge.net/projects/tullibee	Provides Java EE components that work with interactive brokers' financial-trading Java API.	20	4869
Shop	http://www.cs.umd.edu/projects/shop	A hierarchical ordered planner application.	34	5348
JHandBallMoves	https://jhandballmoves.sourceforge.net	A handball tactics board for all platforms.	69	5489
Follow	https://follow.sourceforge.net/	Monitor text files to which information is being appended asynchronously (*e.g.*, log files).	101	4946
Fim1	https://sourceforge.net/projects/fim1/	An instant messenger server and client.	242	10,602
Corina	https://dendro.cornell.edu/corina/corina.php	An open-source dendrochronology program.	1041	42,087
Total			1507	73,341

Table 4
Performance of different design testability prediction models.

Model	Best hyperparameters	MAE	MSE	RMSE	MdAE	R^2
Multi-layer perceptron (MLP)	'hidden_layer_sizes': (256, 100), 'activation': "tanh", 'learning_rate': 0.001, 'max_iter': 150, 'solver': 'adam'	0.1784	0.0510	0.2258	0.1428	0.4137
Random Forest (RF)	'max_depth': 20, 'n_estimators': 100, 'criterion'="squared_error", 'min_samples_leaf': 1, 'min_samples_split':2,	**0.1504**	0.0414	0.2035	**0.1157**	0.5239
Histogram gradient boosting (HGB)	'max_depth': 20, 'n_estimators': 100, 'learning_rate': 0.05, 'loss'="squared_error", 'min_samples_leaf': 25	0.1547	0.0412	0.2031	0.1230	0.5228
Hybrid voting model	'MLP_weight': 0.2803, 'RF_weight': 0.3569, 'HGB_weight': 0.3628	0.1560	**0.0410**	**0.2024**	0.1235	**0.5286**

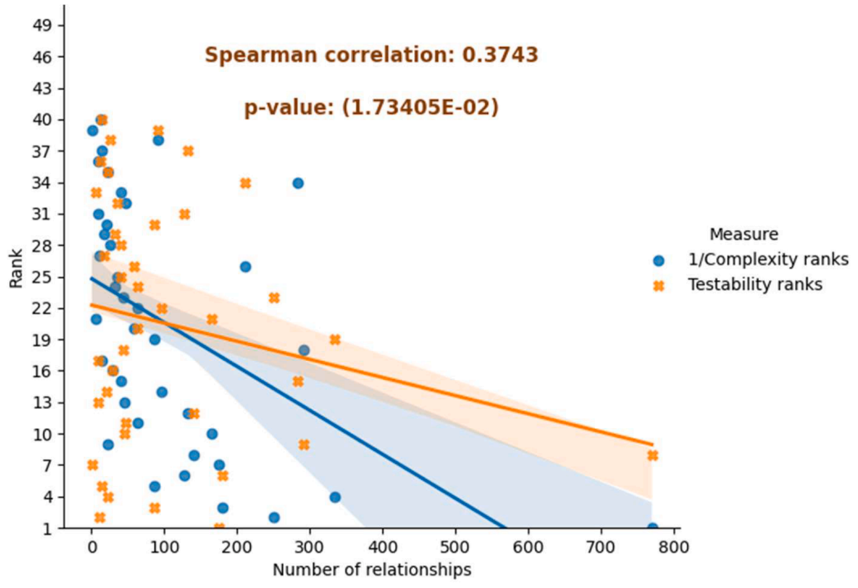


Fig. 5. Interaction complexity and design testability ranks per the number of design relationships.

prediction models on the test set along with the best hyperparameters obtained during the grid search with five-fold cross-validation for each model on the training set. According to Eq. (3) in Section 4.3.3, the weight used to average the result in the hybrid model is proportional to the R^2 score of the base regressors.

It is observed that the performance of the hybrid model is better than the base regressors in three, including MSE, RMSE, and R^2, out of the five evaluation metrics described in Section 4.1. The proposed model can predict the design testability with an MSE of 0.04 and an R^2 score of 0.53. These metrics indicate that the model can capture the relationship between the design features and the testability outcomes in practice. It is worth noting that other machine learning regressors, such as decision tree and support vector machine, have been evaluated in our experiments. However, we noticed that their performance is significantly lower than the models reported in Table 4. Therefore, we did not report their results in this section.

Unfortunately, there is no ground truth data or relevant measurement tool regarding the design testability of software systems to be used as a basis for comparing existing approaches. Therefore, we perform a relative comparison of the efficiency and correctness of our testability prediction method with the design testability measurement method proposed by Baudry and Traon [2]. To this aim, we applied our testability prediction model to measure the testability, $T(D)$, of 110 class diagrams, extracted from 110 projects in the SF110 corpus [46]. Then, we applied a design testability computing method called the interaction complexity [2] to compute the average complexity interaction of all pairs of classes in each class diagram of the SF110 projects. Afterward, we ranked the projects once based on their testability values, measured by our model, and once based on the inverse of the interaction complexities measured by the approach proposed in [2].

The Spearman's rank correlation was used to evaluate the correctness of our testability prediction model compared to the results provided by Baudry and Traon's approach [2]. Spearman's rank correlation is a statistical measure that assesses the association between two ranked (ordered) variables. It is a valuable tool for understanding associations in situations where the rank order matters more than the exact values. Spearman's correlation focuses on the rank order of scores rather than their absolute values. Unlike Pearson's correlation, which assumes a linear relationship between continuous variables, Spearman's correlation is nonparametric and more versatile.

Fig. 5 shows the ranking results of the SF110 projects with the above-mentioned methods. The vertical axis shows the projects' ranks, and the horizontal axis indicates the number of class dependencies in the projects. The regression line between project ranks and the number of the class diagram's relationships are illustrated to infer the correlation between the two resultant ranks. In addition, a 95 % confidence interval computed by the bootstrap sampling technique is shown for each regression line. It is observed that the projects with high connections receive lower ranks than the projects with few relationships in both ranking methods. The Spearman rank correlation between testability prediction results and the inverse of design complexity interaction is 0.37. A p-value lower than 0.05 indicates that this correlation is statistically significant, with a confidence level of 0.95. Therefore, our method can be used instead of Baudry and Traon's interaction complexity approach [2] to estimate design testability while it maintain a positive correlation with that approach.

