The Journal of Supercomputing (2023) 79:1654–1682
https://doi.org/10.1007/s11227-022-04718-7

Probabilistic detection of GoF design patterns

Niloofar Bozorgvar[1] · Abbas Rasoolzadegan[1] · Ahad Harati[1]

Accepted: 11 July 2022 / Published online: 1 August 2022

Abstract
Detecting design patterns from source code of software systems can help to understand the structure and the behavior of the software systems. The better understanding of software systems is helpful in reengineering and refactoring. As software progression, refactoring has become more valuable. One way to reduce the refactoring costs is to detect design patterns. The key criteria for accurately detecting design patterns is signatures. Achieving fine signatures is not an easy forward task. Instead of improving signatures, more accurate detection can be achieved by having probabilistic viewpoints. Since each of the design patterns has variants or may be implemented differently, having a probabilistic approach in detection can increase coverage as well as help in software refactoring. In this study, the main purpose is to identify the design patterns in source code with a non-crisp approach and measuring the possibility of the presence of the design patterns in the source code. Considering main body of design patterns and their corresponding signatures, design patterns are represented as appropriate features. We try to get features from design pattern signatures that do not change in the face of variations that occur during implementation. Then, through these features, the probability of presence of the roles forming the design patterns is determined, using neural network and regression analysis. After this step, using probabilistic graphical models the probability of presenting design patterns in source code is measured. The results of the proposed method show the similarity of each code to the design patterns in the range between 0 and 1. The results of other valid methods are a subset of the results of proposed method. Results that are 50% to 100% similar to the design patterns are presented in the evaluation section.

Keywords GoF design patterns · Design patterns detection · Regression · Probabilistic

✉ Abbas Rasoolzadegan
rasoolzadegan@um.ac.ir

Extended author information available on the last page of the article

1 Introduction

Software design patterns are well-known solutions to repetitive software design problems. Using design patterns makes it possible to reuse the experience of experts in designing software systems. Consequently, in recent decades design patterns and their usefulness have received much attention [1–3]. Using software design patterns has many advantages. One of their most important benefits is to help inexperienced people in software design. Design patterns have increasing qualitative characteristics such as reusability, modularity, understandability, maintenance, efficiency, and flexibility. All these characteristics produce an improvement in software quality when using them [4–6].

After software implementation, accessing the knowledge related to the design patterns used in software design is not an easy task. Due to the lack of documentation, essential design decisions may be lost. Detecting design patterns is an important task in analyzing software system code. It helps us to understand the code, return to initial design decisions, and system re-engineering. More importantly, it enables the identification of crucial design decisions [1, 3, 7, 8]. Adding new functionality, correcting, improving, and fixing errors in the existing design can save a great deal of software development costs. It also enhances the understandability and extensibility of the system [9]. Developers implement software design patterns in different ways. Diverse implementations make it difficult to detect the design patterns used in the source code and harm the accuracy of design patterns detection methods. The main reason for the differences observed in the results of existing detection methods is the different implementations of design patterns [1, 9, 10].

The basis of software quality engineering is utilizing different mechanisms and processes, and to produce a high-quality software product these mechanisms should be apply [11]. Engineering means quantitative measurement using relevant metrics and criteria. On the other hand, design patterns affect various aspects of quality, such as software reusability. Therefore, engineering design patterns in code (such as specifying the number of each design pattern instances presenting in the software system code) can be a practical step for quantifying software quality. Quantitative measurement of quality is an essential factor for quality enhancement and the basis of software quality engineering. For these reasons, taking a probabilistic look at design patterns detection can have a significant effect on the reverse engineering of a software system, thus helping improve software quality.

As current design patterns detection methods does not cover variant use of design patterns, selecting unfit features and metrics, and having a deterministic approach in detecting design patterns, the real instances of design patterns get lost in the detection phase. However, the probability-based approach can detect the presence of design patterns in code. Some parts of code may have minor similarities to design patterns, while other parts of code may be very similar to the body of design patterns. Detecting similar instances is helpful in the refactoring process and reducing costs. As a result, probabilistic detection of design patterns can help in refactoring the software system with its ability to estimate the similarity between design patterns and each section of code. On the other hand, by improving the features and metrics used in design patterns detection process, the accuracy of the detection methods can be further increased.

The purpose of this paper is to provide a way for identifying design patterns in source code probabilistically. In the first phase, design patterns are represented as a set of features suitable for detecting and distinguishing between different roles of design patterns. Then these features are automatically measured in code. Next, using machine learning techniques named MLP and regression, plus measured features from the previous step, we developed a trained MLP model to identify the roles of design patterns and their probability of being derived from source code. Finally, in the second phase, using probabilistic graphical model as an inference system, the probability of each design pattern being presented in each section of the source code is obtained.

This study is suitable for all categories of Gang of Four (GoF) design patterns [12]. Furthermore, our experiments are conducted on real-world open-source software systems, including JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7. The coverage of proposed method is evaluated by comparing the results with other design pattern detection methods. The results of the experiments show an improvement in detection phase.

Briefly, the main contributions of this study for design pattern detection are: (1) having a probabilistic approach in detecting design patterns, (2) better distinguishing between design patterns using regression by representing patterns as the set of features, which are the concepts of design patterns signature, and (3) Covering the variants of design patterns and different implementations.

The rest of the paper is organized as follows: Sect. 2 defines related work on design pattern detection. In Sect. 3 we present different steps of proposed method. Section 4 covers our experiments on three open-source systems. Lastly, the threats to validity and future work are discussed in Sect. 5 and Sect. 6, respectively.

2 Related work

The most remarkable subject of design pattern research is detection [13]. The methods proposed for design pattern detection can be subdivided into three main categories based on their search method: (1) Intermediate model-based methods, (2) Reason-based methods, and (3) Metric-based methods. In the following, we elaborate on each of these sub-categories in more detail.

Intermediate model-based methods: In summary, the most critical methods that fall into this category are: 1) Similarity Scoring Methods, (2) Parsing-based methods, and (3) Query-based methods.

- *Similarity Scoring Methods:* These methods model the structural information of the software system under study as well as design patterns into a suitable intermediate representation. The model that most of these methods have chosen as the intermediate model is a graph or matrix. Then they try to match the structure of software system with the structure of each design pattern [14–17]. These methods are usually unable to distinguish between structurally similar design patterns [15, 16, 18, 19].

- *Parsing-based methods:* Other methods for design pattern detection that use the intermediate model usually map the programming language grammar for each design pattern to its corresponding graph. These methods are highly accurate but are limited to structural patterns [20, 21]. In some studies, authors have attempted to address this limitation by considering behavioral aspects of design patterns [7, 22, 23].
- *Query-based methods:* Other methods of design pattern detection use intermediate models such as AST, ASG, XMI, UML, and ontology structures. These methods are modeling the software system, as well as modeling the design patterns to the intermediate models. Then use the database and query to extract information related to the design patterns and detect them [24][24]. Ontology-based approaches examine the input model semantically and then use semantic queries to detect design pattern instances [23, 24, 26, 27]. The efficacy of these methods is limited to the information available in the middle representation.

  Reason-based methods: In summary, the most critical methods that fall into this category are: (1) Constraint satisfaction-based methods, (2) Formal methods and (3) miscellaneous methods.
- *Constraint satisfaction-based methods:* Constraint satisfaction-based methods use constraint-based programming to find design patterns instances. These methods first turn the detection problem into a constraint-satisfying problem and then describe design pattern instances as constraint-based systems. In these systems, each role is represented as a variable and the relationship between these roles is displayed as a constraint between the corresponding variables [28, 29]. Constraint satisfaction-based methods have high recall but low precision.
- *Formal methods:* These methods use logical and mathematical methods to detect design patterns in software system code. The accuracy of formal methods is not better than the other methods. These methods suffer from a high volume of computation, and are also unable to find patterns that include more than a specified number of classes [30–32].
- *Reasoning-based methods:* These methods are divided into two main groups: logical reasoning and fuzzy reasoning. Logical reasoning methods first present the search conditions and then try to detect the design pattern. These methods usually use techniques such as backtracking and database function. The main limitation of these methods is the lack of detection of approximate matching [33]. The methods in the fuzzy reasoning group present the design patterns as fuzzy reasoning networks. These networks represent the rules required for finding micro-architectures similar to design patterns. Fuzzy reasoning methods can identify uncertain states but suffer from high false positive rates [34, 35].

