symmetry

MDPI

Article

A Feature-Based Method for Detecting Design Patterns in Source Code

Mariam Kouli and Abbas Rasoolzadegan *

Department of Computer Engineering, Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad 9177948974, Iran; kouli.mariam@mail.um.ac.ir
* Correspondence: rasoolzadegan@um.ac.ir

Abstract: Design patterns are common solutions to existing issues in software engineering. In recent decades, design patterns have been researched intensively because they increase the quality factors of software systems such as flexibility, maintainability, and reusability. Design pattern detection refers to the determination of the symmetry between a code fragment and the definition of a design pattern. One of the major challenges in design pattern detection is how to obtain accurate information about the design patterns used in the software system due to the existence of different design pattern variants. Increasing the number of design pattern variants covered by a detection method is one of the main factors that increase its accuracy. In this paper, a step toward solving this challenge was taken by proposing a new feature-based method that builds on concrete definitions of existing design pattern variants and supports the definition and detection of new variants. In this proposed method, the needed features are extracted from the signatures of the design patterns. This method was applied to the 23 Gang of Four (GoF) design patterns and evaluated using four open-source Java projects. Afterward, it was compared with some previous methods using automatically generated testbeds. The experimental results demonstrated that the proposed method has better performance in terms of precision and recall compared to the other methods.

Keywords: design pattern variants; feature-based pattern detection; design patterns' signature analysis; reverse engineering; software quality

Citation: Kouli, M.; Rasoolzadegan, A. A Feature-Based Method for Detecting Design Patterns in Source Code. *Symmetry* **2022**, *14*, 1491. https://doi.org/10.3390/sym14071491

Academic Editor: Kuo-Hui Yeh

Received: 12 June 2022
Accepted: 19 July 2022
Published: 21 July 2022

1. Introduction

Designing reusable object-oriented (OO) software systems is not a straightforward process. The designed system should be symmetric to the problem and generalizable to similar problems at the same time. Design patterns represent general solutions that are symmetric to specific problems in certain contexts. Since the effectiveness of these solutions has been proven, their use has increased the speed of the system development process [1]. To ease the implementation of design patterns, they have been informally defined by means of classes and relations between them using a modeling language such as the unified modeling language (UML).

The appropriate use of design patterns can have a positive impact on many quality factors of a software system [2–4], and developers tend to use design patterns to build more flexible software systems [5]. Moreover, design patterns increase the reusability of the software system [6] and increase its maintainability by providing developers with essential information about the underlying structure of the system [7]. Therefore, design pattern detection is important to maintain and modify systems with inadequate documentation.

One of the major challenges in design pattern detection is to accurately identify the symmetry between the definition of a design pattern and the design pattern instances in software systems. By considering more variants, detection methods become more capable of determining the symmetry between the design pattern instances that particularly apply to these variants and the design pattern definition. Therefore, increasing the number of

design pattern variants covered by a detection method plays an important role in increasing its accuracy [8–11].

Thus far, several detection methods with different approaches have been claimed to be successful in detecting design pattern variants [3,9,12,13]. However, to the best of the authors' knowledge, none of these methods have provided detailed information about the variants covered or detected. Such information is essential to understand important aspects of the variant handling process of the detection method. Later, in Section 5, some examples of design pattern variants that failed to be detected by existing detection methods are illustrated.

In this paper, a solution to address the abovementioned problem is proposed, which is referred to as E_x-DPD_{FE} (Extended Design Pattern Detection based on Feature Extraction) and which builds on the concrete definitions of the existing design pattern variants. Although several variants of the GoF (Gang of Four (Gamma, Helm, Johnson, and Vlissides)) [1] design patterns have been discussed in the literature [14–17], the catalog of Rasool et al. [18] was considered in this research since it is the only catalog of the variants of the GoF design patterns proposed thus far.

E_x-DPD_{FE} consists of two main phases: the feature-based design pattern definition phase and the design pattern detection phase. During the first phase, a set of structural and behavioral features that represent the main roles of a design pattern are extracted from its signature. After that, the extracted features are organized in a structured textual form and added to a vocabulary of reusable features that represent design patterns. The design pattern variants are considered in this phase as well. During the second phase, the detection process tries to determine the symmetry between a code fragment of the input system and the design pattern textual definition obtained during the first phase. To accomplish this, the detection process starts by analyzing the feature-based textual definition of the design pattern. Then, it looks for classes that satisfy the features that define a specific role in the source code of the input system. Finally, the relations between the candidate role classes are validated based on the signature of the design pattern, and only role classes that are related using the specified relations are considered as an instance of that pattern.

E_x-DPD_{FE} uses structural and static behavioral analysis to detect design patterns. Moreover, E_x-DPD_{FE} reduces the number of false negative instances and achieves high accuracy by considering the different variants of the design pattern in the process of the feature-based design pattern definition. Thus, the features are specified appropriately to distinguish the design patterns and cover their variants.

E_x-DPD_{FE} detects the design patterns using a new set of features that is extracted from the signatures of the design patterns, and it can detect more design pattern variants than other detection methods since these variants are considered in the feature extraction process. Therefore, E_x-DPD_{FE} is important because it takes a step toward increasing the accuracy of the design pattern detection methods. Moreover, it tries to address some of the most important challenges faced by the feature-based methods proposed thus far. The considered challenges include choosing the appropriate set of features that define a specific design pattern and determining the proper threshold values for the considered features.

In summary, the main contributions of this paper are as follows:

- A new set of features is proposed that is based on the signatures of the design patterns. By using these features, we try to take a step toward solving some of the problems faced by the feature-based detection methods proposed thus far.
- A new design pattern detection method is proposed that builds on concrete definitions of the design pattern variants.
- The proposed method can detect more design pattern variants than other methods with acceptable accuracy.

Since the proposed method belongs to the feature-based approach, it preserves the main advantages of the methods within this category, such as their low complexity, pattern independence, and distinction ability. On the other hand, the proposed method increases the accuracy of the feature-based methods by using an extended feature set,

which is extracted from the signatures of the design patterns; this method obtains more accurate detection results than other methods with different approaches by considering more variants.

The remainder of this paper is organized as follows. Section 2 presents a literature review of related works, and E_x-DPD_{FE} is explained in Section 3. In Section 4, the implementation of E_x-DPD_{FE} is discussed. In Section 5, E_x-DPD_{FE} is evaluated and the obtained results are compared to other previous methods. Threats to the validity are explained in Section 6. Finally, the conclusion and future work directions are presented in Section 7.

2. Related Work

In recent decades, many design pattern detection methods have been proposed. The main purpose of these methods is to identify a symmetry between the design patterns and their instances in software systems. In the following, the previous methods are highlighted and the proposed method is compared to those previous ones.

2.1. Detection Approaches

Detection methods can differ based on their analysis type, detection process, input type, and detection results. However, they are generally divided into two main categories based on their analysis type [9]: structural and behavioral, as illustrated in Figure 1. Structural analysis methods detect the design patterns based on their structural characteristics, such as class hierarchies, associations, and modifiers of methods and classes, whereas behavioral methods consider the behavior of the system by using static and dynamic techniques. While the static analysis techniques depend on studying the source code of the system to determine its behavioral characteristics, such as method calls and delegations, the dynamic analysis techniques consider the runtime behavior of the system. Behavioral analysis is essential to distinguish the patterns that expose similar structural characteristics [11]. In addition, semantic analysis, which focuses on meanings, can be used to complement the structural and behavioral analysis and improve the detection results. It is worth mentioning here that a detection method can use one or more analysis types.

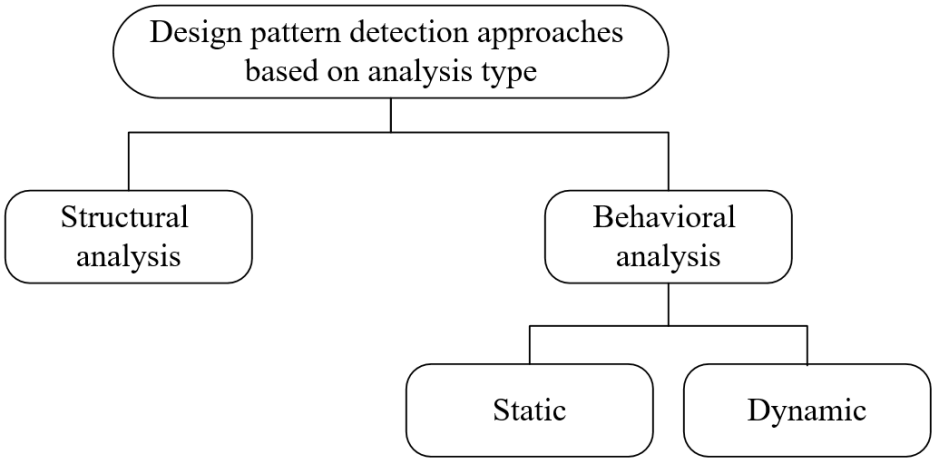


Figure 1. Design pattern detection approaches based on their analysis type.

In addition to the above, detection methods are classified into nine main categories based on their detection process [9,19,20]. In the following, these categories are summarized.

Design pattern detection using structural analysis: This approach depends on transforming both the system under study and the design patterns into intermediate structures such as graphs and matrices. After that, the detection algorithm looks within the system structure for structures that represent specific design patterns [9,13,21–29]. Methods that depend on structural analysis have high accuracy [10,11]. On the other hand, these methods cannot distinguish design patterns with similar structures [9,22]. For this reason, using

behavioral analysis along with structural analysis increases the accuracy of the detection results [7,30–34].

Design pattern detection using database queries: In the query-based approach, the source code of the input system is transformed into an intermediate structure such as AST, ASG, UML, and XMI. After that, database queries are applied to the intermediate structure to extract the information needed to detect the design patterns [35–41]. The accuracy of these methods depends on the used database and the information represented by the intermediate structure [9–11].

Parsing-based approach: In this approach, the source code of the system under study is transformed into scalable vector graphics (SVG). On the other hand, design patterns are represented using a visual language. The visual representation of the design patterns is parsed and mapped to the representation of the system [42–45]. Generally, parsing-based methods demonstrate good detection results, but they cannot detect behavioral design patterns [11,32].

Reasoning-based approach: This approach contains two sub-categories: fuzzy reasoning and logical reasoning. In the fuzzy reasoning methods, the design patterns are represented using fuzzy-reasoning nets that represent the conditions that must be satisfied by a micro-architecture to be considered as a design pattern [8]. These methods usually demonstrate low precision [9]. Meanwhile, in the logical reasoning methods, detection conditions are represented, and design patterns are detected considering these conditions [46–48]. These methods are mostly incapable of detecting the design pattern instances that are slightly different from the specified conditions.

Constraint satisfaction approach: This approach considers the detection problem as a constraint satisfaction problem. For this purpose, the design patterns are represented as constraint systems in which a variable represents a role of a design pattern, and the constraints between these variables represent the relations between their corresponding roles [49–51]. Commonly, these methods demonstrate high recall and low precision, and they show high complexity [9,11].

Formal approach: In this approach, mathematical-based and logical-based techniques are used [52–55]. This approach is not necessarily more accurate than other approaches. At the same time, it has high complexity and it is only capable of detecting design patterns with a specific number of roles in most cases [9].

Feature-based approach: The feature-based approach depends on defining a set of features such as DIT (Depth of the Inheritance Tree), LCOM (Lack of Cohesion in Methods), the number of associations, and the types of attributes. After that, the values of these features are calculated for the classes of the input system and compared with the definitions of the design patterns [56–62]. Feature-based methods have low computational complexity [10,22], but the methods proposed within this category thus far have only considered general OO features to detect the design patterns, except for the method of [56]. For this reason, the accuracy of their detection results is generally low [9,11,22].

Machine learning approach: Machine learning is a multi-disciplinary field that has been recently used for design pattern detection [63]. Learning-based methods map the design pattern detection problem to a learning problem. These methods use supervised and unsupervised learning techniques to detect the design patterns [20,64–77]. The accuracy of these methods depends on the training dataset. Furthermore, they can only detect the design pattern variants available in the training set [22].

Miscellaneous approaches: There are several approaches that cannot be categorized under one of the categories mentioned above, such as data mining [78] and concept analysis [12,79]. These approaches are used basically to improve the results of other approaches.

2.2. Variant Detection

Different design pattern detection methods have tried to address the problem of design pattern variants. One of the recent works is the method proposed in [12], which covers the design pattern variants by determining the necessity of the roles and their

relations in the pattern's structure. Therefore, all the design pattern instances that only have the necessary roles and relations are considered as candidate instances. After that, the candidate instances are verified manually, considering specific conditions. In another recent method [2], the authors concentrated on the implementation variants of the design patterns. They considered the structural, behavioral, and semantic characteristics of the design patterns and applied static analysis and inference techniques in the detection process. In [9], the authors addressed the variant problem by concentrating on the main body of a design pattern, which is shared among its different variants. The method proposed in [73] uses machine learning for design pattern detection and it can detect the design pattern variants available in the training dataset. The authors of [80] used fine-grained rules to capture the structural aspects of the design patterns; they detected the design pattern variants by labeling specific rules as optional. A graph matching method to detect the design patterns was proposed in [76]. This method addresses the variant problem by considering the partial occurrences of the design patterns. In [15], a metamodel for formal design pattern specification was proposed. This metamodel covers the design pattern variants by relaxing the conditions on the design pattern elements and introducing terms such as *Optional*, *In case of*, and *Alternatives*. SSA [13] represents the structural characteristics of the input system using matrices and detects the implementation variants of the design patterns by considering the transitive relations.

2.3. Concluding Remarks

By analyzing the existing detection methods of different approaches (please refer to sqlab:A_detailed_comparison_between_Ex_DPDfe_and_other_detection_methods.pdf) (accessed on 11 June 2022), it has been noticed that the detection methods that have tried to address the variant problem thus far concentrate on a limited number of variants, more specifically, structural variants. Moreover, these methods do not clarify the characteristics of the variants considered and they have been evaluated using systems with certain types of variants, which threatens their evaluation credibility.

In this research, a step forward to solve the above challenges was taken by proposing a new feature-based method, which will be described in detail within the next sections. The proposed method covers more design pattern variants compared to other methods by considering both structural and behavioral characteristics of these variants. In addition, it is based on the concrete definitions of the existing variants and supports new variants. Finally, the proposed method was evaluated using software systems with different variants and compared to other existing methods.

This method employs features because they facilitate the automation of the detection process, and they can capture both structural and behavioral characteristics of the design patterns with acceptable accuracy and computational complexity (please refer to sqlab:A_detailed_comparison_between_Ex_DPDfe_and_other_detection_methods.pdf) (accessed on 11 June 2022). Moreover, the proposed method tries to take a step toward improving the feature-based methods proposed thus far by addressing some of the common issues related to the features used to define the design patterns and their threshold values.

The idea of the proposed method is built on the method of Guéhéneuc [61], which was chosen since it has shown more strengths and exposed fewer limitations compared to other feature-based methods (please refer to sqlab:A_comparison_between_feature_based_methods.pdf) (accessed on 11 June 2022).

The signatures used to detect the design patterns are one of the main factors that affect the accuracy of the detection results. Therefore, several design pattern signatures have been proposed in the literature [8,55,61,81–84]. However, in most cases, these signatures do not cover the design pattern variants. To address this problem, the structural and behavioral signatures proposed in [9] were applied in the proposed method since they demonstrate higher accuracy in comparison with other signatures and allow us to cover more design pattern variants.

3. E_x-DPD$_{FE}$

E_x-DPD$_{FE}$ contains two main phases: the feature-based design pattern definition phase and the design pattern detection phase, as shown in Figure 2. These two phases are explained here using the Adapter design pattern (the Object Adapter) as a running example.

