Front. Comput. Sci.
https://doi.org/10.1007/s11704-017-6424-y

RESEARCH ARTICLE

Software design pattern mining using classification-based techniques

Ashish Kumar DWIVEDI (✉), Anand TIRKEY, Santanu Kumar RATH

Department of Computer Science and Engineering, National Institute of Technology, Rourkela 769008, India

Abstract Design patterns are often used in the development of object-oriented software. It offers reusable abstract information that is helpful in solving recurring design problems. Detecting design patterns is beneficial to the comprehension and maintenance of object-oriented software systems. Several pattern detection techniques based on static analysis often encounter problems when detecting design patterns for identical structures of patterns. In this study, we attempt to detect software design patterns by using software metrics and classification-based techniques. Our study is conducted in two phases: creation of metrics-oriented dataset and detection of software design patterns. The datasets are prepared by using software metrics for the learning of classifiers. Then, pattern detection is performed by using classification-based techniques. To evaluate the proposed method, experiments are conducted using three open source software programs, JHotDraw, QuickUML, and JUnit, and the results are analyzed.

Keywords design patterns, design pattern mining, machine learning techniques, object-oriented metrics

1 Introduction

Design patterns are a set of artifacts that encapsulates knowledge of design problems that occur in a particular context. Various software patterns have been proposed to provide solutions to recurring design problems [1–3]. Pattern-based solutions can be analyzed by considering various tools and techniques [4–6]. Design patterns are helpful in the development of proposed systems. A pattern template is often used to indicate a pattern-based solution for various design problems that recur in nature. The concept of a pattern template was initially proposed by Gamma et al. [1], who organized their design pattern's catalogue into creational, structural, and behavioral patterns. Application of design patterns is intended to improve software maintenance by applying reverse engineering techniques. Design patterns are special types of software architecture that provide information about a system at a higher level of abstraction, whereas software design patterns provide information at a lower level of abstraction.

The idea of software patterns is often used in forward engineering, where various applications often consider pattern-based solutions for problems that recur in nature. It has been shown that a pattern-based solution associated with a pattern's participant plays different roles during the development of a system. The candidate classes sometimes lose their roles during system implementation. Therefore, detecting a pattern's candidate classes and roles, which help in understanding and maintaining original design decisions of an application, is necessary. However, the problem is that the number of design pattern increases as a result of changes in the requirements that were not anticipated in the original design. Thus, a pattern identification method cannot be based on a specific pattern. In this study, mining of design patterns is performed by considering classification-based approaches.

The notation for software patterns is specified by considering a modeling language (i.e., unified modeling language (UML)). It offers several implementation templates, which are also known as variants [7], for design patterns. These

Received August 23, 2016; accepted March 8, 2017

E-mail: shil2007@gmail.com, andy9c@gmail.com, skrath@nitrkl.ac.in

variants may derive from abstract representations of patterns that have a similar structure and behavior. The interpretation and pattern recognition becomes difficult because of the problems with variants. This problem can be solved by providing a metric-based definition of design patterns. Each candidate class of a design pattern is represented as a set of object-oriented metrics.

Design pattern detection is a useful activity that supports the maintenance of object-oriented software by using a reverse engineering approach [8]. Design pattern recognition is similar to any other information retrieval technique that most of the time suffers from false positives (FPs) and false negatives (FNs) [9]. While considering the aspect of detecting patterns, an FP denotes the participants of an identified pattern that do not belong to actual instances. With an FN, actual instances are not correctly identified by the considered technique.

This study intends to overcome these problems by offering an approach that maps the software pattern detection process into a learning process. Classification-based techniques are applied in identifying the design patterns of abstract factory, adapter, bridge, composite, and template method from JHotDraw, QuickUML, and JUnit. For the classification process, supervised machine learning techniques such as artificial neural network (ANN) [10], support vector machine (SVM) [11], and random forest [12] are applied. These classifiers are used to develop a model that becomes useful for predicting the results of testing samples after applying the learning process. The main purpose of a testing sample is to validate the trained model. A developed model that offers greater prediction accuracy may be acquired by using large training samples that can be generalized for new training samples. The presented design pattern mining technique is based on machine learning methods, which consider metrics-based datasets. These metrics help to measure object-oriented system properties that are often considered desirable for the development process [13].

2 Related work

Various design pattern detection techniques have been proposed by different authors [9]. Some of the techniques based on static analysis [14–16] encounter certain problems when two or more patterns have a similar structure. For example, state and strategy patterns have similar pattern structures. This section reviews existing related techniques.

The first software pattern detection approach was proposed by Shull et al. [17]. They performed manual checking, which often turns out to be tedious and time-consuming. Antoniol et al. [18] used object-oriented metrics for detecting design patterns. They examined the source code and class diagrams of object-oriented software and developed an abstract object language (AOL) that includes the descriptions of design patterns. They then developed abstract syntax tree (AST) from AOL. Finally, they detected design patterns based on the structural relationships among candidate classes. Gueheneuc et al. [19] presented a technique for reducing the search space of pattern candidate classes available in the source code of software. They examined the source code of various object-oriented software in order to detect pattern candidate classes. The authors prepared a repository that included pattern instances. They parsed and computed metrics on these instances. The authors then performed experiments that applied machine learning techniques to detect candidate classes that played variant roles. The limitation of their approach is that they did not learn all individual classes of design patterns that exist in software source code. The authors considered classes instead of patterns for the classification process.

Tsantalis et al. [14] presented a pattern detection approach based on the similarity scoring algorithm and applied it to the vertices of a graph. Specifically, they applied their approach to detect pattern instances that correspond to pieces of code residing in various inheritance hierarchies. For an evaluation, the authors performed experiments on three open source projects, namely, JHotDraw, JUnit, and JRefactory. Dong et al. [15] proposed a template matching approach for recognizing pattern instances that exist in software source code. The authors claimed that their approach detected exact matches of pattern instances and identified variations of pattern candidates. They developed a tool to implement their approach. Another structural approach was proposed by Kaczor et al. [20]. The authors considered two approximate string matching algorithms, the first based on automata simulation, the second on the bit-vector processing concept. The authors considered two case studies to detect the exact and approximate occurrence of design motifs using these two algorithms.

Ferenc et al. [21] used classification-based techniques for detecting software patterns. The authors minimized the number of false hits from the results obtained from the structure-based pattern miner algorithm presented in [22]. Their approach is purely static. The authors distinguished true from false pattern instances by using a learning database prepared by manually tagging huge datasets in C++ software. By contrast, our study uses manual as well as automated techniques for the tagging of pattern-based datasets. The authors

used two classifiers, namely, C4.5 decision trees and neural network with back propagation, for pattern classification. Uchiyama et al. [23] used a learning-based algorithm in order to detect design pattern instances. The authors used software metrics in creating datasets. They claimed that their approach suppresses FNs and recognizes patterns containing similar class structures. They used a neural network algorithm as a classification-based method. The authors considered only 12 metrics, whereas the proposed method uses 67 metrics for the process of pattern mining.