Metric-based methods: These methods measure the class level metrics such as the number of attributes, types of operations, dependency, association, and inheritance relationships. Then they use different mechanisms to compare the values of metrics with the expected values for a design pattern. The expected values for each pattern are obtained from the description of that pattern. These methods are usually supported by machine learning techniques. Metric-based methods contain a filtering phase, so they are computationally efficient. These methods usually examine the code statically [36, 37, 38, 39, 40, 41].

According to analyses, detecting design patterns is not a straightforward task [3, 14]. Table 1 contains some of the most important criteria for evaluating a detecting design patterns method [3].

In the following, Table 2 compares some of the researches conducted in the design pattern detection field. These studies are one of the most successful design pattern recovery tools and they are reviewed based on mentioned criteria.

By analyzing the studies conducted in the design patterns detection field, we have concluded that in general, metric-based methods, as well as machine learning-based methods, achieve more accurate results in detecting design patterns. These methods often have been able to detect design patterns' variants more than other methods. As a result, we have also chosen and considered the techniques used in these approaches. By improving the criteria and using probabilistic models, we try to measure the presence of design patterns in the code.

3 The proposed method

In this section, we propose proposed method and explain its details. Figure 1 demonstrates Schematic Overview of the Proposed Method that is divided into two phases. Learning phase (Phase 1): This phase aims to create a trained MLP model by determining appropriate features. Detection phase (Phase II): Following the Learning phase, a trained model is already obtained, by which in this phase, the design patterns from source code will be detected probabilistically. Sections 3.1 and 3.2 describe the steps of each phase in detail.

We focused on GoF design patterns because (1) The signatures of GoF patterns are available [14] and (2) A large number of GoF patterns' variants are studied in the

Table 1 The most important criteria for evaluating a design patterns detection method [3]

Feature	Explanation
Accurate	Achieve appropriate accuracy along with acceptable recall
Low time complexity	Low time complexity and independence of the algorithm from the input size
Covering variants of patterns	Automatic implementation of the method without human intervention
Pattern-type independence	Independence of the source code implementation language
Language-type independence	Detect different software patterns regardless of their type and number of roles
Ability to differentiate	Distinguish between structurally (behaviorally) similar patterns
Automatic	Identify pattern instances in different implementation modes
Appropriate evaluation	Evaluate the detection method using appropriate criteria
(Use of accuracy and recall metrics)	

Table 2 Comparison of GoF design patterns detection methods

#Ref	Accurate	Low time complexity	Covering variants of patterns	Pattern-type independence	Language-type independence	Ability to differentiate	Automatic	Appropriate evaluation	Description
GTM [14]	✓	✗	✓	✗	✗	✓	✗	✓	Cover patterns that are out of standard Focus on GoF patterns Semi-automatic
Sempatrec [24]	✗	✗	✓	✗	✓	✓	✗	✓	Focus on GoF patterns Low accuracy in pattern detection Cover patterns that are out of standard Semi-automatic (user must define different types of patterns)
DEMiMA [27]	✗	✗	✓	✗	✓	✗	✗	✓	Not able to detect structurally similar patterns Low accuracy in detection Focus on GoF patterns Semi-automatic
SSA [42]	✗	✓	✗	✗	✓	✗	✓	✓	Low accuracy in detection Focus on GoF patterns Fully-automatic
[43]	-	✓	✗	✗	✗	✓	✓	✓	Ignore patterns that are out of standard Focus on GoF patterns Java language-dependent
Mbf-PD [31]	✗	✓	✓	✗	✓	✓	✗	✓	Use of semantic metrics Ignore patterns that are out of standard
[44]	✓	✓	✓	✗	✗	✓	✓	✓	Focus on GoF patterns Java language-dependent
[15]	✗	✓	✗	✗	✗	✗	✓	✓	Depending on pattern type (For structural patterns) Java language-dependent Ignore patterns that are out of standard

Table 2 (continued)

#Ref	Accurate	Low time complexity	Covering variants of patterns	Pattern-type independence	Language-type independence	Ability to differentiate	Automatic	Appropriate evaluation	Description
[45]	✗	✗	✓	✗	✓	✗	✓	✓	Limited to 4-element patterns Not able to detect structurally similar patterns High time complexity (check all combinations of 4 input system classes)
[46]	✗	✓	✓	✗	✗	✓	✗	✓	Focus on GoF patterns Java language-dependent Semi-automatic

✓: Having the feature

✗: Lack of the feature

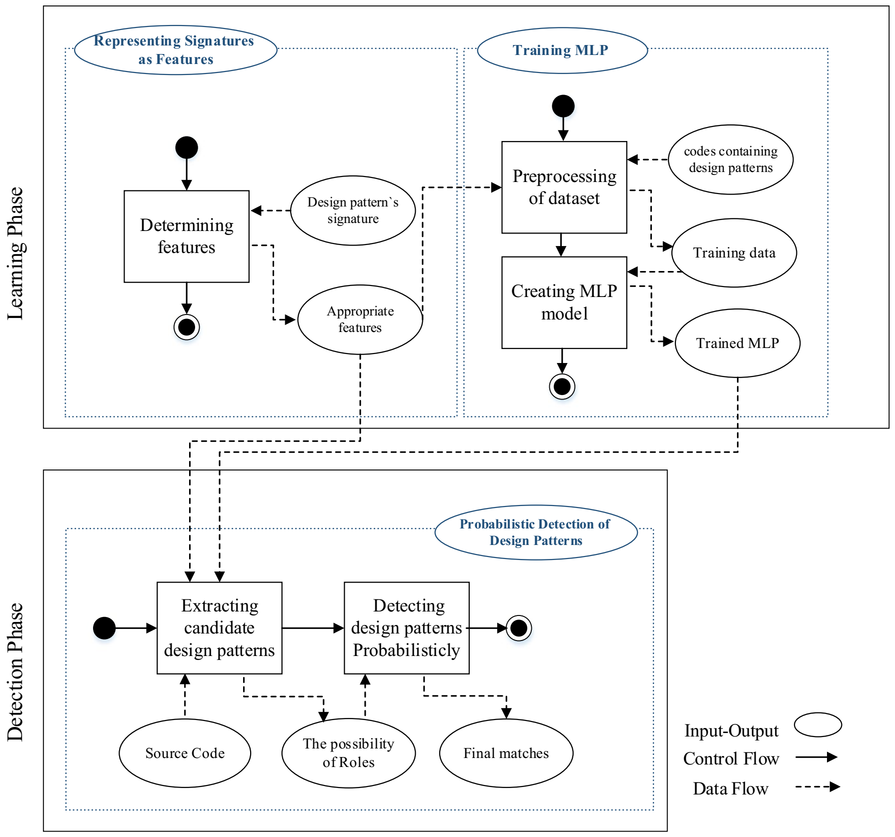


Fig. 1 A Schematic Overview of the Proposed Method

previous researches [47]. GoF deign patterns are divided into three category namely Creational, Structural, and Behavioral [12]. To confirm that the proposed method in this study is efficient for detecting patterns' variant instances of each category, we have selected frequently used patterns of each category in open-source projects. These selected patterns are Singleton, Factory Method, Adapter, Composite, Observer, and State/Strategy and their variants. The first phase was performed only once, and then the second phase, detection of design patterns with probability functions, will be executed each time to detect patterns from source code.

3.1 Learning phase (Phase I)

In this section, we explain the design patterns classifier. As shown in Fig. 1, the phase of learning the classifier consists of two steps. In the first step, a list of features is determined and then in the second step, a trained MLP model is created by measuring the features on a set of design pattern implementations.

