



Source code and design conformance, design pattern detection from source code by classification approach



Abdullah Chihada, Saeed Jalili*, Seyed Mohammad Hossein Hasheminejad, Mohammad Hossein Zangoeei

SCS Lab., Computer Engineering Department, Electrical and Computer Engineering Faculty, Tarbiat Modares University, Tehran, Iran

ARTICLE INFO

Article history:

Received 16 August 2012
Received in revised form 9 September 2014
Accepted 17 October 2014
Available online 27 October 2014

Keywords:

Design pattern detection
Machine learning
Support vector machine
Object-oriented metrics

ABSTRACT

Nowadays, software designers attempt to employ design patterns in software design phase, but design patterns may be not used in the implementation phase. Therefore, one of the challenging issues is conformance checking of source code and design, i.e., design patterns. In addition, after developing a system, usually its documents are not maintained, so, identifying design pattern from source code can help to achieve the design of an existing system as a reverse engineering task. The variant implementations (i.e., different source codes) of a design pattern make hard to detect the design pattern instances from the source code. To address this issue, in this paper, we propose a new method which aims to map the design pattern detection problem into a learning problem. The proposed design pattern detector is made by learning from the information extracted from design pattern instances which normally include variant implementations. To evaluate the proposed method, we applied it on open source codes to detect six different design patterns. The experimental results show that the proposed method is promising and effective.

© 2014 Elsevier B.V. All rights reserved.

1. Introduction

A design pattern encapsulates a proven solution to a recurring design problem [1]. In fact, each design pattern includes some classes with relations like inheritances, aggregations and delegations. Over many years, software developers suggest solutions for satisfying design problems. Then, these experience-based solutions are standardized and have been organized in the form of design patterns. The use of design patterns in software development can provide several advantages, such as increasing reusability, modularization, quality, consistency between the design and the implementation, and relationship between the design and the implementation teams [1–4].

Design pattern detection from source code is an important task in reverse engineering and can provide several advantages, such as understanding the code of program when the documentation is inadequate or absence, providing the design information to help to

reconstruct software architecture, and providing ability to conformance checking of source code and design [5–7].

Several methods for design pattern detection from source codes have been proposed and we divided these methods into five categories: *Logical reasoning*, *Similarity scoring*, *FCA-based methods*, *Ontology-based methods*, and *Learning-based methods*. In the following, a brief description of each category is presented.

1.1. Logical reasoning

Kramer and Prechelt [8] developed the Pat system where both design patterns and designs are represented in Prolog and the Prolog engine do the actual search. The basic design information itself is extracted from source code by a structural analysis mechanism of commercial object-oriented CASE tools. Similar method is proposed in [9] developing the SOUL environment in which design patterns are described as Prolog predicates and program constituents (classes, methods, fields, etc.) as facts. Smith and Stotts [10] presented a *System for Pattern Query and Recognition* (SPQR) which uses *Elemental Design Patterns* (EDPs) and matches formalizations in a logical calculus. Fabry and Mens [11] proposed a language-independent meta-level interface to extract complex information about a structure of source code, and then used the SOUL environment, to specify and identify design motifs.

* Corresponding author. Tel.: +98 021 8288 3374.

E-mail addresses: A.Chihada@Modares.ac.ir (A. Chihada), SJalili@Modares.ac.ir (S. Jalili), SMH.Hasheminejad@Modares.ac.ir (S.M.H. Hasheminejad), MH.Zangoeei@Modares.ac.ir (M.H. Zangoeei).

The prolog facts [12] present both static and dynamic information of source code to achieve the detection of design patterns with high accuracy. In [13,14], the advantage of fuzzy reasoning in dealing with incomplete knowledge to identify variant implementations of the same design pattern are used.

1.2. Similarity scoring

Tsantalis et al. [7] proposed a method based on similarity scoring between vertices of design pattern graphs and a graph correspond to a piece of program that reside in one or more inheritance hierarchies. This method has an ability to detect modified versions of same design pattern. In [15], a similarity is calculated between whole two graphs rather than their vertices, this approach adopts a template matching method from computer vision by calculating the normalized cross correlation between pattern graph matrix and system graph matrix. Kaczor et al. [16] proposed a method to identify classes whose structure and organization match exactly or approximately the structure and organization of design pattern classes. The authors used two classical approximate string matching algorithms based on automata simulation and bit-vector processing. In [17–19], dynamic analysis is integrated with static analysis to achieve high accuracy in design pattern detection. Heuzeroth et al. [17] used dynamic analysis results of a given system to decrease false positives of results obtained by static analysis. Dong et al. [18] presented a method in which design pattern detection applied similar to [15] and then false positive results are eliminated by behavioral and semantic analyses. Ng and Guéhéneuc [19] introduced a trace analysis technique to identify occurrences of creational and behavioral design patterns.

1.3. FCA-based methods

Tonella and Antoniol [20] proposed a method in which *Formal Concept Analysis* (FCA) algorithm is used to infer the presence of class groups which instantiate a common, repeated pattern, without assumption on availability of any predefined design pattern library. In [21,22], the concepts that have been calculated using FCA algorithm are filtered by removing unconnected patterns and merging equivalent ones. Then, the filtered concepts are compared with a reference library of well-known patterns to be assigned as one of the matched patterns. Tripathi et al. [23] proposed a model which solves the problem of performance by using an efficient algorithm called *Concept-Matrix based Concepts Generation* (CMCG) for construction of concepts. In [6], just patterns consisted of four classes or less have been detected because the detection of relatively simple structures in relatively small pieces of source code requires a lot of calculations.

1.4. Ontology-based methods

These methods use the *Web Ontology Language* (OWL) to structure source code knowledge. Dietrich and Elgar [24] proposed a Web of Pattern system which formally defines design patterns by using OWL, then, they used a prototype of a Java client that scans the pattern definitions and detects patterns in Java software. Kirasi and Basch [25] proposed a system that has three subsystems: parser, OWL ontologies and analyzer. The parser subsystem translates the input code to an XML tree. The OWL ontologies define patterns and general programming concepts. The analyzer subsystem constructs instances of the input code as ontology individuals and asks the reasoner to classify them.

1.5. Learning-based methods

Guéhéneuc et al. [26,27] used machine learning for obtaining rules set called signatures for participant classes (roles) of the

design patterns. Then, they integrated these signatures with their constraint-based tools suite to reduce the search space. The authors learned just some individual classes of design patterns. One of the important challenges in learning-based design pattern detection is what piece of code will be given to the learned model as a candidate design pattern. Ferenc et al. [28] used another design pattern tool to candidate some probable design patterns. They used machine learning after structural matching phase for filtering out as many as possible of false hits per pattern rather than classifying them. In [29], the authors instead of analyzing source code directly, they used *Metrics and Architecture Reconstruction Plug-in for Eclipse* (MARPLE) tool to summarize the code into micro architecture that are EDP and *Design Pattern Clues*. After that, they used neural networks and WEKA data mining algorithms [30] to classify design patterns.