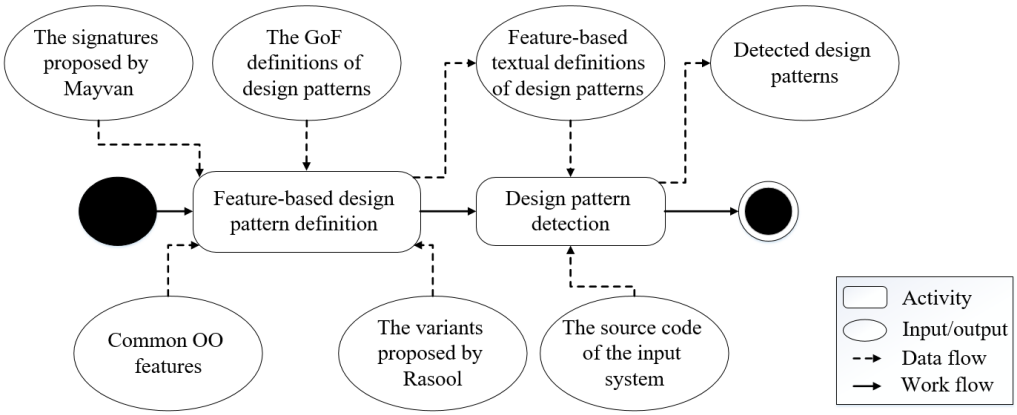


Figure 2. The main phases of E_x-DPD$_{FE}$.

3.1. The First Phase: Feature-Based Design Pattern Definition

In this phase, each design pattern is represented using a structured textual form that contains a set of roles that satisfy specific features. To cover all the GoF categories of design patterns (creational, structural, and behavioral), two types of features are deployed: structural and behavioral. Structural features refer to the features that can be extracted from the structural signature (main body) of a design pattern and include features such as the class abstraction level, inheritance, and association. Meanwhile, behavioral features describe the features that are extracted from the behavioral signature of the design pattern. Some examples of behavioral features include method call, delegation, and method return type.

This phase consists of two steps that are carried out manually, as shown in Figure 3. These two steps are described as the following.

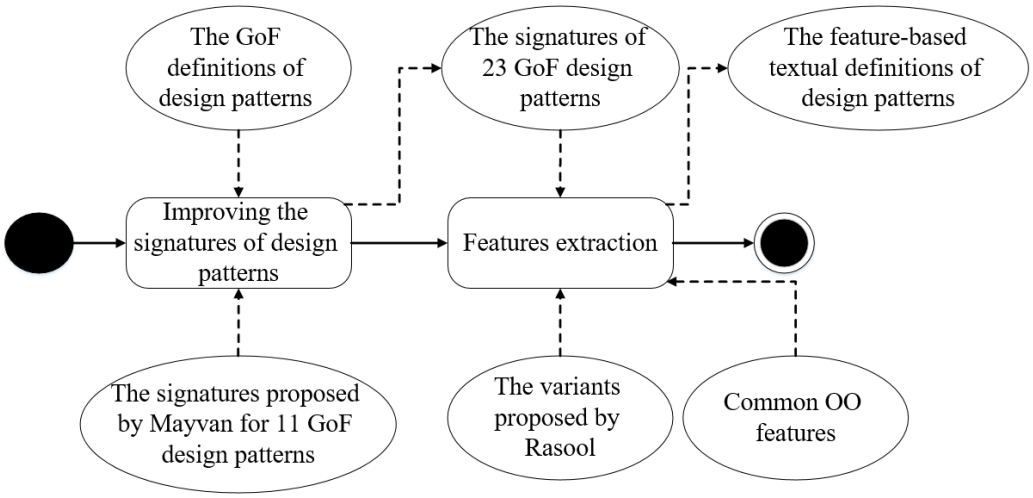


Figure 3. The steps of the feature-based design pattern definition phase.

3.1.1. The First Step: Improving the Signatures of the Design Patterns

The signature of each design pattern is defined based on the design pattern definitions proposed in [1]. The signature definition process is usually carried out manually [8,9,55,81–84]. Here, the signatures proposed in [9] were considered because they facilitate the detection of the design pattern variants by concentrating on the main body and behavioral characteristics

of the design patterns. Additionally, these signatures were partially improved to cover the 23 GoF design patterns since only 11 design patterns were considered in [9].

According to [9], the signature of a design pattern consists of two parts: structural and behavioral. The main body of the design pattern, which represents its structural signature, consists of its main roles. The behavioral signature describes the relations between these roles. Hence, the structural signatures are defined based on the class diagrams of the design patterns, whereas the behavioral signatures are extracted from their sequence diagrams. Figure 4 demonstrates the structural signature (main body) of the Adapter design pattern, and Figure 5 represents its behavioral signature. The signatures of some of the most commonly used GoF design patterns are illustrated in Appendix A; the signatures of the remaining GoF design patterns are available online at sqlab:The-signatures-of-some-GoF-patterns.pdf (accessed on 11 June 2022).

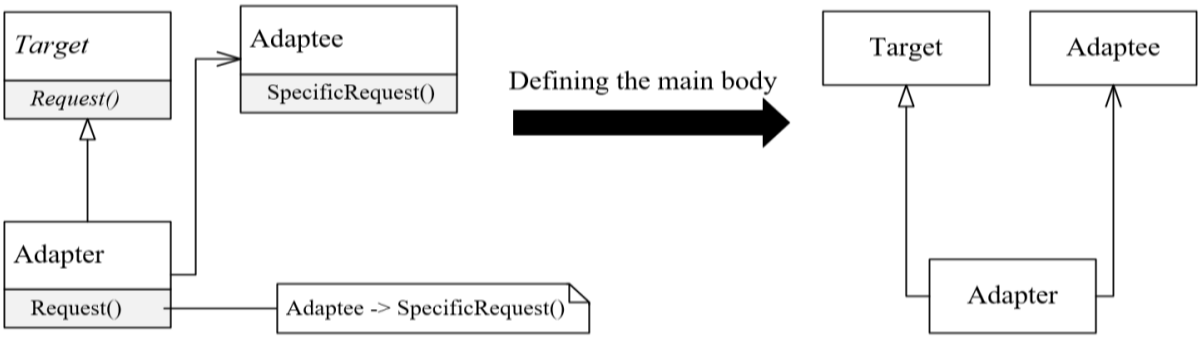


Figure 4. The structural signature (main body) of the Object Adapter design pattern.

Object Adapter	
Behavioral characteristics	A method in the Adapter that ***overrides*** a method in the Target ***calls*** a method in the Adaptee

Figure 5. The behavioral signature of the Object Adapter design pattern.

3.1.2. The Second Step: Feature Extraction

In this step, features are extracted considering the OO features that have been widely used in the literature [85], the signatures defined in the previous step, and the variants proposed in [18].

By analyzing the feature-based methods proposed thus far, it was noticed that adopting features with fixed ranges of values that are determined using specific systems is one of the most significant limitations of these methods. The determined values affect the accuracy of these methods when applied to new systems [57]. Furthermore, the relations between the OO features used thus far and the design patterns have been obtained based on specific software systems [86]. These systems take advantage of the characteristics of the programming languages used for their implementation. For example, some programming languages, such as Java and C++, provide a method for cloning objects, which can be used for implementing the Prototype design pattern. In contrast, other programming languages do not provide such a method. As a result, there is no guarantee that the generalization of the obtained relations to other systems with different programming languages would generate similar detection results [87]. Moreover, selecting the appropriate set of features that represent a design pattern accurately is an open issue [86].

Considering the information mentioned above, the main advantage of E_x-DPD_{FE} over other detection methods, especially these feature-based methods, lies in exploiting the signature of the design pattern to extract the needed features based on the general OO features. Therefore, the extraction of new features eliminates the need for comparing

the calculated values with fixed thresholds, which compromises the accuracy of feature-based methods, as discussed earlier. For example, instead of identifying the Adapter design pattern by measuring features such as Coupling Between Object classes (CBO) and comparing its value with a predefined threshold, which is how the feature-based methods commonly identify the Adapter [20,56,61,69,71], E_x-DPD_{FE} studies the existence of a more specific feature, namely, *method call to an associated class by an overriding method*. Moreover, the relations between the new features and the design patterns are clear and independent of the software system because, unlike the OO features that are commonly used to describe OO software systems, these features are pattern oriented. Adopting such features provides a step toward increasing the accuracy of the feature-based methods while preserving their main advantages, such as time efficiency. Moreover, the proposed features enable the detection of the different categories of design patterns since they are based on the structural and behavioral aspects specified in the signatures of these patterns.

On the other hand, the defined features cover as many design pattern variants as possible since they are considered in the feature extraction process. To the best of the authors' knowledge, the catalog of Rasool et al. [18] is the only catalog on the design patterns' variants proposed thus far in which a total of 107 variants of the 23 GoF design patterns has been presented. For this reason, the variants proposed in [18] were considered for conducting this research. It is notable that compound patterns were out of the scope of this research. Moreover, some variants differ significantly in their structural and behavioral characteristics from the basic form of the design pattern. For these reasons, a total of 58 variants of those specified in [18] can be detected thus far using E_x-DPD_{FE}. The remaining variants can be detected independently after defining the appropriate features.

The extracted features are then organized in a structured textual form that represents a design pattern. Using this form to represent the design patterns is another advantage of E_x-DPD_{FE} since it enables the developers to define new design patterns other than those considered in this research and to apply E_x-DPD_{FE} to them regardless of their implementation aspects.

Considering the main body of the Adapter design pattern shown in Figure 4, it has been noticed that the Adapter class inherits from the Target class. Thus, being a *subclass* is the first feature that defines the Adapter role ($FAR1_1$) (the first Feature of the Adapter's Role1). In addition, the Adapter class is associated with the Adaptee class. As a result, having a *one-to-one association to another class* is the second feature that defines the Adapter role ($FAR1_2$). No more structural features can be extracted from the structural signature of the Adapter design pattern to define the Adapter role. Therefore, studying its behavioral signature is required to define its behavioral features. Considering the behavioral signature of the Adapter design pattern shown in Figure 5, two features have been defined: *overriding a method of its superclass* and *calling a method of an associated class*. The latest two features have been merged to achieve a more accurate definition, which reduces the number of false positive instances. Therefore, the third feature that defines the Adapter is having a *method call to an associated class by an overriding method* ($FAR1_3$). A similar analysis is performed to extract the features that define the remaining roles. The extracted features do not require any additional modifications since they cover one of the variants of the Adapter design pattern (the Pluggable Adapter) [18]. The other variant (the Two-Ways Adapter) is related to the Class Adapter, which is analyzed independently. Figure 6c shows the resulting feature-based textual definition of the Adapter design pattern. The proposed feature-based textual definitions of some of the most commonly used GoF design patterns are shown in Appendix B; the feature-based textual definitions of the 23 GoF design patterns are available online at sqlab:The-definitions-of-GoF-design-patterns.zip (accessed on 11 June 2022).

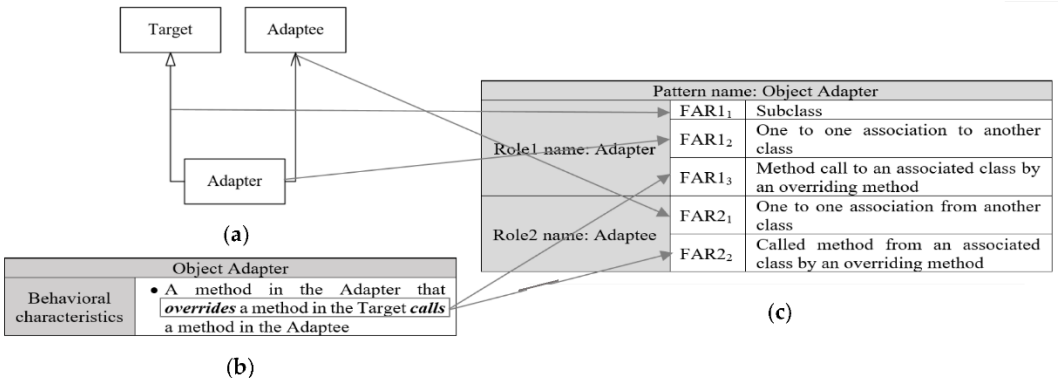


Figure 6. Extracting the features that represent the Object Adapter design pattern from its signature: (**a**) the structural signature of the Object Adapter; (**b**) the behavioral signature of the Object Adapter; (**c**) the feature-based textual definition of the Object Adapter.

Newly defined features are then compared with those previously defined to avoid redundancy. Furthermore, in this step, any feature with two different names is corrected, and any two features with the same name are recognized. After that, the new features are added to a vocabulary of reusable features that define the design patterns. The proposed features consist of one or more of the structural and behavioral OO features summarized in Table 1. Moreover, the basic OO features used to identify some of the most commonly used design patterns are presented in Table 2.

On the other hand, Figure 7 depicts the extracted features that define the different cases of association for the design patterns that use association in their structures and how these features differ to provide an accurate representation for each design pattern. Similarly, Figure 8 illustrates the extracted features that define the different cases of method call.

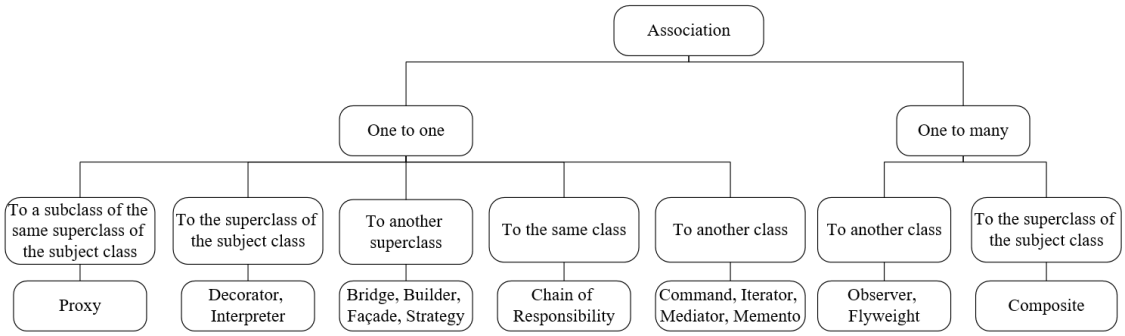


Figure 7. The extracted features that define the different cases of association for different design patterns.

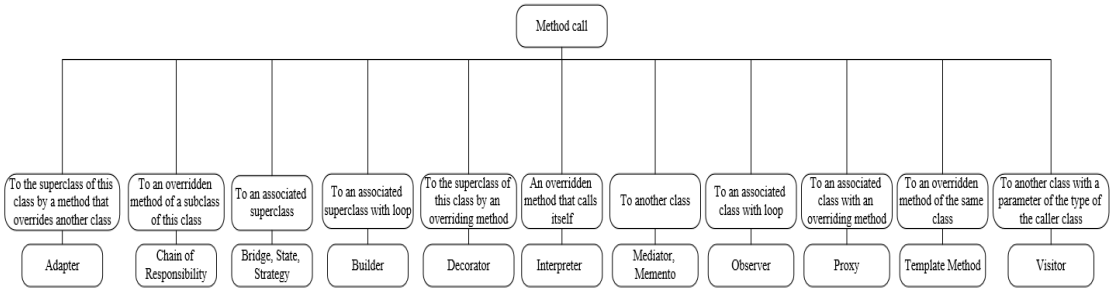


Figure 8. The extracted features that define the different cases of method call for different design patterns.

Table 1. The basic OO features used for defining the new features.

Characteristics / Feature Name	Explanation	Used Values	Type
Class abstraction level	Refers to the type of a class (interface, abstract, or concrete class)	• Interface (abstract class) • Subclass • Superclass • Not superclass	Structural
Constructor visibility	Refers to the accessibility of the constructor of a class (private, protected, or public)	• No public constructor	
Method visibility	Refers to the accessibility of a method (private, protected, or public)	• Public method	
Method modifier	Refers to the modifiers of a method (static, final, synchronized, or abstract)	• Static method	
Inheritance	Refers to a class that exists within an inheritance hierarchy	• Superclass • Subclass • Inheritance • Implementation	
Association	Refers to a class that has a reference to another class	• (One to one/one to many) association (to/from)	
Overriding	Refers to a method that overrides (implements) a method of its superclass	• Overriding (implementing) • Overridden (implemented)	
Method return type	Refers to a method that returns an instance of the same or another class	• Returns a new instance of (the same/another) class • Returns a copy of the same class • An instance returned from	Behavioral
Method call	Refers to a method that calls a method of the same or another class	• Method call to • Called method from	
Dependency	Refers to a class that has a method with a parameter of the type of another class	• Dependency (to/from)	
Delegation	Refers to a method that delegates to another method	• Delegation (with loop) (to/from)	
Composition	Refers to an object that only exists within another object	• Constructor with a parameter of the type of another class	

Table 2. The basic OO features used for defining some design patterns.