Table 3 The Signature of the Composite Design Pattern

Composite	
Constraints	1. *Association* from Composite to Component with *multiplicity* (Composite has a set of objects) 2. Has *inheritance* relationship (Composite, Component) 3. Composite has at least one *overridden method* (the operation is overridden in Composite class and called by it) 4. Has *delegation* in the loop

Table 4 The names of the studied variants

Pattern	Pattern`s variants
Singleton	Eager Instantiation, Lazy Instantiation, Replaceable Instance
Factory method	Delegated construction using factory method, Multiple Product Types Inside Factory Class, Parameterized Factory, Default Product Implementation,
Composite	Pluggable Adapters, Two Way Adapters
Adapter	I-N Relationship using arrays, Composite Implementation using Typical Array Objects of Leaves
Observer	Multiple Instance Observer
State/ Strategy	Flexible Strategy Pattern

3.1.1 Representing design pattern`s signature as features (Step I of the Learning phase)

Since design patterns have abstract descriptions, defining them with a lower level of abstraction such as a signature, results in better understanding. Signatures can indicate the behavioral and structural aspects of design patterns for example the class level information, the number of roles, the interactions between roles, etc. Using the concepts in signatures, design patterns are represented as a set of features and thus achieving more accuracy in pattern detection. It is noteworthy that in this paper, the research presented by B.B Mayvan and A. Rasoolzadegan [14] is considered as the basic study for defining the signatures. Additionally, one of the signatures presented therein, Composite pattern, is improved. Table 3 illustrates the signature of the composite design pattern.

Also, we have used the variants presented in Research [47] as a reference. The names of the studied variants in our research are also announced in Table 4.

After defining the signature of each design pattern, the appropriate features are specified. By appropriate features, we imply features that allow us to differentiate among design patterns well and therefore result in acceptable accuracy for identified design patterns. Corresponding signatures of each design patterns are used to produce these features. The input for this step is signatures obtained in the previous step. Key concepts are used in signatures to specify appropriate features. These key concepts are the explicit properties expressed for design patterns in the signatures.

Then the signature of each design pattern is analyzed to extract key concepts. For each signature all the key concepts are extracted and eventually, a set of features is obtained. Each design pattern can be represented as the specified feature. The 9 features are shown in Table 5.

Although some of the object-oriented metrics [5] are too high or too low, in other words they do not have useful semantics, but our features obtained from the key concepts in the design pattern signatures are consistent with some of the object-oriented metrics. Our features enable us to interpret values correctly. The resemblance of our features and some of object-oriented metrics are shown in Table 6.

The ten features selected in this phase are measured for the roles that form these six design patterns. These design patterns consist of 14 roles in the main body naming Singleton, Creator, ConcreteCreator, ConcreteProduct, Context, State/ Strategy, ConcreteState/ ConcreteStrategy, Target, Adapter, Adaptee, Composite, Component, Observer, and Subject. Also, there are five subsidiary roles Consisted of Product, Client, Leaf, ConcreteObserver, and ConcreteSubject. These ten features are measured on the principal roles expressed in signatures corresponding to the selected design patterns and the subsidiary roles. The output for this step is the training data which will be used as input for the MLP. We use he syntax tree as a middle model for measuring the features, and convert the entire code structure into a syntax tree. All the code details are modeled using the syntax tree, which makes extracting features' values possible. For this purpose, Antler is used to generate a parser for converting Java source code into a syntax tree.

In this step, in order to create the dataset, feature vectors for every instance are constructed by measuring feature (mentioned in Table 5). First, the 10 features are measured for each role. Then the related flags are used to fulfil the feature vectors for training the MLP. They are measured for all the roles in the design pattern instances. Each instance is presented by features vectors and the design pattern name as a label.

3.1.2 Training MLP (Step II of the Learning phase)

In this step, we train a model of a MLP to detect individual design pattern roles. This step required two inputs: 1) The values of the features corresponding to each role of the design patterns in the code given as output of the previous step (training data), and 2) The labels for the feature vectors. The output of this step is a trained multilayer perceptron model. This step consists of two sub-steps, which are explained in the following. The process of creating a trained model is shown in Fig. 2.

- *Pre-processing sub-step:* As shown in Fig. 2, a pre-processing sub-step for labeling training data is first performed to access the trained neural network. As we seek to estimate and predict the similarity of each class under study to the roles that constitute design patterns, we face the problem of regression rather than classification. Therefor the corresponding labels on the data need to include the similarity of each role to the other roles so that the neural network is well trained for estimation For this purpose, the percentage of similarity between roles is taken into account, and is included in the labels corresponding to each class. For example, the role of Composite in the Composite

Table 5 Set of Specified Features to Identify Design Patterns

#No	Feature	Description	Required Condition	Flag Value
1	Association	Investigating the association relationship between the class under Study with other classes	1.1 If this class has an association to another class	F1 = 1
			1.2 If there is an association to this class from another class	F1 = 2
			1.3 Both	F1 = 3
			1.4 Otherwise	F1 = 4
2	Dependency	Investigating the dependency relationship between the class under study with other classes	2.1 If this class has a dependency to another class	F2 = 1
			2.2 If there is a dependency to this class	F2 = 2
			2.3 Both	F2 = 3
			2.4 Otherwise	F2 = 4
3	Inheritance	Investigating the existence of an Inheritance relationship between the class under study and other classes	3.1 If this class is a parent but not a child	F3 = 1
			3.2 If this class is a child but not a parent	F3 = 2
			3.3 If this class is a child and parent	F3 = 3
			3.4 Otherwise	F3 = 4
4	Delegation	Checking for delegation in the class under study	4.1 If there is delegation in class	F4 = 1
			4.2 Otherwise	F4 = 2
5	Overridden Method	Checking for existence of at least one override method in the class under study	5.1 If this class is child class & has at least one overridden method	F5 = 1
			5.2 Otherwise	F5 = 2
6	Method Call	Investigating the existence of method call in the class under study with other related classes	6.1 If there is a method in this class calling another class`s method	F6 = 1
			6.2 If there is a method in this class called from another class`s method	F6 = 2
			6.3 Both	F6 = 3
			6.4 Otherwise	F6 = 4
7	Types Of Constructor	Investigating the types of constructor in the Class under study (Public / Non-Public)	7.1 If this class has non-public constructor	F7 = 1
			7.2 Otherwise	F7 = 2

Table 5 (continued)

#No	Feature	Description	Required Condition	Flag Value
8	Types Of Method	Investigating the types of method in the class under study (Static/ Public/ Public Static/ Otherwise)	8.1	F8 = 1
			8.2 Otherwise	F8 = 2
9	Return Types Of Method	Checking the return types of method in class under study	9.1 The method return an object (User Defined Objects)	F9 = 1
			9.2 Otherwise	F9 = 2
10	Type of Class	Checking the type of class in the class under study	10.1 If this class is abstract class or an interface	F10 = 1
			10.2 If this class is a concrete class	F10 = 2
			10.3 Otherwise	F10 = 3

Table 6 the resemblance of our features and object oriented features

Feature	Object Oriented Metrics	Feature	Object Oriented Metrics
Association	CBO, NOED	Method Cal	CBO
Dependency	CBO, NOED	Types Of Constructor	RFC
Inheritance	MNOL, DIT, NOCC, DOIH	Types Of Method	NOO
Delegation	AOFD, NOED, NORM	Return Types Of Method	WMPC
Overridden Method	NOOM	Method Cal	CBO

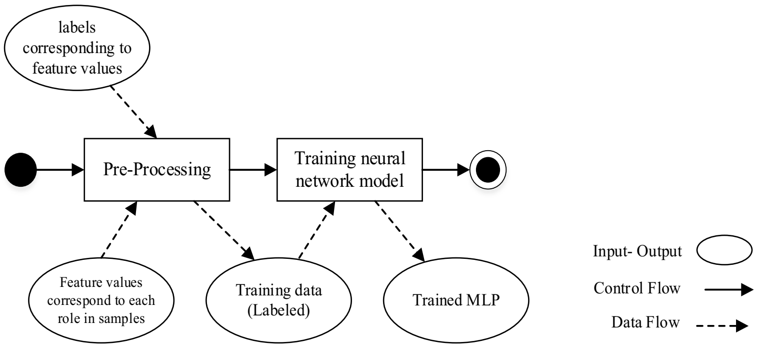

labels corresponding to feature values
Pre-Processing
Training neural network model
Feature values correspond to each role in samples
Training data (Labeled)
Trained MLP
Input- Output
Control Flow
Data Flow

Fig. 2 The Process of Creating a Trained Model

pattern and the role of the Adapter in the Adapter pattern are somewhat similar. The features related to these two roles are shown in Table 7.