In our previous work [4], we proposed a novel learning-based method to select the right design pattern for each design problem. The goal of this paper is to improve limitations of learning based methods, specially [26,27], for detecting design patterns. Therefore, in this paper, we map the design pattern detection problem into a learning problem, without using another design pattern detection tool for preprocessing.

Compared with other learning-based methods [26–29], the proposed method has a number of distinguishing characteristics: (1) It has ability to detect multiple versions of design patterns (i.e., different implementations), (2) as opposed to the other learning-based methods [26–29], it does not use a tool for preprocessing but it proposes a novel, simple and low-cost preprocessing, (3) it learns simultaneously all design pattern elements (i.e., all classes playing role) of a design pattern sample instead of learning each role separately [26,27], and (4) it uses a novel and high precision learning method (i.e., classification method) called SVM-PHGS [31] and customizes it to detect design patterns.

For evaluation, we use samples of design patterns gathered from nine case studies which are manually detected by experts. We apply some different data representation methods and machine learning algorithms on the samples. In our experiments, we use six design patterns, *Adapter*, *Builder*, *Composite*, *Factory Method*, *Iterator*, and *Observer*. The results reveal that for these patterns, the proposed method can identify corresponding source code samples with acceptable Precision, Recall and small False Positive rate.

The paper is organized as follows: Section 2 presents the background needed to understand the proposed method. The proposed design pattern detection method is described in Section 3. The experimental work and results are reported in Section 4 and Section 5 presents a discussion on results and compares the proposed method with the related works. Conclusion and future works are given in Section 6.

2. Background

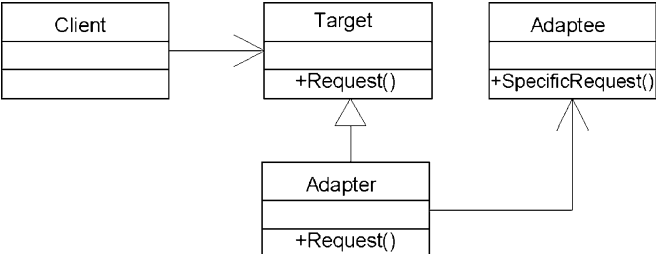
This section describes three categories which are necessary for understanding the proposed method. At first, the definition of design patterns is presented. Then, object-oriented metrics which are used in the proposed method are introduced. Finally, an overview of the classification methods is presented.

2.1. Design pattern definition

In general, a design pattern is described in a template consisting of two sections, *Problem Domain* and *Solution Domain*. The *Problem Domain* describes the problem context where the pattern can be applied. Analogously, the *Solution Domain* describes the structure and collaborations of the pattern solution being applied to

Table 1

The Adapter design pattern description [1].

Pattern name	Adapter (Wrapper) Pattern
Intent	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces
Motivation	Sometimes a toolkit class that is designed for reuse is not reusable only because its interface does not match the domain-specific interface an application requires. Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams. The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape. The rest of Motivation section is available in [1]
Applicability	You want to use an existing class, and its interface does not match the one you need. You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that do not necessarily have compatible interfaces.
Structure	 <pre> classDiagram class Client class Target { +Request() } class Adapter { +Request() } class Adaptee { +SpecificRequest() } Client --> Target Adapter -- > Target Adapter --> Adaptee </pre>
Participants (Roles)	Target, Adaptee, Client, and Adapter
Collaborations	Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.
Consequences	Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter will not work when we want to adapt a class and all its sub classes. Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee. Introduces only one object and no additional pointer in direction is needed to get to the Adaptee.
Related patterns	Bridge, Decorator, and Proxy Pattern
Implementation	<pre> /* This is our Adaptee, a third party implementation of a number sorter that deals with Lists, not arrays.*/ Public class NumberSorter{ public List<Integer>sort(List<Integer>numbers) { //sort and return return new ArrayList<Integer>();} } // This is our Target interface Public interface Sorter { public int[] sort(int[] numbers); } /* The goal of our Client is to sort primitive arrays */ Public class Client{ public static void main(String args[]) { int[] numbers = new int[]{34, 2, 4, 12, 1}; Sorter sorter = new SortListAdapter(); sorter.sort(numbers);} } // This is our Adapter Public class SortListAdapter implements Sorter { @Override public int[] sort(int[] numbers) { //convert the array to a List List<Integer>numberList = new ArrayList<Integer>(); //call the adaptee NumberSorter sorter = new NumberSorter(); numberList = sorter.sort(numberList); //convert the list back to an array and return return sortedNumbers; } } </pre>

the problem. A design pattern consists of *Pattern Name* section, *Intent* section (description of its problem), *Motivation* section (a scenario that illustrates a design problem), *Applicability* section (the situations in which the design pattern can be applied), *Structure* section (a structure of the participants in the *Solution Domain*), *Participants* section (the classes and/or objects participating in the *Solution Domain*), *Collaborations* section (collaboration diagrams between solution participants), *Consequences* section (the results and trade-offs of applying the pattern), *Implementation* section, and the *Related Patterns* section [1]. In this paper, **only the *Solution Domain* of each design pattern document including *Structure*, *Participants*, *Collaborations*, and *Implementation* sections is used to detect automatically design patterns.** To illustrate the template of a design pattern, Table 1 shows the *Adapter* design pattern document from Gamma et al. [1].

It should be noted that there are various implementations for a design pattern. For example, for *Adapter* design pattern mentioned in Table 1, Table 2 shows an alternative implementation of its *Adapter* class. In this implementation, *Adapter* class extends *Target* class instead of implementing it and *Adaptee* class is an attribute of *Adapter* class. For this reason, the relation between *Adapter* and

Adaptee classes is converted to aggregation relationship in spite of a simple dependency.

2.2. Object oriented metrics

Object oriented metrics are measurements of some property of a piece of software or its specifications. We use **JBuilder metrics tool** to

Table 2

An alternative implementation for the Adapter Class.

```

// This is our Adapter
Public class SortListAdapter extends Sorter {
Private NumberSorter ns;
Public SortListAdapter (NumberSorter ns) {
this.ns = ns; }
public int[] sort(int[] numbers){
//convert the array to a List
List<Integer>numberList = new ArrayList<Integer>();
numberList = ns.sort(numberList);
//convert the list back to an array and return
return sortedNumbers; }
}
  
```

Table 3
Object oriented metrics [32].