Feature Name \ Design Pattern	Abstract Factory/ Factory Method	Singleton	Decorator	Strategy/State	Template Method	Visitor	Adapter	Composite	Observer	Command	Proxy
Class abstraction level	✓	×	✓	✓	✓	✓	✓	✓	×	✓	✓
Constructor visibility	×	✓	×	×	×	×	×	×	×	×	×
Method modifier	×	✓	×	×	×	×	×	×	×	×	×
Method return type	✓	✓	×	×	×	×	×	×	×	×	×
Method call	×	×	✓	✓	✓	✓	✓	✓	✓	✓	✓
Inheritance	×	×	×	✓	×	✓	×	×	×	×	×
Dependency	×	×	×	×	×	✓	×	×	×	×	×
Association	×	×	✓	✓	×	✓	✓	✓	✓	✓	✓
Overriding	✓	×	✓	×	×	×	✓	✓	×	✓	×
Delegation	×	×	×	×	×	×	×	✓	×	×	×
Composition	×	×	×	×	×	×	×	×	×	✓	×

Although these features are used to define the GoF design patterns, including their variants, they can be used by the design pattern developers to define other design patterns. Feature reusability refers to the use of different combinations of features to properly define different design patterns, which does not threaten the accuracy of these features when extended to new design patterns. However, for some design patterns, it may be required to define new features using the illustrated definition process. It is remarkable that the results of the detection process depend on the accuracy of the signatures and the features extracted from them.

3.2. The Second Phase: Design Pattern Detection

The design pattern detection phase, which is depicted in Figure 9, focuses on the determination of the symmetry between a code fragment of the input system and the feature-based design pattern definition resulting from the previous phase. This phase consists of the following steps:

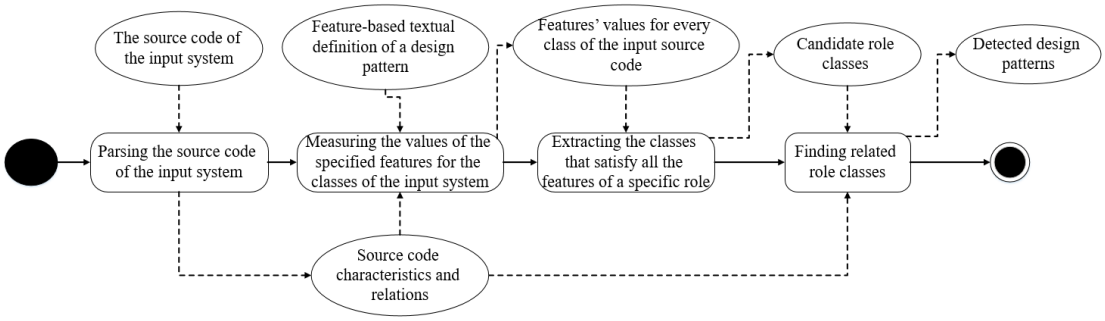


Figure 9. The steps of the design pattern detection phase.

- **Parsing the source code of the input system:** To measure behavioral features, the source code of the input system is parsed, and the needed characteristics are stored in a database. Moreover, the different relations between the classes of the input system are extracted and stored in the database. To be time-efficient, source code parsing is performed only once for all the design patterns.
- **Measuring the values of the specified features for the classes of the input system:** The feature-based textual definition file resulting from the definition phase is parsed,

and the values of the specified features are measured for the classes of the input system. Every class is associated with a flag in the database. This flag refers to the number of features satisfied by its corresponding class for one role. This step is described in detail in Section 4.

- **Extracting the classes that satisfy all the features of a specific role:** System classes that satisfy all the features that define a role in the design pattern are considered as candidate role classes.
- **Finding related role classes:** To eliminate possible false positive instances, only the candidate role classes that are related by utilizing the relations specified in the signature of the design pattern are considered as a design pattern instance.

Considering the Adapter design pattern example, the detection process starts from the first role (the Adapter). It then modifies the flags of the system classes that satisfy one or more of the defined features. Figure 10 represents a code fragment of the class DrawApplet in JHotDraw v5.1 that is identified by E_x-DPD_{FE} as an Adapter role. A field of the type ToolButton, named fSelectedToolButton, is declared, which means that a *one-to-one association* from DrawApplet to ToolButton exists. In addition, the method PaletteUserOver() in DrawApplet, which *overrides* the method PaletteUserOver() in PaletteListener, *calls* the method name() of the class ToolButton for the field fSelectedToolButton. As a result, this class is considered as a candidate Adapter. In the same way, since the class ToolButton satisfies the features defined for the Adaptee role, it is considered as a candidate Adaptee. Because an association between these two classes exists, they form an instance of the Adapter design pattern.

```
public class DrawApplet implements PaletteListener {
    transient private ToolButton fSelectedToolButton;

    public void paletteUserOver(PaletteButton button, boolean inside) {

            showStatus(fSelectedToolButton.name());
    }
```

Figure 10. A code fragment of the class DrawApplet in JHotDraw v5.1, which represents the Adapter role of the Adapter design pattern.

3.3. Implementation Cost

The first phase of E_x-DPD_{FE} (the definition phase) was performed manually, as illustrated in Section 3.1. A team of three master's degree and Ph.D. degree students was responsible for improving the signatures of the design patterns; the same team conducted the feature extraction process. The complexity of this phase depends on the design pattern under study and its variants. Some design patterns, such as the Singleton, are easier to distinguish and define than other design patterns that have similar structural and behavioral characteristics, such as the Bridge and Builder design patterns. Furthermore, some design patterns have a limited number of variants, while other patterns have more variants with different characteristics. However, once design patterns are properly defined, the resulting definitions can be used in the detection process for any input system without any further human intervention.

The second phase of E_x-DPD_{FE} (the detection phase) was completely automated by the authors of this paper. The Java parser used in the first step of this phase was developed by a team of master's degree students for different purposes prior to conducting this research. This parser was partially enhanced to fit the goal of this research. Parser enhancement requires analyzing and understanding the existing code and designing and implementing the additional functionality required. Moreover, some functionalities were removed to reduce the parsing time. To conduct the second step of this phase, a text parser was developed to analyze the textual definition files of the design patterns. Additionally, in this step, the methods that measure the features specified in the definition files are implemented.

Structural features, such as the class abstraction level, are easier to measure than behavioral features, such as method call. After measuring feature values, the candidate role classes can be easily determined, and their relations can be immediately identified based on the information stored in the database.

4. Software Implementation

The second phase of E_x-DPD_{FE} (the detection phase) is implemented in Java. In this section, the developed software system is illustrated; after that, the computational complexity of the developed system is discussed.

4.1. The Software System of E_x-DPD_{FE}

The graphical user interface (GUI) of the developed software system is illustrated in Figure 11. First, the user needs to specify the location of the Java files of the input system; after that, the software system of E_x-DPD_{FE} parses these files using a Java parser developed using ANTLR 4.7. The parser is required for measuring the features that cannot be measured using direct analysis of the input source code, such as delegation and method call. The developed parser matches single statements in the input source code and stores the extracted information in a database. The MySQL Workbench 8.0 was used to store and retrieve this information using standard queries. The extracted information includes the following:

- Class attributes (visibility, parent classes);
- Method attributes (type, visibility, parameter types, return type);
- Method calls (source and destination);
- The relations between classes (association, dependency, generalization, composition).

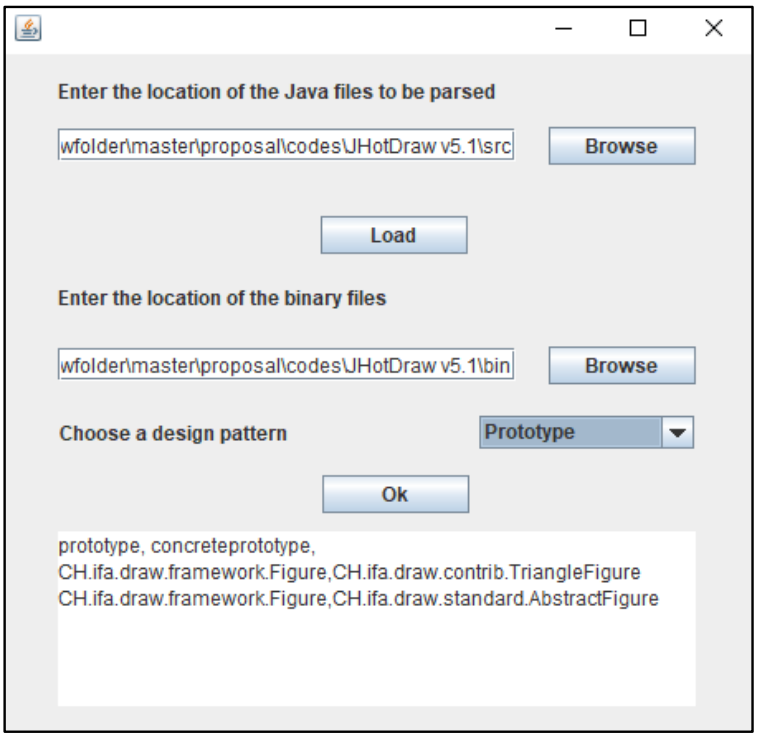


Figure 11. The GUI of the developed software system.

It is notable that source code parsing is only required when the user desires to detect the design patterns in a new software system for the first time. Once the characteristics of the input system are stored in the database, they are accessible until the user enters a different input system.

It is worth mentioning here that the modified implementations of a design pattern are considered. For example, when class A inherits from class B, and class B inherits from class C, then class A inherits from class C. This case is depicted in Figure 12, which shows an instance of the Decorator design pattern in JHotDraw v5.1. As can be noticed, the class AbstractLocator has been added between the class Locator, which represents the Component role of the Decorator design pattern, and the class OffsetLocator, which represents the Decorator role.

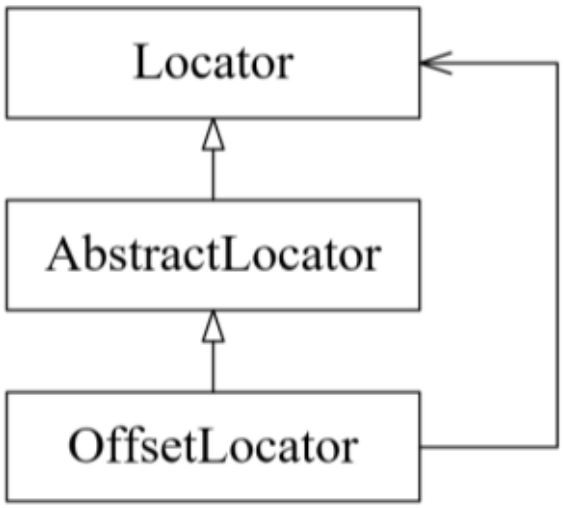


Figure 12. A modified implementation of the Decorator design pattern in JHotDraw v5.1.

Next, the user needs to enter the location of the binary files of the input system and choose a design pattern from the list of patterns. Here, the textual feature-based definition file that represents the design pattern is parsed using a text file parser developed using ANTLR 4.7. Then, the resulting Abstract Syntax Tree (AST) is walked using an ANTLR listener, which is responsible for calling the appropriate methods that measure the specified features. In this research, the Java reflection library was used for measuring structural features, such as inheritance and constructor visibility. Nevertheless, behavioral features, such as delegation and method call, are not supported by Java reflection. For this reason, these features were measured using the information stored in the database. It is worth mentioning here that all features can be measured using the information stored in the database. However, this reduces the accuracy of the detection results since they become entirely dependent on the quality of the parser. Moreover, increasing the number of database queries reduces time efficiency.

It is notable that the binary files of the input system need to be generated in order to apply E_x-DPD_{FE} to it because Java reflection cannot be applied to Java files. For this reason, a customized class loader was used to handle these binary files. Nonetheless, the class loader cannot handle the classes of external online libraries. Therefore, the project's libraries must be locally accessible to enable the class loader to resolve class dependencies.

To identify the candidate role classes, the canonical name of every class of the input system is stored in the database, along with a flag that indicates the number of features satisfied by that class for a particular role. When a class satisfies a specified feature, its associated flag value is incremented by 1; the classes that satisfy all the features that define a particular role are considered as candidate role classes. Algorithm 1 clarifies the process of extracting candidate role classes.

Algorithm 1 Extraction of the candidate role classes

```
Inputs: The feature-based textual definition of a design pattern D, and the characteristics
of the parsed input system S
Output: Candidate role classes
foreach Role r in D do
    foreach Class c in S do
            flag = 0
            foreach Feature f in r do
                if c satisfies f then
                    flag ++
                end
            end
            if flag = featurescount then
                add c to candidateclasses
            end
    end
end
ruturn candidateclasses
```

It is worth mentioning here that some classes can satisfy the features that identify different roles and participate in more than one instance of the same design pattern. For this reason, the different relations between the candidate role classes, which are stored in the database, are mapped to the signature of the design pattern under study, considering all their possible combinations. The correct combinations that satisfy the relations specified in the signature of the design pattern are outputted as instances of this design pattern. After that, the classes that satisfy the specified features and do not participate in any combination are eliminated. Finally, detection results are presented to the user without any human intervention.

In the context of our running example (the Adapter design pattern), the ANTLR listener walks the Abstract Syntax Tree that represents the textual definition file of the Adapter design pattern. At this point, the initial flag values of all the classes of the input system are 0. Afterward, the ANTLR listener calls the method responsible for measuring the first feature '*subclass*'. This method uses Java reflection to identify subclasses and increases the flag values of these classes by 1. Then, the ANTLR listener calls the method responsible for measuring the second feature '*one-to-one association to another class*', which also uses Java reflection to identify the classes that satisfy this feature based on the types of their fields and increases their flag values by 1. After that, the ANTLR listener calls the method responsible for measuring the third feature '*method call to another class by an overriding method*'. This method uses the information stored in the database to analyze the methods of the system classes and increases the flag values of the classes that satisfy the third feature by 1. Finally, system classes whose flag values are equal to 3 are considered as candidate Adapter classes. Figure 13 illustrates the flag values of some JHotDraw v5.1 classes after measuring each feature of the Adapter role of the Adapter design pattern. Moreover, Figure 14 shows some candidate Adapter classes, and Figure 15 demonstrates a code fragment of the class DrawApplet in JHotDraw v5.1, which represents the Adapter role of the Adapter design pattern.