As shown in Table 7, these two roles share some common features. Adapter role that associates with another class, satisfies 3 of the five features correlated with the Composite role. Therefore, about 60% of the Adapter's role, can be similar to the Composite role. It is worth noting that we measure features specified in the previous step on design patterns' main roles and also for all possible and constituent roles of design patterns; eventually including them as training examples. Then, the labels corresponding to each training example are identified, and the similarity of the specified roles to each label is considered.

- *Neural Network training sub-step:* After the pre-processing sub-step and obtaining the training data and corresponding labels, a training model from the neural network is developed. Finally, with this trained model, it is possible to obtain the presence of any roles forming the design patterns and also estimate the similarity of each role to the others. To achieve the goal of this step and obtaining a trained model for predicting and estimating the probability of design patterns' roles, a multilayer perceptron neural network was used to perform regression. This can determine and predict the similarity of each Java class to the design patterns' roles learned by the multilayer perceptron model.

Table 7 Features Related to Composite and Adapter Roles

DP	Roles in DP	General description of role structure and behavior	Features related to the roles
Composite	Component	Has a child class Associate from other classes or child classes with this class	**Inheritance** ** **Association** ** Overridden method **Method call** ** Delegation
	Composite	Is child class Having an associate relationship to another class Has at least one rewritten method There is a method in this class that calls a method from another class Has delegation	
Adapter	Target	Has a child class Existence of a method in this class that is called by a method in another class Dependency relationship to this class	**Inheritance** ** **Association** ** Dependency **Method call** **
	Adaptee	Has a child class Existence of a method in this class that is called by a method in another class (different signature) Association or dependency from other classes	
	Adapter	Is child class Having an association or dependency with another class There is a method in this class that calls a method from another class	

**: Common features between roles

Table 8 The Role Forming the Body of the Design Patterns

Design Pattern	Roles of Design pattern			
Singleton	Singleton			
Composite	Componen	Composite	Leaf	
Strategy/ State	Context	State/ Strategy	ConcreteState	
Adapter	Target	Adaptee	Adapter	Client
Factory Method	Creator	ConcreteCreator	ConcreteProduct	Product
Observer	Subject	Observer	ConcreteSubject	ConcreteObserver

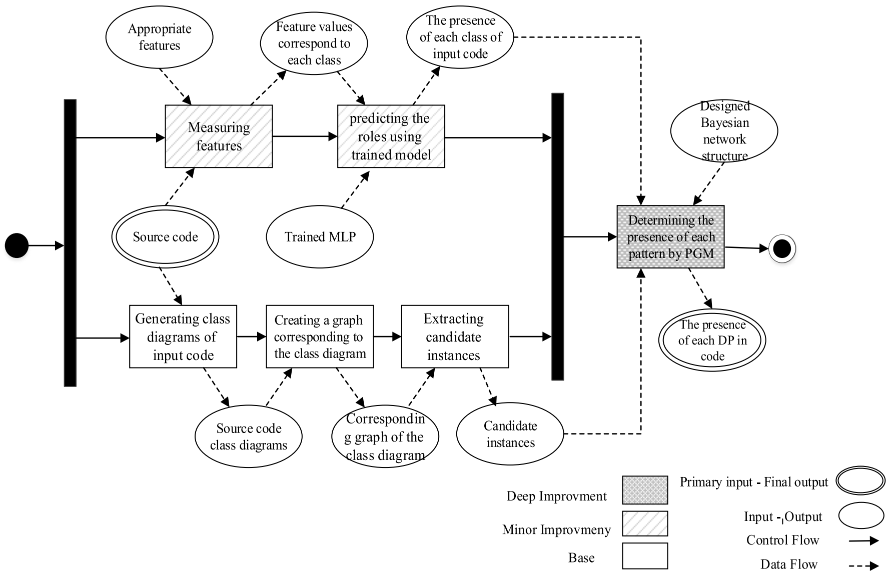

Appropriate features
Feature values correspond to each class
The presence of each class of input code
Measuring features
predicting the roles using trained model
Designed Bayesian network structure
Source code
Trained MLP
Determining the presence of each pattern by PGM
Generating class diagrams of input code
Creating a graph corresponding to the class diagram
Extracting candidate instances
The presence of each DP in code
Source code class diagrams
Corresponding graph of the class diagram
Candidate instances
Deep Improvment
Minor Improvmeny
Base
Primary input - Final output
Input - Output
Control Flow
Data Flow

Fig. 3 The Phase of Probabilistic Detecting of Design Patterns from Source Code

The number of inputs for this network is 10, the same as the number of features specified in the second step of the first phase. These features have been measured on 100 design patterns for the six design patterns individually: Singleton, Observer, Factory Method, Composite, Strategy /State, and Adapter (a total of 600 instances). These ten features have been measured separately for each constituent roles of the six design patterns. These patterns have a total of 19 roles in their body. As a result, the perceptron network has 19 outputs and by using the regression analysis, predict the similarity between each role. Table 8 shows the roles that constitute these patterns.

After measuring the features of these roles in the data, each data is labeled. Eventually, the data and their corresponding labels can be used as training data. The multilayer perceptron has a hidden layer with 12 neurons. The first layer of the network (input layer to hidden layer) uses the Log-Sigmoid function and the second layer of the network (hidden layer to output layer) uses the Tan-Sigmoid transfer function. The multilayer perceptron network has feedforward architecture, and the backpropagation algorithm is used on the network. Mean squared error is also used to estimate the error rate of the trained model.

3.2 Detection phase (Phase II)

This phase is performed each time to determine the probability of design patterns being present in the source code. The input for this phase is the source code under study and the output is the probability of design patterns in the code studied. Details of this phase are shown in Fig. 3.

In this phase, as shown in Fig. 3, the source code is initially received as input. After receiving the source code, several activities are performed in parallel on the source code. First, using the Class Visualizer tool, the code of software system under study is converted to an equivalent class diagram. Then, the class diagram is converted to an equivalent graph, and the graph gets enriched. The next step is to extract the candidate examples of the design patterns from the graph. Candidate examples are all three or four connected nodes of a graph that are connected. The algorithm for selecting candidate samples from the source code class diagram is illustrated by Algorithm 1. For this algorithm we have used the method proposed in [5] with some modifications.

Algorithm: Algorithm for generating candidate samples from source code class diagrams

Algorithm 1: Algorithm for generating candidate samples from source code class diagrams

```
Input: Source code class diagrams
Output: Candidate design pattern instances
    1) Transform given source code class diagrams to a graph G
    2) Enrich G by adding new edges representing parent`s relationships to children according to class
       diagrams (for edge with inherent relationships)
    3) Search all connected subgraphs with b number of vertices from G as candidate design pattern
       instances
```

The input for this algorithm is the source code class diagram. First, the source code class diagram is converted to graph G. Then, this graph is enriched by adding new edges showing the inheritance relationship between parents and children according to the class diagrams (to include the transitive relation). In the next step, all connected subgraphs of the G graph containing b vertices ($b=3, 4$) are identified as candidate instances of design patterns from the source code class diagram. Outputs of the algorithm are candidate instances of design patterns with b roles separately. This output will be used to determine the probability of design patterns being present in the source code by utilizing probabilistic graphical models.

On the other hand, after receiving code of the system under study, ten features are measured for each classes in the code. Then through the trained MLP, we estimate the similarity between these classes to each of the 19 roles in Observer, Factory Method, Composite, Strategy /State, and Adapter design patterns. Since the Singleton pattern has only one role at its body, estimating the presence of this pattern gets completed in this step. For other roles, we estimate this factor using the inference made by designed probabilistic graphical models, and examine the probability of it in each piece of code. For these six design patterns, a probabilistic graphical model corresponding to them is developed, a Bayesian network that uses generalized linear models and, logistic conditional distributions. We use the sigmoid function to have continuous input and output of real values and a probability number between zero and one.