Type	Metric	Description	Type	Metric	Description
Basic	CIW	Class Interface Width	Coupling	AOFD	Access Of Foreign Data
	NOA	Number Of Attributes		AUF	Average Use of Interface
	NOC	Number Of Classes		ChC	Changing Classes
	NOCN	Number Of Constructors		CM	Changing Methods
	NOIS	Number Of Import Statements		COC	Clients Of Class
	NOM	Number Of Members		CBO	Coupling Between Objects
	NOO	Number Of Operations		DAC	Data Abstraction Coupling
	NAM	Number of Accessor Methods		DD	Dependency Dispersion
	CL	Class Locality		FO	FanOut
	LCOM1	Lack of Cohesion Of Methods 1		MIC	Method Invocation Coupling
Cohesion	LCOM2	Lack of Cohesion Of Methods 2	Inheritance	NOED	Number Of External Dependencies
	LCOM3	Lack Of Cohesion Of Methods 3		VOD	Violations of Demeters Law
	TCC	Tight Class Cohesion		WCM	Weighted Changing Methods
	AC	Attribute Complexity		DOIH	Depth Of Inheritance Hierarchy
Complexity	CC	Cyclomatic Complexity	Inheritance-based coupling	NOCC	Number Of Child Classes
	EC	Essential Complexity		TRAp	Total Reuse of Ancestor percentage
	MDC	Module Design Complexity		TRAu	Total Reuse of Ancestor unitary
	NORM	Number Of Remote Methods		TRDu	Total Reuse in Descendants unitary
	RFC	Response For Class	Maximum	TRDu	Total Reuse in Descendants unitary
	WOC	Weight Of a Class		MNOL	Maximum Number Of Levels
	WMPC1	Weighted Methods Per Class 1		MNOP	Maximum Number Of Parameters
	WMPC2	Weighted Methods Per Class 2		NOAM	Number Of Added Methods
			Polymorphism	NOOM	Number Of Overridden Methods

calculate 45 metrics [32] for each class of design pattern instances. Table 3 shows these metrics which are divided into eight collections including Basic, Cohesion, Coupling, Complexity, Inheritance, Inheritance-Based Coupling, Maximum, and Polymorphism.

2.3. Classification methods

In general, machine learning is programming computers to optimize a performance criterion using example data or past experiences [33]. The example data are represented as individuals, where several measurements are made on each one of them generating an observation vector. The sample may be viewed as a data matrix

$$X = \begin{bmatrix} X_1^1 & X_2^1 & \cdots & X_d^1 \\ X_1^2 & X_2^2 & \cdots & X_d^2 \\ \vdots & \vdots & \ddots & \vdots \\ X_1^N & X_2^N & \cdots & X_d^N \end{bmatrix}$$

where d columns correspond to d variables called features denoting the result of measurements made on an individual. In fact, the above matrix includes N row where each row shows d features of the corresponding individual.

Supervised automatic classification [33] is a machine learning technique, i.e., assigning individuals automatically to one or several predefined categories. The classification methods usually use two sets, a training set and a test set. The training set is used for learning some classifiers and requires a primary group of labeled individuals, in which the category related to each individual is obvious from its label. The test set is used to measure the efficiency of the learned classifiers and includes labeled individuals which do not participate in learning classifiers. Classifications methods are usually made in binary mode, which means the label of each individual is either +1 or -1, i.e., each individual is assigned to either one category or its complementary. Note that there are several classification methods [33] like Naïve Bayesian, K-Nearest Neighbor (KNN), C4.5 Decision Trees, and Support Vector Machines (SVM) [34,35] methods.

3. Proposed method

Our main motivation for using classification methods is their abilities in the right design pattern selection [4], therefore, we use it in the proposed method to organize design patterns and to automatically detect right design pattern from a given source code. In addition, another motivation is the fact that design pattern detection problem is similar to an IR (Information Retrieval) problem; therefore, with regard to the literature [26,27,36], we can use the classification methods for solving the design patterns detection problem.

Fig. 1 illustrates the proposed process of design pattern detection from source code. This process is divided into two phases, design pattern organization phase and design pattern detection phase. The following sections describe the steps of the process.

3.1. Design pattern organization phase

In this section, we intend to explain how to learn design pattern classifiers. The input of the design pattern organization phase is a set of design pattern implementations which manually determined by experts and the output of this phase is some design pattern classifiers. In fact, in design pattern organization phase, for each design pattern, we learn a binary classifier, like an expert, which indicates whether the candidate combination of given source code classes is an instance of that design pattern or not. The process of learning the classifiers is divided into three following steps as shown in Fig. 1 (Phase I).

3.1.1. Creating feature vectors

As mentioned in Section 2.3, in classification methods, each instance is presented by some features and classifiers are learned by using the training set having feature vectors and labels for each instance (i.e., the design pattern name). To prepare the training instances, suppose we have a set of design pattern instances per design pattern, then, we construct a feature vector for every design pattern instance. The features for each design pattern instance are the Object-Oriented metrics mentioned in Section 2.2 (Table 3). They are calculated for all classes that play a role in that instance. For example, if a design pattern has b roles and the number of metrics is n , then, the number of features for each design pattern instance

Table 4
An instance of the adapter design pattern in netbeans.

Classes of “Target” role	Classes of “Client” role
org.openide.filesystems.FileChangeListener	org.openide.filesystemds.AbstractFileObject.Invalid org.openide.filesystems.AbstractFolder org.openide.filesystems.FileObject org.openide.filesystems.FileChangeListener
Classes of “Adapter” role	Classes of “Adaptee” role
org.netbeans.core.ClassLoaderSupport org.openide.util.WeakListener.FileChange org.openide.filesystems.FileChangeAdapter org.openide.loaders.MultiDataObject.EntryL org.netbeans.modules.corba.IDLDataObject.FileListener org.openide.loaders.FolderList org.openide.loaders.XMLDataObject.InfoParser org.openide.filesystems.MultiFileObject org.netbeans.core.Packages	org.openide.filesystems.FileObject org.openide.filesystems.FileChangeListener org.openide.loaders.DataObject org.netbeans.modules.corba.IDLDataObject org.openide.loaders.XMLDataObject

3.1.3. Learning classifiers of design patterns

In this section, the proposed method intends to learn appropriate classifiers to detect precisely design pattern samples from a given source code. With this concern, we extend our previous binary classification method, called SVM-PHGS [31] which is an effective classification method, to learn design pattern classifiers. Fig. 3 illustrates the scheme of the extended SVM-PHGS which consists of three steps: (1) finding the optimal values of hyper parameters, (2) finding the best weights to fuse opinions of all kernels, and (3) computing confidence range.

There are two hyper-parameters for SVM kernels: C and γ . The best value of these parameters depends on the nature of the given problem. In real-world applications, selecting the appropriate hyper-parameters for SVM is a difficult and vital step which impacts the generalization capability and classification performance of its learned classifier. To find the optimal value of these parameters, SVM-PHGS uses a parallel hierarchal greedy search in the first step as shown in Fig. 3. Having obtained the best pair of (C, γ), SVM-PHGS builds a binary learning classifier for each design pattern.

In fact, SVM-PHGS uses some kernels that usually are RBF, linear, and polynomial kernels. In the second step shown in Fig. 3, SVM-PHGS calculates dynamically some weights and considers them as the opinion of all three kernels for fusing their opinions.