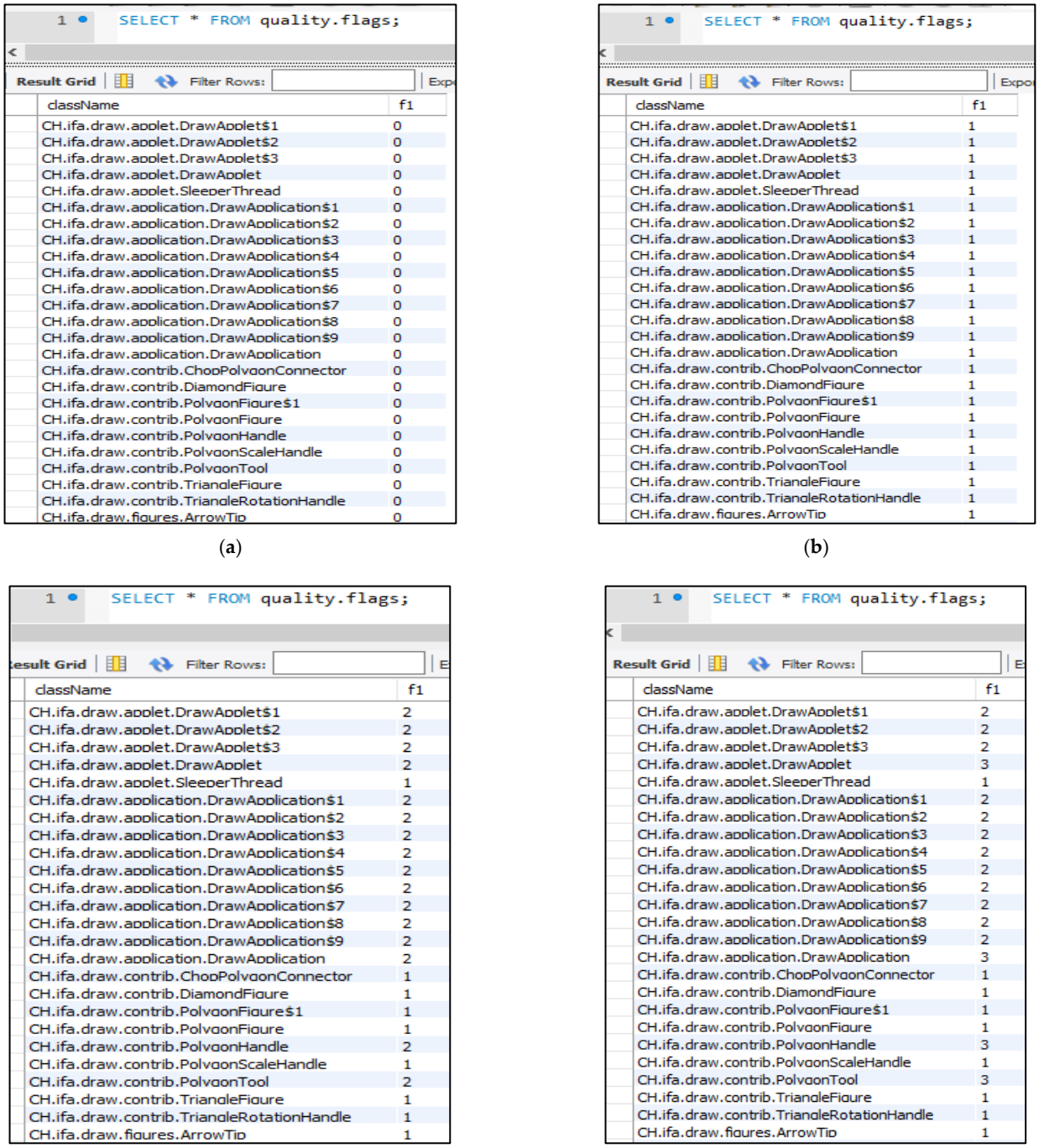


Figure 13. The flag values of some JHotDraw v5.1 classes after measuring each feature of the Adapter role of the Adapter design pattern: (**a**) initial values; (**b**) the first feature (subclass); (**c**) the second feature (one-to-one association to another class); (**d**) the third feature (method call to an associated class by an overriding method).

	role1
	CH.ifa.draw.applet.DrawApplet
	CH.ifa.draw.application.DrawApplication
	CH.ifa.draw.contrib.PolygonHandle
	CH.ifa.draw.contrib.PolygonTool
	CH.ifa.draw.figures.AttributeFigure
	CH.ifa.draw.figures.GroupCommand
	CH.ifa.draw.figures.InsertImageCommand
	CH.ifa.draw.figures.LineConnection
	CH.ifa.draw.figures.RadiusHandle
	CH.ifa.draw.figures.TextFigure
	CH.ifa.draw.figures.TextTool
	CH.ifa.draw.figures.UngroupCommand
	CH.ifa.draw.samples.javadraw.URLTool
	CH.ifa.draw.standard.AbstractConnector
	CH.ifa.draw.standard.AbstractFigure
	CH.ifa.draw.standard.AbstractTool
	CH.ifa.draw.standard.AlignCommand
	CH.ifa.draw.standard.BringToFrontComm...
	CH.ifa.draw.standard.ChangeAttributeCo...
	CH.ifa.draw.standard.ChangeConnection...
	CH.ifa.draw.standard.ConnectionHandle
	CH.ifa.draw.standard.ConnectionTool
	CH.ifa.draw.standard.CreationTool
	CH.ifa.draw.standard.DragTracker
	CH.ifa.draw.standard.HandleTracker

Figure 14. Some of the candidate Adapter classes in JHotDraw v5.1.

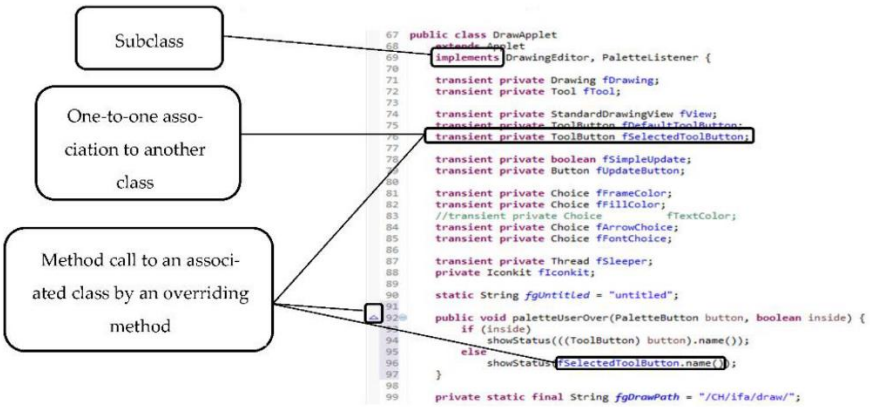


Figure 15. A code fragment of the class DrawApplet in JHotDraw v5.1, which clarifies the features satisfied by the Adapter role of the Adapter design pattern.

After that, the candidate Adaptee classes are identified by applying the same previous steps. Then, the detection process looks in the database for the candidate Adapter classes that are associated with the candidate Adaptee classes using a one-to-one association. Such classes are considered as an instance of the Adapter design pattern.

It is worth mentioning here that no special hardware elements were used in the implementation of the developed software system. The source code of E_x-DPD_{FE} is available online at https://sqlab.um.ac.ir/images-/219/files/detection%20method.zip (accessed on 11 June 2022).

4.2. Computational Complexity

The computational complexity of the developed software system was calculated using Microsoft Windows 10 with Intel(R) Core(TM) i5 2.20 GHz CPU and 8 GB memory. The first step of the detection process (the parsing step) is the most time consuming since the time required for parsing the source code of an input system depends on its size. To avoid this step, the characteristics of the input systems used to conduct this research were stored. Therefore, they can be immediately restored to the database when needed. The computational complexity of this step is $O(LOC)$.

In the second step (feature measurement), the time required to measure a feature depends on its type. Measuring structural features, which is conducted using Java reflection, is more straightforward than measuring behavioral features, which is conducted using database queries. The number of required database queries, in turn, depends on the size of

the input system. Assuming that f is the total number of features used to define a design pattern and n is the number of classes of the input system, the computational complexity of this step is $O(fn)$.

The third step (candidate role class extraction) is a simple step that is the least time consuming, and it only requires one database query. The computational complexity of this step is $O(1)$.

Finally, the execution time of the fourth step (finding related roles) depends on the number of candidate role classes (c) and the number of database queries required to determine their relations (r). Therefore, the computational complexity of this step is $O(cr)$.

5. Evaluation and Discussion

In this section, the efficiency of E_x-DPD_{FE} is assessed and compared with some other detection methods.

5.1. Research Questions

To address the goal of this study, we aimed to answer the following research questions (RQ):

RQ1: What is the accuracy of E_x-DPD_{FE} in detecting design patterns?

RQ2: Can E_x-DPD_{FE} detect more design pattern variants than other existing methods?

RQ3: How do design pattern variants affect the accuracy of detection methods?

While RQ1 concentrates on the accuracy of E_x-DPD_{FE}, RQ2 and RQ3 compare the accuracy of E_x-DPD_{FE} with that of other methods, considering the effect of the number of variants detected by a detection method on the quality of the detection results.

In the following subsection, we clarify the case studies used to respond to the above research questions.

5.2. Case Studies

To answer RQ1, E_x-DPD_{FE} was evaluated using four open-source Java projects, namely, JRefactory v2.6.24, JHotDraw v5.1, JUnit v3.7, and QuickUML 2001. These projects have been used to evaluate many design pattern detection methods, which makes it easier to compare the results of E_x-DPD_{FE} with those of the other methods. The characteristics of these projects are illustrated in Table 3. Moreover, to respond to research questions RQ2 and RQ3, the Java testbeds generated by PDB_{GTGT} [88], which is a benchmark developed as a solution to the problem of the absence of a golden standard, were used. Furthermore, the use of PDB_{GTGT} enables a comprehensive and fair evaluation of the design pattern detection methods and facilitates an accurate comparison between E_x-DPD_{FE} and other methods regarding variants' detection.

Table 3. The characteristics of the open-source Java projects used to evaluate E_x-DPD_{FE}.

Characteristics / Systems	Number of Classes	Number of Methods	KLOC
JRefactory v2.6.24	566	4609	93.1
JHotDraw v5.1	155	1334	13.5
JUnit v3.7	93	681	6.4
QuickUML 2001	217	1094	18.4

In PDB_{GTGT}, testbeds with different levels of complexity that are composed of Java source codes and their corresponding class diagrams are generated automatically using graph theory. These codes are injected with different design patterns and their variants. The types, numbers, and locations of the injected patterns are well defined and controlled. The testbeds generated by PDB_{GTGT} are only applicable to the detection methods that can receive input in the form of a class diagram or a Java source code. To define new design patterns in PDB_{GTGT}, the developers provided a grammar that has an XML structure in which a set of predefined tags is used to specify the role classes that participate in a design

pattern and their relations. The definitions used to generate the design patterns are not published publicly to preserve confidentiality. The generated testbeds were provided as an input to the detection methods under study to measure their accuracy practically. In this research, PDB_{GTGT} was used to generate 11 design patterns and examine the ability of different detection methods to detect their variants. These 11 design patterns were chosen because they can be detected by all the methods under study.

The use of PDB_{GTGT} to evaluate the detection methods provides several advantages: (1) accurate values of recall can be obtained since the number of injected instances of a design pattern is well defined; (2) the ability of each detection method to detect the variants of a design pattern can be evaluated; and (3) different detection methods can be evaluated under the same conditions, including the used testbeds, the considered design patterns, and the calculated evaluation metrics (i.e., precision and recall).

To evaluate the investigated detection methods, 30 auto-generated testbeds were used; the main characteristics of these testbeds are summarized in Table 4. The source codes of the testbeds generated by PDB_{GTGT} are available online at sqlab:SourceCodes_Generated_by_PDBGTGT.zip (accessed on 11 June 2022).

Table 4. The main characteristics of the evaluation testbeds auto-generated using PDB_{GTGT}.

Characteristics / Testbeds	Number of Classes per Project	Average KLOC
$Testbed_1$–$Testbed_{10}$	200	9.87
$Testbed_{11}$–$Testbed_{20}$	300	14.89
$Testbed_{21}$–$Testbed_{30}$	400	19.65

To verify the scalability of E_x-DPD_{FE}, it was applied to five open-source Java projects of different sizes, namely, JHotDraw v5.2, JHotDraw v5.3, JUnit v2, JUnit v3, and JUnit v3.8.1. The main characteristics of these projects are shown in Table 5. It is notable that the design pattern instances available in these projects have not been documented. Therefore, the recall of E_x-DPD_{FE} cannot be calculated using these five projects.

Table 5. The characteristics of the open-source Java projects used to assess the scalability of E_x-DPD_{FE}.

Characteristics / Systems	Number of Classes	Number of Methods	KLOC
JHotDraw v5.2	168	1460	14.7
JHotDraw v5.3	242	2316	24.8
JUnit v2	39	315	2.7
JUnit v3	116	841	7.2
JUnit v3.8.1	56	514	4.8

5.3. Method Selection

E_x-DPD_{FE} was compared with DeMIMA [50], SSA [13], SparT [2], GTM [9], and CA [31]. These methods were chosen since they all satisfy at least two of the following conditions: (1) they have been used to evaluate many design pattern detection methods; (2) their detection results are available online; (3) they provide runnable tools or online support for practical evaluation; and (4) they demonstrate accurate detection results in terms of precision and recall. One thing to mention is that the SparT group link is not working at the moment, although it was accessible when this study was conducted. However, the evaluation results of SparT are still available online at sqlab:The_design_pattern_instances_detected_by_Ex_DPDfe_and_some_other.pdf (accessed on 11 June 2022). Additionally, since GTM is a semi-automated method in which a candidate design pattern is validated manually by mapping its behavior to the proposed behavioral signature, it cannot be evaluated using the considered benchmark. Therefore, we only considered the evaluation results reported by its developers.

5.4. Metrics

In this research, three widely adopted metrics (precision (P), recall (R), and F-measure) were used to measure the accuracy of the evaluation results. These metrics are defined as follows:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \tag{1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \tag{2}$$

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precison} + \text{Recall}}, \tag{3}$$

where TP refers to the number of design pattern instances that are correctly detected, FP refers to the number of design pattern instances that are incorrectly detected, and FN refers to the number of design pattern instances that are incorrectly not detected [89,90]. Generally, the use of these metrics to measure the accuracy of E_x-DPD_{FE} allows a fair comparison between the investigated detection methods.

To calculate the precision obtained from applying E_x-DPD_{FE} to the open-source Java projects, the design pattern instances detected by E_x-DPD_{FE} were verified manually to identify true positive instances. Moreover, to identify all the design pattern instances in the projects under study and calculate the recall of E_x-DPD_{FE}, the union of the true positive instances detected by the investigated methods with the true positive instances detected by E_x-DPD_{FE} was considered as a golden standard after manual verification in addition to the instances reported in the P-MART (http://www.ptidej.net/tools/designpatterns/index_html#2 (accessed on 11 June 2022)) dataset. These verified instances are available online at sqlab:The-gold-standard-of-Ex-DPDfe.pdf. Table 6 summarizes the number of the design pattern instances identified in the used dataset, and Table 7 illustrates the design patterns considered by the methods used to build the dataset.

Table 6. The number of the design pattern instances identified in the used dataset.

Open-Source Projects / Design Patterns	JRefactory v2.6.24	JHotDraw v5.1	JUnit v3.7	QuickUML 2001
Singleton	10	1	0	1
Adapter	40	26	7	10
Abstract Factory/Factory Method	11	17	0	5
Template Method	18	8	3	5
Composite	0	1	1	1
Observer	6	5	3	8
State/Strategy	29	42	3	5
Decorator	0	4	1	1
Visitor	2	0	0	0
Prototype	0	2	0	1
Builder	0	0	0	0
Bridge	0	0	0	0
Facade	0	0	0	0
Flyweight	0	0	0	0
Proxy	11	1	0	6
Chain of Responsibility	0	0	0	0

Table 6. *Cont.*

Open-Source Projects / Design Patterns	JRefactory v2.6.24	JHotDraw v5.1	JUnit v3.7	QuickUML 2001
Command	7	14	0	0
Mediator	0	0	0	0
Iterator	0	0	0	0
Memento	0	0	0	0
Interpreter	0	0	0	0

Table 7. The design patterns considered by the methods used to build the dataset.

Detection Methods / Design Patterns	DeMIMA	SSA	SparT	GTM	CA	E_x-DPD$_{FE}$	P-MART
Singleton	✓	✓	✓	✓	✓	✓	✓
Adapter	✓	✓	✓	✓	✓	✓	✓
Abstract Factory	✓	✓	✓	✓	✓	✓	✓
Factory Method	✓	✓	✓	✓	✓	✓	✓
Template Method	✓	✓	✓	✓	✓	✓	✓
Composite	✓	✓	✓	✓	✓	✓	✓
Observer	✓	✓	✓	✓	✓	✓	✓
State	✓	✓	✓	✓	✓	✓	✓
Strategy	✓	✓	✓	✓	✓	✓	✓
Decorator	✓	✓	✓	✓	✓	✓	✓
Visitor	✓	✓	✓	✓	✓	✓	✓
Prototype	✓	✓	✓	×	✓	✓	✓
Builder	×	×	✓	×	✓	✓	✓
Bridge	×	×	✓	×	✓	✓	✓
Facade	×	×	✓	×	✓	✓	✓
Flyweight	×	×	✓	×	✓	✓	✓
Proxy	×	×	✓	×	✓	✓	✓
Chain of Responsibility	×	×	✓	×	✓	✓	✓
Command	✓	✓	✓	×	✓	✓	✓
Mediator	×	×	✓	×	✓	✓	✓
Iterator	×	×	✓	×	✓	✓	✓
Memento	×	×	✓	×	✓	✓	✓
Interpreter	×	×	✓	×	✓	✓	✓

Similarly, precision and recall were calculated manually when applying E_x-DPD$_{FE}$ to PDB$_{GTGT}$ using the information about the injected design patterns that was provided along with the generated testbeds.