As mentioned earlier, Bayesian networks are a subset of probabilistic graphical models represented by acyclic directed graphs. Bayesian networks are used to express the causal relationship between random variables and also their independent relationship. These networks are used to identify possible relationships and to predict and estimate. Since there is a causal relationship between the roles forming the design patterns and to produce a design pattern, each of these constituent roles is needed, so a Bayesian model for the six cases of design patterns is created. The structure of the Bayesian model designed using the generalized linear model is shown in Fig. 4.

For the design of this Bayesian network, we draw inspiration from the BN2O. Although the model designed in this study differs from the BN2O model, the network helped designing the framework. One of the current applications of the BN2O model is in medical field, such as investigating the relationship between fever and various diseases. As mentioned earlier, in this study, we used the BN2O model for designing the Bayesian network framework by generalized linear models and estimating the presence of design patterns in the code. Generalized linear models are a set of models created due to the independence of causes. The independence of

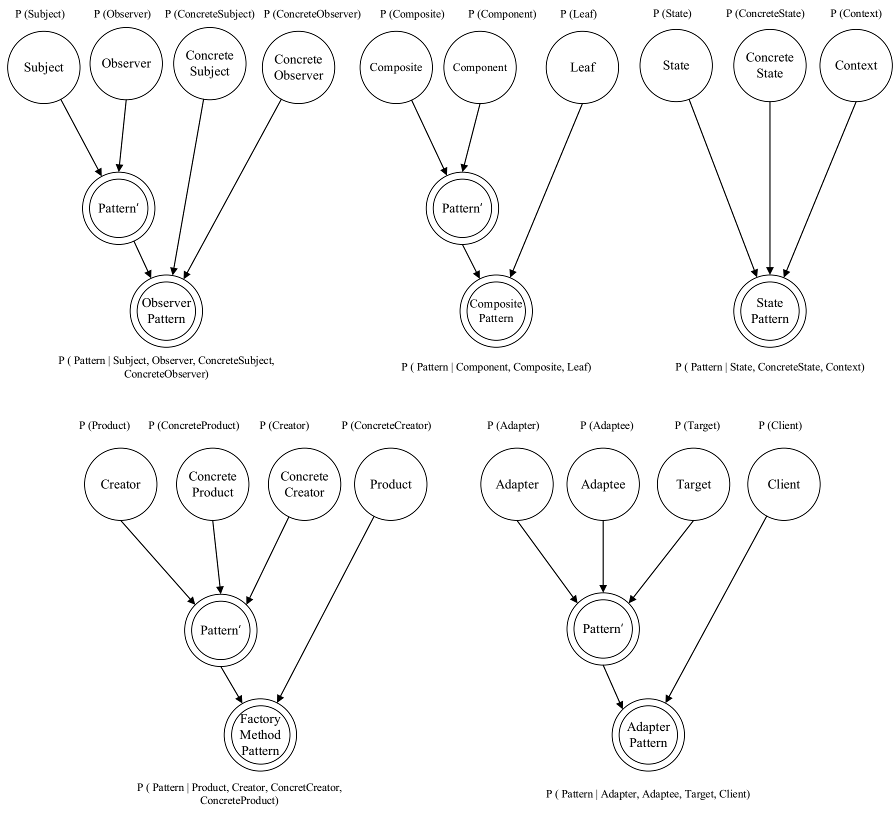


Fig. 4 Bayesian Network Structure with Generalized Linear Model for Six Design Patterns

causes meaning that if we consider the variable Y and the values of X_1, X_2, …, X_k as a parent, in the generalized linear model, it is assumed that the effect of each X_i on Y is independent of the other parent and It is linear. So assuming inputs are active for value one and inactive for value zero, the sum of the stimuli applied is:

$$f\left(X_1 + X_2 + X_3 + \ldots + X_k\right) = w_0 + \sum_{i=1}^{k} \left(w_i X_i\right) \tag{1}$$

W_i's determine the intensity of each parent's impact. W_0 also indicates the likelihood of a leak, meaning that if either parent is inactive, the probability may still be some.

As illustrated in Fig. 4, we seek the possibility of a design pattern if the roles that constitute the pattern are looked at. The neural network trained by regression obtains the probability of design patterns' constitutive roles. When designing this Bayesian network, both the main body roles and the subsidiary roles of design patterns have been considered so that in cases where there exists a boundary in the probability found, these subsidiary roles can be helpful in decision-making and inference.

As shown in Fig. 4, an intermediate variable is initially considered to estimate the presence of a design paradigm whose parent variables affecting these intermediate variables are the same constituent roles of the main body of the design patterns. The cumulative function in this intermediate variable is the "Min" operator or the "And" operator. Apart from the independent influence of the parent on the variable, each role of the main body is necessary to achieve the design pattern. We also consider the impact of subsidiary roles on the presence of design patterns. The cumulative function used in this part is the "Addition" operator. Finally, a smooth model called sigmoid is used to determine the probability.

In this process, generally, after estimating the presence of the roles forming the design patterns by regression, and simultaneously identifying the classes associated with each other from the class diagram corresponding to the code under study, the classes for each of which are regressed over 50% similarity to each of the 19 roles being investigated, along with other related classes identified by the corresponding graph of the code, the corresponding Bayesian model is designed to model the probability of that class being present inferences are drawn and finally the probability of the design pattern being represented in the source code is expressed separately.

4 Evaluation of the proposed method

This section evaluates the proposed method and its steps. As we already mentioned, the proposed method consists of three phases. In the first phase, it is designed to represent design patterns to appropriate features. To this end, the signatures of design patterns were identified, which, as mentioned earlier, have been used in the paper presented by B.B Mayvan and A. Rasoolzadegan [14], and only some modifications were made. In this section, the learning phase and the output of the probabilistic detection phase are evaluated.

4.1 Evaluation of the Learning phase of the proposed method

In this section, the multilayer perceptron network is evaluated. After designing and training the MLP, we evaluate the Regression learning phase, and discuss the error rates for learning, testing, and validation. To create this trained model of the MLP, the code and the design patterns generated by the testbed [48] was used. To this end, and for having a balanced number of training data, 100 design pattern instances (a total of 1900 training samples) for each of the six design patterns, comprising a total of 19 roles were used. 70% of these instances were used as training data, 15% as test data, and the remaining 15% as validation data. This ratio is based on the amount of training data, testing, and validation of the ratios are used in the field. The number of training samples for the six evaluated models is shown in Table 9.

After training the MLP model to apply regression, the performance of this model is examined. The performance of the trained model is shown in Fig. 5.

As illustrated in Fig. 5, the trained model has a good performance, and the errors are reduced and eventually converged. In addition, the test rate is lower than the training error and the validation error, which indicates a good performance for the trained model.

One of the most valuable methods and criteria for determining the suitability of the regression function is to examine the mean square error or MSE. In mathematics and statistics, the mean square error is a method of estimating the error rate that examines the difference between the target values and what is estimated. So in MSE, which always has a positive value (not zero), the lower value we have, the lower error rate we get. The MSE formula is as follows:

$$\mathrm{MSE} = \frac{1}{n}\sum_{n=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \tag{2}$$

In the above formula $\frac{1}{n}\sum_{n=1}^{n}.$ performs the averaging operation and $\left(Y_i - \hat{Y}_i\right)^2$ calculates the square value of the error for each data set. The MSE value for the training model in this study is 0.0027, which is close to zero, indicating a good performance in the regression performed by the trained model. We had run the neural network many times with different hyper-parameter to achieve stable and best results as possible. We briefly illustrated the ranges in Table 10.

4.2 Evaluation of the detection phase of the proposed method

The proposed method is examined on five open source software systems: JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7. The reason for using these three projects for evaluation is that these three projects have been widely used in the field literature and help to compare the proposed method with other existing methods. Due to the availability of source code for these three projects, some experts have identified different design patterns. Thus, they are beneficial in evaluating design patterns detection tools. As other validated methods in the field have often evaluated their method with these three projects, we have also used

Table 9 Number of samples for Six Evaluated Design Patterns

Number of evaluated roles	Number of sample for each role	Total number of data	Number of training data	Number of test data	Number of validation data
19	100	1900	1330	285	285

these same projects to evaluate our proposed approach. The Characteristics of these three software systems are shown in Table 11.