Alhusain et al. [24] used a learning-based algorithm to detect software design patterns. The authors developed a training dataset by using several existing pattern detection tools and applied a feature selection approach to identify feature vectors. For evaluation, six design patterns, namely, adapter, command, composite, decorator, observer, and proxy, were considered. The authors trained an ANN for the pattern instances with various input feature vectors by considering selection methods. The authors considered only JHotDraw when experimenting with their technique. They provided a comparative analysis for the pattern instances recognized by various techniques. Chihada et al. [25] described a detector developed by training data from pattern instances to assist in the implementation of variants. They used SVM for classifying selected patterns as well as identifying correct patterns from open source software. They considered six design patterns, namely, adapter, builder, composite, factory method, iterator, and observer in evaluating their approach. The authors used 45 metrics in preparing their dataset, whereas we use 67 metrics and provide better accuracy for common design patterns such as adapter and composite.

A different technique was proposed by Yu et al. [26], who presented a pattern detection process by using sub-patterns and the concept of method signatures. First, the authors translated source codes and patterns based on Gamma et al. [1] into graphs, and then identified the instances of sub-patterns. Then, the sub-pattern instances were merged by the join classes to check the matches with predefined patterns. Pradhan et al. [27] employed a graph-based pattern detection method based on normalized cross-correlation and graph-isomorphism techniques. They conducted experiments on four open source software programs. Di Martino and Esposito [28] proposed a design pattern detection approach through which they exploited the semantic specifications of design patterns to be detected, which was based on the web ontology language (OWL). The authors defined a set of first-order logic rules that were independent of any other patterns.

3 Background

In this section, we introduce some basic elements of design pattern mining, including software design patterns, software metrics, and classification-based algorithms.

3.1 Software design patterns

Five design patterns, namely, abstract factory, adapter, bridge, composite, and template method, were considered in this study for detecting instances of design patterns from the source code of open source projects. These patterns cover all three categories in the pattern catalogue, namely, creational, structural, and behavioral design patterns [1]. We selected these five design patterns for two reasons. First, these design patterns are used in existing techniques [9], which makes it easier to compare the proposed method with these techniques. Second, the bridge and composite design pattern are the most and least frequently used patterns [29]. Thus, we can analyze both the frequency of use of patterns as well as the efficiency of the proposed method in correctly identifying both the least and most frequently occurring design patterns in the source code of software systems.

The abstract factory design pattern belongs to the creational pattern class. It provides an interface that is responsible for generating a factory of dependent objects without explicitly specifying their classes. The adapter belongs to the structural pattern class. The adapter converts the interface of one class into that of another. A bridge pattern is in a similar class of structural design pattern. It separates the abstract concept from the concrete implementation of patterns, where they can alter their representation individually. Composite design patterns are also a type of structural design pattern that aggregates objects of a class into tree structures. The template method is a behavioral design pattern used to represent the skeleton of an algorithm in a base class template method. The template method provides a common implementation of individual methods specified in a base class. These selected set of design patterns are presented in Fig. 1.

3.2 Software metrics for an object-oriented system

Software design patterns have inspired solutions to problems related to object-oriented software development. The quality of software can be quantitatively measured by certain object-oriented metrics such as afferent coupling (AC), package size (PZ), weight of a class (WOC). In this study, we consider the 67 metrics listed in Table 1 in our experiments with the pattern mining process. These metrics for an open source

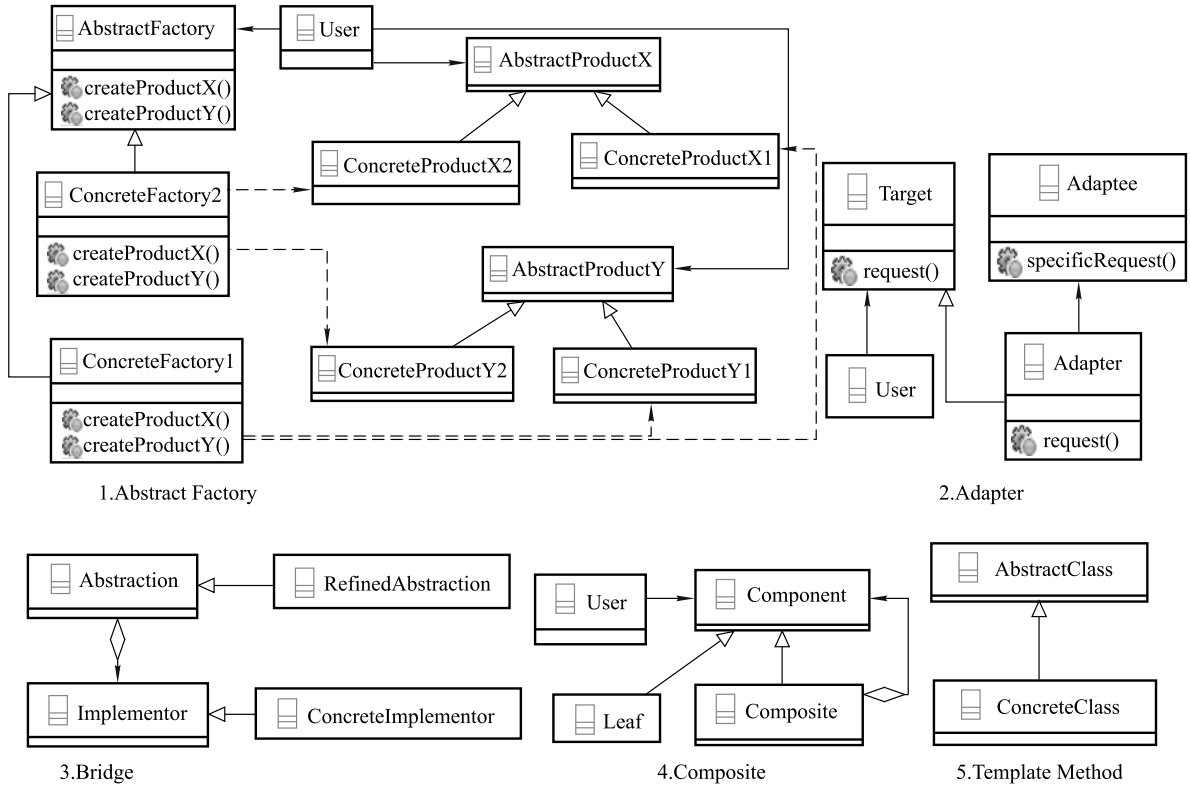

Fig. 1 Selected software design patterns

Table 1 List of selected metrics

No.	Metrics	Meaning	No.	Metrics	Meaning	No.	Metrics	Meaning
1	A	Abstractness	24	FO	FanOut	47	NOCP	Number of Client Packages
2	AC	Attribute Complexity	25	HDiff	Halstead Difficulty	48	NOED	Number of External Dependencies
3	AHF	Attribute Hiding Factor	26	HEff	Halstead Effort	49	NOIS	Number of Import Statements
4	AID	Access of Import Data	27	HPLen	Halstead Program Length	50	NOLV	Number of Local Variables
5	AIF	Attribute Inheritance Factor	28	I	Instability	51	NOM	Number of Members
6	AIUR	Average Inheritance Usage Ratio	29	IUR	Inheritance Usage Ratio	52	NOO	Number of Operations
7	ALD	Access of Local Data	30	LCOM3	Lack of Cohesion of Methods 3	53	NOOM	Number of Overridden Methods
8	AOFD	Access of Foreign Data	31	LOC	Lines of Code	54	NOP	Number of Parameters
9	AUF	Average Use of Interface	32	MDC	Module Design Complexity	55	NOPA	Number of Public Attributes
10	CA	Afferent Coupling	33	MHF	Method Hiding Factor	56	NORM	Number of Remote Methods
11	CBO	Coupling Between Objects	34	MIC	Method Invocation Coupling	57	PC	Package Cohesion
12	CC	Cyclomatic Complexity	35	MIF	Method Inheritance Factor	58	PF	Polymorphism Factor
13	CE	Efferent Coupling	36	MNOB	Maximum Number of Branches	59	PIS	Package Interface Size
14	CF	Coupling Factor	37	MNOL	Maximum Number of Levels	60	PS	Package Size
15	CIW	Class Interface Width	38	MPC	Message Passing Coupling	61	PUR	Package Usage Ratio
16	CM	Changing Methods	39	MSOO	Maximum Size of Operation	62	RFC	Response For Class
17	COC	Clients of Class	40	NAM	Number of Accessor Methods	63	RMD	Normalized Distance
18	CR	Comment Ratio	41	NCC	Number of Client Classes	64	TCC	Tight Class Cohesion
19	ChC	Changing Classes	42	NIC	Number of Import Classes	65	WCM	Weighted Chancging Methods
20	DAC	Data Abstraction Coupling	43	NOA	Number of Attributes	66	WMPC1	Weighted Methods Per Class 1
21	DD	Dependency Dispersion	44	NOAM	Number of Added Methods	67	WOC	Weight of a Class
22	DOIH	Depth of Inheritance Hierarchy	45	NOC	Number of Classes			
23	EC	Essential Complexity	46	NOCC	Number of Child Classes			

software are extracted using the JBuilder tool and represent a variety of measuring parameters such as cohesion, coupling, program complexity, polymorphism, and inheritance. During the recognition of design pattern instances, object-oriented metrics are used to identify pattern constituent candidate sets, when combinatorial explosion occurs in examining all possible class combinations [18]. The selected set of 67 metrics becomes helpful in identifying right kind of pattern role.

3.3 Machine learning methods

In this study, specific supervised learning methods, namely, ANN, SVM, and random forest are considered when detecting software design patterns and a help in building a model that facilitates that process based on known results. ANN belongs to a family of biological neural networks that are used in learning models from experience [10]. ANN models have the capability of learning and must be trained. In an ANN, various nodes are contained in input, hidden, and output layers, in which each node processes data based on function and produces output. In this study, input values based on object-oriented metrics provide a weight when they move from one node to another and adjust to a function by changing the weights. SVM is another supervised learning method used to learn pattern structure from a metrics-based dataset. SVM generates a set of hyperplanes for classifying different classes of datasets [11]. To generate an optimal hyperplane, SVM executes an iterative training algorithm that helps to reduce the error rate. In this study, multiclass SVM is used, which assigns labels to different classes of pattern instances. The third classifier (i.e., random forest) is an ensemble learning method used to classify a OO metrics-based dataset. It works by creating a multitude of decision trees during training and generating the classes composed of subclasses of the individual trees [12]. Random forest helps to reduce the over-fitting problem of individual decision trees.

4 Proposed work

In this study, the mining of design patterns is categorized into two main phases, creating a metrics-based dataset and detecting software patterns, as presented in Fig. 2.

4.1 Creating a metrics-based dataset

In the first phase, a metrics-based dataset is created to train the learning-based classifiers applied in the presented approach. Creation of metrics-based dataset includes three other processes: defining a design pattern, identifying pattern participants, and creating a software metrics-based dataset.

4.1.1 Software pattern definition

Software design patterns are associated with various classes in order to create a larger structure. Sometimes they have a similar structure, but they are different in terms of behavior. The concept of a design pattern is often specified by considering different aspects inherent in a pattern template such as the problem, consequences, motivation, structure, and behavior. However, these descriptions may be ambiguous and inconsistent. Thus, a design patterns is often defined by system analysts who observe pattern structures and behaviors.

4.1.2 Identifying pattern participants

The definition of design patterns provides information about pattern structure as well as behavior and in turn help to identify pattern participants. These participants have certain roles and responsibilities in a design pattern instance. Software system source code is provided as input for a metrics measurement system (i.e., JBuilder) that uses object-oriented metrics for pattern participants. Each design pattern is associated with one or more pattern participants. For example, the abstract factory design pattern is associated with

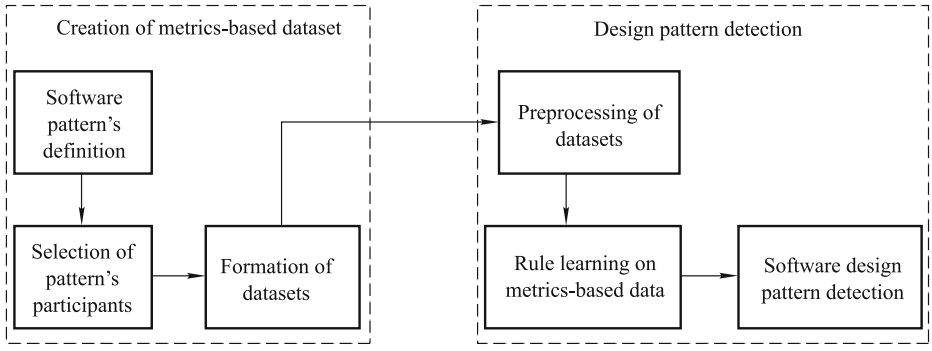

Fig. 2 Software design pattern mining model

abstract factory, concrete factory, abstract product, and *concrete product*. Similarly, the adapter design pattern is associated with *adapter, adaptee*, and *target*. The bridge pattern includes *abstraction, refinedAbstraction, implementor*, and *concreteImplementor* as pattern participants. The composite pattern contains *component* and *composite* as pattern participants. The template method includes *abstractClass* and *concreteClass* as pattern participants. The instances of pattern participants often create role permutations based on the number of participants in the source code. In this step, the candidate search space may also be minimized by removing the classes that do not play a major role as participants. This process is helpful in improving the accuracy of the proposed method during the design pattern detection process.

4.1.3 Formation of datasets

Two techniques are often used to develop a pattern-based dataset: manually tagging candidate classes and using pattern detection tools. The first technique is time-consuming and requires expert knowledge in how to tag pattern instances. The manual technique may be infested with issues such as FPs and FNs because of poor understanding of the target software. The second technique is faster than the first. However, the second requires proper analysis of detected pattern instances in terms of candidate classes. Sometimes, a tool may detect a candidate class as a pattern participant, while another tool does not recognize it at all. In this case, the proper definition of a pattern is necessary to correctly identify a particular candidate class manually. Thus, the dataset is created by using both automatic and manual tagging.

To develop a metrics-based feature vector, design pattern detection tools such as web of patterns [4], MARPLE [8], and the similarity scoring algorithm [14] are considered. Metrics-based datasets of design patterns are prepared first by extracting design pattern instances from the aforementioned tools, where are then mapped with object-oriented metrics obtained from the JBuilder tool. A pattern instance is tagged only when it is detected by at least two pattern detection tools. However, if a pattern instance is tagged differently by all tools, then we tag it either manually or by referring to the P-MARt repository [30]. The process of metrics-based dataset preparation is given in the following algorithm.

The algorithm (metrics-based dataset preparation) consists of three procedures. In the first procedure, source code from the selected software (JHotDraw, QuickUML, and JUnit) are used as input in various design pattern detection systems in order to extract pattern instances. In this process, pattern instances that are detected by at least two pattern detection tools are stored in repository-1. Otherwise, pattern instances are tagged manually and then stored in repository-1. In the second procedure, the JBuilder tool extracts object-oriented metrics of all candidate classes in open source projects and are then stored in repository-2. In the third procedure, pattern instances stored in repository-1 are mapped with corresponding object-oriented metrics associated with instances in repository-2. The mapping process creates feature vectors that are in turn used to build metrics-based feature vectors for pattern participants. These feature vectors are stored in repository-3.

Algorithm Metrics-based dataset preparation

```
1: procedure EXTRACTPATTERNINSTANCES
2: Take the source code of object-oriented software as input
3: Extract XML file containing pattern instances from tools
4: Extract pattern instances from XML file
5:    for each INSTANCE ∈ instances from XML file do
6:       if INSTANCE exist in at least two tools then
7:          Store INSTANCE into a repository-1
8:       else
9:          Store INSTANCE into a repository-1 after
            manually tagging it
10:      end if
11:   end for
12: end procedure
13: procedure EXTRACTMETRICSVALUES
14: Take the source code of object-oriented software as input
15: Extract OO metrics values from source code
16:   for each METRICS ∈ metrics values from source code do
17:      Store METRICS into a repository-2
18:   end for
19: end procedure
20: procedure CREATEMETRICSBASEDPATTERNINSTANCES
21:   for each INSTANCE from repository-1 do
22:      for each METRICS from repository-2 do
23:         if INSTANCE maps to METRICS then
24:            Store METRICS into a repository-3
25:         end if
26:      end for
27:   end for
28: Take repository-3 as metrics-based dataset
29: end procedure
```

In this study, 268 features are considered for all of the selected sets of design patterns as shown in Fig. 1. The abstract factory design pattern with four participants (abstract product, concrete product, abstract factory, and concrete factory) are assigned with their $(4 \times 67) = 268$ metric values. In the case of patterns having fewer than four participants, the missing participant feature values are assigned with the mean of the features of available participants. For ex-

ample, the adapter design pattern contains three participants (adapter, adaptee, and target) having $(3 \times 67) = 201$ features associated with their respective metric values. The remaining $(268 - 201) = 67$ features are assigned the mean metric value $\overline{m}$ as shown in Eq. (1), where n denotes the number of pattern participants. In the case of the adapter design pattern, there are 201 features from the available three participants; hence, $n = 3$.

$$\overline{m} = \frac{\sum_{i=1}^{(n\times 67)} metric_i}{(n \times 67)}, \quad 1 \leqslant n \leqslant 4. \tag{1}$$

4.2 Detection of software patterns

During the detection process, classification methods are trained for the mining of design pattern instances collected from the source code of open source software. The algorithm checks whether the candidate classes existing in the source code are instances of actual design patterns. The detection of software patterns is conducted in three processes (preprocessing of dataset, learning of metrics-based dataset, detecting software design pattern), and finally conformance of result has been performed. Results are analyzed by evaluating pattern instances with existing pattern repositories such as P-MARt [30].

4.2.1 Preprocessing of dataset

To begin the learning process, we must first preprocess the dataset. The considered datasets of JHotDraw, QuickUML, and JUnit are of sizes 3956×269, 79×269, and 64×269, respectively. These datasets are further categorized into input datasets of sizes 3956×268, 79×268 and 64×268 and output datasets of sizes 3956×1, 79×1, and 64×1 for JHotDraw, QuickUML, and JUnit, respectively. Eighty percent of this metrics-based dataset was used in the training of the model; the remaining 20% of the data is used to test the accuracy of the model. These two partitions of the dataset are used to achieve higher prediction accuracy.

4.2.2 Rule learning on metrics-based data

In this study, three classifiers (ANN, SVM, and random forest) are used for the learning process. The main advantage of ANN is that it may contain any number of outputs. ANN can be helpful for nonlinear statistical modeling and is a viable alternative for other methods such as logistic regression. SVM helps to address over-fitting problems by choosing a suitable kernel, tuning the kernel parameters, and applying k-fold cross-validation. SVM does not attempt to control model complexity by keeping the number of features small. Another advantage of SVM is that it automatically chooses its model size. Random forest is another family of methods (i.e., tree ensembles). The advantage of this method is that it does not treat features as part of a linear structure. Another benefit of random forest is that it handles multi-dimensional spaces of features as well as a large number of training datasets. These classifiers detect similarities between classes that have the same roles in design patterns.

To perform the learning process, the datasets of JHotDraw, QuickUML, and JUnit are partitioned into training and test datasets. A model validation technique (i.e., cross-validation) is used to determine how the accuracy of learning models can be generalized for an independent dataset. The cross-validation technique is used on the training data, and the model is tested against the test dataset. In general, a test dataset helps to estimate the performance of learned classifiers. The objective in using selected learning classifiers of ANN, SVM, and random forest is to minimize the number of FNs and to increase precision values in the design pattern detection process.

4.2.3 Design pattern mining

In the pattern detection process, the candidate classes of design patterns in the training dataset are checked against trained candidate classes. ANN, SVM, and random forest are used for the learning process, in which the cross-validation process has previously been performed to reduce underfitting and overfitting of the model. In this study, five- and ten-fold cross-validation methods are applied on the datasets for the 80-20 and 70-30 settings, respectively. Then, accuracy values are measured on the test datasets, and validation is conducted for result conformance by considering the pattern definition as well as the results obtained from the detection process.

5 Experimental results

In this study, various experiments were performed to determine the accuracy by implementing the previously described classification techniques. Testing the superiority of the proposed approach in relation to a baseline method is necessary. This study identifies a suitable classification approach by comparing methods based on performance. In our experiments, JHotDraw, QuickUML, and JUnit were considered for detecting pattern instances of the selected set of design patterns. During the dataset preparation process, 59 samples of abstract factory, eight of composite, 160 of adapter, 3629

Table 2 Recognized pattern instances from open source projects

Projects	Design patterns	No. of instances	80-20 setting		70-30 setting	
			Training dataset	Testing dataset	Training dataset	Testing dataset
JHotDraw	Abstract factory	59	48	11	42	17
	Adapter	160	128	32	112	48
	Bridge	3,629	2,903	726	2,540	1,089
	Composite	8	6	2	5	3
	Template method	100	80	20	70	30
QuickUML	Abstract factory	6	5	1	4	2
	Bridge	55	44	11	39	16
	Template method	18	14	4	13	5
JUnit	Abstract factory	6	5	1	4	2
	Adapter	11	9	2	8	3
	Bridge	9	7	2	6	3
	Template method	38	30	8	27	11

of bridge, and 100 of template method patterns were identified for JHotDraw. For QuickUML and JUnit, the pattern instances are listed in Table 2. These pattern instances were partitioned into training and testing datasets, as shown in Table 2. For the evaluation of the proposed method, a similar set of existing techniques and the P-MARt repository [30] were used. The P-MARt repository includes a pattern's candidate classes from nine open source projects, including JHotDraw, QuickUML, and JUnit.

To assess the greater accuracy of learning methods, a cross-validation process was performed on metrics-based datasets. First, the datasets were divided into 80% and training and 20% testing, respectively. We then applied five-fold cross validation on these datasets. For a comparative analysis, the original metrics-based datasets were partitioned into two datasets of 70% and 30%, respectively, and ten-fold cross validation was applied. In the five-fold cross-validation, both input and target data were divided into five equal datasets in order to perform the learn process up to five iterations. In the testing phase, a precision value was calculated for the final accuracy measurement. The proposed design pattern recognition process then determined the values for training and testing accuracy. The training accuracy was determined after cross-validation, whereas the testing accuracy was determined by considering the testing dataset.

Detection of software design patterns can be measured by considering precision, recall, and the F-measure (i.e., harmonic mean of precision and recall), as shown in Eqs. (2)–(4) respectively, where "NDP" denotes the number of instances of patterns. The true positive (TP) determines the available patterns in the metrics-based dataset that are correctly identified by the classifier. By contrast, the FP determines the pattern instances that do not exist in the metrics-based dataset and the classifier has correctly identified them. The FN determines the pattern instances in the metrics-based dataset that are not identified by the classifiers. A higher value for the F-measure indicates greater accuracy of the design pattern detection process.

In this study, a confusion matrix is used to visualize the accuracy parameters. Results for ANN using JHotDraw, QuickUML, and JUnit are presented in Figs. 3(a)–3(c) respectively. Confusion matrices for the SVM using JHotDraw, QuickUML, and JUnit are presented in Figs. 4(a)–4(c) respectively. Confusion matrices for the random forest using JHotDraw, QuickUML, and JUnit are presented in Figs. 5(a)–5(c) respectively. The presented matrices show the overall accuracies determined by the respective methods. The diagonal cells of the matrices represent TP values. In the confusion matrices, the last row and column represent the precision values and recall values for the respective design patterns, respectively. The last diagonal cell value indicates the overall accuracy of the classifier.

$$\text{Precision} = \frac{\sum_{i=1}^{NDP} TP_i}{\sum_{i=1}^{NDP} TP_i + FP_i}. \tag{2}$$

$$\text{Recall} = \frac{\sum_{i=1}^{NDP} TP_i}{\sum_{i=1}^{NDP} TP_i + FN_i}. \tag{3}$$

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{4}$$

The overall accuracy for the abstract factory, adapter, bridge, composite, and template method patterns were generated from the trained classifiers of ANN, SVM, and random forest and are presented in Tables 3–5 respectively. These represent the values for both training and testing accuracy. For JHotDraw, ANN yielded 99.8% training and 98.7% testing

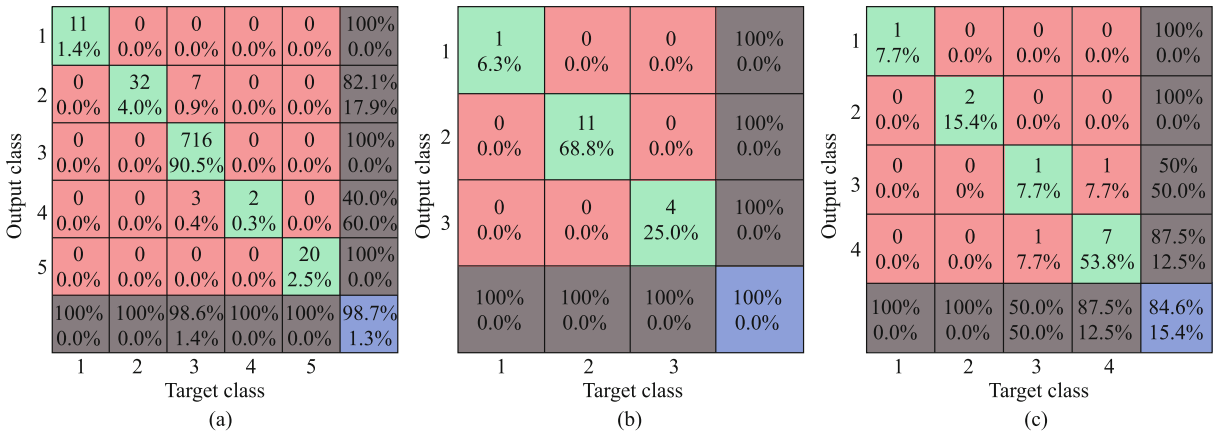

Fig. 3 Confusion matrix generated from ANN using the 80-20 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

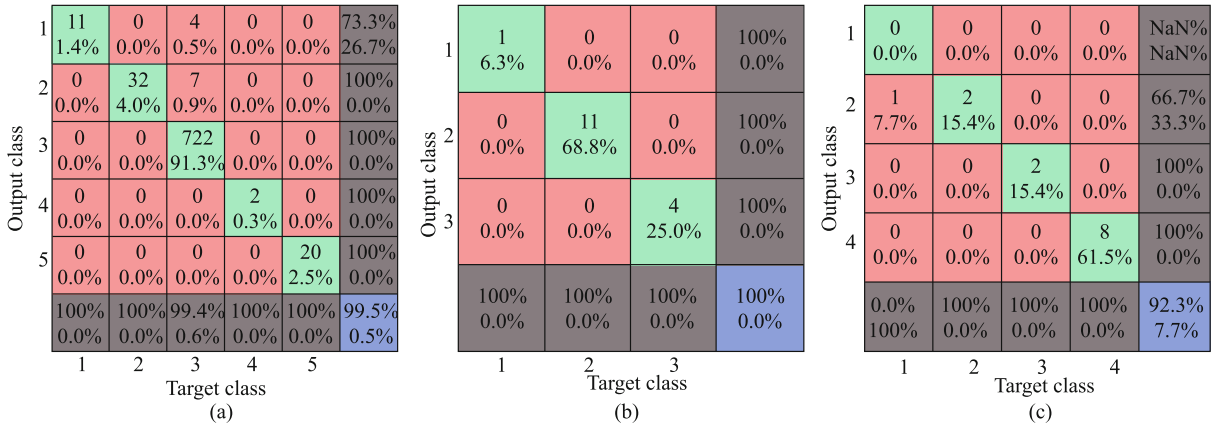

Fig. 4 Confusion matrix generated from SVM using the 80-20 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

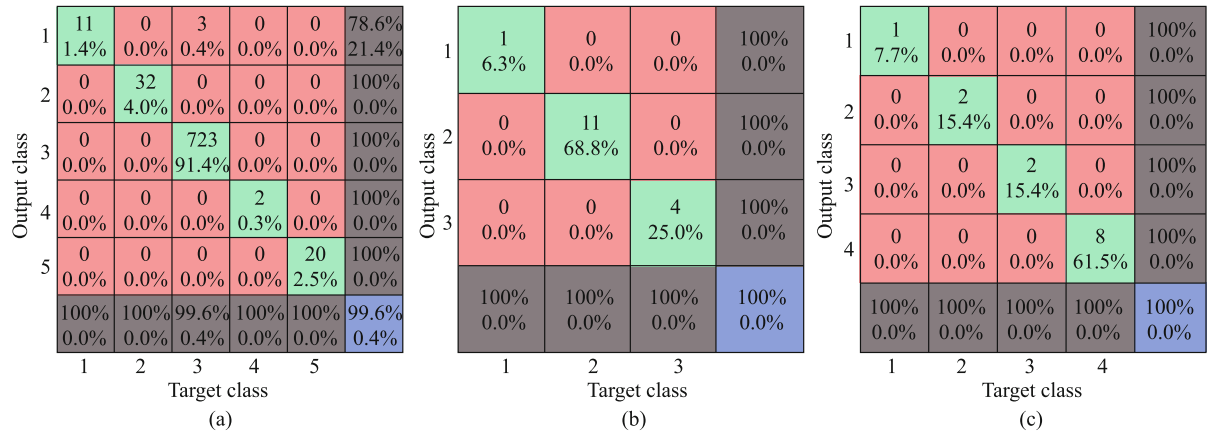

Fig. 5 Confusion matrix generated from random forest using the 80-20 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

accuracies, SVM produced 99.9% training and 99.5% testing accuracies, and random forest showed 99.9% training and 99.6% testing accuracies. Similarly for QuickUML and JUnit, results are presented in Tables 4 and 5, respectively. These results show the individual precision, recall, and F-measure values for the design patterns of abstract factory, adapter, bridge, composite, and template method.

In the case of JHotDraw, the classifiers ANN, SVM, and random forest yielded 100% precision for all patterns except the bridge, as shown in Tables 3–5, respectively. The recall value of the bridge pattern was observed to be 100% for all the methods because of the fact that the learning methods

Table 3 Results for ANN

Project	Software patterns	Precision/%	Recall/%	F-measure/%	Train accuracy/%	Test accuracy/%
JHotDraw	Abstract factory	100	100	100	99.8	98.7
	Adapter	100	82.1	90.17		
	Bridge	98.6	100	99.3		
	Composite	100	40	57.14		
	Template method	100	100	100		
QuickUML	Abstract factory	100	100	100	100	100
	Bridge	100	100	100		
	Template method	100	100	100		
JUnit	Abstract factory	100	100	100	90	84.6
	Adapter	100	100	100		
	Bridge	50	50	50		
	Template method	87.5	87.5	87.5		

Table 4 Results for SVM

Project	Software patterns	Precision/%	Recall/%	F-measure/%	Train accuracy/%	Test accuracy/%
JHotDraw	Abstract factory	100	73.3	84.6	99.9	99.5
	Adapter	100	100	100		
	Bridge	99.4	100	99.7		
	Composite	100	100	100		
	Template method	100	100	100		
QuickUML	Abstract factory	100	100	100	100	100
	Bridge	100	100	100		
	Template method	100	100	100		
JUnit	Abstract factory	0	NaN	NaN	98	92.3
	Adapter	100	67.7	80		
	Bridge	100	100	100		
	Template method	100	100	100		

Table 5 Results for Random Forest

Project	Software patterns	Precision/%	Recall/%	F-measure/%	Train accuracy/%	Test accuracy/%
JHotDraw	Abstract Factory	100	78.6	88	99.9	99.6
	Adapter	100	100	100		
	Bridge	99.6	100	99.8		
	Composite	100	100	100		
	Template Method	100	100	100		
QuickUML	Abstract Factory	100	100	100	100	100
	Bridge	100	100	100		
	Template Method	100	100	100		
JUnit	Abstract Factory	100	100	100	100	100
	Adapter	100	100	100		
	Bridge	100	100	100		
	Template Method	100	100	100		

suppress the FN values. This means that the created metrics-based dataset effectively assists the selected classification methods in distinguishing individual patterns where two or more patterns have identical structures. Higher values of precision mean that the proposed approach helped to suppress FP values, thereby achieving greater accuracy. In JHotDraw, ANN yielded the lowest recall value (i.e., 40% for the composite design pattern). This means that the actual number of candidate classes of composite design patterns was not recognized by the particular method. In this experiment, a 100% F-measure value was obtained for the template method design pattern with all classifiers except for JUnit using ANN, which means the proposed method identified all pattern instances correctly.

In the case of QuickUML, all classifiers provided a 100% F-measure for design pattern detection. However, in the case of JUnit, the classifiers yielded distinct results. ANN produced a 100% F-measure for the abstract factory and adapter design patterns, but 50% and 87.5% F-measures for the bridge and template method patterns, respectively. SVM produced a 100% F-measure for the bridge and template method design patterns, but an 80% F-measure for the adapter design pattern. With the abstract factory pattern, the value of the F-measure was not-a-number (NaN), because the precision value for the abstract factory was 0. JUnit contained only one test sample, as shown in Table 2 and it was misclassified in this case. Random forest yielded a 100% F-measure for all selected patterns in JUnit.

6 Comparison

We observed that because of the lack of a standard benchmark, a proper evaluation and validation of accuracy was difficult [29]. Existing approaches often tag design pattern data manually, which enables the creation of redundant design pattern instances. These repetitions of design pattern instances may increase the amount of FP rates. In this study, we evaluated the performances by compared the results of our proposed method with those of the existing approaches of: Tsantalis et al. [14], Dong et al. [29], Chihada et al. [25], Zanoni et al. [8], and Yu et al. [26]. Chihada et al. [25] and Zanoni et al. [8] used supervised machine learning algorithms. Zanoni et al. [8] used the method of Weka [31] and applied 10-fold cross validation, whereas Chihada et al. [25] divided datasets into 70% and 30% for training and validation sets, respectively. The same setting for the datasets (i.e., 70%–30% and 10-fold cross-validation) were used in our study, and the results are shown in Figs. 6–8. In our comparative analysis, our approach provided significantly better pattern detection results than those obtained by Chihada et al. [25] and Zanoni et al. [8] for the same experimental settings.

Fig. 6 Confusion matrix generated from ANN using the 70-30 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

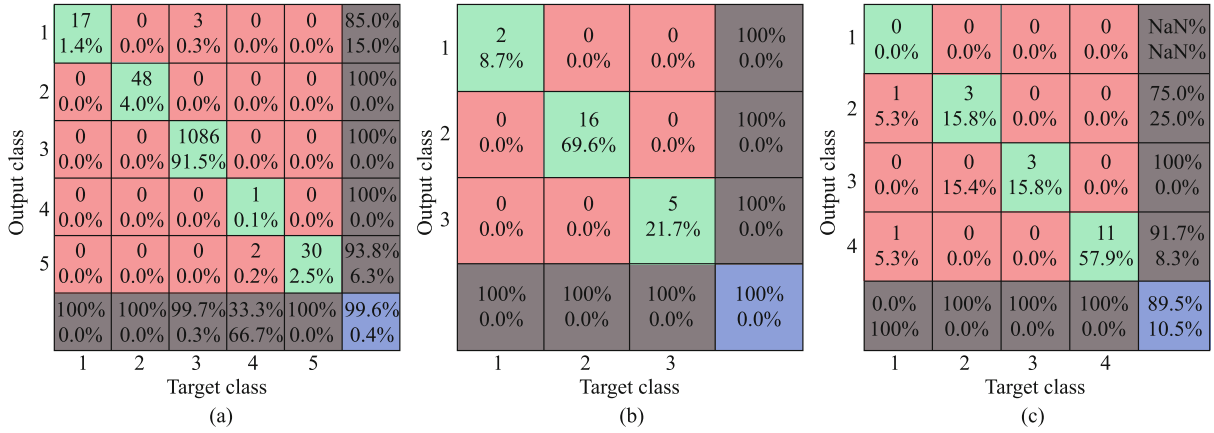

Fig. 7 Confusion matrix generated from SVM using the 70-30 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

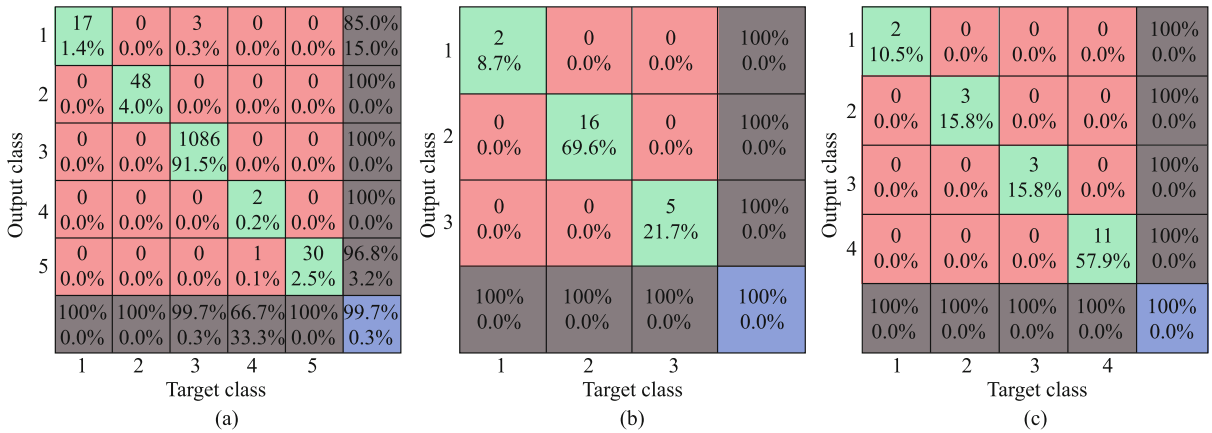

Fig. 8 Confusion matrix generated from random forest using the 70-30 dataset partition. (a) Results for JHotDraw; (b) results for QuickUML; (c) results for JUnit

Table 6 Comparative analysis of results for JHotDraw

No.	Design pattern	Tsantalis et al. [14]			Dong et al. [29]			Chihada et al. [25]			Zanoni et al. [8]			Yu et al. [26]			Proposed approach		
		P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%
1	Abstract factory	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*	100	78.6	88
2	Adapter	48	100	65	100	*	*	86	86	86	*	*	85	100	*	*	100	100	100
3	Bridge	-	-	-	91.4	*	*	-	-	-	-	-	-	*	*	*	99.6	100	99.8
4	Composite	100	100	100	100	*	*	74	74	74	*	*	56	100	*	*	100	100	100
5	Template method	100	100	100	-	-	-	-	-	-	-	-	-	100	*	*	100	100	100

Note: P is short for Precision, R is short for Recall, F is short for F-measure. "-" denotes that the particular design pattern has not been considered, whereas "*" denotes that the particular design pattern has been considered but that the results have not been presented by the authors

Table 7 Comparative analysis of results for JUnit

No.	Design pattern	Tsantalis et al. [14]			Dong et al. [29]			Chihada et al. [25]			Zanoni et al. [8]			Yu et al. [26]			Proposed approach		
		P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%	P/%	R/%	F/%
1	Abstract factory	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*	100	100	100
2	Adapter	17	100	29	*	*	*	*	*	*	*	*	*	100	100	100	100	100	100
3	Bridge	-	-	-	-	-	-	-	-	-	-	-	-	*	*	*	100	100	99.8
5	Template method	100	100	100	-	-	-	-	-	-	-	-	-	100	100	100	100	100	100

Note: P is short for Precision, R is short for Recall, F is short for F-measure. "-" denotes that the particular design pattern has not been considered, whereas "*" denotes that the particular design pattern has been considered but that the results have not been presented by the authors.

Tables 6 and 7 present a comparative analysis of results by considering the values of precision, recall, and F-measure for the case studies of JHotDraw and JUnit, respectively. In this section, a comparative analysis of QuickUML is not given because some of the existing techniques have not considered it as a case study. Tsantalis et al. [14] presented a similarity scoring algorithm for identifying patterns. The authors applied a graph-based technique on JHotDraw, JRefactory, and JUnit to detecting design patterns. According to Table 6, our approach yielded better F-measure values for the adapter pattern with JHotDraw. Dong et al. [29] presented a matrix-based design pattern detection approach. They performed experiments on six open-source systems for detecting the four design patterns of adapter, bridge, strategy, and composite. In the case of JHotDraw, the authors presented 91.4% precision for the bridge pattern, whereas the proposed method showed 99.6% precision, as shown in Table 6.

Chihada et al. [25] proposed pattern identification technique by using a machine learning technique (i.e., SVM). They considered 45 OO metrics in developing the training dataset. However, our proposed study considers 67 metrics. Chihada et al. achieved an 86% F-measure for the adapter pattern, whereas the proposed method achieved a 100% F-measure with the same setting, as shown in Fig. 7(a). Zanoni et al. [8] proposed a pattern identification technique by considering several learning-based techniques. The authors indicated that their approach did not perform better with the composite design patterns. By contrast, in our study, all selected classification methods yielded a 100% precision value with the composite pattern. Zanoni et al. achieved an 85% F-measure for the adapter pattern, whereas the proposed method achieved a 100% F-measure. Yu et al. [26] pre-

sented sub-patterns and a method-signature-based pattern detection approach. They performed experiments on various open source projects for the detection of Gamma et al. [1] design patterns. They considered graph mining and matching approaches for extracting design pattern instances.

Our study presents the results of the random forest classification technique in comparison because it provided the highest value of accuracy. From our comparative analysis, we observed that our study yielded significantly better precision, recall, and F-measure values for the considered set of design patterns. For design pattern detection, results were obtained for two experimental settings: 80%–20% dataset partition considering five-fold cross-validation and 70%–30% dataset partition considering ten-fold cross-validation. In our study, 67 object-oriented metrics were considered in preparing the metrics-based dataset that yielded greater accuracy.

7 Discussion

From the experiments and literature [29], we observed that the composite pattern is the least used pattern, whereas the bridge and template method patterns are the most frequently used. In addition, these five design patterns were often considered by various authors in their studies [26, 30, 32]. Thus, to evaluate the proposed method through a comparative analysis with the results obtained by other authors is worthwhile. In this study, three open source projects were considered, where JHotDraw and JUnit were considered with various techniques [8, 14, 25, 26, 29]. However, QuickUML was not considered except by Zanoni et al. [8] for pattern detection. In our experiment, QuickUML produced a 100% F-measure for all selected patterns when using ANN, SVM, and random forest. Thus, it proved helpful in identifying pattern instances when evaluating QuickUML. During the proposed dataset preparation phase, instances of the composite design were not found in JUnit and QuickUML, whereas instances of the adapter design pattern was not found in QuickUML. The proposed method analyzed pattern instances found in JHotDraw, as shown in Table 8.

From [29], we observed that pattern interpretation may be ambiguous when considering its structure. For example, state and strategy design patterns have a similar structure. Thus, different pattern detection tools tag them differently. However, in our study, a pattern instance was tagged only when it was detected by at least two pattern detection tools. However, if a pattern instance was tagged differently by all tools, then we tagged it either manually or by referring to the P-MARt repository. The pattern detection accuracy was much better with our method than with existing approaches because the prepared object-oriented metrics-based dataset helped to minimize FNs and successfully distinguished the role of different pattern participants.

Table 8 Instances of design patterns identified by various techniques for JHotDraw

Design patterns	Dong et al. [29]	Zanoni et al. [8]	Proposed method
Abstract factory	-	-	59
Adapter	27	135	160
Bridge	74	-	3,629
Composite	0	5	8
Template method	-	-	100

Note: "-" denotes that the particular design pattern has not been considered by the authors.

In this study, variant pattern detection results were revealed by using the three classifiers of ANN, SVM, and random forest. For each dataset of JHotDraw, QuickUML, and JUnit, ANN produced the lowest value of pattern detection, whereas random forest provided the highest value. During the experiment, ANN considered various hidden layers and we observed that a value of 15 for the hidden layers yielded the best pattern detection accuracy. Unlike SVM, ANN contains a multiple number of outputs and can be used for nonlinear statistical modeling. SVM provides robust classification when input data are non-linearly separable. SVM can linearize input data by using kernel transformation. The advantage of random forest is that it does not expect linear features and its algorithms such as bagging or boosting handle high-dimensional spaces.

8 Threats to validity

This study employed two phases: dataset preparation and pattern detection. The first threat is related to dataset preparation, in which pattern instances are considered from various pattern detection tools as well as from manual analysis. The tagging of pattern instances is based on both the roles of pattern participants and the outcomes of various pattern detection tools. However, this process does not always provide correct pattern instances, because no standard benchmark is available for the evaluation process. The second threat is associated with classification techniques. In this study, the three classifiers of ANN, SVM, and random forest are considered for the classification process. These classifiers provided a higher value of precision and recall for the datasets of JHotDraw, QuickUML, and JUnit. The proposed dataset preparation and classification processes were not applied to other

existing open source systems.

9 Conclusion and future work

This study presented a method of software design pattern mining by employing supervised learning-based classification techniques ANN, SVM, and random forest. This study considered abstract factory, adapter, bridge, composite, and template method design patterns for the process of software design pattern mining. These patterns were taken from all three categories of Gamma et al. [1] design patterns, namely, creational, structural, and behavioral. A set of experiments were performed using the proposed method on the object-oriented software of JHotDraw, QuickUML, and JUnit. The proposed approach utilized a scheme for preprocessing a prepared dataset and yielded better accuracy by minimizing the number of candidate classes. The main benefits of using machine learning algorithms for the mining of software design patterns is that they map the pattern detection process into the learning process, which retrieves pattern-based data. Thus, this method can be used to detect design patterns in different versions of software systems. The advantage of using object-oriented metrics for design pattern mining is that it helps to reduce the search space and improves its quality parameters by recognizing the right kinds of candidate classes.

We suggest extending the process of design pattern mining in future studies by considering other open source software.

References

1. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995
2. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002
3. Dwivedi A K, Rath S K. Incorporating security features in service-oriented architecture using security patterns. ACM SIGSOFT Software Engineering Notes, 2015, 40(1): 1–6
4. Dietrich J, Elgar C. Towards a Web of patterns. Web Semantics: Science, Services and Agents on the World Wide Web, 2007, 5(2): 108–116
5. Zhu H, Bayley I. On the composability of design patterns. IEEE Transactions on Software Engineering, 2015, 41(11): 1138–1152
6. Dwivedi A K, Rath S K. Formalization of web security patterns. INFOCOMP Journal of Computer Science, 2015, 14(1): 14–25
7. Niere J, Schäfer W, Wadsack J P, Wendehals L, Welsh J. Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering. 2002, 338–348
8. Zanoni M, Fontana F A, Stella F. On applying machine learning techniques for design pattern detection. Journal of Systems and Software, 2015, 103: 102–117
9. Dong J, Zhao Y, Peng T. A review of design pattern mining techniques. International Journal of Software Engineering and Knowledge Engineering, 2009, 19(06): 823–855
10. Hagan M T, Demuth H B, Beale M H, De Jesús O. Neural Network Design. Vol 20. Boston: PWS publishing Company, 1996
11. Cortes C, Vapnik V. Support-vector networks. Machine learning, 1995, 20(3): 273–297
12. Breiman L. Random forests. Machine Learning, 2001, 45(1): 5–32
13. Arvanitou E M, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P. Software metrics fluctuation: a property for assisting the metric selection process. Information and Software Technology, 2016, 72: 110–124
14. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis S T. Design pattern detection using similarity scoring. IEEE Transactions on Software Engineering, 2006, 32(11): 896–909
15. Dong J, Sun Y, Zhao Y. Design pattern detection by template matching. In: Proceedings of ACM symposium on Applied Computing. 2008, 765–769
16. 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. 2005, 224–232
17. Shull F, Melo W L, Basili V R. An inductive method for discovering design patterns from object-oriented software systems. Technical Report UMIACS-TR-96-10, 1998
18. Antoniol G, Fiutem R, Cristoforetti L. Using metrics to identify design patterns in object-oriented software. In: Proceedings of the 5th International Software Metrics Symposium. 1998, 23–34
19. Gueheneuc Y G, Sahraoui H, Zaidi F. Fingerprinting design patterns. In: Proceedings of the 11th Working Conference on Reverse Engineering. 2004, 172–181
20. Kaczor O, Guéhéneuc Y G, Hamel S. Identification of design motifs with pattern matching algorithms. Information and Software Technology, 2010, 52(2): 152–168
21. Ferenc R, Beszedes A, Fülöp L, Lele J. Design pattern mining enhanced by machine learning. In: Proceedings of the 21st IEEE International Conference on Software Maintenance. 2005, 295–304
22. Balanyi Z, Ferenc R. Mining design patterns from c++ source code. In: Proceedings of International Conference on Software Maintenance. 2003, 305–314
23. Uchiyama S, Washizaki H, Fukazawa Y, Kubo A. Design pattern detection using software metrics and machine learning. In: Proceedings of the 1st International Workshop on Model-Driven Software Migration. 2011, 38–47
24. Alhusain S, Coupland S, John R, Kavanagh M. Towards machine learning based design pattern recognition. In: Proceedings of the 13th UK Workshop on Computational Intelligence. 2013, 244–251
25. 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. Applied Soft Computing, 2015, 26: 357–367
26. Yu D, Zhang Y, Chen Z. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. Journal of Systems and Software, 2015, 103: 1–16
27. Pradhan P, Dwivedi A K, Rath S K. Detection of design pattern using graph isomorphism and normalized cross correlation. In: Proceed-

ings of the 8th International Conference on Contemporary Computing. 2015, 208–213

28. Di Martino B, Esposito A. A rule-based procedure for automatic recognition of design patterns in uml diagrams. Software: Practice and Experience, 2015
29. Dong J, Zhao Y, Sun Y. A matrix-based approach to recovering design patterns. IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans, 2009, 39(6): 1271–1282
30. Guéhéneuc Y G. P-MARt: Pattern-like micro architecture repository. In: Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories. 2007
31. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I H. The weka data mining software: an update. ACM SIGKDD Explorations Newsletter, 2009, 11(1): 10–18
32. Shi N, Olsson R A. Reverse engineering of design patterns from java source code. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. 2006, 123–134

Ashish Kumar Dwivedi received his Bachelor of Technology degree in computer science and engineering from Uttar Pradesh Technical University, Lucknow, India and his Master of Technology by Research in computer science and engineering from NIT Rourkela, India. Presently, he is pursuing his PhD in computer science and engineering from NIT Rourkela. His areas of interest are design patterns, formal methods and machine learning techniques. He is an IEEE member.

Anand Tirkey received his Bachelor of Engineering degree in computer science and engineering from Birla Institute of Technology Mesra, Ranchi, India. Presently, he is pursuing his Master of Technology degree in computer science and engineering from NIT Rourkela, India. His areas of interest are data mining, machine learning, Internet of Things and robotics.

Santanu Kumar Rath is a professor in the Department of Computer Science and Engineering, NIT Rourkela, India since 1988. His research interests are in software engineering, bioinformatics and management. He has published a large number of papers in international journals and conferences in these areas. He is a senior member of the IEEE, USA and ACM, USA and Petri Net Society, Germany.