It should be noted that a timeout of 60 min was given to compute design interaction complexity for each project. Computing the design complexity interaction for 69 out of 110 projects was not finished during this timeout, while testability prediction was completed for 95 out of 110 projects. Therefore, Fig. 5 only shows the ranks of 41 projects for which both measurement techniques were finished reasonably. As a result, our approach can be used instead of Baudry and Traon's approach [2] when improving class interaction complexity throughout the refactoring process.

The main advantage of testability prediction over the complexity of interaction is the interpretability of results. The proposed method predicts design testability in the interval $[0, 1]$. This value is explainable

Table 5
Testability prediction and design interaction complexity computation time statistics.

Method	# Projects	Computation time (second)				
		Sum	Mean	Minimum	Maximum	Standard deviation
Interaction complexity [2]	49	1137.119	25.269	0.008	936.354	140.246
Testability prediction	95	1686.837	37.480	1.404	417.006	84.504

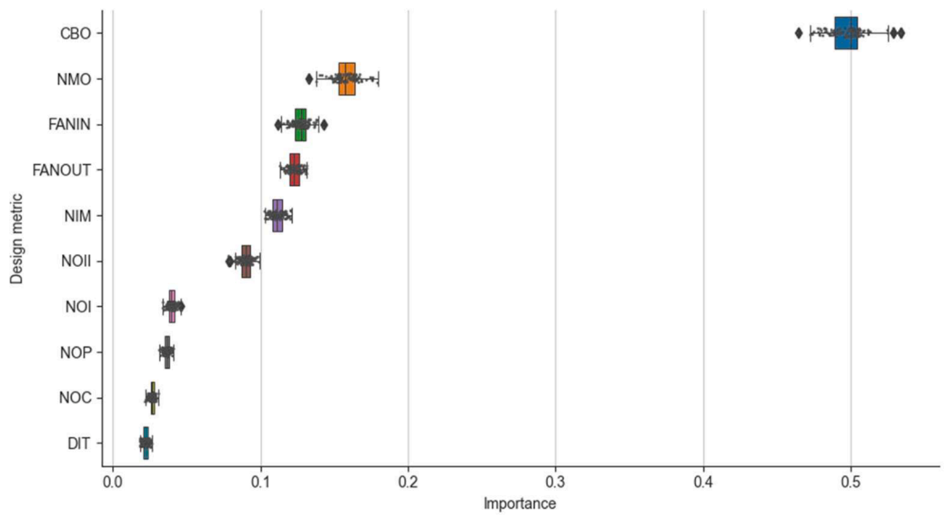


Fig. 6. Design metrics' importance in predicting design testability.

from the software engineer's viewpoint. Moreover, the proposed testability prediction approach is more efficient than computing the interaction complexity. The complexity interaction method calculates all possible paths between two nodes (classes or interfaces) in a class diagram which is an NP-Hard problem. The time complexity of computing the interaction complexity is $O(v!)$, where v is the number of nodes in the class dependency graph. Therefore, it takes a long time to compute the interaction complexity between all pairs of classes for projects with a large number of nodes (classes).

Table 5 shows descriptive statistics of the computation time for design testability protection and design interaction complexity. We observe that the maximum time and standard deviation of calculating design interaction complexity are greater than the testability prediction time. However, its average, minimum, and total time are still better than the testability prediction. It is worth noting that the computation time for projects whose processing was not finished in the 60-minute timeout has not been reported in the descriptive statistics of Table 5. Therefore, we conclude that the overall efficiency of testability prediction is better than design interaction complexity, specifically for large-scale projects.

Similar to the testability prediction model, the interaction complexity method also needs to be executed after applying each refactoring to measure the changes in the design testability, making it impractical in large software projects. On the other hand, testability prediction is efficient but always estimates testability with some error rate.

5.4. *RQ2: important testability design metrics*

One of the main benefits of the proposed testability prediction model is the ability to find the most important design metrics automatically by interpreting the learned model. We proposed a hybrid regressor model combining the results of three base regressors. Therefore, no straightforward feature analysis can be performed on such a model. Permutation feature importance [50] is a valuable model inspection technique used to assess the contribution of individual features to a fitted model's statistical performance on a given tabular dataset. Permutation feature importance is indeed a model-agnostic technique, unlike impurity-based methods such as Gini impurity or information gain. It can be applied to any machine learning model, regardless of its type (*e.g.*, decision trees, neural networks, linear regression). For each feature, it randomly

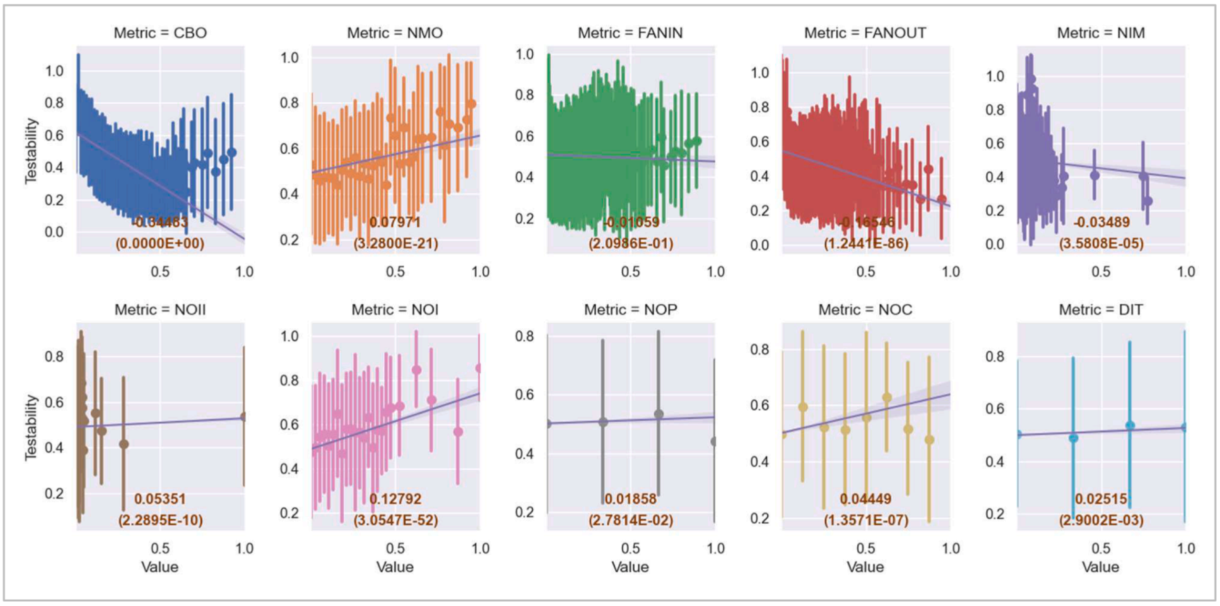


Fig. 7. Relationship between design metrics and design testability.