The proposed method is evaluated on six design patterns: Observer, Factory Method, Composite, Strategy/ State, and Adapter. These selected patterns are considered from different categories of design patterns. We also implement our proposed method on similar platforms to other studies to compare our results.

Much research has been done on detecting design patterns, and many tools have been provided to makes detection of design patterns from the source code possible. In this study, the results of the proposed method are compared with DEMiMA [27], Sempatrec [24], GTM [14], and SSA method [42]. The reason we chose these methods is that they are: (1) valid and successful methods for detecting design patterns, (2) covering a significant number of design patterns when evaluating their results, (3) similar platforms to the proposed method for evaluating results that make results comparable, and (4) their tools are available. Details of performing the proposed method and measuring the metrics, by which we determine how much each section of code resembles a design pattern, for example for the composition pattern is shown in Fig. 6. As traceable examples, we measured these features for 12 instances of design patterns are accessed via http://sqlab.um.ac.ir/images/219/files/Details_of_performing_the_probabilistic_detection_of_GoF_design.pdf.

Although it is not possible to compare the results with other methods due to the probabilistic approach in this study by accuracy and recall criteria, we have attempted to investigate whether the instances detected in other studies by the proposed method have also been observed, and if so, how likely we are to reach them. In this regard, the proposed method is implemented on five open source code JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7. In Table 12 the Comparison of the numbers of design pattern instances in the proposed method and other mentioned studies is illustrated. Also, the number of design patterns instances that each of the four mention methods accepted as a correct instance, and then the number of instances that we have correctly identified are shown in Table 12.

As shown in Table 12, the four mentioned methods have obtained design pattern instances in five open source projects: JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7 with deterministic approaches. It is noticeable that in Table 12, the numbers of the recovered instances expressed for the proposed method are just for the instances which have over 50% probability and the instances with lower probability are not considered as design pattern instances.

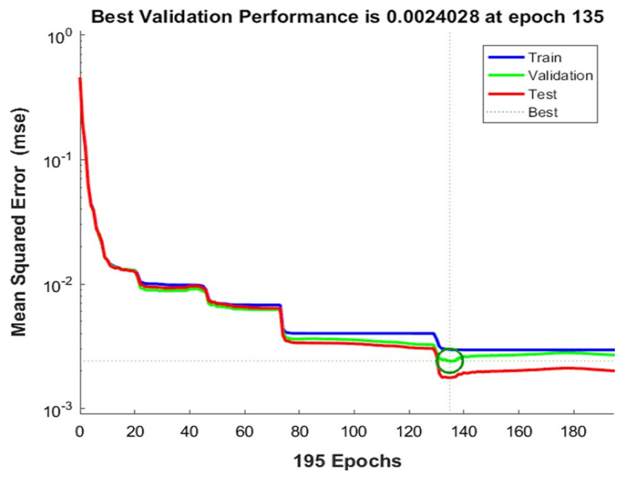


Fig. 5 Performance of the Trained Model

Table 10 The range of neural network hyper parameters

Hyper parameter	Min	Max
Learning rate	0.00001	0.01
Bach Size	10	1000
Momentum	0.1	0.9
Layer size	10	25
Epoch	10	10,000
Weight decay (Log)	0.05	0.0005
Gaussian weight init σ	0.00001	0.1
Loss function	Binary Crossentropy-MSE	
Activation function	Sigmoid- Softmax-Relu- Tanh Sigmoid	

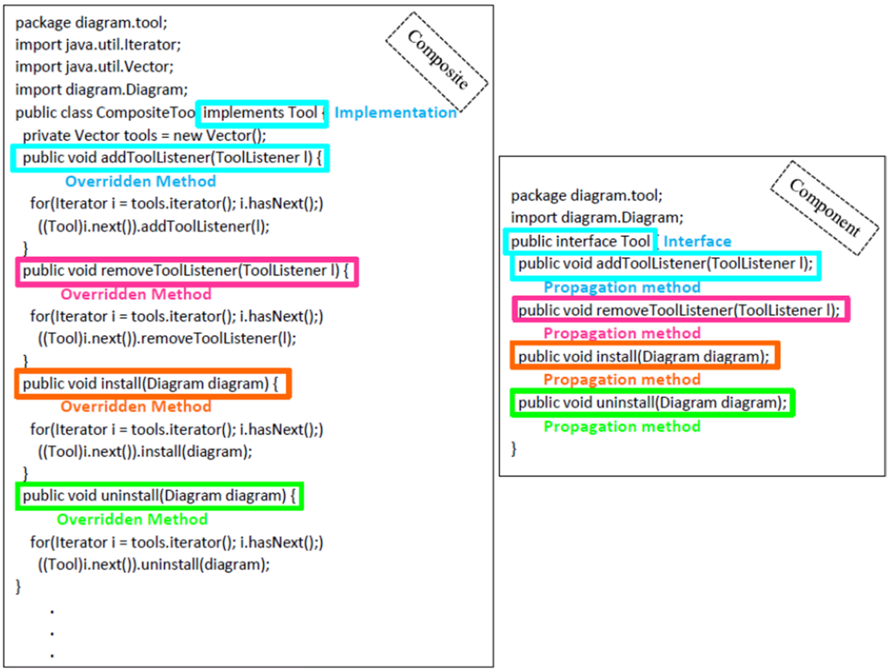
Table 11 Characteristics of the Software System Used in the Evaluation

Characteristic Project	Version	# Classes	# Attribute	# Method
JHotDraw	5.1	155	331	1314
Junit	3.7	51	11	422
JRefactory	2.6.24	569	1364	4234
QuickUML	2001	204	1082	422
MapperXML	1.9.7	346	1469	4741

In all cases where each of the proposed methods identified a specific pattern instance, the proposed method estimated the pattern as 98% to 100% probable, which indicates the correct operation of the proposed method in these cases. In all cases where each of the proposed methods did not identify a specific pattern instance, the proposed method identified the pattern instance, the estimated number for the pattern was 50% to 70%. This is not correctly detected as a design pattern because of the deterministic approach in other methods.

In all cases where the more acceptable method, namely the GTM method [14], has detected the pattern instances, the proposed method estimates the design patterns with a probability of between 80 and 100%. In cases where the proposed method is above 90%, the results are consistent. However in cases where this percentage is about 80%, it is less consistent with the results obtained by the GTM method. The reason for this disagreement is that in the proposed method, the features are not strictly selected. For example, for the observer pattern, both association and dependency relationships were considered necessary, so that a similar pattern would not be missed. This decision sometimes causes noise at the output of the proposed method. However, in the end, the probability of 86% and 88% is not much different from what is expected.

In summary, the proposed method is in line with GTM [14] in design pattern detection, and all instances detected by the GTM [14] method are estimated by the proposed method with high probability. While the instances were also detected by the methods of DEMiMA [27], Sempatrec [24], and the SSA method [42], overall



```
package diagram.tool;
import java.util.Iterator;
import java.util.Vector;
import diagram.Diagram;
public class CompositeToo implements Tool {
 private Vector tools = new Vector();
 public void addToolListener(ToolListener l) {
   for(Iterator i = tools.iterator(); i.hasNext();)
    ((Tool)i.next()).addToolListener(l);
 }
 public void removeToolListener(ToolListener l) {
   for(Iterator i = tools.iterator(); i.hasNext();)
    ((Tool)i.next()).removeToolListener(l);
 }
 public void install(Diagram diagram) {
   for(Iterator i = tools.iterator(); i.hasNext();)
    ((Tool)i.next()).install(diagram);
 }
 public void uninstall(Diagram diagram) {
   for(Iterator i = tools.iterator(); i.hasNext();)
    ((Tool)i.next()).uninstall(diagram);
}
      .
      .
      .
```

```
package diagram.tool;
import diagram.Diagram;
public interface Tool
 public void addToolListener(ToolListener l);
 public void removeToolListener(ToolListener l);
 public void install(Diagram diagram);
 public void uninstall(Diagram diagram);
}
```

Fig. 6 Performance of the Trained Model

Table 12 The Comparison of the numbers of design pattern instances in the proposed method and other mentioned studies