5.5. Results

The results are organized as follows. Table 8 summarizes the results of applying E_x-DPD$_{FE}$ to the four open-source projects mentioned above, while Table 9 presents the detection results of E_x-DPD$_{FE}$ in comparison with the other methods under study according to the results reported by their developers. Detailed evaluation results are available at sqlab: The_design_pattern_instances_detected_by_Ex_DPDfe.pdf (accessed on 11 June 2022). Table 10 illustrates the results of evaluating E_x-DPD$_{FE}$ using 30 testbeds, automatically generated using PDB$_{GTGT}$, and Table 11 presents a comparison between E_x-DPD$_{FE}$ and the other methods under study using the same testbeds. In these tables, the use of '0' means that no design pattern instances were correctly detected (TP = 0). Finally, Table 12 shows the implementation cost of E_x-DPD$_{FE}$, and Table 13 illustrates its time complexity.

Table 8. The results of applying E_x-DPD_{FE} to four open-source Java projects.

Detection Results / Design Patterns	JHotDraw v5.1				JUnit v3.7				QuickUML 2001				JRefactory v2.6.24			
	TP	**FP**	**P%**	**F%**	**TP**	**FP**	**P%**	**F%**	**TP**	**FP**	**P%**	**F%**	**TP**	**FP**	**P%**	**F%**
Singleton	1	0	100	100	0	0	100	100	1	0	100	100	10	0	100	100
Adapter	26	1	96.3	98	7	1	87.5	93	10	1	90.9	95	40	4	90.9	95
Abstract Factory/ Factory Method	17	2	89.5	94	0	0	100	100	5	0	100	100	11	1	91.7	96
Template Method	8	0	100	100	3	0	100	100	5	0	100	100	18	2	90	95
Composite	1	0	100	100	1	0	100	100	1	0	100	100	0	0	100	100
Observer	5	1	83.3	91	3	1	75	86	8	2	80	89	6	1	85.7	92
State/Strategy	42	2	95.5	98	3	0	100	100	5	2	71.4	83	29	4	87.9	94
Decorator	4	0	100	100	1	0	100	100	1	0	100	100	0	0	100	100
Visitor	0	0	100	100	0	0	100	100	0	0	100	100	2	0	100	100
Prototype	2	0	100	100	0	0	100	100	1	0	100	100	0	0	100	100
Builder	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Bridge	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Facade	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Flyweight	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Proxy	1	0	100	100	0	0	100	100	6	0	100	100	11	0	100	100
Chain of Responsibility	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Command	14	2	87.5	93	0	0	100	100	0	0	100	100	7	3	70	82.4
Mediator	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Iterator	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Memento	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100
Interpreter	0	0	100	100	0	0	100	100	0	0	100	100	0	0	100	100

Table 9. The results of E_x-DPD_{FE} in comparison with those reported by the developers of the other methods when applied to four open-source Java projects.

Detection Results / Design Patterns	DeMIMA			SSA			SparT			GTM			CA			E_x-DPD_{FE}		
	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%	P%	R%	F%
Singleton	78.6	100	88	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Adapter	9.95	100	18.1	100	100	100	85.8	83.2	84.5	100	100	100	100	100	100	91.4	100	95.5
Abstract Factory/ Factory Method	26.35/ 0.75	100/ 100	41.7/ 1.5	100	63.9	78	100	100	100	100	100	100	100	100	100	95.3	100	97.6
Template Method	1.6	100	3.1	100	100	100	100	100	100	100	100	100	100	100	100	97.5	100	98.7
Composite	67.7	100	80.7	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Observer	25	100	40	100	100	100	93.8	100	96.8	100	100	100	100	50	66.7	81	100	89.5
State/Strategy	9.4	100	17.2	100	95.7	97.8	63.3	94.2	75.7	100	100	100	100	100	100	88.7	100	94
Decorator	51.9	100	68.3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Visitor	87.5	100	93.3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Prototype	75	100	85.7	100	100	100	100	100	100	-	-	-	100	100	100	100	100	100
Builder	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Bridge	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Facade	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Flyweight	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Proxy	-	-	-	-	-	-	95.8	97.7	96.7	-	-	-	100	100	100	100	100	100
Chain of Responsibility	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Command	8.53	100	15.7	100	100	100	100	100	100	-	-	-	100	100	100	89.4	100	94.4
Mediator	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Iterator	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Memento	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100
Interpreter	-	-	-	-	-	-	100	100	100	-	-	-	100	100	100	100	100	100

Table 10. The results of applying E_x-DPD_{FE} to 30 testbeds generated using the PDB_{GTGT} benchmark.

Detection Methods / Design Patterns		DeMIMA	SSA	SparT	GTM	CA	E_x-DPD_{FE}
Singleton		87	87	0	100	100	100
Class Adapter		34	34	0	100	100	100
Object Adapter		42	42	0	100	100	100
Abstract Factory/ Factory Method	**Variant F1**	34	34	0	100	100	100
	Variant F2	121	121	0	100	100	100
Template Method		48	48	0	100	100	100
Composite	**Variant C1**	39	39	0	100	100	100
	Variant C2	42	42	0	100	100	100
	Variant C3	33	33	0	100	100	100
Observer	**Variant O1**	41	41	0	100	100	100
	Variant O2	38	38	0	100	100	100
State/Strategy		157	157	0	100	100	100
Decorator		40	40	0	100	100	100
Visitor		34	34	0	100	100	100

Table 11. The results of comparing E_x-DPD_{FE} with other methods using the PDB_{GTGT} benchmark.

Detection Results / Design Patterns	DeMIMA			SSA			SparT			CA			E_x-DPD_{FE}		
	P%	**R%**	**F%**	**P%**	**R%**	**F%**	**P%**	**R%**	**F%**	**P%**	**R%**	**F%**	**P%**	**R%**	**F%**
Singleton	100	53.8	70	100	100	100	100	100	100	56.7	78.3	65.8	100	100	100
Adapter	0	0	na	100 *	100 *	100	100 *	100 *	100	100	94.3	97	100	100	100
Abstract Factory/ Factory Method	9.9	24.4	14	100	21.9	36	0	0	na	89	19.8	32.4	100	100	100
Template Method	0	0	na	0	0	na	0	0	na	0	0	na	100	100	100
Composite	0	0	na	100	34.2	51	100	71.1	83.1	72.3	24.3	36.4	100	100	100
Observer	0	0	na	100	48.1	65	100	48.1	65	21	30.2	24.8	100	100	100
State/Strategy	91.1	100	95.3	100	100	100	100	100	100	50.3	100	66.9	100	100	100
Decorator	0	0	na	100	100	100	100	100	100	27.7	22	24.5	100	100	100
Visitor	0	0	na	0	0	na	0	0	na	0	0	na	100	100	100

* Only Object Adapter.

Table 12. The implementation cost of E_x-DPD_{FE}.

	Cost / Step	Average Human Hours
The definition phase	Signatures' improvement	2 h per pattern
	Feature extraction	2 h per pattern
The detection phase	Source code parsing	120 h
	Features' measurement	4 h per feature
	Candidate Role classes' extraction	1 h
	Finding related role classes	1 h

Table 13. The time complexity of E_x-DPD_{FE}.

Cost / Step	Feature Type	KLOC	Time (s)
Source code parsing	-	2.7	90
		4.8	135
		6.4	210
		7.2	300
		9.87	390
		13.5	540
		14.7	600
		14.89	615
		18.4	750
		19.65	870
		24.8	1130
		93.1	2245
Features' measurement(features' average)	Structural	No major effect	0.03
	Behavioral	2.7	0.05
		4.8	0.09
		6.4	0.11
		7.2	0.14
		9.87	0.17
		13.5	0.21
		14.7	0.23
		14.89	0.24
		18.4	0.29
		19.65	0.32
		24.8	0.37
		93.1	0.63
Candidate Role classes' extraction	No major effect	No major effect	0.01
Finding related role classes(patterns' average)	No major effect	2.7	0.02
		4.8	0.023
		6.4	0.027
		7.2	0.028
		9.87	0.031
		13.5	0.037
		14.7	0.039
		14.89	0.04
		18.4	0.061
		19.65	0.063
		24.8	0.08
		93.1	0.14

Since the State and Strategy design patterns share the same signature, they could not be identified independently using E_x-DPD_{FE}. Therefore, to distinguish between them, a further study of the purpose of using the design patterns is needed. Similarly, since the Abstract Factory design pattern contains a Factory Method in its Concrete Factory class that creates and returns an instance of the Concrete Product class, these two design patterns were identified together.

E_x-DPD_{FE} demonstrated an average precision of 97% on the projects used to evaluate its scalability, which are presented in Table 5. Detailed results are available online at sqlab:The_precision_of_Ex_DPDfe.pdf (accessed on 11 June 2022).

In the following, we answer the research questions according to the obtained results.

5.5.1. RQ1: What Is the Accuracy of E_x-DPD_{FE} in Detecting Design Patterns?

Considering Tables 8 and 9 and comparing the detected design pattern instances of E_x-DPD_{FE} with those of other methods, some cases in which the results did not match were noticed. For instance, both SSA and SparT considered the class Iconkit in JHotDraw v5.1 as

an instance of the Singleton design pattern even though it has a public constructor. This is a good indication of the high accuracy of the signatures considered by E_x-DPD_{FE} as well as the extracted features. In addition, since E_x-DPD_{FE} covers more variants, it identified the classes AbstractFigure, CompositeFigure, and GroupFigure in JHotDraw v5.1, which are depicted in Figure 16, as three independent instances of the Concrete Creator of the Factory Method design pattern. In contrast, SSA and SparT only identified the AbstracFigure class.

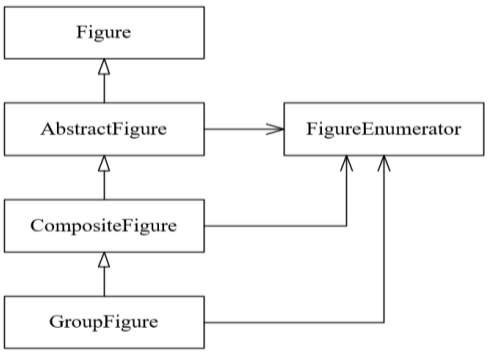


Figure 16. Some instances of the Factory Method design pattern in JHotDraw v5.1 that were not detected by other methods.

Similarly, considering that GTM and SSA have not been evaluated using QuickUML 2001, it was noticed that neither DeMIMA nor SparT could detect the classes shown in Figure 17 as instances of the Factory Method design pattern. These instances were only detected by E_x-DPD_{FE}.

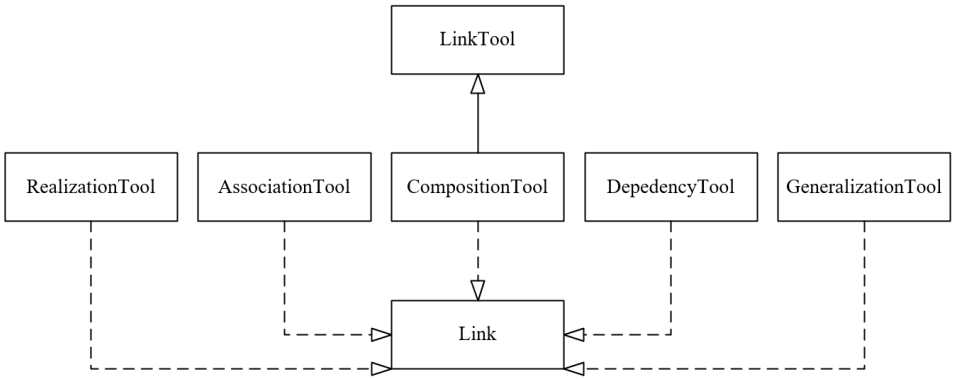


Figure 17. Some instances of the Factory Method design pattern in QuickUML2001 that were not detected by other methods.

On the other hand, GTM only considers three types of relations between classes (inheritance, aggregation, and association) in its first phase (the structural analysis phase). The other types of relations, such as dependency and method call, which are essential to distinguish design patterns with similar structures, are checked manually in its second phase (the behavioral analysis phase). As a result, many false positive instances are left for design pattern experts to exclude manually based on the proposed signatures. Moreover, in certain cases, GTM mislabels some classes and, therefore, misses some design pattern instances. Such a case is depicted in Figure 18, which shows an example of the Strategy design pattern in JHotDraw v5.1. In this example, the class ConnectionFigure was not labeled correctly by GTM. As a result, it was not recognized as a Concrete Strategy class; it was only recognized as a Strategy class. For this reason, the class Figure was not detected as a Strategy.

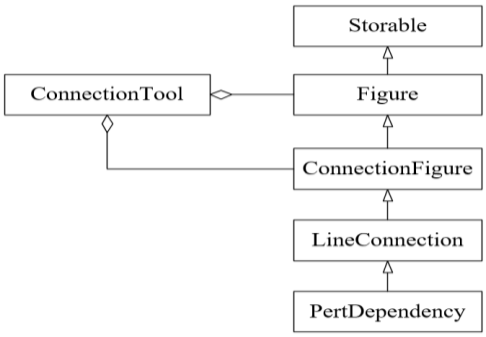


Figure 18. An example of an instance of the Strategy design pattern in JHotDraw v5.1 that was not detected by GTM.

It is notable that E_x-DPD_{FE} was able to avoid false negatives by utilizing three strategies:

- **Using the appropriate features that accurately define design patterns:** The used features are extracted from the signatures of design patterns. Therefore, these features represent each design pattern properly.
- **Considering both structural and behavioral characteristics of the design patterns in the feature extraction process:** The extracted features cover the different aspects of the design patterns (structural and behavioral). Therefore, they enable us to detect the design patterns of all categories (creational, structural, and behavioral).
- **Considering the design pattern variants in the process of feature extraction and covering as many variants as possible:** By including the characteristics of the design pattern variants within the extracted features, the instances that do not apply to the standard form of a design pattern can be detected.

As a result of the above, all the instances of the design patterns were detected, and the values of recall were equal to 100% for all the design patterns.

On the other hand, in some cases, an instance can satisfy all the features that define a design pattern. Nonetheless, this instance may not satisfy the purpose of this pattern. Therefore, a false positive instance will be detected by E_x-DPD_{FE}. An example that clarifies this case is depicted in Figure 19, which shows a code fragment of the class ChangeConnectionHandle in JHotDraw v5.1. This class is a subclass of the class AbstractHandle and it is associated with the class Figure by the field fTarget. Moreover, the method invokeStep(), which is an overriding method, calls the method connectorVisibility() of the class Figure. However, this class does not satisfy the purpose of the Adapter design pattern.

```
28 public abstract class ChangeConnectionHandle extends AbstractHandle {
29
30     protected Connector         fOriginalTarget;
31     protected Figure            fTarget;
32     protected ConnectionFigure  fConnection;
33     protected Point             fStart;
34
35     public void invokeStep (int x, int y, int anchorX, int anchorY, DrawingView view) {
36         Point p = new Point(x, y);
37         Figure f = findConnectableFigure(x, y, view.drawing());
38         // track the figure containing the mouse
39         if (f != fTarget) {
40             if (fTarget != null)
41                 fTarget.connectorVisibility(false);
42             fTarget = f;
43             if (fTarget != null)
44                 fTarget.connectorVisibility(true);
45         }
```

Figure 19. A code fragment of the class ChangeConnectionHandle in JHotDraw v5.1, which represents a false positive instance of the Adapter design pattern detected by E_X-DPD_{FE}.

The total number of false positive instances detected by E_x-DPD_{FE} in the four open-source Java projects used for evaluation was 30. On the other hand, the total number of true positive instances was 316, as clarified in Table 8. Therefore, the average precision of E_x-DPD_{FE} was equal to 91.3. Appendix C illustrates some examples of true positive and false positive instances (if any) of some design patterns detected in JHotDraw v5.1, with an explanation of the main reasons for their detection. Furthermore, all the detected design pattern instances are available at sqlab:The_design_pattern_instances_detected_by_Ex_DPDfe.pdf (accessed on 11 June 2022). It is worth mentioning here that such false positive instances were also detected by the other detection methods under study for similar reasons. However, the number of false positive instances can be reduced in the future by considering semantic analysis.

Since such a case was not encountered when evaluating E_x-DPD_{FE} using PDB_{GTGT}, E_x-DPD_{FE} reported accurate detection results, although this benchmark generates codes containing more variants than other open-source projects. Nevertheless, obtaining lower precision using the open-source Java projects does not threaten the validity of the results obtained using PDB_{GTGT} since it provides a fair comparison between detection methods that is conducted under the same conditions.

E_x-DPD_{FE} obtained relatively low precision for the Observer design pattern (81%) when evaluated using the open-source projects since the total number of Observer design pattern instances in these projects is low. Therefore, detecting a false instance results in a rapid drop in precision. Moreover, the design patterns are detected based on their structural and behavioral signatures. Therefore, E_x-DPD_{FE} cannot distinguish between design patterns with similar structural and behavioral characteristics, such as the State and Strategy design patterns.

5.5.2. RQ2: Can E_x-DPD_{FE} Detect More Design Pattern Variants than Some Other Existing Methods?

Based on the evaluation results shown in Tables 9–11, it was noticed that both SSA and SparT did not detect some common variants of some design patterns. DeMIMA, being a constraint satisfaction method, could not detect most of the variants since these variants were not considered in the constraint specification process. Moreover, CA was not able to detect any of the injected variants.

Except for E_x-DPD_{FE}, none of the other detection methods could detect the classes FileSummary and PackageSummary in JRefactory v2.6.24 as instances of the Singleton design pattern because the private static variables in these classes are of the HashMap type. Unlike E_x-DPD_{FE}, which considers all types of variables, the other methods only detect the Singleton class when its private static variable represents a single object and not a list of objects. This, in turn, demonstrates the wide variant coverage of E_x-DPD_{FE}.

In addition to the above, after analyzing the variants injected within the codes generated by PDB_{GTGT}, which are defined and reported, two variants were recognized for the Observer design pattern. In the first variant (O1), the Observer class is a concrete class that does not participate in any inheritance hierarchy, while in the second variant (O2), the Observer class is an abstract class (interface) that exists within an inheritance hierarchy. These two variants are illustrated in Figure 20. Both SSA and SparT were only able to detect variant O2. In comparison, E_x-DPD_{FE} could detect both O1 and O2, and DeMIMA could not detect either of them.

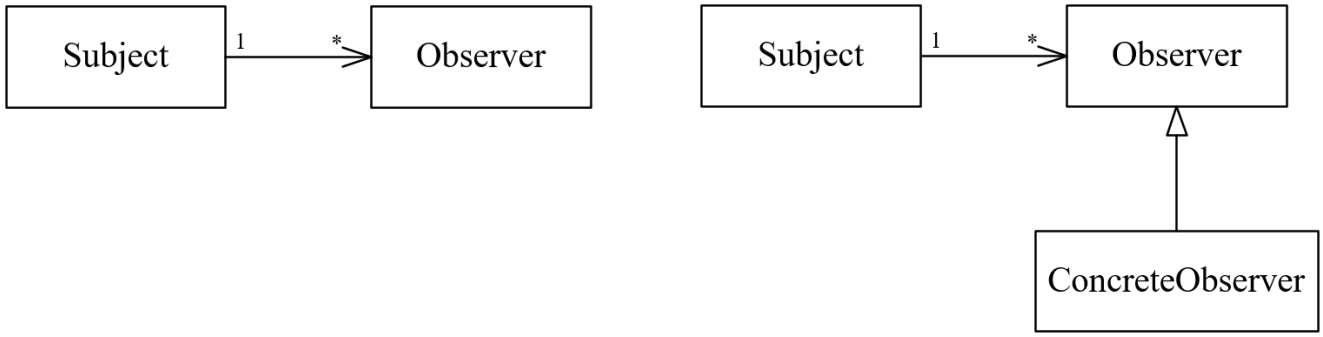


Figure 20. The variants of the Observer design pattern that were used in the codes generated by the PDB_{GTGT} benchmark: (**a**) Variant O1; (**b**) Variant O2.

Similarly, three variants were identified for the Composite design pattern, as illustrated in Figure 21. In the first variant (C1), the Component class is abstract (interface), while in the second variant (C2), both the Component and the Composite are concrete classes. The third variant (C3) is called *Reflexive connection between Leaf and Composite* [18]. Both SSA and SparT were able to detect C1. However, only SparT was able to detect C2. In comparison, variant C3 was only detected by E_x-DPD_{FE} after extracting the appropriate features independently. It is worth mentioning here that the three Composite instances detected in JHotDraw v5.1, Junit v3.7, and QuickUML 2001 were of C1 type.

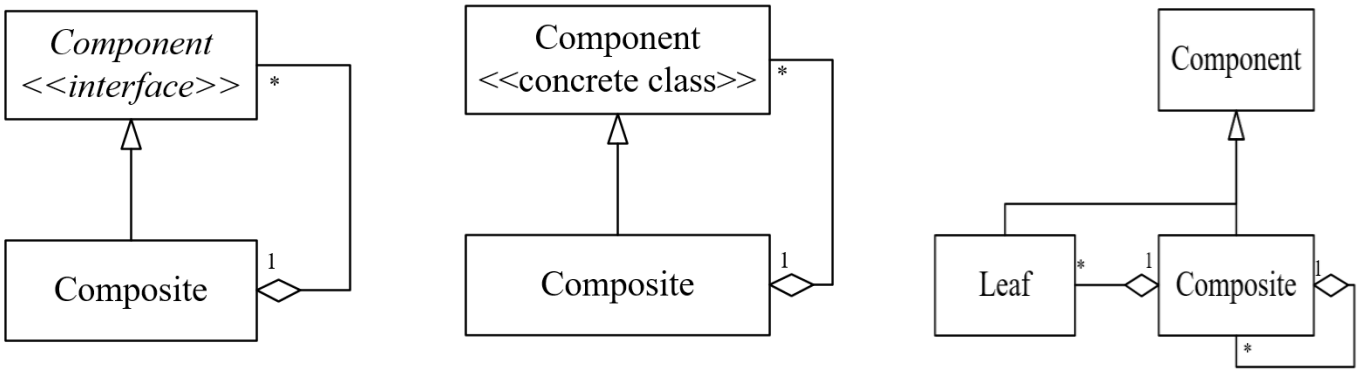


Figure 21. The variants of the Composite design pattern that were used in the codes generated by the PDB_{GTGT} benchmark: (**a**) Variant C1; (**b**) Variant C2; (**c**) Variant C3.

Moreover, two variants of the Factory Method design pattern were recognized, as depicted in Figure 22. The first variant (F1) consists of four roles: Creator, Concrete Creator, Product, and Concrete Product. In the second variant (F2), the Product class is eliminated. Unlike E_x-DPD_{FE}, which was able to detect both F1 and F2, SSA was only able to detect F1, while SparT could not detect either case.

Furthermore, unlike E_x-DPD_{FE}, in which both cases of the Adapter design pattern (the Object Adapter and the Class Adapter) were considered, the remaining methods could only detect the Object Adapter. Since none of the considered methods detected the Template Method and Visitor design patterns in the generated codes, improving the used benchmark is recommended.

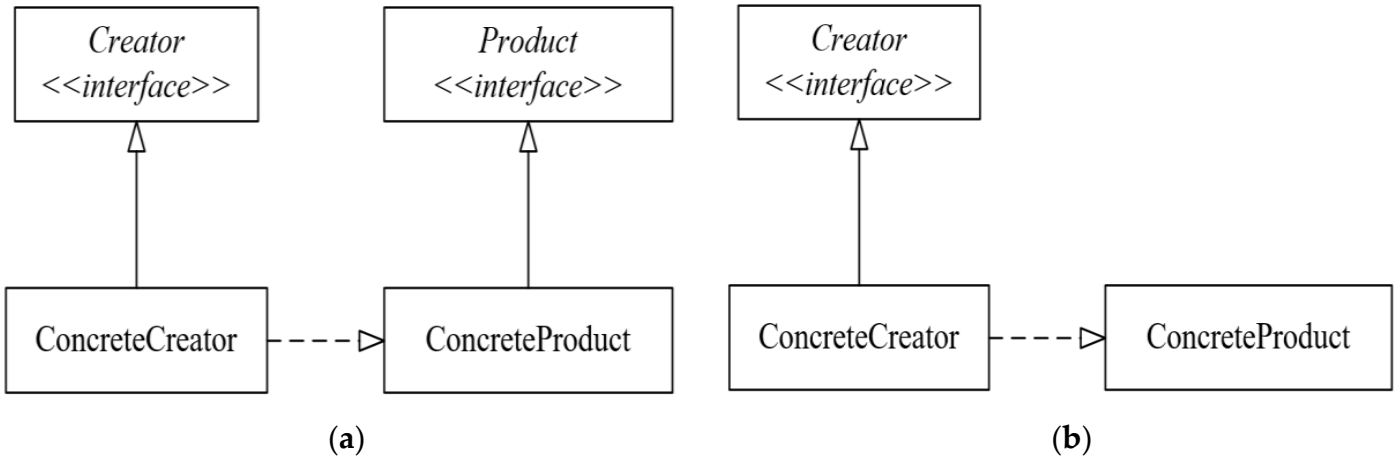


Figure 22. The variants of the Factory Method design pattern that were used in the codes generated by the PDB_{GTGT} benchmark: (**a**) Variant F1; (**b**) Variant F2.

5.5.3. RQ3: How Do Design Pattern Variants Affect the Accuracy of Detection Methods?

By analyzing the design pattern instances detected in the four open-source projects considering the variants proposed in [18], it can be concluded that the open-source Java projects only include specific design pattern variants. The results of this analysis are shown in Table 14, which illustrates the variants covered by E_x-DPD_{FE} thus far and their use in the projects under study. Based on these results, it can be noticed that the developers of a software project tend to use similar elements to reduce the complexity of the project and increase its maintainability and coherence. For example, the developers of JHotDraw v5.1 and QuickUML 2001 use the eager instantiation variant of the Singleton design pattern, while the developers of JRefactory v2.6.24 use the lazy instantiation variant. Moreover, developers focus on variants with low complexity rather than more complex variants. Therefore, it can be concluded that the evaluation results obtained using these projects are not deterministic and there is no guarantee that the evaluation of the same detection methods using other projects would generate similar results.

Table 14. The variants considered by E_x-DPD_{FE} thus far and their use in the four open-source Java projects under study.

Covered Variants / Design Patterns	Variant Name	Usage	Project
Adapter	Pluggable Adapters	×	-
	Two-way Adapters	×	-
Composite	Composites of Composites	✓	JH
	1-N Relationship using arrays	×	-
	1-N Relationship using hash tables	×	-
	Reference participant	×	-
	Supplementary relationship	×	-
	Association implementation	×	-
Decorator	Omitting the abstract Decorator class	✓	JH

Table 14. *Cont.*

Covered Variants / Design Patterns	Variant Name	Usage	Project
Abstract Factory/ Factory Method	Different Product types inside Factory class	✓	JH, JR
	Default Product implementation	×	-
	Parameterized Factory Method	×	-
	One Concrete Creator for all	×	-
	Single Concrete class for Product selection without Client	×	-
Observer	Multiple instance Observer	✓	JH, QUML
	Compound implementation	×	-
Singleton	Eager instantiation	✓	JH, QUML
	Lazy instantiation (non-thread safe)	✓	JR
	Lazy instantiation (thread safe)	×	-
	Lazy instantiation with double lock mechanism	✓	JR
	Replaceable instance	×	-
	SubClassed Singleton	×	-
	Delegated construction	✓	JR
	Different placeholder	×	-
	Limiton	×	-
State/Strategy	Statemaps	×	-
	Three-level finite state machine	×	-
	Flexible Strategy pattern	×	-
Template method	Enhanced Template design pattern	×	-
Visitor	Visitor combinators	×	-
	Distributed monitoring using Visitor pattern	×	-
	Extended Visitor pattern	×	-
Builder	Nested Builder	×	-
Prototype	Only has a basic form	-	-
Bridge	Cascading Bridge	×	-
	Folded cascading Bridge	×	-
	Partially folded cascading Bridge	×	-
	Architectural cascading Bridge	×	-
	Bi-directional cascading Bridge	×	-
Façade	Encapsulating layered Facade	×	-
	Wrapper Facade	×	-
	Subsystem Facade	×	-
Flyweight	Constrainedly shared Flyweight	×	-
	Externalizing extrinsic State	×	-
Proxy	Pipe and filter implementation	×	-
	Dynamic Proxies	×	-
Chain of Responsibility	Handling strategy	×	-
	Forwarding strategy	×	-
	Bureaucracy pattern	×	-
Command	Basic form only	-	-
Interpreter	Only has a basic form	-	-
Mediator	Traffic generator Mediator	×	-

Table 14. *Cont.*

Design Patterns \ Covered Variants	Variant Name	Usage	Project
Iterator	External Iterators	×	-
	Static structure Iterators	×	-
	Nested object Iterator	×	-
	Single integral Iterator	×	-
	Multiple integral Iterator	×	-
	Magic cookie	×	-
	External magic cookie Iterator	×	-
	Internal Iterator	×	-
Memento	HybridPrM	×	-

JH = JHotDraw v5.1; JR = JRefactory v2.6.24; JU = JUnit v3.7; QUML = QuickUML 2001.

Considering Tables 9 and 11, it is remarkable that the recall values obtained by evaluating the detection methods using the PDB_{GTGT} benchmark were generally lower than those obtained by evaluating them using the open-source Java projects. Furthermore, it can be noted that, by covering more variants, E_x-DPD_{FE} was able to achieve more accurate detection results in comparison with the other detection methods.

It should be mentioned here that compound design patterns and variants that have different structural and behavioral characteristics were not considered in this research. As a result, a total of 58 variants could be detected using E_x-DPD_{FE}. Detailed information about the variants listed in Table 14 is available online at https://sqlab.um.ac.ir/images/219/files/G.rasool%20design%20-pattern%20variants.zip (accessed on 11 June 2022). In addition, the variants detected by E_x-DPD_{FE} share the essential structural and behavioral characteristics specified by the extracted features that define the design patterns.

6. Threats to Validity

Threats to internal validity refer to the factors that affect the results. In this paper, the design patterns were defined based on their structural and behavioral features. However, extracting a set of features that is shared among the different variants of a specific design pattern is challenging. Moreover, considering the absence of full documentation of the design pattern instances existing in the open-source software systems used for evaluation, the reported variants are not deterministic and there may still be other undetected variants. Furthermore, detection results can be affected by the quality of the parser and the information stored in the database. This threat was alleviated by using the Java reflection library to measure structural features. The information extracted by the Java parser was only used to measure the behavioral features, which are not supported by the Java reflection library.

Threats to external validity refer to the ability to generalize the evaluation results. In this paper, E_x-DPD_{FE} was only applied to the GoF design patterns. Therefore, there is no guarantee that the detection results obtained by applying it to other design patterns with different structural and behavioral characteristics will be similar to those obtained by its application to the GoF design patterns. This threat can be addressed in the future by applying E_x-DPD_{FE} to a wider variety of design patterns. Additionally, another threat to external validity results from only considering input systems with the Java programming language. Finally, it is worthwhile to apply E_x-DPD_{FE} to input systems with other programming languages. Since both Java functions and source code parsing were used to measure the different features, the application of E_x-DPD_{FE} to input systems with a different programming language only requires using the appropriate parser in addition to modifying features' implementation appropriately. Moreover, we cannot claim that E_x-DPD_{FE} generates similar results on software systems that are based on principles such as microservices and distribution. Therefore, it is desirable to apply E_x-DPD_{FE} to such systems.

7. Conclusions and Future Work

In this paper, a new feature-based design pattern detection method, E_x-$DPD_{FE,}$ was proposed. This method is composed of two phases. Within the first phase, the required features are extracted from the signatures of the design patterns, and the variants of those design patterns are considered in the feature extraction process. These features, in turn, are then organized into a feature-based textual design pattern definition form and added to a reusable vocabulary. In the second phase, the resulting design pattern definition is applied to the input system to detect the design pattern instances.

E_x-DPD_{FE} was applied to four open-source Java projects in addition to 30 auto-generated testbeds injected with different design pattern variants. E_x-DPD_{FE} demonstrated high precision and recall because of its wide coverage of the design pattern variants. The detected design pattern variants were analyzed and reported intensively.

As future work, it is intended to expand the reusable vocabulary and apply E_x-DPD_{FE} to other design patterns to assess its extendibility. Furthermore, it is intended to consider semantic analysis to reduce the number of false positive instances and make it possible to differentiate between the design patterns that share the same signature, such as the Strategy and State design patterns. In addition, E_x-DPD_{FE} can be improved by considering input systems with other programming languages such as C and C++. Finally, it is planned to apply E_x-DPD_{FE} to similar contexts, such as systems that are based on microservices, distributed systems, and cloud design patterns.

Author Contributions: Conceptualization, A.R.; methodology, M.K.; software, M.K.; validation, M.K.; formal analysis, M.K.; investigation, M.K.; resources, M.K.; data curation, M.K.; writing—original draft preparation, M.K.; writing—review and editing, A.R.; visualization, M.K.; supervision, A.R.; project administration, A.R.; funding acquisition, M.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

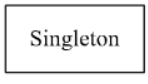
Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Codes and results are available online at https://sqlab.um.ac.ir/index.php (accessed on 11 June 2022).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. The Signatures of Some of the GoF Design Patterns

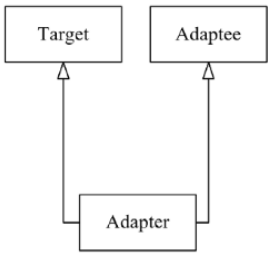


(**a**)

Singleton	
Behavioral characteristics	• Has ***no public constructor*** • Has ***a static public method*** that ***returns an instance of the same class***

(**b**)

Figure A1. The signature of the Singleton design pattern: (**a**) the structural signature of the Singleton; (**b**) the behavioral signature of the Singleton.

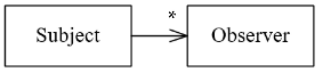


(**a**)

Class Adapter	
Behavioral characteristics	• A method in Adapter ***overrides*** a method in Target and ***calls*** a method in Adaptee

(**b**)

Figure A2. The signature of the Class Adapter design pattern: (**a**) the structural signature of the Class Adapter; (**b**) the behavioral signature of the Class Adapter.

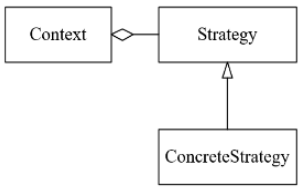


(**a**)

Observer	
Behavioral characteristics	• A method in Subject ***calls*** a method in Observer ***for all or some*** members of the mentioned set

(**b**)

Figure A3. The signature of the Observer design pattern: (**a**) the structural signature of the Observer; (**b**) the behavioral signature of the Observer.

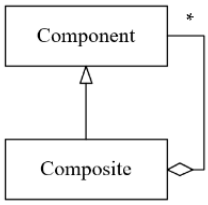


(**a**)

Strategy (State)	
Behavioral characteristics	• A method in Context ***calls*** a method in Strategy • A method in ConcreteStrategy ***overrides*** a method in Strategy

(**b**)

Figure A4. The signature of the Strategy (State) design pattern: (**a**) the structural signature of the Strategy (State); (**b**) the behavioral signature of the Strategy (State).

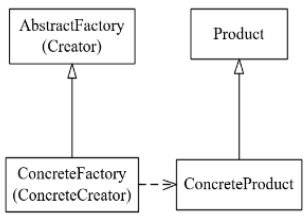


(**a**)

Composite	
Behavioral characteristics	• A method in Composite ***overrides*** a method in Component and ***delegates to it with loop***

(**b**)

Figure A5. The signature of the Composite design pattern: (**a**) the structural signature of Composite; (**b**) the behavioral signature of the Composite.

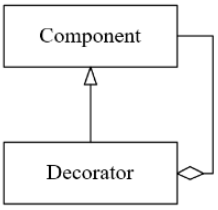


(**a**)

Abstract Factory (Factory Method)	
Behavioral characteristics	• A method in ConcreteFactory ***overrides*** a method in AbstractFactory and ***returns a new instance*** of ConcreteProduct

(**b**)

Figure A6. The signature of the Abstract Factory (Factory Method) design pattern: (**a**) the structural signature of Abstract Factory (Factory Method); (**b**) the behavioral signature of the Abstract Factory (Factory Method).

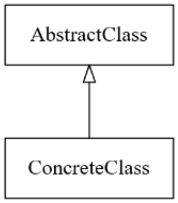


(a)

Decorator	
Behavioral characteristics	• A method in Decorator ***overrides*** a method in Component and ***calls*** it

(b)

Figure A7. The signature of the Decorator design pattern: (**a**) the structural signature of Decorator; (**b**) the behavioral signature of the Decorator.

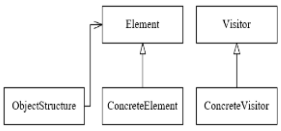


(a)

Template Method	
Behavioral characteristics	• A method in ConcreteClass ***overrides*** a method in AbstractClass • A method in AbstractClass ***calls*** another method in the same class

(b)

Figure A8. The signature of the Template Method design pattern: (**a**) the structural signature of Template Method; (**b**) the behavioral signature of the Template Method.



(a)

Visitor	
Behavioral characteristics	• A method in Element has a ***dependency*** to Visitor • A method in ConcreteElement ***overrides*** a method in Element and ***calls*** a method in Visitor

(b)

Figure A9. The signature of the Visitor design pattern: (**a**) the structural signature of Visitor; (**b**) the behavioral signature of the Visitor.

Appendix B. The Proposed Feature-Based Textual Definitions of Some of the GoF Design Patterns

Table A1. The definition of the Singleton design pattern.

Pattern Name: Singleton		
Role1 name: Singleton	$FSR1_1$	Public static method
	$FSR1_2$	Method returns an object of the same class
	$FSR1_3$	Non-public constructor

Table A2. The definition of the Class Adapter design pattern.

Pattern Name: Class Adapter		
Role1 name: Adapter	$FAR1_1$	Inheritance
	$FAR1_2$	Implementation
	$FAR1_3$	Method call to its superclass by a method that overrides another class
Role2 name: Adaptee	$FAR2_1$	Called method from its subclass by a method that overrides another class

Table A3. The definition of the Observer design pattern.

Pattern Name: Observer		
Role1 name: Subject	$FOR1_1$	One-to-many association to another class
	$FOR1_2$	Method call to an associated class in loop
Role2 name: Observer	$FOR2_1$	One-to-many association from another class
	$FOR2_2$	Called method from an associated class in loop

Table A4. The definition of the Strategy (State) design pattern.

Pattern Name: Strategy (State)		
Role1 name: Context	$FSR1_1$	One-to-one association to an interface
	$FSR1_2$	Method call to an associated interface
Role2 name: Strategy	$FSR2_1$	Interface
	$FSR2_2$	One-to-one association from another class
	$FSR2_3$	Called method from an associated class

Table A5. The definition of the Composite design pattern.

Pattern Name: Composite		
Role1 name: Composite	$FCR1_1$	Subclass
	$FCR1_2$	One-to-many association to its superclass
	$FCR1_3$	Delegation to its superclass with loop by an overriding method
Role2 name: Component	$FCR2_1$	Superclass
	$FCR2_2$	One-to-many association from its subclass
	$FCR2_3$	Delegation from its subclass with loop by an overriding method

Table A6. The definition of the Abstract Factory (Factory Method) design pattern.

Pattern Name: Abstract Factory (Factory Method)		
Role1 name: Concrete Product	$FFR1_1$	A new instance returned from another class by an overriding method
Role2 name: Concrete Factory (Concrete Creator)	$FFR2_1$	Subclass
	$FFR2_2$	Not interface
	$FFR2_3$	Returns a new instance of another class by an overriding method

Table A7. The definition of the Decorator design pattern.

Pattern Name: Decorator		
Role1 name: Decorator	$FDR1_1$	Subclass
	$FDR1_2$	One-to-one association to its superclass
	$FDR1_3$	Method call to its superclass by an overriding method
Role2 name: Component	$FDR2_1$	Superclass
	$FDR2_2$	Called method from its subclass
	$FDR2_3$	One-to-one association from its subclass

Table A8. The definition of the Template Method design pattern.

Pattern Name: Template Method		
Role1 name: Abstract Class	$FTR1_1$	Interface
	$FTR1_2$	Method call to an overridden method of the same class

Table A9. The definition of the Visitor design pattern.

Pattern Name: Visitor		
Role1 name: Visitor	$FVR1_1$	Superclass
	$FVR1_2$	Called method from another class with an overriding method that has a parameter of the type of the caller
	$FVR1_3$	One-to-one association from another class
	$FVR1_4$	Method with a parameter of the type of another class
Role2 name: Element	$FVR2_1$	Superclass
	$FVR2_2$	Method with a parameter of the type of another class
	$FVR2_3$	Method call to another class with a parameter of the type of the caller class
Role3 name: Concrete Element	$FVR3_1$	Subclass
	$FVR3_2$	Method with a parameter of the type of another class
	$FVR3_3$	Method call to another class with a parameter of the type of the caller class

Appendix C. Some Examples of True Positive and False Positive Instances (If Any) of Some Design Patterns Detected in JHotDraw v5.1

Instance Details / Design Patterns	Detected Instance	Instance Type	The Reason of Detection
Singleton	Singleton: CH.ifa.draw.util.Clipboard	TP	Singleton: • Public static method: getClipboard(). • Method returns an object: getClipboard() returns a Clipboard. • Nonpublic constructor: The class has a private constructor.
Template Method	Abstract Class: CH.ifa.draw.standard.AbstractFigure	TP	Abstract Class: • Interface: The class AbstractFigure is an abstract class. • Method call to the same class: The method moveBy() of the class AbstractFigure calls the methods willChange(), basicMoveBy (), and changed() of the same class.
Observer	Subject: CH.ifa.draw.standard.StandardDrawing, Observer: CH.ifa.draw.framework.DrawingChangeListener	TP	Subject: • One-to-many association to another class: The class StandardDrawing is associated with the class DrawingChangeListener using the vector named fListeners. • Method call to an associated class in loop: The class StandardDrawing calls the drawingRequestUpdate() method of the class Drawing Change Listener for every element in the vector fListeners.
Composite	Composite: CH.ifa.draw.standard.CompositeFigure, Component: CH.ifa.draw.framework.Figure	TP	Composite: • One-to-many association to its superclass: The class CompositeFigure extends the class AbstractFigure, which, in turn, implements the class Figure. Additionally, the class CompositeFigure is associated with the class Figure using the vector named fFigures. • Delegation with loop to its superclass by an overriding method: The method draw() of the class CompositeFigure overrides a method of the class Figure. Additionally, this method delegates to the method draw() in the class Figure with a loop.

Instance Details / Design Patterns	Detected Instance	Instance Type	The Reason of Detection
Strategy (State)	Context: CH.ifa.draw.applet.DrawApplet Strategy: CH.ifa.draw.framework.Drawing	TP	Strategy: • Interface: The class is an interface. • One-to-one association from another class: The class DrawApplet is associated with the class Drawing using the field named fDrawing. • Called method from an associated class: The class DrawApplet calls the release() method of the class Drawing for the field fDrawing.
	Context: CH.ifa.draw.standard.DragTracker Strategy: CH.ifa.draw.framework.Figure	FP	These classes satisfy the features used to identify the Strategy design pattern without meeting the goal of the pattern.
Decorator	Decorator: CH.ifa.draw.standard.DecoratorFigure, Component: CH.ifa.draw.framework.Figure	TP	Decorator: • One-to-one association to its superclass: The class DecoratorFigure extends the class AbstractFigure, which, in turn, implements the class Figure. Additionally, the class DecoratorFigure is associated with the class Figure using the field named fComponent. • Method call to associated superclass by an overriding method: The class DecoratorFigure calls the connectionInsets() method of the class Figure for the field fComponent by a method that overrides the same called method.
Factory Method (Abstract Factory)	Concrete Product: CH.ifa.draw.framework.Connector, Concrete Creator: CH.ifa.draw.standard.AbstractFigure	TP	Concrete Creator: • Subclass: AbstractFigure implements Figure • Returns a new instance of another class by an overriding method: The method connectorAt() of the class AbstractFigure overrides a method of the class Figure and returns a new instance of the class Connector.
	Concrete Product: CH.ifa.draw.framework.Locator Concrete Creator: CH.ifa.draw.contrib.PolygonFigure	FP	These classes satisfy the features used to identify the Factory Method design pattern without meeting the goal of the pattern.

References

1. Gamma, E.; Helm, R.; Johnson, R.E.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995.
2. Xiong, R.; Li, B. Accurate design pattern detection based on idiomatic implementation matching in Java language context. In Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019.
3. Ampatzoglouaba, A. A methodology to assess the impact of design patterns on software quality. *Inf. Softw. Technol.* **2012**, *54*, 331–346. [CrossRef]
4. Shilintsev, D.; Dlamini, G. A study: Design patterns detection approaches and impact on software quality. In Proceedings of the International Conference on Frontiers in Software Engineering, Innopolis, Russia, 17–18 June 2021.
5. Jaafar, F.; Guéhéneuc, Y.-G.; Hamel, S.; Khomh, F.; Zulkernine, M. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empir. Softw. Eng.* **2016**, *21*, 896–931. [CrossRef]
6. Hussain, S.; Keung, J.; Khan, A.A. Software design patterns classification and selection. *Appl. Soft. Comput.* **2017**, *58*, 225–244. [CrossRef]
7. Lucia, A.D.; Deufemia, V.; Gravino, C.; Risi, M. Detecting the behavior of design patterns through model checking and dynamic analysis. *ACM Trans. Softw. Eng. Methodol.* **2018**, *26*, 1–41. [CrossRef]
8. Niere, J.; Schafer, W.; Wadsack, J.P.; Wendehals, L.; Welsh, J. Towards pattern-based design recovery. In Proceedings of the 24th International Conference on Software Engineering, Orlando, FL, USA, 19–25 May 2002.
9. Mayvan, B.B.; Rasoolzadegan, A. Design pattern detection based on the graph theory. *Knowl.-Based Syst.* **2017**, *120*, 211–225. [CrossRef]
10. Al-Obeidallah, M.G.; Petridis, M.; Kapetanakis, S. A survey on design pattern detection approaches. *Int. J. Softw. Eng.* **2016**, *7*, 41–59.
11. Rasool, G.; Streitfdert, D. A survey on design pattern recovery techniques. *Int. J. Comput. Sci. Issues* **2011**, *8*, 251–260.
12. Shahbazi, Z.; Rasoolzadegan, A.; Purfallah, Z.; Horestani, S.J. A new method for detecting various variants of GoF design patterns using conceptual signatures. *Softw. Qual. J.* **2021**. [CrossRef]
13. Tsantalis, N.; Chatzigeorgiou, A.; Stephanides, G.; Halkidis, S.T. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.* **2006**, *32*, 896–909. [CrossRef]
14. Sahoo, S.K. Social object—A software design pattern. In Proceedings of the IEEE 2nd International Conference on Software Engineering and Service Science, Beijing, China, 15–17 July 2011.
15. Bayley, I.; Zhu, H. Formal specification of the variants and behavioral features of design patterns. *J. Syst. Softw.* **2010**, *83*, 209–221. [CrossRef]
16. Gao, C. Application of design patterns to control system of digital photofinishing. In Proceedings of the Third International Symposium on Intelligent Information Technology Application, Nanchang, China, 21–22 November 2009.
17. Stencel, K.; Węgrzynowicz, P. Detection of diverse design pattern variants. In Proceedings of the 15th Asia-Pacific Software Engineering Conference, Beijing, China, 3–5 December 2008.
18. Rasool, G.; Akhtar, H. Towards a catalog of design patterns variants. In Proceedings of the International Conference on Frontiers of Information Technology (FIT), Islamabad, Pakistan, 16–18 December 2019.
19. Yarahmadi, H.; Hasheminejad, S.M.H. Design pattern detection approaches: A systematic review of the literature. *Artif. Intell. Rev.* **2020**, *53*, 5789–5846. [CrossRef]
20. Dwivedi, A.K.; Tirkey, A.; Rath, S.K. Applying learning-based methods for recognizing design patterns. *Innov. Syst. Softw. Eng.* **2019**, *15*, 87–100. [CrossRef]
21. Zein, S.; Rimawi, D. A static analysis of android source code for design patterns usage. *Int. J. Adv. Trends Comput. Sci. Eng.* **2020**, *9*, 2178–2186.
22. Liu, W.; Zhang, C.; Wang, F.; Yang, Y. Combining network analysis with structural matching for design pattern detection. In Proceedings of the Evaluation and Assessment in Software Engineering (EASE '20), Trondheim, Norway, 15–17 April 2020.
23. Guimaraes, E.; Cai, Y. Understanding software systems through interactive pattern detection. In Proceedings of the IEEE International Conference on Software Architecture Companion (ICSA-C), Salvador, Brazil, 2–6 November 2020.
24. Xiong, R.; Lo, D.; Li, B. Distinguishing similar design pattern instances through temporal behavior analysis. In Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020.
25. Oruc, M.; Akal, F.; Sever, H. Detecting design patterns in object-oriented design models by using a graph mining approach. In Proceedings of the 4th International Conference in Software Engineering Research and Innovation (CONISOFT), Puebla, Mexico, 27–29 April 2016.
26. Dong, J.; Sun, Y.; Zhao, Y. Design pattern detection by template matching. In Proceedings of the ACM Symposium on Applied Computing, Fortaleza, Ceara, Brazil, 16–20 March 2008.
27. Yu, D.; Ge, J.; Wu, W. Detection of design pattern instances based on graph isomorphism. In Proceedings of the IEEE 4th International Conference on Software Engineering and Service Science, Beijing, China, 23–25 May 2013.
28. Singh, J.; Gupta, M. Design pattern detection using Dpdetect algorithm. *Int. J. Innov. Technol. Explor. Eng.* **2019**, *8*, 2278–3075.

29. Pande, A.; Pant, V.; Gupta, M.; Mishra, A. Design patterns discovery in source code: Novel technique using substring match. *TEM J.* **2021**, *10*, 1166–1174. [CrossRef]
30. Huang, H.Y.; Zhang, S.S.; Cao, J.; Duan, Y.H. A practical pattern recovery approach based on both structural and behavioral analysis. *J. Syst. Softw.* **2005**, *75*, 69–87. [CrossRef]
31. Yu, D.; Zhang, Y.; Chen, Z. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *J. Syst. Softw.* **2015**, *103*, 1–16. [CrossRef]
32. Bernardi, M.L.; Cimitile, M.; Lucca, G.D. Design pattern detection using a DSL-driven graph matching approach. *J. Softw.-Evol. Process* **2014**, *26*, 1233–1266. [CrossRef]
33. Singh, J.; Chowdhuri, S.R.; Bethany, G.; Gupta, M. Detecting design patterns: A hybrid approach based on graph matching and static analysis. *Inf. Technol. Manag.* **2021**. [CrossRef]
34. Liu, C. A general framework to detect design patterns by combining static and dynamic analysis techniques. *Int. J. Softw. Eng. Knowl. Eng.* **2021**, *31*, 21–54. [CrossRef]
35. Rasool, G.; Mäder, P. A customizable approach to design pattern recognition based on feature types. *Arab. J. Sci. Eng.* **2014**, *39*, 8851–8873. [CrossRef]
36. Mohamed, K.A.; Kamel, A. Reverse engineering state and strategy design patterns using static code analysis. *Int. J. Adv. Comput. Sci. Appl.* **2018**, *9*, 568–576.
37. Martino, B.D.; Esposito, A. A rule-based procedure for automatic recognition of design patterns in UML diagrams. *Softw.-Pract. Exp.* **2016**, *46*, 983–1007. [CrossRef]
38. Alnusair, A.; Zhao, T.; Yan, G. Rule-based detection of design patterns in program code. *Int. J. Softw. Tools Technol. Transf.* **2014**, *16*, 315–334. [CrossRef]
39. Thongrak, M.; Vatanawood, W. Detection of design pattern in class diagram using ontology. In Proceedings of the International Computer Science and Engineering Conference (ICSEC), Khon Kaen, Thailand, 30 July–1 August 2014.
40. Ren, W.; Zhao, W. An observer design-pattern detection technique. In Proceedings of the IEEE International Conference on Computer Science and Automation Engineering (CSAE), Zhangjiajie, China, 25–27 May 2012.
41. Vokác, M. An efficient tool for recovering Design Patterns from C++ Code. *J. Object Technol.* **2006**, *5*, 139–157. [CrossRef]
42. Lucia, A.D.; Deufemia, V.; Gravino, C.; Risi, M. Design pattern recovery through visual language parsing and source code analysis. *J. Syst. Softw.* **2009**, *82*, 1177–1193. [CrossRef]
43. Lucia, A.D.; Deufemia, V.; Gravino, C.; Risi, M. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In Proceedings of the IEEE International Conference on Software Maintenance, Timisoara, Romania, 12–18 September 2010.
44. Costagliola, G.; Lucia, A.D.; Deufemia, V.; Gravino, C.; Risi, M. Design pattern recovery by visual language parsing. In Proceedings of the Ninth European Conference on Software Maintenance and Reengineering, Manchester, UK, 23 March 2005.
45. Lucia, A.D.; Deufemia, V.; Gravino, C.; Risi, M. Behavioral pattern identification through visual language parsing and code instrumentation. In Proceedings of the 13th European Conference on Software Maintenance and Reengineering, Kaiserslautern, Germany, 24–27 March 2009.
46. Hayashi, S.; Katada, J.; Sakamoto, R.; Kobayashi, T.; Saeki, M. Design pattern detection by using meta patterns. *IEICE Transactions on Information and Systems* **2008**, *E91.D*, 933–944. [CrossRef]
47. Wuyts, R. Declarative reasoning about the structure of object-oriented systems. In Proceedings of the Technology of Object-Oriented Languages, Santa Barbara, CA, USA, 3–7 August 1998.
48. Kramer, C.; Prechelt, L. Design recovery by automated search for structural design patterns in object-oriented software. In Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '96), Monterey, CA, USA, 8–10 November 1996.
49. Albin-Amiot, H.; Cointe, P.; Guéhéneuc, Y.-G.; Jussien, N. Instantiating and detecting design patterns: Putting bits and pieces together. In Proceedings of the 16th Annual International Conference on Automated Software Engineering, San Diego, CA, USA, 26–29 November 2001.
50. Guéhéneuc, Y.-G.; Antoniol, G. DeMIMA: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.* **2008**, *34*, 667–684. [CrossRef]
51. Guéhéneuc, Y.-G.; Albin-Amiot, H. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39), Santa Barbara, CA, USA, 29 July–3 August 2001.
52. Zhu, H.; Bayley, I.; Shan, L.; Amphlett, R. Tool support for design pattern recognition at model level. In Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, Seattle, WA, USA, 20–24 July 2009.
53. Wierda, A.; Dortmans, E.; Somers, L. Pattern detection in object-oriented source code. In Proceedings of the International Conference on Software and Data Technologies, Barcelona, Spain, 22–25 July 2007.
54. Mens, K.; Tourwé, T. Delving source code with formal concept analysis. *Comput. Lang. Syst. Struct.* **2005**, *31*, 183–197. [CrossRef]
55. Blewitt, A.; Bundy, A.; Stark, I. Automatic verification of design patterns in Java. In Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering, New York, NY, USA, 7–11 November 2005.
56. Issaoui, I.; Bouassida, N.; Ben-Abdallah, H. Using metric-based filtering to improve design pattern detection approaches. *Innov. Syst. Softw. Eng.* **2014**, *11*, 39–53. [CrossRef]

57. Guéhéneuc, Y.-G.; Guyomarc'h, J.-Y.; Sahraoui, H. Improving design-pattern identification: A new approach and an exploratory study. *Softw. Qual. J.* **2010**, *18*, 145–174. [CrossRef]
58. Kim, H.; Boldyreff, C. A method to recover design patterns using software product metrics. In Proceedings of the International Conference on Software Reuse (ICSR), Vienna, Austria, 27–29 June 2000.
59. Antoniol, G.; Fiutem, R.; Cristoforetti, L. Using metrics to identify design patterns in object-oriented software. In Proceedings of the Fifth International Software Metrics Symposium Metrics, Bethesda, MD, USA, 20–21 March 1998.
60. Antoniol, G.; Casazza, G.; Penta, M.D.; Fiutem, R. Object oriented design patterns recovery. *J. Syst. Softw.* **2001**, *59*, 181–196. [CrossRef]
61. Guéhéneuc, Y.-G.; Sahraoui, H.; Zaidi, F. Fingerprinting design patterns. In Proceedings of the 11th Working Conference on Reverse Engineering, Delft, The Netherlands, 8–12 November 2004.
62. Nazar, N.; Aleti, A.; Zheng, Y. Feature-based software design pattern detection. *J. Syst. Softw.* **2021**, *185*, 111179. [CrossRef]
63. Huang, X.-L.; Ma, X.; Hu, F. Editorial: Machine Learning and Intelligent Communications. *Mob. Netw. Appl.* **2018**, *23*, 68–70. [CrossRef]
64. Dwivedi, A.K.; Tirkey, A.; Rath, S.K. Software design pattern mining using classification-based techniques. *Front. Comput. Sci.* **2018**, *12*, 908–922. [CrossRef]
65. Detten, M.V.; Becker, S. Combining clustering and pattern detection for the reengineering of component-based software systems. In Proceedings of the Joint ACM SIGSOFT Conference—QoSA and ACM SIGSOFT Symposium—ISARCS on Quality of Software Architectures—QoSA and Architecting Critical Systems—ISARCS, New York, NY, USA, 20–24 June 2011.
66. Uchiyama, S.; Washizaki, H.; Fukazawa, Y.; Kubo, A. Design pattern detection using software metrics and machine learning. In Proceedings of the Fifth International Workshop on Software Quality and Maintainability (SQM2011), Oldenburg, Germany, 1–4 March 2011.
67. Uchiyama, S.; Kubo, A.; Washizaki, H.; Fukazawa, Y. Detecting design patterns in object-oriented program source code by using metrics and machine learning. *J. Softw. Eng. Appl.* **2014**, *7*, 983–998. [CrossRef]
68. Gupta, M.; Singh, S. Comparative analysis of software design patterns-based design metrics using machine learning algorithms. *Int. J. Comput. Eng. Technol.* **2018**, *9*, 32–41.
69. Dwivedi, A.K.; Tirkey, A.; Rath, S.K. Applying software metrics for the mining of design pattern. In Proceedings of the IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics Engineering (UPCON), Varanasi, India, 9–11 December 2016.
70. Dwivedi, A.K.; Rath, S.K.; Satapathy, S.M. Neural network-based patterns detection in object-oriented program. In Proceedings of the 9th Annual Information Technology, Electromechanical Engineering and Microelectronics Conference (IEMECON), Jaipur, India, 13–15 March 2019.
71. Dwivedi, A.K.; Rath, S.K.; Satapathy, S.M. Applying neural network to determine patterns in open-source software. In Proceedings of the IEEE 5th International Conference for Convergence in Technology (I2CT), Bombay, India, 29–31 March 2019.
72. Paakki, J.; Karhinen, A.; Gustafsson, J.; Nenonen, L.; Verkamo, A.I. Software metrics by architectural pattern mining. In Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), Beijing, China, 21–24 August 2000.
73. Chihada, A.; Jalili, S.; Hasheminejad, S.M.H.; Zangooei, M.H. Source code and design conformance, design pattern detection from source code by classification approach. *Appl. Soft Comput.* **2015**, *26*, 357–367. [CrossRef]
74. Chaturvedi, A.; Gupta, M.; Gupta, S.K. Design pattern detection using genetic algorithm for sub-graph isomorphism to enhance software reusability. *Int. J. Comput. Appl.* **2016**, *135*, 33–36. [CrossRef]
75. Chaturvedi, S.; Chaturvedi, A.; Tiwari, A.; Agarwal, S. Design pattern detection using machine learning techniques. In Proceedings of the 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 29–31 August 2018.
76. Gupta, M. Design pattern mining using greedy algorithm for multi-labelled graphs. *Int. J. Inf. Commun. Technol.* **2011**, *3*, 314–323. [CrossRef]
77. Fontana, F.A.; Zanoni, M. A tool for design pattern detection and software architecture reconstruction. *Inf. Sci.* **2011**, *181*, 1306–1324. [CrossRef]
78. Arcelli, F.; Christina, L. Enhancing Software Evolution through Design Pattern Detection. In Proceedings of the Third International IEEE Workshop on Software Evolvability, Paris, France, 1 October 2007.
79. Tonella, P.; Antoniol, G. Object oriented design pattern inference. In Proceedings of the IEEE International Conference on Software Maintenance, Oxford, UK, 30 August–3 September 1999.
80. Lebon, M.; Tzerpos, V. Fine-grained design pattern detection. In Proceedings of the IEEE 36th Annual Computer Software and Applications Conference, Izmir, Turkey, 16–20 July 2012.
81. Dietrich, J.; Elgar, C. A formal description of design patterns using OWL. In Proceedings of the Australian Software Engineering Conference, Brisbane, QLD, Australia, 29 March–1 April 2005.
82. Elaasar, M.; Briand, L.C.; Labiche, Y. A metamodeling approach to pattern specification. In Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MODELS 2006), Genoa, Italy, 1–6 October 2006.
83. Dietrich, J.; Elgar, C. An ontology-based representation of software design patterns. In *Design Pattern Formalization Techniques*; IGI Global: Hershey, PA, USA, 2007; pp. 258–279.

84. Blewitt, A. Spine: Language for pattern verification. In *Design Pattern Formalization Techniques*; IGI Global: Hershey, PA, USA, 2007; pp. 109–122.
85. Chidamber, S.R.; Kemerer, C.F. A metrics suite for object-oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [CrossRef]
86. Hernandez, J.; Kubo, A.; Washizaki, H.; Yoshiaki, F. Selection of metrics for predicting the appropriate application of design patterns. In Proceedings of the 2nd Asian Conference on Pattern Languages of Programs, Tokyo, Japan, 5–8 October 2011.
87. Derezińska, A. Metrics in software development and evolution with design patterns. In Proceedings of the Computer Science On-line Conference (CSOC2018), Vsetin, Czech Republic, 25–28 April 2018.
88. Mayvan, B.B.; Rasoolzadegan, A.; Ebrahimi, A.M. A new benchmark for evaluating pattern mining methods based on the automatic generation of testbeds. *Inf. Softw. Technol.* **2019**, *109*, 60–79. [CrossRef]
89. Vokac, M. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Eng.* **2004**, *30*, 904–917. [CrossRef]
90. Pettersson, N.; Löwe, W.; Nivre, J. Evaluation of accuracy in design pattern occurrence detection. *IEEE Trans. Softw. Eng.* **2010**, *26*, 575–590. [CrossRef]