Table 6
Results of applying factory method and dependency injection on design testability.

Project	Factory sensitivity (best value)	Number of refactoring		Testability			Relative improvement $\Delta T(X)_{relative}$ (%)		
		Factory	DI	Initial	After factory	After DI	Factory	DI	Factory + DI
Shop	0.2	7	23	0.40442	0.48145	0.48169	19.047	0.050	19.106
Tullibee	0.1	2	2	0.59128	0.66321	0.67313	12.165	1.496	13.843
Fim1	0.1	5	6	0.59186	0.63202	0.65733	6.785	4.005	11.062
JHandBallMoves	0.1	3	7	0.54228	0.56116	0.59357	3.482	5.776	9.458
Corina	0.1	6	49	0.53462	0.54689	0.57846	2.295	5.773	8.210
Follow	0.1	1	5	0.51742	0.53362	0.55802	3.131	4.573	7.847
Mean	0.1	4	15.33	0.53031	0.56973	0.59037	7.818	3.612	**11.588**

shuffles its values within the dataset to break the relationship between that specific feature and the target variable. Thereafter, it observes and reports how this shuffling affects the model's performance (*e.g.*, a decrease in accuracy or an increase in error). The drop in performance quantifies the importance of the feature that has been shuffled.

Fig. 6 illustrates the box-whisker plot of design metrics based on their importance. The results have been obtained by applying the permutation importance algorithm [50] to the hybrid design testability prediction model 100 times for each metric. The horizontal axis, labeled by 'importance', shows the decreases in the R^2 score of the model evaluated on the test set after shuffling the corresponding metrics shown by the vertical axis. The CBO metric is the most important design metric affecting testability prediction, which belongs to dependency metrics. Indeed, according to Fig. 6, if we remove this metric from our testability prediction feature set, the R^2 score of the machine learning model is reduced on average by 0.50. The NMO metric is the second important metric that belongs to the inheritance metrics. Removing this metric decreases the learned model R^2 score by about 0.16. It concludes that both the types of dependency and inheritance metrics are affecting design testability. These importance scores have been computed automatically and therefore they are not biased towards any human judgment. However, Fig. 6 does not specify the direction of relationships between important design metrics and testability.

Fig. 7 shows the relationship between design metrics and design testability in our proposed dataset, complementing the results of Fig. 6. All metrics values have been normalized in the interval of $[0, 1]$ for better visualization. The Pearson correlation coefficient along with the p-value of correlation, have been shown on each plot in the figure. According to p-values, the reported correlation is significant in all metrics except FANIN and NOP, with a confidence level of 0.99. However, the correlation coefficient is pretty low for half of these metrics. It indicates that testability improvement techniques must focus on metrics highly correlated with testability. The CBO and NMO metrics have, respectively, negative and positive correlations with testability. The high CBO value for a given class in a design shows that the class is coupled with many other classes which violates the low coupling or modular design principle. The high NMO value indicates that a child class refused to use many methods of its parent class.

As a result, both types of negative and positive correlations emphasize that decreasing relations increase design testability. However, despite CBO, the high value of NMO in an inheritance hierarchy is deemed as a sign of poor object-oriented design violating the hierarchy principle. Hence, it concludes that the following object-oriented principles do not necessarily result in a testable design. Indeed, testability must be measured after refining the design to indicate whether the applied changes improve testability or not.

Increasing the NMO, NOI, and NOC metrics improve design testability while increasing CBO, FANOUT, and NIM decreases it. Both proposed automated refactoring operations, *i.e.*, refactoring to the factory method and dependency injection, increase the number of interfaces and overridden methods while decreasing the coupling between objects and outgoing dependencies. Therefore, applying these refactorings is expected to improve the design testability predicted by the machine learning model.

5.5. RQ3: impacts of automated refactoring on design testability

Table 6 shows the testability value before and after applying the refactoring to the factory method and dependency injection (DI) on the extended class diagram of each project in our benchmark. The second column denotes the sensitivity threshold used in refactoring to the factory method for which the maximum improvement in testability is achieved. The last column shows the total relative improvement after applying the batch of factory method and DI refactoring operations. It is observed that, on average, the applied batch refactoring has improved the design testability by about 11.59 %. A maximum of 19.11 % improvement is observed for the 'Shop' project, indicating a relatively high improvement in design testability when considering the factory method and DI refactoring operations in a batch mode.

The main reason for the improvement is that during refactoring to factory and dependency injection some strong coupling relations between concrete classes are removed and replaced by coupling between concrete classes and interfaces which are more flexible. As a result, the values of design metrics such as CBO and FANOUT decrease, and metrics such as NOI and NMO increase after refactoring. According to Fig. 7, such changes in important testability predictors lead to improving the overall design testability.

According to the results shown in Table 6, the contribution of DI refactoring on design testability improvement is lower than the factory method refactoring. The reason is that when applying factory method refactoring, some DI refactoring opportunities are also covered and fixed as a result of the refactoring design to the factory method. On the other

Table 7
Results of applying dependency injection on design testability.

Project	Number of DI refactoring	Number of constructor parameters		Testability		Relative improvement $\Delta T(X)_{relative}$ (%)
		Initial	After DI	Initial	After DI	DI
Shop	23	14	37	0.40442	0.41466	2.532
Tullibee	2	5	6	0.59128	0.6012	1.678
Fim1	6	82	88	0.59186	0.61717	4.276
JHandBallMoves	7	65	72	0.54228	0.57469	5.977
Corina	49	292	333	0.53462	0.56619	5.905
Follow	5	56	60	0.51742	0.54182	4.716
Mean	13.14	85.67	99.33	0.53031	0.55262	**4.181**

Table 8
Overlap between dependency injection and factory method refactoring on testability improvement.

Project	Testability relative improvement by refactoring to DI	Testability relative improvement by refactoring to DI after refactoring to the factory method	Overlap	Relative overlap (%)
Shop	2.532	0.050	2.482	98.025
Tullibee	1.678	1.496	0.182	10.846
Fim1	4.276	4.005	0.271	6.338
JHandBallMoves	5.977	5.776	0.201	3.363
Corina	5.905	5.773	0.132	2.235
Follow	4.716	4.573	0.143	3.032
Mean	4.181	3.612	0.569	13.609

hand, we observed less improvement in design testability when applying DI refactoring before the factory method. Indeed, DI refactoring removes some objects created by a class (in its constructors), which may lead to the loss of a factory method refactoring opportunity. For these reasons, the best sequence to apply a batch of factory methods and DI refactoring operations for improving design testability is the sequence in which the factory method refactoring operations are detected and applied first, *i.e.*, before dependency injection.

Table 7 shows the results of applying the dependency injection method without considering the factory method effects. The average design testability improvement is about 4.81 %, which is lower than that obtained by applying both the factory method and dependency injection refactoring operations. As a result, refactoring to DI might not directly improve design testability as much as a factory method. However, it facilitates the creation of test doubles essential for an effective test in isolation. The third column of the table shows the number of constructor parameters whose type is not primitive before and after refactoring to DI. On average, the number of constructor parameters has increased by 15.95 % after refactoring to DI. Indeed, the test doubles can be created for an average of 14 new parameters in addition to previous constructor parameters at each project. Therefore, refactoring to DI improves software testability at the unit level.

It should be noted that we cannot combine factory methods and dependency injection into one refactoring or perform only one of these refactoring operations without degrading the testability improvement. They are different design patterns with relatively different goals and impacts. Moreover, not all dependency injection opportunities form a factory method refactoring opportunities. To understand the impact of both refactorings on testability, we measured the overlap between two refactoring operations, *i.e.*, dependency injection and factory method on testability improvement. By overlap, we mean how refactoring to the factory method covers improvement achieved by refactoring to the dependency injection. In other words, is there any need to apply dependency injection after applying factory method refactoring?

To measure the overlap, we computed the testability improvement achieved by applying dependency injection in isolation and the testability improvement achieved by applying dependency injection after performing factory method refactorings. Table 8 shows the absolute and relative overlap values between dependency injection and factory method refactoring for each project and on the overage. The fourth column, 'overlap' is the difference between testability improvement by refactoring to DI after the factory method (shown in column 3) and testability improvement by refactoring to DI without factory (shown in column 2).

The relative overlap shown in the last column is computed by dividing the overlap value by the testability improvement by refactoring to DI (*i.e.*, the value reported in column 2). For example, in the 'Shop' project if we apply refactoring to DI after applying all factory method opportunities, testability improvement by DI would be about 0.05 %.

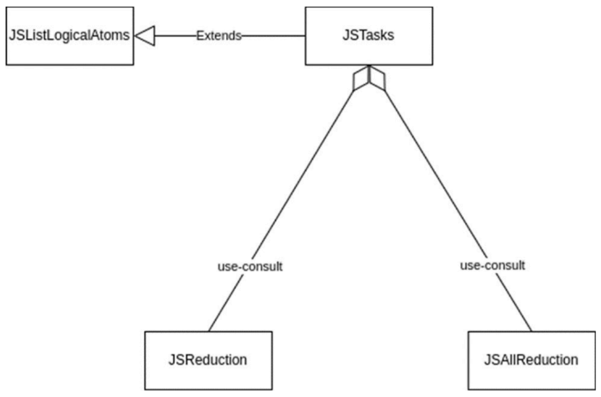


Fig. 8. "JSTasks" class in the Shop project and its dependencies before applying the factory method refactoring.

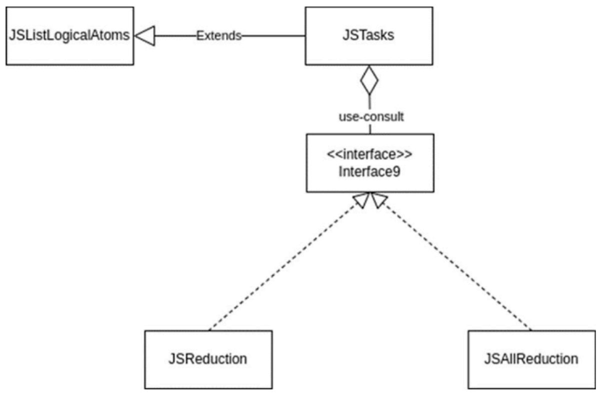


Fig. 9. "JSTasks" class in the Shop project and its dependencies after applying the factory method refactoring.

However, testability improvement by refactoring to DI before the factory is about 5.53 %. Therefore, it concludes that many dependency injection opportunities are indeed factory methods and are covered by the factory. As shown in Table 8, the overlap between the two refactorings is about 98 % in the 'Shop' project which is pretty high. However, on average, the overlap between dependency injection and factory method is about 13.6 %, indicating that both of these refactoring operations are required and influential when improving design testability.

An example of refactoring to the factory method identified and applied by our approach in the 'Shop' project is illustrated to indicate the applicability of the proposed method on real-life projects. Fig. 8 shows the "JSTasks" class, identified as a creator class by the factory method refactoring algorithm, along with its dependencies. It uses "JSReduction" and "JSAllReduction" detected as product classes in the applied factory method refactoring. The "JSTasks" class also extends the "JSListLogicalAtoms" class.

The factory method refactoring algorithm found that "JSReduction" and "JSAllReduction" have common methods. Therefore, the refactoring algorithm creates an interface named "Interface9" for the "JSReduction" and "JSAllReduction" classes. The "JSTasks" use the created interface instead of the "JSReduction" and "JSAllReduction" concrete classes. This way, the direct dependencies between "JSTasks" and its product classes, "JSReduction" and "JSAllReduction," are broken.

Fig. 9 illustrates the class diagram of "JSTasks" and the new dependencies after applying refactoring to the factory method pattern. As shown in the figure, the design is now more comfortable to test due to reducing the number of 'use-consult' relationships after refactoring. Moreover, the newly introduced interface allows the software tester to use the test doubles (mock objects) [57] instead of concrete products,

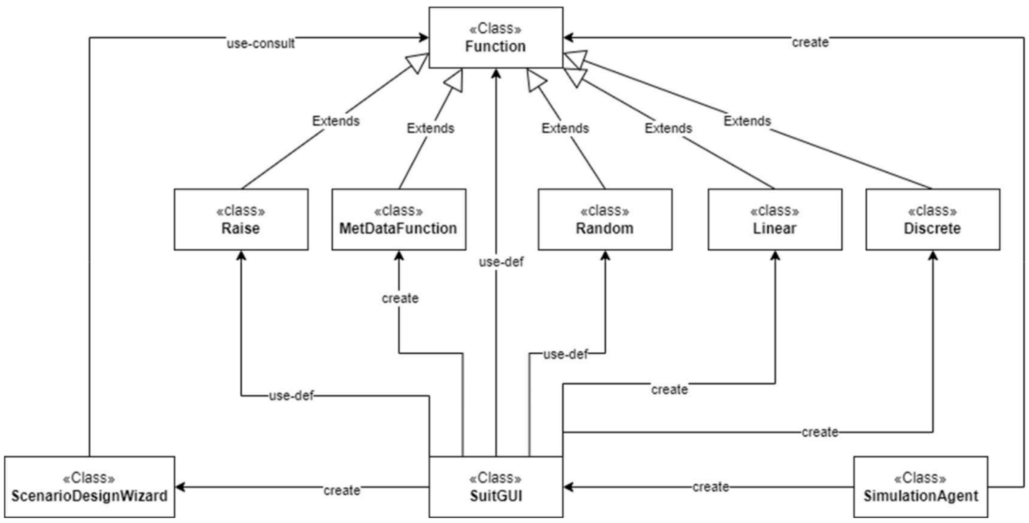


Fig. 10. Extended class diagram for the “SuiteGUI” class and its dependencies before refactoring.

facilitating unit testing of the creator class.

The complexity of interaction does not change in the above example since the number of descendents-path equals one for each class in a design. As the design becomes more complex, the effect of automated refactoring on the complexity of interactions and design testability enhancement increases. A more complicated design demonstrating the “SuitGUI” class and its dependencies is shown in Fig. 10. The “SuitGUI” class initializes instances from the “Linear”, “Random”, “Discrete”, “Raise”, and “MetDataFunction” classes, and also initializes an array of the “Function” class, shown in the following code snippet:

```
public class SuiteGUI extends JFrame {
    . . .
  private Function[] availableFunctions = new Function[7];

  private Linear linear = new Linear();

  private Random random1 = new Random();

  private Random random2 = new Random();

  private Discrete discrete = new Discrete();

  private Raise raise = new Raise();

  private MetDataFunction metData = new MetDataFunction();

  private Raise price = new Raise();
```

The above “Linear”, “Random”, “Discrete”, “Raise”, and “MetDataFunction” classes have been identified as product classes for the factory method refactoring by our approach. Tables 9 and 10 show the design metrics and testability values before and after refactoring to the factory method, respectively. An average improvement of 0.0232 (3.9 %) is observed in design testability after refactoring to the factory method. This improvement is due to decreases in the design metrics that deteriorate testability, such as FANIN and FANOUT, and increases in the metrics that enhance testability, such as NMO and NOI. The values changed after refactoring have been highlighted in Table 10. Fig. 11 shows the extended class diagram for the “SuiteGUI” class and its dependencies after refactoring. The concrete dependencies between the “SuiteGUI” class and the production class have been removed as a result of refactoring to the factory method pattern.

According to Baudry and Traon [1], the complexity of interactions corresponding to the “SuitGUI” class is computed as the sum of the complexity of each dependent path in the design. Using Baudry and Traon’s approach the value of “SuitGUI” interaction complexity equals 286 before applying any refactoring operations. The complexity of interactions corresponding to the “SuitGUI” class decrease to 235 after applying factory method refactoring.

5.6. Discussion and implications

Refactoring to creational patterns mainly factory method and dependency injection helps separate the responsibilities of creating and using objects at the design and source code level and increases testability. Refactoring to the factory method is prioritized over dependency injection where possible since it results in better design testability as

Table 9
Design metrics and testability values of the SuiteGUI class and its dependencies before refactoring.

Class	DIT	NOC	NOP	NIM	NMO	NOII	FANIN	FANOUT	CBO	NOI	Testability
SuiteGUI	2	0	2	0	0	0	51	69	4	0	0.6729
Discrete	2	0	2	13	3	0	12	15	1	1	0.6120
Function	1	5	2	0	0	0	53	17	1	1	0.7206
Linear	2	0	2	13	3	0	12	12	1	1	0.6414
MetDataFunction	2	0	2	13	3	0	5	9	1	1	0.6243
Raise	2	0	2	13	3	0	8	10	1	1	0.6364
Random	2	0	2	13	3	0	12	16	1	1	0.6122
SimulationAgent	2	0	1	0	0	0	141	187	19	0	0.3154
ScenarioDesignWizard	1	0	1	0	0	0	17	13	4	0	0.5151
Average	1.78	0.56	1.78	7.22	1.67	0.0	34.56	38.67	3.67	0.67	0.5945

Table 10
Design metrics and testability values of the SuiteGUI class and its dependencies after refactoring.

Class	DIT	NOC	NOP	NIM	NMO	NOII	FANIN	FANOUT	CBO	NOI	Testability
SuiteGUI	2	0	2	0	0	0	51	**62**	**6**	**2**	0.7401
Discrete	2	0	2	13	**5**	0	12	15	1	3	0.6259
Function	1	5	2	0	0	0	53	17	1	3	0.7307
Linear	2	0	2	13	**5**	0	12	12	1	3	0.6601
MetDataFunction	2	0	2	13	**5**	0	5	9	1	3	0.6484
Raise	2	0	2	13	**5**	0	**7**	10	1	3	0.6511
Random	2	0	2	13	**5**	0	**11**	16	1	3	0.6495
SimulationAgent	2	0	1	0	0	0	141	187	**21**	3	0.3384
ScenarioDesignWizard	1	0	1	0	0	0	17	13	4	0	0.5152
Average	1.78	0.56	1.78	7.22	2.78	0.0	34.33	37.89	4.11	2.56	0.6177

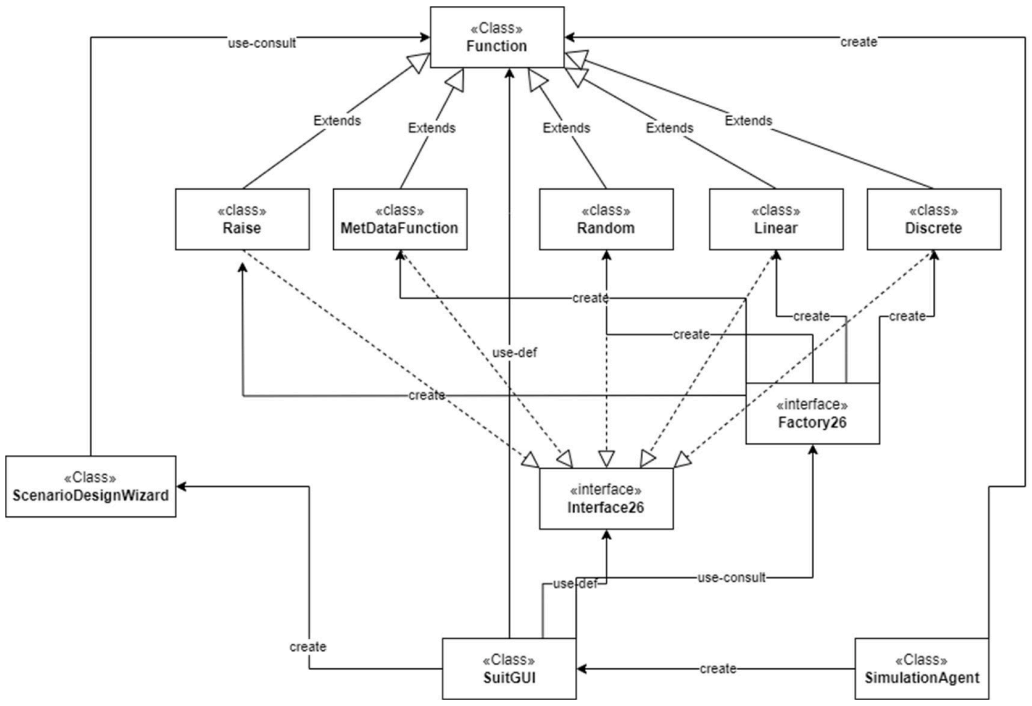


Fig. 11. Extended class diagram for the "SuiteGUI" class and its dependencies after refactoring.

well as modularity. Software developers and testers both benefit from factory method refactoring pattern as it makes the design more testable and flexible. Besides, there are various schemes for refactoring to factory methods and dependency injection patterns which may affect testability improvement. Developer preference may be considered when different schemes of refactoring to a pattern are applied. For instance, the update_class_diagram function in Algorithm 3 creates many static methods in the body of the creator class during refactoring to the factory method pattern. It generates a method for each constructor found in the products responsible for creating an object of that product. However, the standard scheme of the factory method often uses one static method to generate all types of products leveraging switch cases or if-then-else statements [12].

It would be beneficial to ask the developer to choose the desired refactoring scheme after identifying a refactoring opportunity. Likewise, the dependency injection pattern is typically performed in three schemes, including constructor injection, interface injection, and setter injection [13]. The proposed algorithm in this paper automates the constructor injection scheme. However, these three schemes are only slightly different and can be automated considering some minor changes in Algorithm 4. Constructor injection is a fundamental aspect of dependency injection frameworks like Spring [59], where it is often used to ensure that the necessary components of an application are properly provided and managed.

As a final note, the proposed testability prediction model uses 10 metrics at the design level to predict class testability. Whatever source code artifacts are available more software metrics can be extracted and computed for each class. Enlarging the size of the feature space improves the performance of the machine learning model in the case of our large dataset [60]. Therefore, it would be helpful if we had separate prediction models for working at the source code level when refactoring to improve design testability.

6. Threats to validity

Several factors mainly threaten the validity of the proposed approach in this paper. We outline these threats along with the actions performed to minimize them in terms of construction and internal and external validity.

6.1. *Threats to construction validity*

Regarding the validity of construction, the most important threat is how we define and measure testability at the design level. We consider two ingredients, the test effectiveness and test efficiency, mentioned by ISO/IEC 25010 standard [5] in defining our mathematical testability model. Such formulations have been successfully applied to measure and improve testability at the source code level [5,32]. The three most used test adequacy criteria, including mutation score, branch coverage, and statement coverage, are used to estimate test effectiveness. However, other criteria, such as prime path coverage [44], could be added to the model to improve its accuracy.

The proposed testability prediction model predicts the testability of each class and then averages the results to obtain design testability according to Eq. (2), which is an indirect measurement approach. It is possible to use Eq. (2) to label each class diagram with the corresponding testability and build a model that directly predicts design testability. However, this approach requires many projects (greater than 110 projects used in our method), which can be considered in future work. We use a set of ten design metrics to vectorize each class in the class diagram. These metrics have been used for different software measurement tasks [61] and are supported by most software analysis tools [62–64]. Nevertheless, other metrics can be added to the vector representation to improve the feature space of testability prediction models.

Finally, this paper considers the class diagram as the main important software artifact in the design phase. It is suitable for machine learning objectives since it can be extracted from the source code. Nevertheless, design artifacts such as the activity diagram and sequence diagram providing behavioral information about the program are beneficial in both the measurement and improvement processes.

6.2. *Threats to internal validity*

The most important threat to internal validity is the use of EvoSuite for test data generation and computing test information. EvoSuite works on the program's byte code to generate test data. Therefore, it considers the dependencies between classes and generates test data that maximize different coverage criteria. EvoSuite has achieved the highest overall score in the ninth unit testing competition at SBST 2021 [65]. However, other test data generation tools, such as JDart [66] and Randoop [67], are available for Java programs that can be used to improve the accuracy of the mathematical testability model.

EvoSuite generates test data with an evolutionary algorithm. It means that the test results are different each time due to the stochastic nature of the initial population and operators applied during the test data generation process. To minimize the randomness effect, Test data generation with EvoSuite was repeated five times on each class with default setting and different random seeds, and output parameters were averaged for use in the testability model. Finally, to mitigate the randomness of the machine learning models, the five-fold cross-validation method was applied during hyperparameter optimization, and the model with the highest R^2 score was selected for each regressor.

6.3. *Threats to external validity*

We implemented and evaluated our proposed approach to work on software systems written in Java language. The proposed extended class diagram, design metrics, and refactoring operation are applicable to other programming languages that support the object-oriented paradigm. However, further empirical investigation is required to evaluate the effectiveness of the proposed approach in other programming languages. Some languages support specific design features, which may result in a different distribution of design metrics used for testability precision. For instance, C++ supports multiple inheritances, which most presumably affects testability at the design level.

To improve testability, we mainly rely on the creational pattern responsible for creating and initials objects. There are different types of design patterns whose impact on testability can be studied with our proposed framework. Specifically, the behavioral and structural patterns that are responsible for assembling and manipulating objects may be applied to improve design testability.