			Proposed method	GTM		Sempatrec		DeMIMA		SSA	
			T	T	TP	T	TP	T	TP	T	TP
Creational	Singleton	JU	0	0	0	0	0	0	0	0	0
		JH	2	1	1	2	2	2	2	2	2
		JR	12	10	10	4	4	14	2	12	8
		QU	1	1	1	1	1	2	1	3	1
		MA	2	1	1	1	1	1	1	1	1
	Factory Method	JU	3	1	1	0	0	23	0	0	0
		JH	10	10	10	3	3	189	3	4	4
		JR	17	16	16	2	1	71	1	1	1
		QU	2	2	2	1	1	2	1	2	1
		MA	9	8	7	6	5	8	5	5	3
Structural	Composite	JU	1	1	1	1	1	1	1	1	1
		JH	1	1	1	1	1	3	1	1	1
		JR	2	0	0	0	0	0	0	0	0
		QU	1	0	0	0	0	0	0	0	0
		MA	1	1	1	1	1	1	1	1	1
	Adaptor	JU	4	4	4	1	1	9	0	6	2
		JH	22	18	18	18	8	28	1	23	10
		JR	21	17	17	22	13	47	17	26	17
		QU	3	3	3	2	1	3	1	3	1
		MA	8	7	7	6	4	8	5	8	5
Behavioral	Observer	JU	3	3	3	1	1	4	1	1	1
		JH	4	2	2	4	2	7	2	3	1
		JR	3	3	3	0	0	1	0	0	0
		QU	3	3	3	3	1	2	1	3	1
		MA	8	6	6	5	1	5	0	7	3
	State/ Strategy	JU	3	3	3	4	3	8	0	3	2
		JH	38	30	30	39	8	21	6	44	11
		JR	33	31	31	7	0	22	2	11	0
		QU	2	1	1	1	0	1	1	2	1
		MA	3	2	2	2	1	3	2	2	1

JU Junit, *MA* MapperXML, *JH* JHotDraw, TP: True Positive, *JR* JRefactory, *T* The total number of the recovered instances, *QU* QuickUML

with probabilities of between 60 and 100%, the instances were also detected by the proposed method. It can be concluded that the more consensus on the selection of a pattern instances by the methods, the proposed method has a higher probability of the likelihood of that pattern being present.

The results of performing the proposed method compared with DEMiMA [27], Sempatrec [24], GTM [14], and SSA method [42] on JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7 are publicly accessed via http://sqlab.um.ac.ir/images/219/files/The_results.pdf.

5 Threats to validity

The following section discusses the threats that may be posed to this research. The study consists of two phases that are threatened by the first phase (Learning phase) and the second phase (Learning phase). These cases are discussed in the following.

5.1 Threats related to phase i of the proposed method: learning phase

In this phase, a dataset is created to obtain a trained model. Since this design pattern instances were generated by the code generator [48], one still cannot be sure of the complete validity of this dataset. The accuracy of these instances of design patterns is limited by the accuracy of these tools. Also, the data used to create this dataset and to build a trained model included design patterns and design patterns variants. However, in this dataset, alternative design patterns were not considered due to the lack of instances. Because if these instances were included as training data, we would have faced unbalanced data sets.

On the other hand, the training data had to be labeled non-binary because of having a look at regression analysis rather than classification in creating the trained model. However, in order to determine the similarity of each role to the other roles for use in the label corresponding to each sample, the extent to which each of the feature is fulfilled by the roles of the design patterns were considered. However, there are some drawbacks to this kind of feature evaluation in determining the similarity between different roles.

5.2 Threats related to Phase II of the proposed method: detection phase

In this phase, probabilistic detection of design patterns from source code is discussed. This is done by converting the source code of the software system studied to its corresponding class diagram. This is done automatically by the tool. The completeness of the class diagram created from the source code will depend on the ability of the tool to create a complete class diagram. There is no assurance that the result of tool will generate a comprehensive and complete UML class diagram, if we face the complex, or semantical relationships between UML constructs. Threats to external validity belong to the capability of generalizing the results to a larger population outside the experimentation environment. Here, we performed our method, on five Java projects. We cannot argue that our method induces the same results on larger systems.

6 Conclusion and future work

Software design patterns are common ways to solve problems in large software systems. These patterns are crucial in forward engineering (designing a system) and in reverse engineering (understanding a software system). Detecting the design patterns used in a system helps to understand and improve its quality. Identifying design patterns in code can help to understand the structure and behavior of a software system. On the other hand, a probabilistic detection of design patterns can significantly assist the software system reconstruction process and reduce its costs.

In this study, a new method for detecting design patterns with the probabilistic approach is presented. To this end, design patterns were represented as a set of features, and then identify and estimate the presence of the constituent roles of design patterns by using regression analysis. Using regression as one of the machine learning techniques allows us to cover design patterns variants. In the next step, using the probabilistic graphical model and Bayesian network and generalized linear model to estimate design patterns in the source code is discussed. In this research, six design patterns, including Singleton, Observer, Factory Method, Composite, Strategy/ State, and Adapter from the category of ≪creational≫, ≪structural≫and≪behavioral≫design patterns of GoF on five open source projects named JHotDraw 5.1, JRefactory 2.6.24, Junit 3.7, QuickUML 2001, and MapperXML 1.9.7 were evaluated.

Next, we are interested in implementing this research on a larger number of design patterns. Also, in the learning phase that we created the dataset as inputs for multilayer perceptron for regression purposes, we considered the design patterns and their variants. We are going to use alternate design patterns as training data. We are also interested in refining the Bayesian network designed in this study.

Data availability All the datasets analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Conflict of interest The authors have no conflicts of interest to declare. All co-authors have seen and agree with the contents of the manuscript and there is no financial interest to report. We certify that the submission is original work.

References

1. Dong J, Zhao Y, Peng T (2009) a Review of design pattern mining techniques. Int J Softw Eng Knowl Eng 19(06):823–855. https://doi.org/10.1142/S021819400900443X
2. Rasool G, Streitfdert D (2011) A survey on design pattern recovery techniques. J Comput Sci Issues 8(6):251–260
3. Mayvan BB, Rasoolzadegan A, Yazdi ZG (2016) The state of the art on design patterns: a systematic mapping of the literature, J. Syst. Softw. 37. https://doi.org/10.1016/j.jss.2016.11.030
4. Ampatzoglou A, Chatzigeorgiou A, Charalampidou S, Avgeriou P (2015) The effect of GoF design patterns on stability: a case study. IEEE Trans Softw Eng 41(8):781–802. https://doi.org/10.1109/TSE.2015.2414917
5. Chihada A, Jalili S, Hasheminejad SMH, Zangooei MH (2015) design pattern detection from source code by classification approach. Appl Soft Comput J 26:357–367. https://doi.org/10.1016/j.asoc.2014.10.027
6. Yu D, Zhang Y, Chen Z (2015) A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. J Syst Softw 103:1–16. https://doi.org/10.1016/j.jss.2015.01.019
7. Ren W, Zhao W (2012) An observer design-pattern detection technique, CSAE 2012—Proceedings, 2012 IEEE Int Conf Comput Sci Autom Eng, 3: 544–547, https://doi.org/10.1109/CSAE.2012.6273011