In original SVM-PHGS, each binary classifier has a confidence value that expresses the certainty of belonging a given sample to its category. To achieve a suitable confidence range for each classifier, in the third step shown in Fig. 3, we divide the corresponding design patterns pool (the input to SVM-PHGS) into 70% and 30% as training set and validation set, respectively. After that, for each design pattern belonging to the training set, one classifier is learned according to the best obtained parameters for SVM-PHGS. Then, the learned design pattern classifiers are evaluated by the validation set and for each sample of the validation set; its confidence value is calculated. Then, the confidence range for each classifier is calculated according to Eq. (5). In Eq. (5), $ConfidenceRange_m$ denotes the confidence range of classifier m and $ConfidenceValueSet_m$ denotes a confidence

value set of the samples of the design pattern m that correctly identified by the learned design pattern classifiers. These confidence ranges help the proposed method to reduce false positive detections. In the design pattern detection phase of the proposed method (see Fig. 1), when a design pattern classifier informs about the design pattern of a given source code if the confidence value of this classifier is not in the learned confidence ranges, the proposed method does not identify any design pattern for the given source code.

$$ConfidenceRange_m = [Min(ConfidenceValueSet_m), Max(ConfidenceValueSet_m)] \tag{5}$$

3.2. Design pattern detection phase

The input of this phase is a given source code and the output is design pattern instances existing in the given source code. To perform this phase, the proposed method uses the classifiers learned in the previous phase to detect what groups of classes of the given source code are design pattern instances. This phase is divided into two steps, preprocessing and detection.

3.2.1. Preprocessing

In this section, we try to partition a given system source code into suitable chunks as candidate design pattern instances. Tsantalis et al. [7] presented a method for partitioning a given source code based on inheritance hierarchies, so each partition has at most one or two inheritance hierarchy. This method has a problem when some design pattern instances involving characteristics that extend beyond the subsystem boundaries (such as chains of delegations) cannot be detected. Furthermore, in a number of design patterns, some roles might be taken by classes that do not belong to any inheritance hierarchy (e.g., Context role in the State/Strategy design patterns [1]).

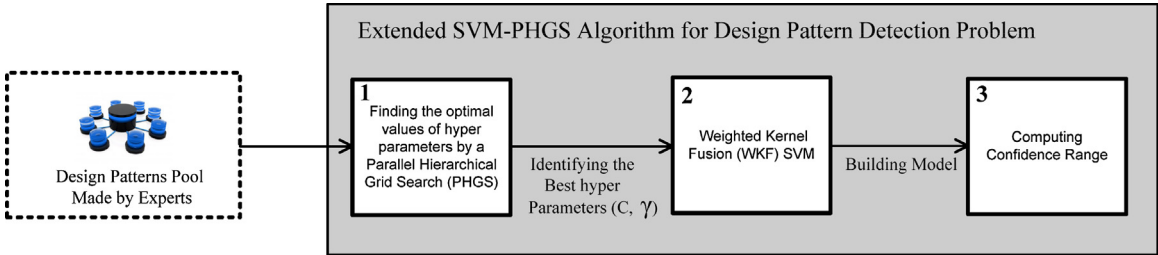


Fig. 3. The process of extended SVM-PHGS in the proposed design pattern detection method.

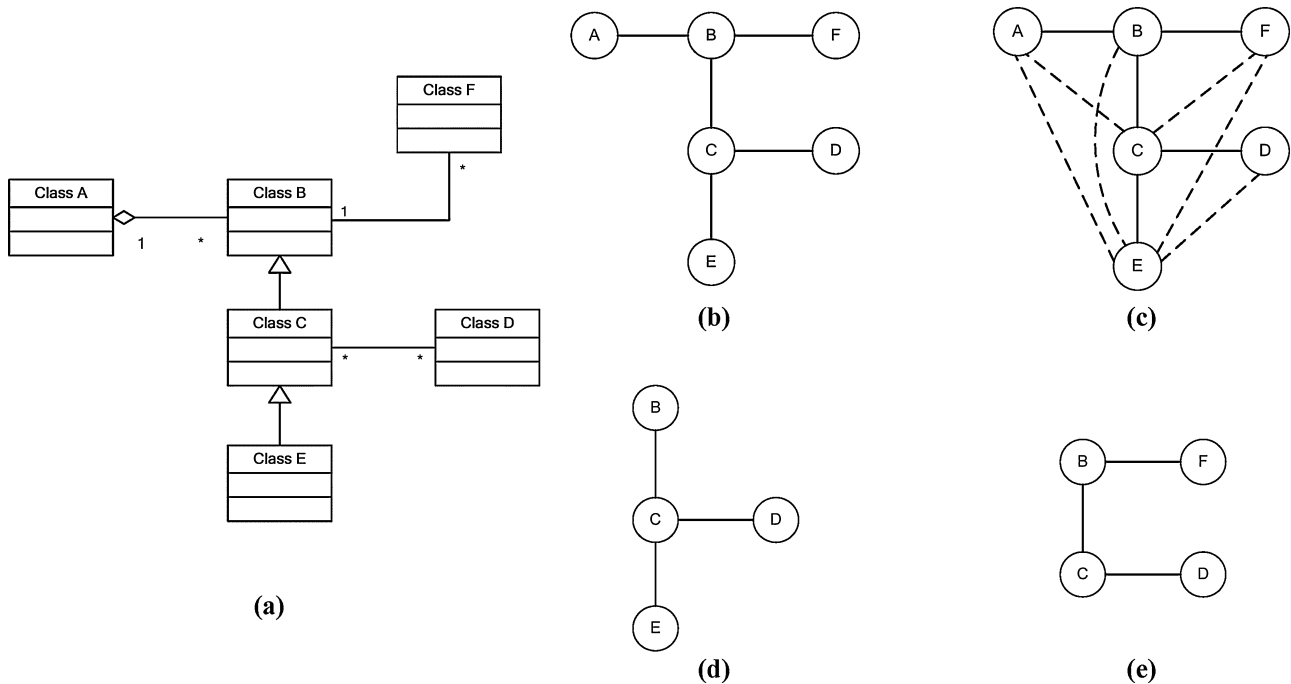


Fig. 4. Example of translating a source code class diagram to an association graph.

In order to improve the limitations of the method presented in [7], we propose a new procedure that candidates each combination of b classes as a design pattern instance, where b is the number of roles of the desired design pattern. Algorithm 1 gives the pseudo code for the proposed preprocessing procedure.

Algorithm 1. The proposed preprocessing procedure

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
3. Search all connected subgraphs with b number of vertices from G as candidate design pattern instances
4. Filter candidate design pattern instances that haven't any abstract classes or interfaces

In the proposed preprocessing procedure, we look for b -tuple of classes in which all classes are connected together by some association, generalization or realization relationships. To candidate b number of classes as input to a design pattern classifier that has b roles too, we initially transform the given source code class diagrams to its correspond graph; For example, Fig. 4(b) depicts an association graph for a class diagram illustrated in Fig. 4(a), where each vertex is a distinct class, and each edge shows a dependency between two classes. Then, we explore the association graph of the source code to find the connected subgraphs that have b vertices, i.e., if we intend to detect a design pattern with four roles, we have to look for any connected subgraphs that have four vertices. For example, two connected subgraphs with four vertices for the original graph shown in Fig. 4(b) are depicted in Fig. 4(d) and (e).

According to several design pattern implementations, we find out that in inheritance hierarchy of a source code class diagram, a child class along with some classes that have a dependency with its parent class may be a design pattern instance, for example, in Fig. 4(a), there may be a design pattern instance, includes four classes, A, B, F, and E. Therefore, to help overcome this difficulty, we enrich the extracted association graph by adding the parent's relationships to the child class. For example, we add a new edge A–C to enrich the graph presented in Fig. 4(b), because in Fig. 4(a),

Class A is composed of some Class B (Parent class) using aggregation relationship and Class C is a child of Class B, therefore; between vertices C and A in the graph presented in Fig. 4(b), a new edge is added. The result of enriching graph for the class diagram shown in Fig. 4(a) is depicted in Fig. 4(c).

It is worth mentioning that design patterns usually include at least one abstract or interface class in one of their roles, so to reduce the number of subgraphs, we filter them by choosing the subgraphs which have an abstract class or interface.

3.2.2. Design pattern detection

In this step, each candidate combination of classes produced in the preprocessing step is given to each design pattern classifier learned in Phase I of the proposed method in order to identify whether the candidate combination of classes is related to the design pattern that the classifier is expert on. Then, each classifier states its opinion with a confidence value. Finally, if the confidence value of the candidate combination of classes is located in the confidence range of that design pattern, then, the combination is a design pattern, otherwise it is not.

4. Results of the proposed method evaluation

To evaluate the proposed method, in this section, we learn some classifiers for six design patterns and these classifiers are evaluated with some design patterns instances. In the following, the obtained results of the proposed method are reported in two subsections: In the first subsection, different classification methods are evaluated and the best classification method for detecting the right design patterns along with its best parameter values will be selected. Next, in the second subsection, the proposed method is evaluated with three open source projects and is compared with other related works.

4.1. Evaluation results of design pattern organization phase

The evaluation of classifiers is typically conducted experimentally, rather than analytically. Note that there are several methods

Table 5
Learning samples of design patterns extracted from P-mart [26].

Design pattern	Number of training samples	Number of test samples	Total number of samples
Adapter	20	11	31
Builder	4	2	6
Composite	4	3	7
Factory method	5	3	8
Iterator	5	3	8
Observer	7	4	11

for classification, but it cannot be said which one is better than the others [33]. The reason is that there is no classification method that achieves good results for all problems. In real life, some of them may achieve good results for some problems and bad results for the rest [33]. Therefore, for a new problem (i.e., design pattern detection from source code) to find out the best classification method, we evaluate some classification methods so that the best classification method is identified. Thus, one of the contributions of this paper is to suggest the best classification method compatible with the design pattern detection problem. In this section, we apply some classification methods [33] such as Simple Logistic, C4.5 decision tree, original SVM, in addition to the extended SVM-PHGS described in Section 3.1.3. Among the well-known machine learning tools, we use WEKA data mining tool [30], which includes different types of classification methods.

In this section, the effectiveness of the proposed method is evaluated using six design patterns that have 4 roles. We learn some classifiers for *Adapter*, *Builder*, *Composite*, *Factory Method*, *Iterator*, and *Observer* patterns. We use a repository of pattern-like micro-architectures, called P-mart [26], which is manually detected and gathered from 9 case studies. In this evaluation, both training and tests set are obtained from the P-mart repository, where approximately 70 percent are randomly selected for training the classifiers and the rest (approximately 30 percent), are considered for testing the classifiers. Table 5 illustrates the number of these learning samples. We ran each classification method 30 times, where in each run

the training and test samples are randomly selected from the original P-mart repository. In the following, the best results of these runs are reported.

We used *JBuilder* metric tool to calculate 45 metrics for each class playing a role in the design pattern instances shown in Table 5. Moreover, when a role is played by more than one class, we apply all methods that are described in Section 3.1.1 to aggregate metrics of all classes that play the same role in a design pattern instance. Note that to make the real training set, we made up a sequence of 4×45 features for each design pattern instance derived from the original training set, and then, add 23 of its permutation samples to the real training set.

The classification effectiveness is usually measured in terms of four parameters [36], Precision (P), Recall (R), F_1 (as combination of P and R using Eq. (6)), and False Positive error rate (FP). To estimate P and R for learned classifiers, either micro-averaging equations or macro-averaging equations can be used. We use micro-averaging equations (see Eqs. (7) and (8)), because they are more precise than macro-averaging equations [36]. In the following equations, C is the number of design patterns, TP is the number of design pattern instances which belong to a design pattern and the classifier has correctly identified them, FP is the number of design pattern instances which do not belong to a design pattern but the classifier has incorrectly identified them as belonging to that design pattern, and FN is the number of design pattern instances which belong to a design pattern but the classifier has not identified them as belonging to that design pattern.

$$F_1 = \frac{2 \times P \times R}{(P + R)} \quad (6)$$

$$P = \frac{\sum_{i=1}^{|C|} TP_i}{\sum_{i=1}^{|C|} TP_i + FP_i} \quad \text{Micro-Averaging} \quad (7)$$

$$R = \frac{\sum_{i=1}^{|C|} TP_i}{\sum_{i=1}^{|C|} TP_i + FN_i} \quad \text{Micro-Averaging} \quad (8)$$

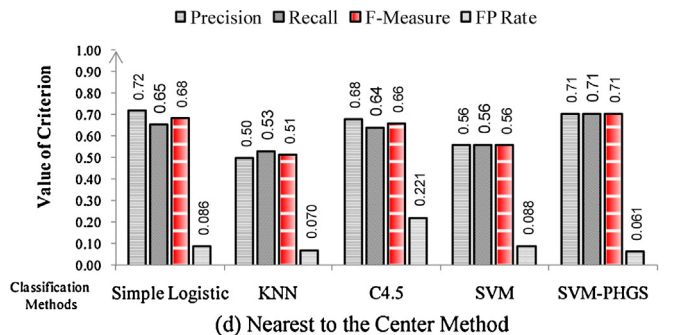
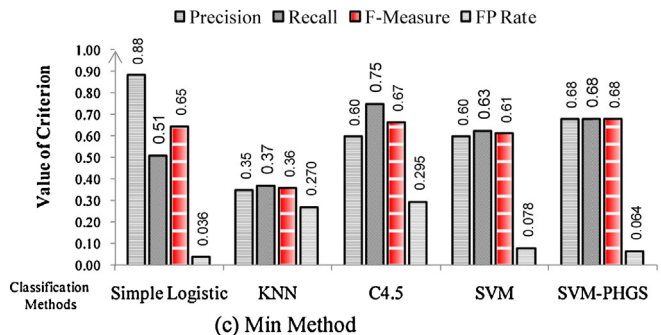
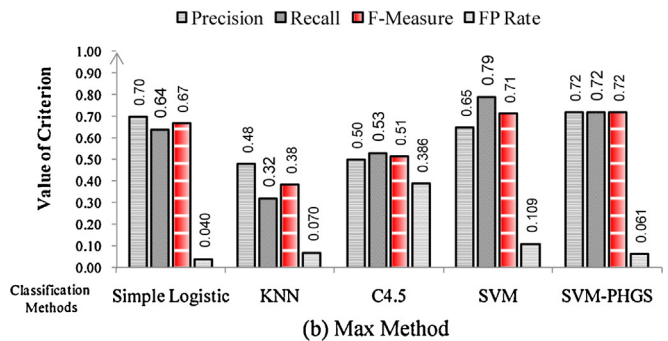
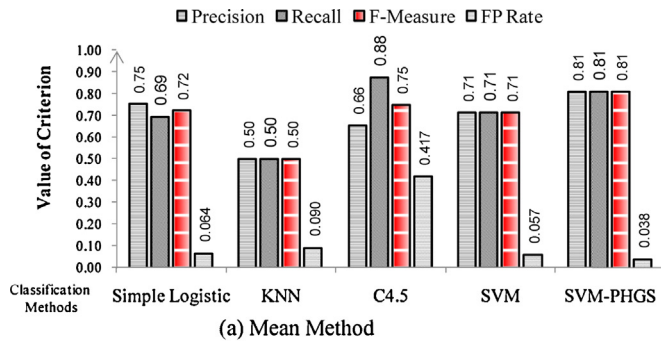


Fig. 5. The evaluation results of learning design patterns.

Table 6

The confidence range of used design pattern classifiers.

Design pattern	The confidence range for its classifier
Adapter	[0.874, +∞)
Builder	[0.999, +∞)
Composite	[0.072, 1.582]
Factory method	[0.874, 1.308]
Iterator	(-∞, +∞)
Observer	(-∞, 1.044]

Fig. 5 compares the evaluation results of the five classification methods, Simple Logistic, C4.5, KNN, SVM, and the extended SVM-PHGS. In this paper, in KNN classification method, we assumed $K=3$.

Fig. 5(a), (b), (c), (d) show the results of learning design patterns, when the *Mean*, *Max*, *Min*, and *Nearest to the Center Methods* are used, respectively, for aggregating metrics of all classes that play the same role in a design pattern instance. As shown in Fig. 5, KNN achieves the worst results, Simple Logistic and C4.5 achieve reasonable results according to P and R criteria, respectively, and according to F_1 criterion, the extended SVM-PHGS achieves the best results. Comparing these results reveals two distinguishing outlines:

- A. In all experiments, according to F_1 criterion, the extended SVM-PHGS achieves the best results. Additionally, in all 30 experiments, the extended SVM-PHGS achieves the same value for P and R criteria. In fact, SVM-PHGS attempts to make a trade-off between P and R criteria and as opposed to other classification methods like C4.5, it does not sacrifice R criterion to P criterion and vice versa (like Simple Logistic). Determining the best classification method, F_1 criterion is chosen, because it is a common criterion to fuse both P and R criteria.
- B. Among four methods for aggregating metrics introduced in Section 3.1.1, using the *Mean Method* (Fig. 5(a)) leads to achieve the best result.

As mentioned in Section 3.1.3, in order to correctly detect design patterns, we need to compute a confidence range for each design pattern classifier, so that the confidence value of the accepted samples must be located in this range. For the case of *Mean Method* and extended SVM-PHGS (see Fig. 5(a)), Table 6 shows the obtained confidence range for each design pattern classifier. It should be noted that the confidence range has an important impact on the trade-off between precision and recall criteria. In fact, when a wide range is considered as the confidence range, the recall criterion is increased, but the precision criterion is decreased.

As shown in Table 6, for *Iterator* design pattern there is no particular range so any confidence value is accepted, because in the performed experiments, this classifier has not any false positive.

4.2. Evaluation results of design patterns detection phase

In this section, the proposed method is evaluated using three open source projects: *JHotDraw 5.1*, which is a GUI framework for technical and structured Graphics, *JRefactory 2.6.24*, which is a refactoring tool for the Java programming language, and *JUnit 3.7*, which is a regression testing framework for implementing unit tests in Java. These case studies are selected for the following reasons: (1) they are mainly used in the literature and this leads to make a comparison with other works, (2) for these open source projects, formerly some experts determined their design patterns from their source codes, therefore, in this way it is possible to evaluate the results of the proposed method using expert opinions, and (3) all of them are open source projects with their source code publicly available.

Table 7

Results of the preprocessing step of the proposed method.

Case study	Number of classes	Number of possible design patterns ($k=4$)	Number of candidate design patterns obtained by the proposed method	Rate of reduction samples using the proposed method
JHotDraw	155	23,130,030	9355	99.96%
JRefactory	575	4,507,327,825	39,422	99.99%
JUnit	94	3,049,501	2688	99.91%

In the preprocessing step of the proposed method (Section 3.2.1), for each case study, (i.e., source code) at first, a graph representing the relationships between its classes is constructed, then, all connected subgraphs that have 4 vertices are determined. Each connected subgraph represents one candidate design pattern. Since design patterns usually have one abstract or interface class in one of their roles, then, candidate design patterns are filtered by choosing the subgraphs which have an abstract or interface class. The number of possible k -combinations of a set of n element is calculated by Eq. (9) as follows:

$$C(n, k) = \binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (9)$$

For example, the number of possible 4-combinations of classes from the *JUnit* case study, which has 94 classes, is equal to $C(94, 4) = 3,049,501$. Note that the order of the classes in each candidate combination is not matter because all the permutations of each design pattern are learned. For the *JUnit* case study, the preprocessing step reduces the candidate 4class-combinations to 2688 samples, which means that the candidate design patterns are reduced to less than 1% of all possible candidates. Table 7 shows the number of candidate samples produced in the preprocessing step.

As shown in Table 7, the proposed method in the preprocessing step is able to greatly reduce (more than 99%) candidate design patterns from a given source code. This reduction has two advantages: improving the performance and decreasing the overall execution time of the proposed method.

After calculating the metrics for the candidate combinations of 4 classes, a feature vector for each candidate design pattern is created according to Section 3.1.1. Then, each feature vector is given to the classifiers learned in Section 4.1 to identify the right design pattern instances.

Table 8 compares the results of the proposed method and other related works for *JHotDraw*, *JRefactory*, and *JUnit* open source projects. It should be noted that the results of DPD Tool [7], PINOT Tool [37], FUJABA Tool [14], and Web of pattern Tool [24] for these open source projects in Table 8 are reported from [38] and [7].

As illustrated in Table 8, for some design patterns like *Factory Method* and *Observer* patterns, the obtained results of the proposed method are better than the results of other methods. However, we believe that for the other design patterns, we cannot compare quantitatively the effectiveness of the proposed method to other methods, because there are three main problems which are discussed below.

- A. It seems likely that the results of the presented design pattern detection tools like PINOT [37] are not promising, because when some researchers [38] evaluated experimentally the performance of this tool, unfortunately, the reported results are not achieved.
- B. Pettersson et al. [39] survey some problems usually faced in evaluating the accuracy in design pattern detection methods. For example, the problem of *Pattern Occurrence Type* has a main impact on the calculated precision and recall criteria. In

Table 8
Results of design pattern detection using three used open source projects.

Case studies	Design patterns	DPD tool (Similarity scoring) [7]			PINOT Tool (Logical reasoning) [37]		FUJABA Tool (Logical reasoning) [14]		Web of pattern (Ontology based) [24]		Proposed method (Learning based)		
		P	R	F ₁	P	R	P	R	P	R	P	R	F ₁
JHotDraw	Adapter	48%	100%	65%	100%	–	19%	–	0%	–	86%	86%	86%
	Composite	100%	100%	100%	0%	–	–	–	0%	–	74%	74%	74%
	Factory method	50%	66%	59%	31%	–	50%	–	–	–	61%	61%	61%
JRefractory	Adapter	–	100%	–	–	–	–	–	–	–	76%	76%	76%
JUnit	Iterator	–	–	–	–	–	–	–	–	–	100%	100%	100%
	Observer	–	100%	–	–	–	–	–	–	–	100%	100%	100%

the field of design pattern detection, the pattern occurrence is represented as a tuple of the participating classes, interfaces, and possibly methods. The problem of *Pattern Occurrence Type* is defined as the problem of what roles of the pattern are represented in the list of matching occurrences [39]. For example, in the DPD tool [7], an *Adapter* design pattern instance is detected when only two roles (*Adapter* and *Adaptee*) are matched, then, the user needs to manually find the missing roles (*Target* and *Client*). Analogously, in the proposed method, an *Adapter* design pattern instance is detected only when all the four roles (*Adapter*, *Adaptee*, *Target*, and *Client*) are matched. It should be noted that experiments show that the precision criterion usually decreases when the pattern occurrence size is increased [39]. On the other hand by increasing the pattern occurrence size, the number of expected occurrences will be increased. For example, consider the *Adapter* design pattern instances presented in Table 4, in the proposed method, we expect to have 180 occurrences of this instance, however, this instance can be seen as only 45 occurrences if some roles (i.e., *Target* and *Client*) are omitted. Then, we forced to report more false positive and false negative occurrences than the other methods.

- C. The presented papers usually do not report both the precision and recall criteria. It should be noted that when the precision criterion is increased, the recall criterion usually is decreased. As mentioned earlier in Section 4.2, the extended SVM-PHGS uses the confidence range and this confidence range make a trade-off between precision and recall criteria. Taking into consideration that we can increase the precision by reducing the recall. The problem of trading off between precision and recall is one of the problems usually faced in evaluation of design pattern detection methods [39].

As a conclusion, making an accurate comparison between the proposed method and the other methods is so hard.

5. Discussion

The attempts made to automatically detect a design pattern from a given source code can be divided into five categories whose drawbacks are discussed in detail below.

Logical reasoning – Logical reasoning based methods cannot identify incomplete occurrences [8–12]. Although, [13,14] use the fuzzy logic to identify approximate matching, but they suffer from high false positive rate.

Similarity scoring – Similarity scoring methods intend to match the structural aspect of a design pattern, so they cannot differentiate between design patterns that are structurally similar, such as *Composite/Decorator*, *Strategy/State* patterns. In [17–19], the authors compound the dynamic analysis with static analysis to achieve high accuracy; however, they do not grantee to detect incomplete occurrences.

FCA-based method – FCA-based methods use the FCA algorithm to infer the presence of class groups which instantiate repeated pattern, however, they have the problem of large quantity of computation, for example, in [18] the detection of relatively simple structures in relatively small pieces of source code required a lot of calculations, so they cannot detect the patterns which consist of more than four classes.

Ontology-based method – Ontology based methods provide template ontology descriptions that must be extended to become more expressive. Additionally, it lacks support for dynamic analysis [25].

Learning-based method – Learning-based methods have the ability to detect incomplete occurrences in comparison to logical reasoning and similarity scoring methods. Also, they are faster than FCA-based method and they need a lower cost preprocessing in comparison to ontology-based methods. However, there are some weaknesses in the previous methods based on machine learning. Guéhéneuc et al. [27] learned just some individual classes of design patterns and not the whole. Ferenc et al. [28] present another design pattern tool to candidate the probable design patterns. However, they use learning methods after structural matching phase to filter possible false hits per pattern rather than classifying them. Arcelli and Cristina [29], at first, classify design patterns using neural networks and WEKA data mining algorithms, then, they test the classifier on manually collected samples and do not test their method on real source codes.

The main contribution of this paper is to map the design pattern detection problem into learning problem; the proposed design pattern detector is made by learning from the information extracted from design pattern instances which normally include variant implementations. Therefore, the proposed method has the ability of detecting variant implementations and incomplete occurrences. Additionally, not only it does not use another design pattern detections tool for the preprocessing like [28], but also it uses a novel preprocessing idea to greatly reduce the number of candidate design patterns from a given source code.

6. Conclusion

In this paper, a machine learning based method for design pattern detection from source code is proposed. The advantage of using machine learning is that the proposed design pattern detector is made by learning from the information extracted from design pattern instances which normally include variant implementations, so the proposed method is able to identify different versions of design patterns. The proposed method uses a novel method for preprocessing source codes and the obtained results show that it is able to greatly reduce (more than 99%) the number of candidate design patterns from a given source code. In addition, the proposed method extends the SVM-PHGS classification method to identify the right design patterns. To evaluate the proposed method, we used a repository of pattern-like micro-architectures, called P-mart, which is manually detected and gathered from 9 case studies. The obtained

results show that the extended SVM-PHGS classification method outperforms the other classification methods and achieves suitable results (i.e., $P=0.81$, $R=0.81$, $F_1=0.81$, and $FP=0.038$) for six design patterns. In future works, we intend to apply the proposed method on the other design patterns by different number of roles and for improving the results, and we intend to use it in software components identification methods [40–42].

Acknowledgment

This research in part was supported by ITRC, Iran Telecommunication Research Center.

References

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
- [2] G. Antoniol, G. Casazza, M. Di Penta, R. Fiutem, Object-oriented design patterns recovery, *J. Syst. Software* 59 (2001) 181–196.
- [3] S.M.H. Hasheminejad, S. Jalili, Selecting proper security patterns using text classification, in: *International Conference on Computational Intelligence and Software Engineering*, CiSE, 2009, pp. 1–5.
- [4] S.M.H. Hasheminejad, S. Jalili, Design patterns selection: an automatic two-phase method, *J. Syst. Software* 85 (2012) 408–424.
- [5] F.A. Fontana, M. Zaroni, A tool for design pattern detection and software architecture reconstruction, *Inform. Sci.* 181 (April) (2011) 1306–1324.
- [6] A. Wierda, E. Dortmans, L. Somers, Pattern detection in object-oriented source code, *Software Data Technol.* 22 (2009) 141–158.
- [7] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, *IEEE Trans. Software Eng.* 32 (November) (2006) 896–909.
- [8] C. Kramer, L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in: *The 3rd Working Conference on Reverse Engineering*, 1996, p. 208.
- [9] R. Wuyts, Declarative reasoning about the structure of object-oriented systems, in: *Technology of Object-Oriented Languages*, 1998, pp. 112–124.
- [10] J.M.C. Smith, D. Stotts, SPQR: flexible automated design pattern extraction from source code, in: *18th IEEE International Conference on Automated Software Engineering (ASE'03)*, 2003, p. 17.
- [11] J. Fabry, T. Mens, Language-independent detection of object-oriented design patterns, *Comp. Lang. Syst. Struct.* 30 (2004) 21–33.
- [12] S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, M. Saeki, Design pattern detection by using meta patterns, *IEICE Trans. Inform. Syst.* E91d (April) (2008) 933–944.
- [13] J. Jahnke, A. Zündorf, Rewriting poor design patterns by good design patterns, in: *1st ESECFSE Workshop on Object-Oriented Reengineering*, 1997.
- [14] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: *24th International Conference on Software Engineering*, 2002, pp. 338–348.
- [15] J. Dong, Y. Sun, Y. Zhao, Design pattern detection by template matching, in: *2008 ACM Symposium on Applied Computing*, 2008, pp. 765–769.
- [16] O. Kaczor, Y.G. Guéhéneuc, S. Hamel, Identification of design motifs with pattern matching algorithms, *Inform. Software Technol.* 52 (2010) 152–168.
- [17] D. Heuzeroth, T. Holl, W.L. We, Combining static and dynamic analyses to detect interaction patterns, in: *presented at the the 6th World Conference on Integrated Design and Process Technology*, 2002.
- [18] J. Dong, D.S. Lad, Y. Zhao, DP-Miner: design pattern discovery using matrix, in: *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, 2007, pp. 371–380.
- [19] J.K.Y. Ng, Y.G. Guéhéneuc, Identification of behavioral and creational design patterns through dynamic analysis, in: *3rd International Workshop on Program Comprehension through Dynamic Analysis*, 2007, pp. 34–42.
- [20] P. Tonella, G. Antoniol, Inference of object oriented design patterns, *J. Software Maint. Evol.: Res. Pract.* 13 (2001) 309–330.
- [21] F. Buchli, *Detecting Software Patterns Using Formal Concept Analysis*, University of Bern, Bern, Switzerland, 2003.
- [22] G. Arevalo, F. Buchli, O. Nierstrasz, Detecting implicit collaboration patterns, in: *Working Conference on Reverse Engineering*, 2004, pp. 122–131.
- [23] V. Tripathi, T.S.G. Mahesh, A. Srivastava, Performance and language compatibility in software pattern detection, in: *Advance Computing Conference, IEEE International*, 2009, pp. 1639–1643.
- [24] J. Dietrich, C. Elgar, Towards a web of patterns, *Web Semant.: Sci. Serv. Agents World Wide Web* 5 (2007) 108–116.
- [25] D. Kirasi, D. Basch, Ontology-based design pattern recognition, *Knowledge-Based Intell. Inform. Eng. Syst.* (2008) 384–393.
- [26] Y. Guéhéneuc, P-MART: pattern-like micro architecture repository, in: *1st Euro-PLoP Focus Group on Pattern Repositories*, Bavaria, Germany, 2008, pp. 1–3.
- [27] Y. Guéhéneuc, J. Guyomarc'h, H. Sahraoui, Improving design-pattern identification: a new approach and an exploratory study, *Software Qual. J.* 18 (2010) 145–174.
- [28] R. Ferenc, Á. Beszédes, L. Fülöp, J. Lele, Design pattern mining enhanced by machine learning, in: *International Conference on Software Maintenance (ICSM 2005)*, 2005, pp. 295–304.
- [29] F. Arcelli, L. Cristina, Enhancing software evolution through design pattern detection, in: *Third International IEEE Workshop on Software Evolvability*, 2007, pp. 7–14.
- [30] *Java Programs for Machine Learning*, University of Waikato, <http://www.cs.waikato.ac.nz/~ml/weka>, 1998–2006.
- [31] M.H. Zangoeei, S. Jalili, PSSP with dynamic weighted kernel fusion based on SVM-PHGS, *Knowledge-Based Syst.* 27 (2011) 424–442.
- [32] M. Lanza, R. Marinescu, *Object-oriented Metrics in Practice*, Springer-Verlag, Berlin Heidelberg, 2006.
- [33] E. Alpaydm, *Introduction to Machine Learning*, The MIT Press Cambridge, Massachusetts, 2010.
- [34] H.M. Azamathulla, F.C. Wu, Support vector machine approach to for longitudinal dispersion coefficients in streams, *Appl. Soft Comput.* 11 (2011) 2902–2905.
- [35] A. Guven, A predictive model for pressure fluctuations on sloping channels using support vector machine, *Int. J. Numer. Methods Fluids* 66 (2011) 1371–1382.
- [36] F. Sebastiani, Machine learning in automated text categorization, *J. ACM Comp. Surv. (CSUR)* 34 (2002) 1–47.
- [37] N. Shi, R. Olsson, Reverse engineering of design patterns from java source code, in: *International Conference on Automated Software Engineering*, 2006, pp. 123–134.
- [38] F.A. Fontana, M. Zaroni, S. Maggioni, Using design pattern clues to improve the precision of design pattern detection tools, *J. Object Technol.* 10 (2011) 1–31.
- [39] N. Pettersson, W. Lowe, J. Nivre, Evaluation of accuracy in design pattern occurrence detection, *IEEE Trans. Software Eng.* 36 (July–August) (2010) 575–590.
- [40] S.M.H. Hasheminejad, S. Jalili, SCI-GA: software component identification using genetic algorithm, *J. Object Technol.* 12 (2013) 1–34.
- [41] S.M.H. Hasheminejad, S. Jalili, An evolutionary approach to identify logical components, *J. Syst. Software* 96 (2014) 24–50.
- [42] G.R. Shahmohammadi, S. Jalili, S.M.H. Hasheminejad, Identification of system software components using clustering approach, *J. Object Technol.* 9 (2010) 77–98.