7. Conclusion

Design testability is an important quality attribute that directly affects the cost and quality of testing software systems at the design level. The class diagram is the most-known design artifact denoting the relationships between different entities in the program. A flexible and non-ambiguous class diagram provides a testable design that not only facilitates integration testing but also results in a testable implementation. Due to the complexity of interactions in the class diagram of large codebases, machine learning techniques could be best applied to estimate class diagram testability and quantify testability at the design level.

The design testability can be best measured in terms of test effectiveness, qualifying by test adequacy criteria and test efficiency denoted by test suite size such that it realizes the standard testability definition. To avoid the actual testing of the program to find its design testability, a machine learning model is trained on a large number of software design artifacts, specifically, extended class diagrams, labeled with their testability. The proposed hybrid machine learning regression model in this paper predicts design testability based on ten design metrics with a mean squared error of 0.04 and an R^2 score of 0.58. The top four most important design metrics affecting class design testability are coupling between objects (CBO), number of overridden methods (NMO), number of incoming invocations (FANIN), and number of outgoing invocations (FANOUT). All metrics except NMO negatively correlate with design testability. It concludes that both the dependency (*e.g.*, CBO, FANIN, and FANOUT) and inheritance (*e.g.*, NMO) relationship metrics influence design testability.

Applying refactoring to creational patterns in a batch mode separates the concerns of object creation and usage and makes design flexible. The results of automatic identification and application to two well-known patterns, factory method and dependency injection (DI), indicate an 11.59 % improvement in design testability. During refactoring to these patterns, robust coupling connections between concrete classes are eliminated and replaced with more flexible couplings between concrete classes and interfaces. Consequently, design metrics like CBO and FANOUT decrease, while metrics like NMO and number of interfaces (NOI) increase after the refactoring. Refactoring to creational patterns, specifically dependency injection, also increases class constructor parameters for each Java project by an average of 15.95 %, which facilitates the creation of test doubles. In summary, using the proposed approach, developers can predict the testability of their design, automatically refactor classes in the design, and observe potential improvements.

There is a vast opportunity for future works on the direction of this paper. First of all, automated refactoring to other design patterns, including behavioral and structural patterns, is offered to improve the testability of the class diagram. Other types of refactoring to dependency injection, including interface injection and setter injection, must be automated besides the constructor injection proposed in this paper. Search-based refactoring methods can be applied in batch refactoring with three or more refactoring operations to maximize the design quality in particular testability. To this aim, the proposed testability prediction model is used as a fitness function of the search algorithm. Finally, other design artifacts, including but not limited to the activity diagram, sequence diagram, and collaboration diagram, can be considered in addition to the class diagram when measuring and improving design testability.

Data availability statement

The datasets generated and analyzed during the current study are available in Zenodo, https://doi.org/10.5281/zenodo.7948064.

Funding

This study has received no funding from any organization.

Ethical approval

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

CRediT authorship contribution statement

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

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.

References

[1] 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.
[2] B. Baudry, Y. Le Traon, Measuring design testability of a UML class diagram, Inf. Softw. Technol. 47 (13) (2005) 859–879, https://doi.org/10.1016/j.infsof.2005.01.006.
[3] R. Sharma, A. Saha, A systematic review of software testability measurement techniques, in: 2018 International Conference on Computing, Power and Communication Technologies (GUCON), IEEE, 2018, pp. 299–303, https://doi.org/10.1109/GUCON.2018.8675006.
[4] M. Harman, Refactoring as testability transformation, in: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 414–421.
[5] 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.
[6] 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.
[7] B. Baudry, Y. L. Traon, G. Sunye and J.-M. Jezequel, "Measuring and improving design patterns testability," Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No.03EX717), Sydney, NSW, Australia, 2003, pp. 50-59, 10.1109/METRIC.2003.1232455.
[8] J. Kerievsky, Refactoring to Patterns, Addison-Wesley Professional, 2005.
[9] S. Chiba and R. Ishikawa, "Aspect-oriented programming beyond dependency injection," 2005, pp. 121–143. doi: 10.1007/11531142_6.
[10] J. Bézivin, "Model driven engineering: an emerging technical space," 2006, pp. 36–64. doi: 10.1007/11877028_2.
[11] 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) 34.
[12] E. Gamma, R. Helm, R. Johnson, R.E. Johnson, J. Vlissides, Design patterns: Elements of Reusable Object-Oriented Software, others, Addison-Wesley Professional, 1995.
[13] S. Van Deursen, M. Seemann, Dependency injection: principles, practices, and Patterns, Manning Publications, 2019.
[14] M. Zakeri-Nasrabadi, S. Parsa, and S. Jafari, "Design for testability." 2024. doi: https://doi.org/10.5281/zenodo.11076642.
[15] 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.
[16] I. Bluemke, A. Malanowska, Software testing effort estimation and related problems, ACM. Comput. Surv. 54 (3) (2021) 1–38, https://doi.org/10.1145/3442694.
[17] T. Sharma, S. Georgiou, M. Kechagia, T.A. Ghaleb, F. Sarro, Investigating developers' perception on software testability and its effects, Empir. Softw. Eng. 28 (5) (2023) 120, https://doi.org/10.1007/s10664-023-10373-0.
[18] 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.
[19] M. Alenezi, Software testability: recovering from 2 decades of research, Int. J. Comput. Appl. 113 (7) (2015) 1–5, https://doi.org/10.5120/19835-1694.
[20] R.S. Freedman, Testability of software components, IEEE Trans. Softw. Eng. 17 (6) (1991) 553–564, https://doi.org/10.1109/32.87281.
[21] J.E. Payne, R.T. Alexander, C.D. Hutchinson, Design-for-testability for object-oriented software, Object Mag. 7 (5) (1997) 34–43.
[22] R.V. Binder, Design for testability in object-oriented systems, Commun. ACM 37 (9) (1994) 87–101, https://doi.org/10.1145/182987.184077.
[23] A. Baldini, P. Prinetto, G. Denaro, M. Pezzè, Design for testability for highly reconfigurable component-based systems, Electron. Notes. Theor. Comput. Sci. 82 (6) (2003) 199–208, https://doi.org/10.1016/S1571-0661(04)81038-7.
[24] L. Badri, M. Badri, F. Toure, An empirical analysis of lack of cohesion metrics for predicting testability of classes, Int. J. Softw. Eng. Appl. 5 (2) (2011) 69–85.
[25] M. Badri, F. Toure, Empirical analysis of object-oriented design metrics for predicting unit testing effort of classes, J. Softw. Eng. Appl. 05 (07) (2012) 513–526, https://doi.org/10.4236/jsea.2012.57060.
[26] 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.
[27] M. Bruntink, M.Sc. thesis, University van Amsterdam, 2003.
[28] M. Bruntink and A. van Deursen, "Predicting class testability using object-oriented metrics," Fourth IEEE International Workshop on Source Code Analysis and Manipulation, Chicago, IL, USA, 2004, pp. 136-145, 10.1109/SCAM.2004.16.
[29] 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.
[30] E. Ribeiro Rocha Camila AND Martins, "A strategy to improve component testability without source code," in *Testing of component-based systems and software quality*, V. A. N. D. M. J. A. N. D. R. R. A. N. D. S. F. Beydeda Ksami AND Gruhn, Ed., Bonn: Gesellschaft für Informatik e.V., 2004, pp. 47–62.
[31] C. Szyperski, Component software: beyond object-oriented programming. ACM Press books, ACM Press, 1997 [Online]. Available, https://books.google.com/books?id=ZKlQAAAAMAAJ.
[32] M. Zakeri-Nasrabadi, S. Parsa, Learning to predict test effectiveness, Int. J. Intell. Syst. (2021), https://doi.org/10.1002/int.22722.
[33] G. Grano, T.v. Titov, S. Panichella, H.C. Gall, Branch coverage prediction in automated testing, J. Softw. Evol. Process 31 (9) (2019), https://doi.org/10.1002/smr.2158.
[34] 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.
[35] W. Liu, Z. Hu, H. Liu, L. Yang, Automated pattern-directed refactoring for complex conditional statements, J. Cent. South. Univ. 21 (5) (2014) 1935–1945, https://doi.org/10.1007/s11771-014-2140-z.
[36] V.E. Zafeiris, S.H. Poulias, N.A. Diamantidis, E.A. Giakoumakis, Automated refactoring of super-class method invocations to the Template Method design pattern, Inf. Softw. Technol. 82 (2017) 19–35, https://doi.org/10.1016/j.infsof.2016.09.008.
[37] A. Ouni, M. Kessentini, M. Ó Cinnéide, H. Sahraoui, K. Deb, K. Inoue, MORE: a multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells, J. Softw. Evol. Process 29 (5) (2017) e1843.
[38] M.Ó. Cinnéide, Automated refactoring to introduce design patterns, in: Proceedings of the 22nd international conference on Software engineering - ICSE '00, ACM Press, New York, New York, USA, 2000, pp. 722–724, https://doi.org/10.1145/337180.337612.
[39] K. Deb, H. Jain, An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems

with box constraints, IEEE Trans. Evolut.Comput. 18 (4) (2014) 577–601, https://doi.org/10.1109/TEVC.2013.2281535.
[40] Y. Zhao, Y. Yang, Y. Zhou, Z. Ding, DEPICTER: a design-principle guided and heuristic-rule constrained software refactoring approach, IEEe Trans. Reliab. 71 (2) (2022) 698–715, https://doi.org/10.1109/TR.2022.3159851.
[41] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 4th edition, MIT Press, 2022. Accessed: Jul. 24, 2022. [Online]. Available, https://mitpress.mit.edu/books/introduction-algorithms-fourth-edition.
[42] T. Parr, "ANTLR (ANother Tool for Language Recognition)." Accessed: Jan. 10, 2022. [Online]. Available: https://www.antlr.org.
[43] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: principles, techniques, and Tools, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
[44] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, Cambridge, 2016, https://doi.org/10.1017/9781316771273.
[45] 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.
[46] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using EvoSuite, ACM Trans. Softw. Eng. Methodol. 24 (2) (2014) 1–42, https://doi.org/10.1145/2685612.
[47] A. Nanthaamornphong, J.C. Carver, Test-driven development in scientific software: a survey, Softw. Qual. J. 25 (2) (2017) 343–372, https://doi.org/10.1007/s11219-015-9292-4.
[48] A. Arcuri, J. Campos, G. Fraser, Unit test generation during software development: evoSuite plugins for Maven, IntelliJ and Jenkins, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), 2016, pp. 401–408, https://doi.org/10.1109/ICST.2016.44.
[49] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016 [Online]. Available, http://www.deeplearningbook.org/.
[50] L. Breiman, Random forests, Mach. Learn. 45 (1) (2001) 5–32, https://doi.org/10.1023/A:1010933404324.
[51] 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.
[52] 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.
[53] 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.
[54] R.C. Martin, *Clean architecture: A craftsman's Guide to Software Structure and Design*. in Martin, Robert C. Prentice Hall, 2018. [Online]. Available: https://books.google.az/books?id=8ngAkAEACAAJ.
[55] G. Suryanarayana, G. Samarthyam, T. Sharma, Refactoring For Software Design smells: Managing Technical Debt, 1st ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2014.
[56] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, L. Teixeira, Detecting overly strong preconditions in refactoring engines, IEEE Trans. Softw. Eng. 44 (5) (2018) 429–452, https://doi.org/10.1109/TSE.2017.2693982.
[57] 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.
[58] M. Zakeri-Nasrabadi, "Testability prediction dataset: design level." Accessed: Aug. 07, 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7948064.
[59] "Spring | Home." Accessed: Apr. 30, 2024. [Online]. Available: https://spring.io/.
[60] P. Sondhi, Feature construction methods: a survey, sifaka. cs. uiuc. edu 69 (2009) 70–71.
[61] A.S. Nuñez-Varela, H.G. Pérez-Gonzalez, F.E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: a systematic mapping study, J. Syst. Softw. 128 (2017) 164–197, https://doi.org/10.1016/j.jss.2017.03.044.
[62] SciTools, "Understand." Accessed: Sep. 11, 2020. [Online]. Available: https://www.scitools.com/.
[63] "PMD: an extensible cross-language static code analyzer." Accessed: Sep. 21, 2021. [Online]. Available: https://pmd.github.io/.
[64] R. Ferenc, P. Siket, and M. Schneider, "OpenStaticAnalyzer," University of Szeged. Accessed: Jun. 23, 2021. [Online]. Available: https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer.
[65] 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.
[66] K. Luckow et al., "JDart: a dynamic symbolic analysis framework," 2016, pp. 442–459. doi: 10.1007/978-3-662-49674-9_26.
[67] C. Pacheco and M.D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion,* Montreal, Canada, 2007, pp. 815–816.