8. Shi N (2005) Reverse engineering of design patterns for high performance computing, Patterns High Perform. Comput, pp. 1–6, 2005, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.782&rep=rep1&type=pdf
9. Travassos GH, Shull F, Fredericks M, Basili VR (1999) Detecting defects in object oriented designs: using reading techniques to increase software quality. Acm Sigplan Not 34(10):47–56. https://doi.org/10.1145/320384.320389
10. Arcelli Fontana F, Zanoni M (2011) A tool for design pattern detection and software architecture reconstruction. Inf Sci (Ny) 181(7):1306–1324. https://doi.org/10.1016/j.ins.2010.12.002
11. Kludt SR (1996) Metrics and models in software quality engineering—kan, sh. J Prod Innov Manag 13(2):182–183. https://doi.org/10.1016/0967-0661(96)81493-6
12. Gamma E, Helm R, Johnson R, Vlissides J (1996) *Design Patterns: Elements of Reusable Software*
13. Ampatzoglou A, Charalampidou S, Stamelos I (2013) Research state of the art on GoF design patterns: a mapping study. J Syst Softw 86(7):1945–1964
14. Bafandeh Mayvan B, Rasoolzadegan A (2017) Design pattern detection based on the graph theory. Knowledge-Based Syst 120:211–225. https://doi.org/10.1016/j.knosys.2017.01.007
15. De Lucia A, Deufemia V, Gravino C, Risi M (2009) Design pattern recovery through visual language parsing and source code analysis. J Syst Softw 82(7):1177–1193. https://doi.org/10.1016/j.jss.2009.02.012
16. Rasool G, Philippow I, Mäder P (2010) Design pattern recovery based on annotations. Adv Eng Softw 41(4):519–526. https://doi.org/10.1016/j.advengsoft.2009.10.014
17. Nazar N, Aleti A, Zheng Y (2022) Feature-based software design pattern detection. J Syst Softw 185:111179
18. Dong J, Zhao Y (2009) Sun Y (2009) A matrix-based approach to recovering design patterns. IEEE Trans Syst Man Cybern Part A Syst Humans 39(6):1271–1282. https://doi.org/10.1109/TSMCA.2009.2028012
19. Liu C (2021) A general framework to detect design patterns by combining static and dynamic analysis techniques. Int J Softw Eng Knowl Eng 31(1):21–54. https://doi.org/10.1142/S0218194021400027
20. Rasool G, Mader P (2011) Flexible design pattern detection based on feature types, *2011 26th IEEE/ACM Int Conf Autom. Softw Eng (ASE 2011)*, 243–252, 2011, https://doi.org/10.1109/ASE.2011.6100060
21. Thongrak M, Vatanawood W (2014) Detection of design pattern in class diagram using ontology, *2014 International Computer Science and Engineering Conference, ICSEC*. 97–102. https://doi.org/10.1109/ICSEC.2014.6978176
22. De Lucia A, Deufemia V, Gravino C, Risi M (2009) Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation, *2009 13th Eur Conf Softw Maint Reengineering*, https://doi.org/10.1109/CSMR.2009.29
23. Shahbazi Z, Rasoolzadegan A, Purfallah Z, Jafari Horestani S (2021) A new method for detecting various variants of GoF design patterns using conceptual signatures, Softw Qual J, https://doi.org/10.1007/s11219-021-09576-9
24. Alnusair A, Zhao T, Yan G (2014) Rule-based detection of design patterns in program code. Int J Softw Tools Technol Transf 16(3):315–334. https://doi.org/10.1007/s10009-013-0292-z
25. Hayashi S, Katada J, Sakamoto R, Kobayashi T (2008) Design pattern detection by using meta patterns. IEICE—Trans Inf Syst 4:933–944
26. Cointe HAP, Jussien YGN, Cedex N (2001) Instantiating and detecting design patterns : putting bits and pieces together, Proc 16th Annu Int Conf Autom Softw Eng (ASE 2001), pp. 166–173
27. Guéhéneuc YG, Antoniol G (2008) DeMIMA: a multilayered approach for design pattern identification. IEEE Trans Softw Eng 34(5):667–684. https://doi.org/10.1109/TSE.2008.48
28. Sahraoui H, Zaidi F (2004) Fingerprinting Design Patterns, *11th Work Conf Reverse Eng*, pp. 1–10
29. von Detten M, Becker S (2011) Combining clustering and pattern detection for the reengineering of component-based software systems, *Proc Jt ACM SIGSOFT Conf.—QoSA ACM SIGSOFT Symp—ISARCS Qual. Softw. Archit.—QoSA Archit Crit Syst—ISARCS—QoSA-ISARCS '11*, p. 23, https://doi.org/10.1145/2000259.2000265
30. Oruc M, Akal F, Sever H (2016) Detecting Design Patterns in Object-Oriented Design Models by Using a Graph Mining Approach, *2016 4th Int Conf Softw Eng Res Innov*, pp. 115–121, https://doi.org/10.1109/CONISOFT.2016.26.
31. Issaoui I, Bouassida N, Ben-Abdallah H (2014) Using metric-based filtering to improve design pattern detection approaches. Innov Syst Softw Eng 11(1):39–53. https://doi.org/10.1007/s11334-014-0241-3

32. Lanza M, Marinescu R, Ducasse S (2006) *Object-Oriented Metrics in Practice*. Springer. https://doi.org/10.1007/3-540-39538-5
33. Blewitt A, Bundy A, Stark I (2005) Automatic Verification of Design Patterns in Java Categories and Subject Descriptors, *ASE '05 Proc 20th IEEE/ACM Int Conf Autom Softw Eng*, pp. 224–232
34. Niere J, Sch W, Wadsack JP, Wendehals L, Welsh J (2002) Towards Pattern-Based Design Recovery, *ICSE '02 Proc 24th Int Conf Softw Eng*, pp. 338–348
35. Jamali N, Sadegheih A, Lotfi MM, Wood LC, Ebadi MJ (2021) Estimating the depth of anesthesia during the induction by a novel adaptive neuro-fuzzy inference system: a case study. Neural Process Lett 53(1):131–175. https://doi.org/10.1007/s11063-020-10369-7
36. Yu D, Zhang Y, Chen Z (2015) A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures, J Syst Softw, 103(C): 1–16, 2015, https://doi.org/10.1016/jjss.2015.01.019
37. Issaoui I, Bouassida N, Ben-abdallah H (2016) Predicting the existence of design patterns based on semantics and metrics. Int Arab J Inf Technol 13(2):310–319. https://doi.org/10.5772/30826
38. Ferenc R, Beszédes Á, Fülöp L, Lele J (2005) Design pattern mining enhanced by machine learning. IEEE Int Conf Softw Maintenance ICSM 2005:295–304. https://doi.org/10.1109/ICSM.2005.40
39. Uchiyama S, Kubo A, Washizaki H, Fukazawa Y (2014) Detecting design patterns in object-oriented program source code by using metrics and machine learning. J Softw Eng Appl J Softw Eng Appl 7(7):983–998. https://doi.org/10.4236/jsea.2014.712086
40. Bafandeh Mayvan B, Rasoolzadegan A, Javan Jafari A (2020) Bad smell detection using quality metrics and refactoring opportunities, J Softw Evol Process, 32(8): 1–33, 2020, https://doi.org/10.1002/smr.2255
41. Nazar N, Aleti A, Zheng Y (2020) Feature-Based Software Design Pattern Detection, pp. 1–15, https://doi.org/10.1016/j.jss.2021.111179
42. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST (2006) Design pattern detection using similarity scoring. IEEE Trans Softw Eng 32(11):896–909. https://doi.org/10.1109/TSE.2006.112
43. Dwivedi AK, Tirkey A, Ray RB, Rath SK (2017) Software design pattern recognition using machine learning techniques, in *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, 2017, pp. 222–227. https://doi.org/10.1109/TENCON.2016.7847994
44. Zanoni M, Perin F, Fontana FA, Viscusi G (2014) Design pattern detection using a DSL-driven graph matching approach. J Softw Evol Process 26(12):1172–1192. https://doi.org/10.1002/smr
45. Chihada A, Jalili S, Hasheminejad SMH, Zangooei MH (2015) Source code and design conformance, design pattern detection from source code by classification approach. Appl Soft Comput J 26:357–367. https://doi.org/10.1016/j.asoc.2014.10.027
46. Nazar N, Aleti A, Zheng Y (2020) Feature-Based Software Design Pattern Detection, https://doi.org/10.1016/j.jss.2021.111179.
47. Rasool G, Akhtar H (2019) Towards a catalog of design patterns variants," *Proc—2019 Int Conf Front Inf Technol FIT 2019*, 156–161, https://doi.org/10.1109/FIT47737.2019.00038.
48. Mayvan BB, Rasoolzadegan A, Ebrahimi AM (2019) A new benchmark for evaluating pattern mining methods based on the automatic generation of Testbeds, Inf Softw Technol

Authors and Affiliations

Niloofar Bozorgvar[1] · **Abbas Rasoolzadegan**[1] · **Ahad Harati**[1]

Niloofar Bozorgvar
n_bozorgvar@mail.um.ac.ir

Ahad Harati
a.harati@um.ac.ir

[1] Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran