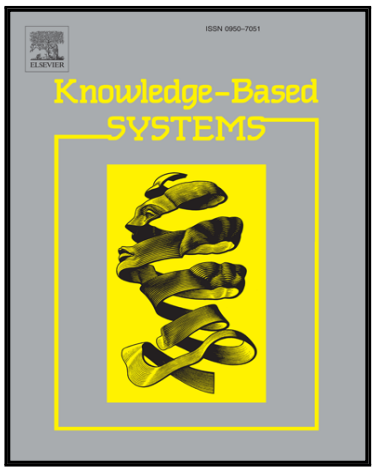


Design pattern detection based on the graph theory

Bahareh Bafandeh Mayvan, Abbas Rasoolzadegan

PII: S0950-7051(17)30008-4
DOI: 10.1016/j.knosys.2017.01.007
Reference: KNOSYS 3784

To appear in: *Knowledge-Based Systems*

Received date: 4 July 2016
Revised date: 2 January 2017
Accepted date: 3 January 2017

Please cite this article as: Bahareh Bafandeh Mayvan, Abbas Rasoolzadegan, Design pattern detection based on the graph theory, *Knowledge-Based Systems* (2017), doi: 10.1016/j.knosys.2017.01.007

Design pattern detection based on the graph theory

Bahareh Bafandeh Mayvan, Abbas Rasoolzadegan*

Faculty of Engineering, Ferdowsi University of Mashhad, Mashhad, Iran

Abstract

Design patterns are strategies for solving commonly occurring problems within a given context in software design. In the process of re-engineering, detection of design pattern instances from source codes can play a major role in understanding large and complex software systems. However, detecting design pattern instances is not always a straightforward task. In this paper, based on the graph theory, a new design pattern detection method is presented. The proposed detection process is subdivided into two sequential phases. In the first phase, we concern both the semantics and the syntax of the structural signature of patterns. To do so, the system under study and the patterns asked to be detected, are transformed into semantic graphs. Now, the initial problem is converted into the problem of finding matches in the system graph for the pattern graph. To reduce the exploration space, based on a predetermined set of criteria, the system graph is broken into the possible subsystem graphs. After applying a semantic matching algorithm and obtaining the candidate instances, by analyzing the behavioral signature of the patterns, in the second phase, final matches will be obtained. The performance of the suggested technique is evaluated on three open source systems regarding precision and recall metrics. The results demonstrate the high efficiency and accuracy of the proposed method.

Keywords: Design pattern detection, Pattern signature, Graph theory, Semantic graph

1. Introduction

Design patterns are the description of classes, objects, and relations between them that are customized to overcome a typical design issue in a particular context. In recent years, design patterns have attracted increasing attention in the software engineering research area and practice, as they encapsulate valuable knowledge, and influence the design quality heavily[1].

*Corresponding author

Email addresses: bahareh.bafandehmayvan@stu.um.ac.ir (Bahareh Bafandeh Mayvan), rasoolzadegan@um.ac.ir (Abbas Rasoolzadegan)

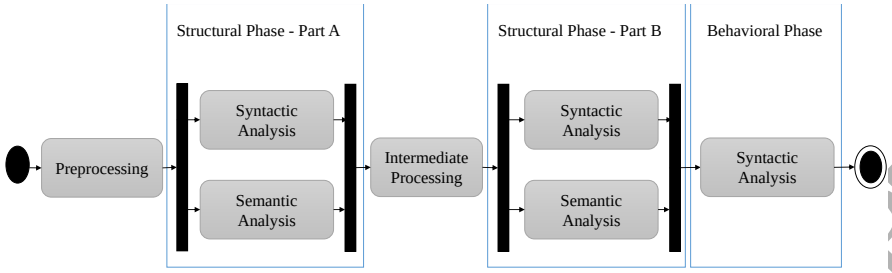


Figure 1: The abstract scheme of the proposed detection process

Searching, selecting, applying, verifying, and detecting design patterns are important issues related to this concern [2, 3, 4, 5]. By searching patterns, we mean obtaining information about existing patterns and by selecting patterns we mean choosing a pattern in a list of candidate ones [3]. Applying design patterns is the process of applying and incorporating patterns in the software development process [5]. Verification and detection of design patterns are two vital steps in reverse engineering process that can aid program comprehension. Design pattern verification is defined as the issue of checking whether a given implementation conforms to its specification or not whereas design pattern detection is the problem of finding motifs of the pattern in the code [4].

Here, we focus on the issue of the design pattern detection. The main motivation to detect design pattern motifs is providing a better perspective to recognize the original design decisions. Moreover, design pattern detection is significant in refactoring, re-engineering, maintenance, software quality measurement, program understanding, and improving the software documentation. However, it is not a straightforward task, furthermore, the missing information about implemented system, the ad-hoc nature of programming, and pattern variants, often cause the low accuracy of design pattern detection [6].

In this paper, we are going to take a step towards resolving major concerns which design pattern detection methods encountered with. In the related work section, we will discuss these concerns in details. To simplify and accelerate the process of detecting design patterns, we utilized a graph-based approach.

In the proposed method, we detect design patterns in two phases: structural and behavioral. Figure 1 demonstrates the overall process of the proposed method. In the structural phase, we apply both syntactic and semantic analysis whereas in the behavioral phase we just use syntactic analysis.

Here, in the preprocessing step, the UML class diagram corresponding to the system design is recovered first. UML is a popular and general-purpose modeling language, which provide a standard way to specify, construct, document, and visualize the artifacts of a system [7].

In the structural phase (Part A), after recovering the class diagrams corresponding to the system under study and the requested patterns, each of the diagrams will be converted into a semantic graph. In these graphs, nodes act as the classes and edges act as the relationship between two corresponding classes.

We used graph, because in software design, we need to use modeling languages which can combine rigor with simplicity and graphs are well-known and general structures for this concern [4]. Furthermore, by using graph structures, we can use numerous and efficient graph theory algorithms that can be employed to solve different problems[8, 9].

In the case of large software systems, the size of the exploration space grows. Therefore, in the intermediate processing step, based on a predetermined set of criteria, the system graph is partitioned into the clusters of candidate edges and their connected nodes. Consequently, in the structural phase (Part B), the matching algorithm is applied to smaller subgraphs rather than to the entire graph. The matching algorithm which is used here, is the Strong Simulation [10] that we revised it to fit our problem. Finally, in the behavioral phase, we compare candidates behavior with the predefined behavioral signatures of the patterns. In this way, we can obtain the balanced high precision and recall.

This study is applicable for all categories of Gang of Four (GoF) patterns [11] and any other design pattern catalogs. We present a precise and unambiguous specification of our method. Also, our experiments are conducted on real-world software systems, including JUnit 3.7, JHotDraw 5.1, and JRefactory 2.6.24. Finally, the performance of our method is evaluated by using two widely-adopted metrics, namely precision and recall. The results of experiments show improvement in detection performance.

Briefly, the main contributions of the proposed method for design pattern detection are: 1) distinguishing between patterns with similar structures, 2) covering patterns with any number of the roles, 3) handling variants of the patterns, 4) obtaining optimal and exact solutions in a polynomial time, 5) achieving high accuracy, and 6) being language independent.

The rest of the paper is organized as follows: Section 2 describes related work on design pattern detection. In Section 3 we define some concepts and terminologies that are used throughout the paper. In Section 4 we present an overview of our method. Section 5 describes our experiments on some large open-source systems. Threats to validity and future work are discussed in Section 6 and Section 7, respectively.

2. Related work

The most active subtopic of design pattern research is detection [12]. Figure 2 classifies the main characteristics of a design pattern detection approach. The key characteristics are “structural” and “behavioral”. Structural approaches are based on inter-class relationships and identify the structural properties of patterns (e.g., accessibility and type of attributes, interface hierarchies, method delegations, class inheritance, return types and parameters, modifiers of classes and methods) [13]. Behavioral approaches extract behavioral aspects of patterns and are based on dynamic or static program analysis techniques. They play a key role in differentiating between patterns that are structurally identical [14]. Both the structural and the behavioral characteristics can be divided further into “syntactic” and “semantic”. In the syntactic approaches, we concentrate

on the external features and the form of programming languages, whereas, in the semantic ones, we focus on the meaning [15]. It is obvious that an approach can include one or more of these characteristics.

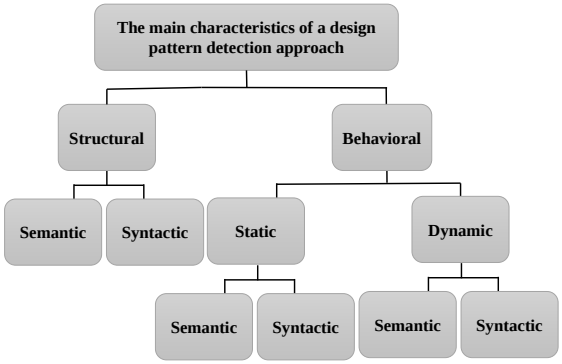


Figure 2: The main characteristics of a design pattern detection approach

2.1. Classification of the design pattern detection methods

In the following, the related works in the field of design pattern detection are classified into eight main groups based on their search method. We explain each group in details.

Quantitative approaches: These approaches compute class level metrics (such as numbers of generalizations, aggregations, associations, attributes, and operations) and then use different techniques to compare the metric values with the expected values for a design instance. The expected values are derived from the descriptions of the design patterns [16, 17, 18, 19, 20, 21]. Quantitative approaches are computationally efficient due to the filtration phase. These techniques do not consider the behavioral aspects of the design patterns, and usually analyze code or design syntactically. Based on the best knowledge of authors, just Issaoui et al. [20] have considered semantic metrics in their analysis. In this work, a *semantic-coverage* metric is defined. This metric determines the most probable correspondence between the design elements and the design pattern. The lack of interactivity and usually low precision and recall are the main drawbacks of quantitative approaches.

Query-based approaches: A number of design pattern detection techniques use database queries for extracting pattern related information and detecting design patterns [22, 23, 24, 25]. Query-based approaches are applied to an intermediate representation (like AST, ASG, XMI, UML, and ontology structures). Approaches, which are based on ontology, analyze the input model semantically. They use semantic queries to detect design pattern motifs [26, 27, 25]. The performance of query-based methods is limited by the information that is available in the intermediate representation. However, a

representation which could present all the information in the code is not rendered yet.

Similarity scoring approaches: Similarity scoring approaches represent the structural information of the software system under study and the input design patterns, as an appropriate intermediate representation (such as graphs) and then try to find matches between the structure of each of the design patterns and the system one [13, 28, 29]. These techniques cannot differentiate structurally similar design patterns. Therefore, they are usually supplemented by the behavioral analysis methods [30, 31, 32, 33].

Reasoning-based approaches: These approaches can be divided into two main groups: logical reasoning and fuzzy reasoning. Logical reasoning methods, at first, represent the detection conditions and then try to detect design patterns. These techniques usually use some mechanisms such as backtracking and database function. The main limitation is that they cannot identify approximate matching [34, 35, 36]. Methods in the other group describe design motifs as fuzzy-reasoning nets which express rules of detecting micro-architectures similar to design motifs. These methods can identify incomplete occurrences, but they suffer from the high false positive rate. Moreover, they are based on users assumptions and require the description of all possible variations of a design motif [37].

Constraint satisfaction approaches: Constraint satisfaction approaches use constraint programming to detect design motifs. They first translate the problem of pattern detection into a constraint satisfaction problem. Next, they describe the design instances as constraint systems where each role is represented as a variable and relationship among roles is represented as constraints among the corresponding variables. These techniques can ensure high recall rate but they usually scarify the precision rate, and the combinatorial complexity of them is prohibitive [38, 39, 40].

Formal approaches: These approaches use logical and mathematical methods. Therefore, formal techniques may be more expressive in some way, but they are not necessarily, more precise than the other matching methods. Formal approaches suffer from the large quantity of computation. Hence, they cannot find the design patterns which consist of more than a certain number of classes [41, 42, 43, 44, 45, 46, 47].

Parsing-based approaches: Parsing-based approaches usually map the visual language grammar of each design pattern to its corresponding graph representation. Then, try to detect pattern motifs by using visual language parsing methods. These techniques have a high precision, but they are, on their own, limited to the structural patterns [48, 49]. In [14, 50] authors try to eliminate this limitation by considering the behavioral aspects of the pattern.

Miscellaneous approaches: There are some techniques (e.g. machine learning [21], bit vector compression [6], minimum key structure method [51], model checking [52, 53], clustering and lexical information [54], etc.), for the detection of design pattern motifs that cannot be categorized in the above groups. However, they can be considered as a complementary to enhance the results of structural approaches.

According to the discussion of related work, there are some major concerns which design pattern detection methods encountered with (see Table A in Appendix A): 1) covering all kind of the patterns, 2) covering variants of the patterns, 3) being fully automatic, 4) achieving high accuracy, 5) being language independent, 6) having low complexity, 7) distinguishing between patterns with similar structures, and 8) evaluating the detection algorithm by a proper method. In this paper, we are going to take a step towards resolving the above mentioned concerns to improve the detection process.

2.2. Properties of design pattern detection tools

Design pattern detection methods can be classified using the following properties: analysis type, input type, recognition type and intermediate representation [55].

Analysis type: Most of the methods available in the literature, rely on static analysis. In a static analysis, we usually focus on the structural aspects of the source code. However, some methods are based on the behavior of the system at execution time and use dynamic analysis. The effectiveness of dynamic approaches relies on the used test case [56]. Moreover, a comprehensive static approach, which could recognize any patterns, is not proposed yet. In this paper, we use static analysis for both structural and behavioral aspects. In the first step, using graph theory, we concentrate on the structural aspects of the source code such as association, inheritance, and aggregation. Then in the next step, by checking the behavioral signatures of the patterns, we focus on the behavior of the system.

Input type: Design pattern detection tools operate on different systems as their input. Input systems can be coded in various programming language. In this context, we choose UML class diagram as our input type. Therefore, in a preprocessing step, we convert the system source code to its corresponding class diagram. According to experts, the visual notation is the best way to articulate design decisions, and UML is the most common visual modeling language in the software industry today.

Intermediate representation: To facilitate the analysis of the system, it is often useful to transform the information obtained from the analyzed system in a more abstract format. Here, to simplify the analysis of the system, we convert the UML class diagrams into the semantics graphs. We choose graph for the system intermediate representation, because graphs are easy to understand and interpret data. In this way, we ignore unnecessary details that impose extra complexity to the system.

Recognition type: The recognition process of design pattern instances can be classified into two main categories: 1- exact or inexact, 2- optimal or approximate. We detail each class in the following. Throughout our algorithm, by applying an exact matching, we obtain optimal solutions in a polynomial time.

Exact vs. inexact matching: An exact design pattern detection algorithm returns only results that detect a specified pattern exactly. In contrast, an inexact one returns a ranked list of the most similar motifs.

Optimal vs. approximate solutions: Optimal detection algorithms are guaranteed to find a true solution, but they often have exponential worst-case complexity. In contrast, approximate algorithms have polynomial complexity, but they are not guaranteed to find a true solution.

Recovered Patterns types: GoF design patterns can be categorized into three main groups: creational, structural, and behavioral. Tools which detect design motifs, concentrate on one or in some cases more than one category. Our method can cover all three groups. Regardless of the type of the patterns looking for, at first, we analyze structural information of them, then, we define design pattern behavioral signature by manually analyzing design pattern architectures and sample implementations.

Variant Handling: There are different methods to implement a design pattern. Therefore, instantiation of design patterns usually leads to generate pattern variants. The detection of these variants is a challenge in reverse engineering. In our method, we consider the cases of variants in the structural analysis. Because in the structural phase, we are going to find the main body (the structural signature) of the pattern which is usually fixed among different variations.

3. Definitions

To define our method, we need some preliminary definitions that will be used in the rest of the paper.

Semantic graph: In particular, a graph is a set of homogeneous nodes connected by homogeneous edges, but we can extend this idea to a collection of individual entities, each with their type and characteristics. A semantic graph is a data representation in which nodes present concepts, and edges present relationships between them[57]. Figure 3 depicts an example of the semantic graph.

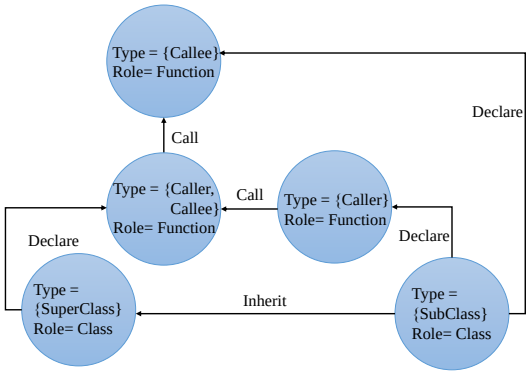


Figure 3: An example of semantic graph

Diameter: The diameter of a graph ($D(a, b)$) is the length of the shortest

path between the most distanced nodes. The distance from node a to node b, is the length of the shortest path from a to b.

Balls: A ball with center v and radius r that is denoted by $\hat{G}[v, r]$ is a subgraph of G, which for all nodes v', $D(v, v') \leq r$, and it has exactly the edges that appear in G over the same node set.

Subgraphs: A subgraph H of a graph G is a graph whose vertices and edges are subsets of G.

Maximum match: In a graph, there may be multiple matches for a specific pattern. Maximum match $MaxM$ in graph G for pattern P is a unique match such that for any match M in G for P, $M \subseteq MaxM$.

Subgraph isomorphism: Subgraph isomorphism is the problem of deciding whether a graph G has a subgraph $G' \subset G$ that is isomorphic to a graph P. Since subgraph isomorphism is NP-complete, it is impossible to solve it in large graphs. Therefore for fast pattern matching, in general graphs, two options have been proposed: (1) use an approximate algorithm or (2) apply matching to only a subset of the data. The first approach may yield non-optimal solutions while the second one is an optimal algorithm. In the second approach, to filter out unimportant parts of a data set, some preprocessing will be performed. This filtering step is known as candidate selection.

Semantic matching algorithm: In the graph theory, matching approaches usually identify matches based on the graph structure. However, structural matching is often inadequate for identifying similar semantic graphs because the meaning of semantic graphs depends rigorously on information stored in nodes and edges. Semantic matching approaches attempt to match graphs based on node and edge information, as well as the structure of the graph [57].

4. The proposed method

In this paper, we can define the problem as follows: given a certain system source code, identify all the motifs of design patterns, which are asked to be detected in the system. To solve this problem, we suggested a method that consists of five steps. Figure 4 is the concrete version of Figure 1 which elaborates it in details.

4.1. Preprocessing

Before the first phase of the detection process, system structure which consists of the classes and the relationship between them is extracted from the source code. Here, we use Visual Paradigm in Eclipse [58] to parse the source code. Afterward, the class diagram and its equivalent XML file, which represent the system structure, are produced for further analysis.

4.2. Constructing Graph

In this step, a set of directed semantic graphs will be constructed for the system and the specified patterns. In these graphs, nodes are the classes whereas edges are the relationship between them. Both edges and vertices of these graphs

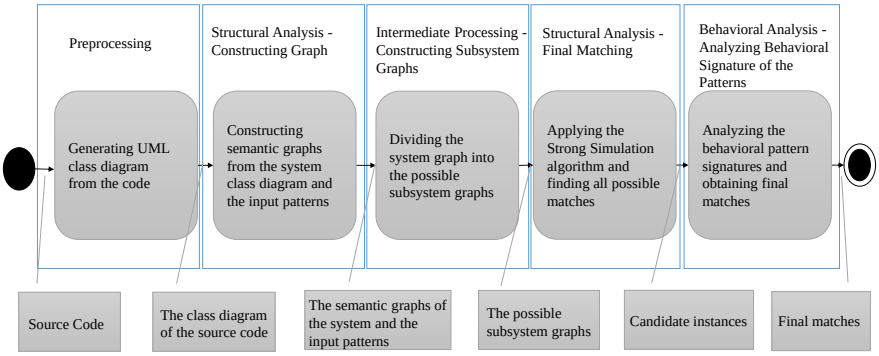


Figure 4: The concrete scheme of the proposed detection process

have a type. Here, edges are typed based on the kind of the relationship between their two connected classes, and nodes are typed based on their connected edges. We have taken three types of relationship for the edges: inheritance, aggregation, and association. According to our method, the pattern graphs are constructed first. Note that, in constructing pattern graphs, we focused only on the main body of the pattern, which is usually constant among different variations. In this way, we can cover variants of the pattern. In other words, main body of a pattern, as an abstract version of it, is the structural signature of the pattern which defined as key participants of its structure. For example, Figure 5 demonstrates main body of the Strategy design pattern introduced in [11]. Also, In Appendix B we present the main body of the creational, structural, and behavioral design patterns which are considered in the evaluation phase.

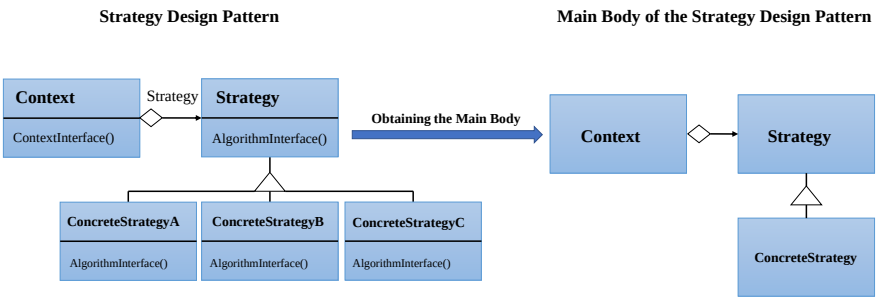


Figure 5: Main body of the Strategy design pattern

There are six modes for an edge, which we assign a specific weight (a prime number) to each of them (Table 1). For each node, the weights of its connected edges are multiplied together, and the resulting number is assigned to the corresponding node as its type. After constructing the pattern graphs, a set of node

Table 1: Weights assigned to the different modes of an edge

Type		Weight
Inheritance	Incoming edge	2
	Outcoming edge	3
Aggregation	Incoming edge	5
	Outcoming edge	7
Association	Incoming edge	11
	Outcoming edge	13

types (P) will be obtained.

In some instances of design patterns, roles may participate in an indirect relationship between other roles. Therefore, before assigning types to the system graph nodes, we should enrich the graph by adding new edges representing parents' relationships according to the system class diagram. We formally explain the enriching process as follows.

$\forall$ A,B,C $\in$ Classes:
if (A$\mapsto$ B)$\in$ inherit & (B$\mapsto$ C)$\in$ assoc$\Rightarrow$ (A$\mapsto$C)$\in$ assoc
if (A$\mapsto$ B)$\in$ inherit & (B$\mapsto$ C)$\in$ inherit$\Rightarrow$ (A$\mapsto$C)$\in$ inherit

In Figure 6, graph G is enriched by adding all edges which represent the indirect relationships between vertices. Graph G' is the result of the enriching process.

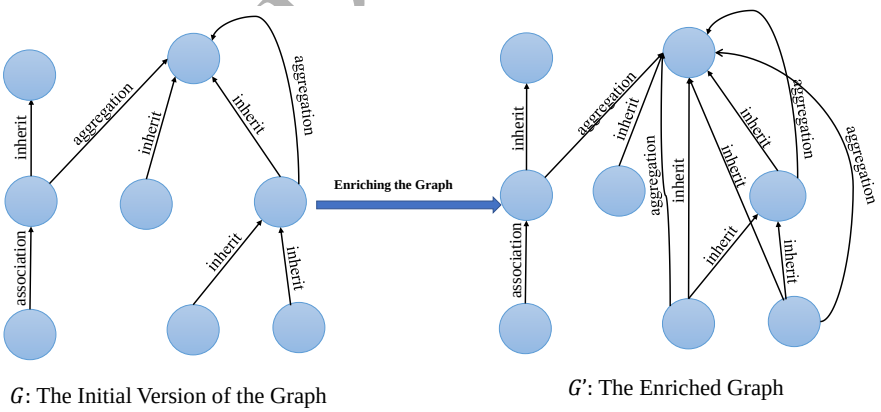


Figure 6: Enriching the graph

After enriching the system graph, according to Algorithm 1, a set of types will be assigned to each of the system nodes. For each node, if it satisfies the constraints of type p_i in the set P" ($0 < i <$ Number of types), we will add

p_i to its set of types. Constraints of a type is defined as the set obtained from the prime factorization of that type. In Example 1, we describe the process in details.
Assume n be the number of system nodes and m be the number of vertices in the pattern graph. In the worth case (when the number of members in the set P is equal to m) Algorithm 1 will iterate $n * m$ times. Therefore, the order of complexity of the algorithm is $O(nm)$. Note that in general, pattern graph is very smaller than the system graph. So, $m \ll n$.

```
Inputs: G(V, E) (Edge typed system graph), and P (Set of node types obtained from
pattern graphs)
Output: Set of node types t(v) for each node v in the system graph
foreach v in V do
    t(v) = Ø                                          ▷ t(v): Set of types for v
    foreach p in P do
        if v satisfies the constraints of the type p then
            Add p to t(v)
        end
    end
end
return t
```

Algorithm 1: Node type allocating algorithm for the system graph

Example 1. Cases (a) and (b) of Figure 7 represent the UML class diagrams of the design pattern and the system, respectively, whereas cases (c) and (e) represent their corresponding semantic graphs. As shown, after constructing the pattern graph (case (c)), the set P will have three elements $\{20, 3, 84\}$. The prime factorization of type 20 is $\{2, 2, 5\}$, according to the Table 1, it indicates nodes that have three incoming edges. Two of type inheritance and one of type aggregation. Prime factorization of type 3 designates nodes that have one outcoming edge of type inheritance. Finally, type 84 indicates nodes that have two outcoming edges, one of type aggregation and one of type inheritance, and also two incoming edges of type inheritance (Note: prime factorization of 84 is $\{2, 2, 3, 7\}$). After enriching the system graph (case (d)), by considering set P, system nodes are typed based on Algorithm 1.

4.3. Constructing subsystem graphs

To reduce the exploration space for each pattern, we partition the system graph into a set of possible subsystem graphs. For this purpose, all of the valid edges are characterized first. An edge is valid, if it satisfies the constraints of one of the edges in the pattern graph. The constraints of an edge is defined as its type and the types of its connected nodes.

Example 2. As seen in Figure 7 (c) there exists four types of valid edges: Edges of type “inherit” that connect a node of type “3” to a node of type “20”. Edges of type “inherit” that connect a node of type “3” to a node of type “84”. Edges of type “inherit” that connect a node of type “84” to a node of type “20”. Edges of type “aggregation” that connect a node of type “84” to a node of type

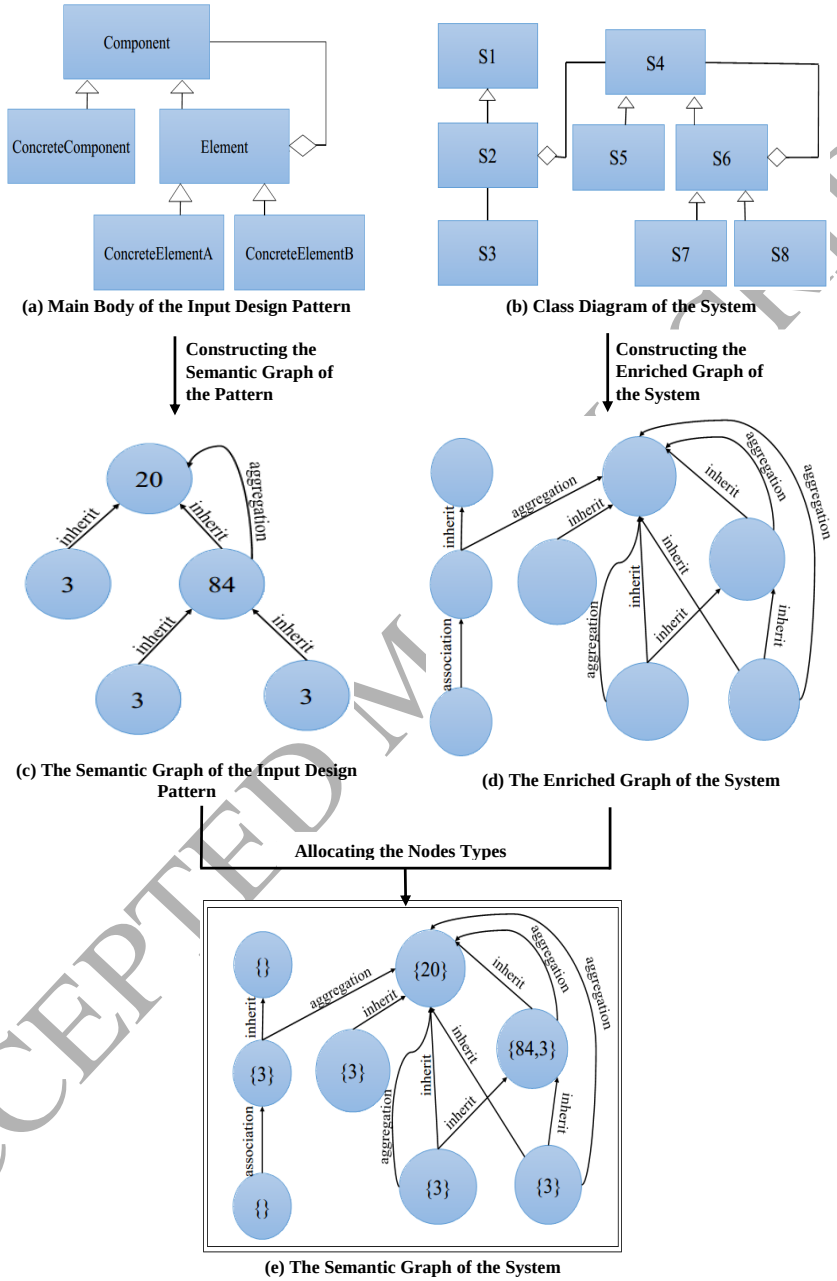


Figure 7: The process of constructing the system semantic graph

"20".

In the next step, in the system graph, valid edges will remain and any other edges will be removed. After that, we will have a set of subsystem graphs. Algorithm 2 illustrates the process of producing subsystems. Assume E_S be the number of edges in the system graph, and E_p be the number of edges in the pattern graph. In the worth case, number of valid edges will be equal to E_p. Therefore, complexity of Algorithm 2 is $O(E_S E_p)$. Note, since the system and the pattern graph are spars, $E_S \ll n^2$ and $E_p \ll m^2$.

```
Inputs: G'(V, E) (Edge and node typed system graph), and ValidEdges (Set of valid
edges)
Output: Set of subsystems
foreach e in E do
    IsValid=0
    foreach e_v in ValidEdges do
        if e satisfies constraints of e_v then
            IsValid =1
        end
    end
    if IsValid then
        add e to ValidSet
    end
end
remove any edges which is not in the ValidSet        ▷ the remained graph contains
subsystems
return remained graph
```

Algorithm 2: Subsystems production

4.4. Final Matching

In this step, we are going to find pattern instances in each of the subsystem graphs, known as the problem of graph pattern matching. Given a pattern graph Q and a data graph G, graph pattern matching is the problem of finding all matches in G such that, they are isomorphic to Q. It is often defined regarding subgraph isomorphism, an NP-complete problem [57]. Therefore, Strong Graph Simulation algorithm was introduced to reduce the complexity [10]. In this paper, using the semantic graphs and applying this concept to the design pattern detection field, we are able to use a revision of Strong Simulation algorithm and allow graph pattern matching to be conducted in polynomial time (cubic-time complexity) [10].

To the best of our knowledge, this study is the first work, in the area of design pattern detection, in which by applying an exact matching, optimal solutions in a polynomial time are obtained for the patterns with any number of the roles. There exists some other techniques with polynomial computational complexity, in the literature. However, they are appropriate only for patterns with a certain number of roles or they return inexact and approximate solutions, which cause generation of many false positives [13, 32, 21].

Algorithm 3 is the matching algorithm used in this paper. It is Strong Simulation algorithm that is modified to fit our problem. The Strong Simulation

algorithm is designed for node-labeled directed graphs. Therefore, we applied some changes in the original version of the algorithm to support directed graphs with typed edges and typed vertices.

As mentioned, the algorithm is applied to directed graphs with typed edges and typed vertices. This kind of graphs defined as $G(V, E, l, t)$ which V is a set of nodes, $E \subseteq V \times V$ is a set of edges and l and t are labeling functions map each node and each edge to a label, respectively.

In algorithm 3, pattern graph Q and a data graph G are introduced as inputs while a set of matched subgraphs, G_s, are the outputs of algorithm. The algorithm has two procedures: Procedure DualSim and Procedure ExtractMaxPG.

Procedure DualSim takes a pattern graph $Q[V_q, E_q]$ and a ball $\hat{G}[w, d_Q]$ with center w and radius d_Q, and finds the maximum match relation S_w in $\hat{G}[w, d_Q]$ for Q. Procedure ExtractMaxPG takes a pattern graph Q, a ball $\hat{G}[w, d_Q]$, and the maximum match relation S_w, and returns the maximum perfect subgraph G_s in the ball if there is one.

4.5. Analyzing behavioral signature of the patterns

Patterns can be characterized by their unique structural and behavioral signature. All of the candidate pattern instances are detected already, conform to the structural signature of the pattern which expressed in the form of its main body. However, they may contain many false positives. Because the conducted process takes into account the class level information only. False positives can be eliminated by validating the behavioral signature of the pattern [32]. In Appendix C, we introduce the behavioral signature for some GoF design patterns by using the notations introduced in [59]. As illustrated, they describe the method invocations, the methods access type, etc. After obtaining the candidate pattern motifs, the roles of the classes are determined. Then, only those instances in which their classes conform to the corresponding behavioral signature remain.

5. Experiments

To evaluate our method, we consider three open source projects: JHotDraw 5.1, JRefactory 2.6.24 and JUnit 3.7. These projects have been selected because they widely used as the benchmarks of design pattern detection, and we can compare our results with results of different tools on the same cases. Table 2 shows the characteristics of the software systems used in the experiment.

The pattern detection methodology effectiveness is usually measured by counting the number of accurately identified patterns (True Positives, TP), number of detected pattern instances which do not satisfy the predefined criterion (False Positives, FP) and number of pattern instances that are not being detected by the applied methodology (False Negatives, FN). Here, the results are also evaluated by using the metrics of precision (P) and recall (R), which typically used in pattern recognition and information retrieval [60]. These two

Inputs: The data graph $G(V, E)$, and the pattern graph Q with diameter d_Q.
Output: The set M of matched subgraphs of G for Q.
M:= $\emptyset$;
foreach *ball* $\hat{G}[w, d_Q]$ *in* G **do**
  $S_w := DualSim(Q, \hat{G}[w, d_Q])$;
  $G_s := ExtractMatchedSubGraph(Q, \hat{G}[w, d_Q], S_w)$;
  if $G_s \neq nil$ **then**
    $M := M \cup \{G_s\}$;
  end
end
return M
Procedure $DualSim(Q, \hat{G}[w, d_Q])$
Inputs: The pattern graph $Q(V_q, E_q)$, and the ball $\hat{G}[w, d_Q]$.
Output: The maximum match relation S_w in $\hat{G}[w, d_Q]$ for Q.
foreach $u \in V_q$ *in* Q **do**
  $sim(u) := \{v \mid v$ is in $\hat{G}[w, d_Q]$ and $l_Q(u) \in l_G(v)\}$;
  while *there are changes* **do**
    foreach *edge* $e : (u, u')$ *of type* t *in* E_q *and each node* $v \in sim(u)$ **do**
      if *there is no edge* $g : (v, v')$ *of type* t *in* $\hat{G}[w, d_Q]$ *with* $v' \in sim(u')$ **then**
        $sim(u) := sim(u) \backslash v$; ▷ node v is removed from $sim(u)$
      end
    end
    foreach *edge* $e : (u', u)$ *of type* t *in* E_q *and each node* $v \in sim(u)$ **do**
      if *there is no edge* $g : (v', v)$ *of type* t *in* $\hat{G}[w, d_Q]$ *with* $v' \in sim(u')$ **then**
        $sim(u) := sim(u) \backslash v$; ▷ node v is removed from $sim(u)$
      end
    end
    if $sim(u) = \emptyset$ **then**
      return $\emptyset$
    end
  end
  $S_w := \{(u; v) | u \in Vq; v \in sim(u)\}$;
end
return S_w
Procedure $ExtractMatchedSubGraph(Q, G[w, d_Q], S_W)$
Inputs: The pattern Q, and the ball $\hat{G}[w, d_Q]$ with maximum match relation S_w.
Output: The matched subgraph G_s in $\hat{G}[w, d_Q]$ for Q if any.
if w *does not appear in* S_w **then**
  return *nill*
end
Construct the matching graph G_m w.r.t. S_w;
return the connected component G_s containing w in G_m.

Algorithm 3: Matching algorithm

Table 2: Characteristics of the software systems used in the experiment

Characteristics / Project	Version	#Classes	KLOC
JHotDraw	5.1	155	8.300
JUnit	3.7	43	9.7
JRefactory	2.6.24	562	216.2

metrics are defined as follows:

$$Precision = \frac{\mid TrueInstancesRecovered(TP) \mid}{\mid AllInstancesRecovered(T) \mid}\% \quad (1)$$

$$Recall = \frac{\mid TrueInstancesRecovered(TP) \mid}{\mid AllTrueInstances(TP+FN) \mid}\% \quad (2)$$

Over the last two decades, many design pattern detection tools and techniques are provided for mining the patterns instances from the source code. In this paper, we compared our results with SSA [13], DeMIMA [40], Sempatrec [25], and n_rP[24]. The motivations behind the selection of these tools are: (1) they are one of the most successful design pattern recovery tools, (2) they consider more patterns, in the evaluation phase, compared to the rest of the detection tools and techniques, and (3) they apply their experiments on the same cases as presented in this work. The results of the proposed detection process are summarized in Table 3.

Since the numbers of detected pattern motifs were rather small (not more than several dozens), we had a group of master students to check manually if the detected instances are true positives. Also, to identify all true instances of the patterns in the systems under study, we considered as a gold standard, the union of the true positives produced by our approach, with the common instances extracted by different tools [61], after manual verification by our master student team. Some cases in which, we could not achieve agreement based on the claims, we assigned them unknown values and we represented these cases by "-". Also, we used "na" to show division by zero. All the results are published at `http://sqlab.um.ac.ir/index.php?lang=en` for reference.

As Table 3 illustrates, in most cases, the proposed approach achieved precision and recall rates of 100%. The main reason is that, in our experiments, all of the false positives were eliminated during the behavioral analysis, so the true positives (TP) and the total number of the recovered instances (T) were equal and according to equation 1 precision was obtained 100%. Furthermore, in the structural phase, since we focused only on the main body of the patterns, all possible candidate motifs had been recovered, and we did not miss any true instances. Therefore, based on equation 2 the recall metric achieved to its highest value.

We also found that our method missed some instances, which other tools claimed to recover. Cases, in which our true positives were less than other approaches, encouraged us to check the results of the other methods manually. Using [62] and the available information about tools results, we found, there are some instances diagnosed as true positives which are controversial. For example, Iconkit in JHotDraw 5.1 and UndoStack in JRefactory 2.6.24 recognized as Singleton instances, but their constructors are public. Furthermore, for example, in Factory Method, we believe, instances which their ConcreteCreators are in an inheritance relationship and have the same CocreteProduct should be merged. Some tools did not cover this situation. In Figure 8, we note one

Table 3: Comparison of the results of our method and that of other methods

			Our Method				n_rp				Sempatrec				DeMIMA				SSA			
			T	TP	FN	P	R	T	TP	FN	P	R	T	TP	FN	P	R	T	TP	FN	P	R
Creational	Singleton	JU	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na
		JH	1	1	0	100%	100%	2	2	0	100%	100%	2	2	0	100%	100%	2	2	0	100%	100%
		JR	10	10	-	100%	-	12	12	-	100%	-	4	4	-	100%	-	14	2	-	14%	-	12	8	-	67%	-
	Factory Method	JU	1	1	0	100%	100%	0	0	0	na	na	0	0	0	na	na	23	0	0	0%	na	0	0	0	na	na
		JH	10	10	0	100%	100%	3	3	-	100%	-	3	3	0	100%	100%	189	3	0	2%	100%	2	2	1	100%	67%
		JR	16	16	-	100%	-	1	1	-	100%	-	2	1	-	50%	-	71	1	-	1%	-	1	1	-	100%	-
	Abstract Factory	JU	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	27	0	0	0%	na	na	na	na	na	na
		JH	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	na	na	na	na	na
		JR	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-	167	0	-	0%	-	na	na	na	na	na
Structural	Composite	JU	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%
		JH	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	3	1	0	33%	100%	1	1	0	100%	100%
		JR	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-	0	0	-	na	-
	Adapter	JU	4	4	0	100%	100%	6	6	0	100%	100%	1	1	0	100%	100%	9	0	1	0%	0%	6	1	0	17%	100%
		JH	18	18	-	100%	-	19	19	-	100%	-	18	8	-	45%	-	28	1	-	4%	-	23	10	-	44%	-
		JR	17	17	-	100%	-	16	16	-	100%	-	22	13	-	59%	-	47	17	-	36%	-	26	17	-	65%	-
	Decorator	JU	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%
		JH	3	3	0	100%	100%	3	3	0	100%	100%	2	1	2	50%	33%	13	1	2	8%	33%	3	1	2	33%	33%
		JR	0	0	-	100%	-	0	0	-	100%	-	0	0	-	na	-	1	0	-	0%	-	0	0	-	na	-
Behavioral	Template Method	JU	1	1	0	100%	100%	1	1	0	100%	100%	1	1	0	100%	100%	11	0	1	0%	100%	1	1	0	100%	100%
		JH	5	5	0	100%	100%	5	5	0	100%	100%	2	1	0	50%	100%	31	1	0	3%	100%	5	1	0	20%	100%
		JR	17	17	-	100%	-	17	17	-	100%	-	6	6	-	100%	-	49	0	-	0%	-	17	17	-	100%	na
	Observer	JU	3	3	0	100%	100%	3	3	0	100%	100%	1	1	0	100%	100%	4	1	0	25%	100%	1	1	0	100%	100%
		JH	2	2	-	100%	-	2	2	-	100%	-	4	2	3	50%	40%	7	2	3	29%	40%	3	1	4	33%	20%
		JR	3	3	-	100%	-	0	0	-	na	-	0	0	-	na	-	1	0	-	0%	-	0	0	-	na	-
	Visitor	JU	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na
		JH	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na	0	0	0	na	na
		JR	2	2	-	100%	-	2	2	-	100%	-	2	1	-	50%	-	4	2	-	50%	-	2	1	-	50%	-
	State/Strategy	JU	3	3	0	100%	100%	3	3	0	100%	100%	4	3	0	75%	100%	8	0	3	0%	0%	3	2	1	67%	67%
		JH	30	30	-	100%	-	20	20	-	100%	-	39	8	-	21%	-	21	6	-	29%	-	44	11	-	25%	-
		JR	31	31	-	100%	-	15	15	-	100%	-	7	0	-	0%	-	22	2	-	9%	-	11	0	-	0%	-

JU: JUnit, JH: JHotDraw, JR: JRefactory.

sample in JHotDraw 5.1. We have considered this case, as a single motif, while some other techniques have taken it three instances.

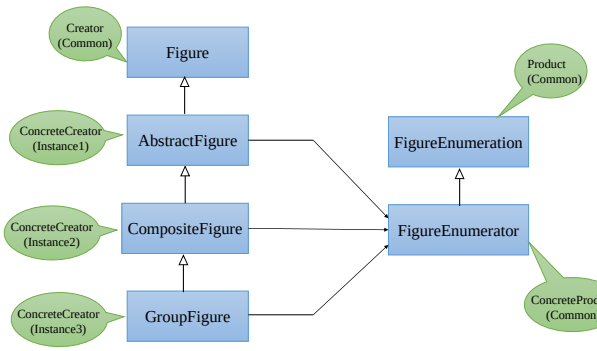


Figure 8: An example of merging Factory Method instances in JHotDraw 5.1

6. Threats to validity

In this section, we discuss possible treats into construct validity, internal validity, and external validity. Threats to construct validity refer to the relationship between theory and measures. There could be inaccuracy and omissions in the measurements for several reasons. We make use of two commonly used evaluation measures: precision and recall. The detected instances were manually checked to see if they are true positives. However, it could affect the number of false negatives and the number of false positives because our results are exposed to human mistakes and bias or interpretation. Therefore, it can pose a threat to the validity of the empirical evaluation. This threat, in future work, can be alleviated using publicly available gold standards (in a gold standard all actual occurrences of a specific design pattern in a particular software system are identified [60]). Threats to internal validity concern conditions that influence the confidence level of the results. In this context, this is mainly due to the variants of different patterns. In our study, the pattern instances are detected based on the main body of the pattern and its behavioral signature. However, for one pattern, finding an appropriate main body which is constant among all of the pattern's variations is challenging. Another threat to internal validity is due to the preprocessing step in the proposed method. In this step, using tool, the system source code is converted into its corresponding UML class diagram. In the cases of the implicit, or complex, or semantical relationships between UML constructs, there is no guarantee that the tool result will lead to generate a complete and perfect UML class diagram. Threats to external validity relate to the ability of generalizing the findings to a larger population outside the experiment setting. Here, we conducted the evaluation, on three Java projects. We cannot claim that our method generates the same results on larger systems

or other projects having different language constructs. Therefore, replication of the evaluation on other systems to confirm the obtained results is desirable.

7. Conclusion & Future work

Software design patterns encapsulate common ways of solving problems in the large software systems. They are very important in the forward engineering when a software engineer decides to design a system from scratch, and also in the reverse engineering when understanding a software system is crucial. Detecting design patterns from a system, assists understanding it and improving its quality. In this paper, we have presented a new method for the detection of design motifs. It is based on the semantic graphs and the pattern signature which characterize each design pattern. In this work, in the final phase, the behavioral signature of the patterns are checked. Therefore, this method is intended to discard the identified false positives in the structural phase, hence improving the precision of the detection process. In the future, we plan to discuss more about a standard way to describe both the structural and the behavioral signatures of the patterns. Also, investigating the problem of obtaining an exact UML class diagram from source code is noteworthy. Another interesting area would be extending the semantic analysis of design motifs. The presented study used semantic analysis partially in the structural phase, that can be investigated deeper on future works. In this way, we can differentiate between the patterns that have the similar structural and behavioral characteristics (e.g. Bridge and Strategy). Also, in future, the empirical study can be improved by considering more software systems (e.g. other versions of JHotDraw, MapperXML, Swing, and QuickUML).

References

[1] M. Riaz, T. Breaux, L. Williams, How have we evaluated software pattern application? a systematic mapping study of research design practices, Information and Software Technology 65 (2015) 14–38. doi:http://dx.doi.org/10.1016/j.infsof.2015.04.002.

[2] J. Dong, Y. Zhao, T. Peng, A review of design pattern mining techniques, International Journal of Software Engineering and Knowledge Engineering 19 (6) (2009) 823–855. doi:http://dx.doi.org/10.1142/S021819400900443X.

[3] A. Birukou, A survey of existing approaches for pattern search and selection, in: Proceedings of the 15th European Conference on Pattern Languages of Programs, EuroPLoP '10, ACM, New York, NY, USA, 2010, pp. 2:1–2:13. doi:10.1145/2328909.2328912.

[4] J. Nicholson, A. H. Eden, E. Gasparis, R. Kazman, Automated verification of design patterns: A case study, Science of Computer Programming

80, Part B (2014) 211–222. doi:http://dx.doi.org/10.1016/j.scico.2013.05.007.

[5] A. H. Eden, J. Gil, A. Yehudai, Automating the application of design patterns, Journal of Object Oriented Programming 10 (2) (1997) 44–46.

[6] Y.-G. Guéhéneuc, J.-Y. Guyomarc'h, H. Sahraoui, Improving design-pattern identification: a new approach and an exploratory study, Software Quality Journal 18 (1) (2009) 145–174. doi:10.1007/s11219-009-9082-y.

[7] G. Booch, J. Rumbaugh, I. Jacobson, Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series), Addison-Wesley Professional, 2005.

[8] B.-k. Lee, S.-h. Yang, D.-H. Kwon, D.-Y. Kim, PGNIDS(Pattern-Graph Based Network Intrusion Detection System) Design, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 38–47. doi:10.1007/11751595_5.

[9] S. Samanta, M. Pal, Fuzzy planar graphs, IEEE Transactions on Fuzzy Systems 23 (6) (2015) 1936–1942. doi:10.1109/TFUZZ.2014.2387875.

[10] S. Ma, Y. Cao, W. Fan, J. Huai, T. Wo, Strong simulation: Capturing topology in graph pattern matching, ACM Transactions on Database Systems 39 (1) (2014) 4:1–4:46. doi:10.1145/2528937.

[11] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[12] A. Ampatzoglou, S. Charalampidou, I. Stamelos, Research state of the art on gof design patterns: A mapping study, Journal of Systems and Software 86 (7) (2013) 1945–1964. doi:10.1016/j.jss.2013.03.063.

[13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, Design pattern detection using similarity scoring, IEEE Transactions on Software Engineering 32 (11) (2006) 896–909. doi:10.1109/TSE.2006.112.

[14] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis, in: Proceedings of the IEEE International Conference on Software Maintenance, 2010, pp. 1–6. doi:10.1109/ICSM.2010.5609707.

[15] F. A. Turbak, D. K. Gifford, Design Concepts in Programming Languages, The MIT Press, 2008.

[16] G. Antoniol, R. Fiutem, L. Cristoforetti, Design pattern recovery in object-oriented software, in: Proceedings of the 6th International Workshop on Program Comprehension, 1998, pp. 153–160. doi:10.1109/WPC.1998.693342.

[17] Y.-G. Guéhéneuc, H. Sahraoui, F. Zaidi, Fingerprinting design patterns, in: Proceedings of the 11th Working Conference on Reverse Engineering, 2004, pp. 172–181. doi:10.1109/WCRE.2004.21.

[18] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A. I. Verkamo, Software metrics by architectural pattern mining, in: Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), 2000, pp. 325–332.

[19] M. Von Detten, S. Becker, Combining clustering and pattern detection for the reengineering of component-based software systems, in: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS, 2011, pp. 23–32. doi:10.1145/2000259.2000265.

[20] I. Issaoui, N. Bouassida, H. Ben-Abdallah, Using metric-based filtering to improve design pattern detection approaches, Innovations in Systems and Software Engineering 11 (1) (2015) 39–53. doi:10.1007/s11334-014-0241-3.

[21] A. Chihada, S. Jalili, S. M. H. Hasheminejad, M. H. Zangooei, Source code and design conformance, design pattern detection from source code by classification approach, Applied Soft Computing 26 (2015) 357 – 367. doi:http://dx.doi.org/10.1016/j.asoc.2014.10.027.

[22] G. Rasool, I. Philippow, P. Mäder, Design pattern recovery based on annotations, Advances in Engineering Software 41 (4) (2010) 519–526. doi:10.1016/j.advengsoft.2009.10.014.

[23] M. Vokác, An efficient tool for recovering design patterns from c++ code, Journal of Object Technology 5 (1) (2006) 139–157. doi:10.5381/jot.2006.5.1.a6.

[24] G. Rasool, P. Mäder, Flexible design pattern detection based on feature types, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, 2011, pp. 243–252. doi:10.1109/ASE.2011.6100060.

[25] A. Alnusair, T. Zhao, G. Yan, Rule-based detection of design patterns in program code, International Journal on Software Tools for Technology Transfer 16 (3) (2014) 315–334. doi:10.1007/s10009-013-0292-z.

[26] M. Thongrak, W. Vatanawood, Detection of design pattern in class diagram using ontology, in: Proceedings of the International Computer Science and Engineering Conference, 2014, pp. 97–102. doi:10.1109/ICSEC.2014.6978176.

[27] W. Ren, W. Zhao, An observer design-pattern detection technique, in: Proceedings of the IEEE International Conference on Computer Science and Automation Engineering, Vol. 3, 2012, pp. 544–547. doi:10.1109/CSAE.2012.6273011.

[28] J. Dong, Y. Sun, Y. Zhao, Design pattern detection by template matching, in: Proceedings of the ACM Symposium on Applied Computing, SAC '08, 2008, pp. 765–769. doi:10.1145/1363686.1363864.

[29] D. Yu, J. Ge, W. Wu, Detection of design pattern instances based on graph isomorphism, in: Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS), 2013, pp. 874–877. doi:10.1109/ICSESS.2013.6615444.

[30] J. Dong, Y. Zhao, Y. Sun, A matrix-based approach to recovering design patterns, IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans 39 (6) (2009) 1271–1282. doi:10.1109/TSMCA.2009.2028012.

[31] J. Dong, D. Lad, Y. Zhao, Dp-miner: Design pattern discovery using matrix, in: Proceedings of the 14th Annual IEEE International Conference and Workshops on Engineering of Computer-Based Systems, 2007, pp. 371–380. doi:10.1109/ECBS.2007.33.

[32] D. Yu, Y. Zhang, Z. Chen, A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures, Journal of Systems and Software 103 (2015) 1–16. doi:http://dx.doi.org/10.1016/j.jss.2015.01.019.

[33] M. L. Bernardi, M. Cimitile, G. A. D. Lucca, Design pattern detection using a dsl-driven graph matching approach, Journal of Software: Evolution and Process 26 (12) (2014) 1233–1266. doi:10.1002/smr.1674.

[34] C. Kramer, L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in: Proceedings of the Third Working Conference on Reverse Engineering, 1996, pp. 208–215. doi:10.1109/WCRE.1996.558905.

[35] R. Wuyts, Declarative reasoning about the structure of object-oriented systems, in: Proceedings of the Conference on Technology of Object-Oriented Languages, 1998, pp. 112–124. doi:10.1109/TOOLS.1998.711007.

[36] S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, M. Saeki, Design pattern detection by using meta patterns, IEICE Transactions on Information and Systems E91-D (4) (2008) 933–944. doi:10.1093/ietisy/e91-d.4.933.

[37] J. Niere, W. Schafer, J. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: Proceedings of the 24rd International Conference on Software Engineering, 2002, pp. 338–348. doi:10.1145/581380.581382.

[38] Y.-G. Guéhéneuc, N. Jussien, Using explanations for design-patterns identification, in: Proceedings of the 1 st IJCAI workshop on Modeling and Solving Problems with Constraints, 2001, pp. 57–64. doi:10.1.1.150.4976.

[39] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien, Instantiating and detecting design patterns: putting bits and pieces together, in: Proceedings of the 16th Annual International Conference on Automated Software Engineering, 2001, pp. 166–173. doi:10.1109/ASE.2001.989802.

[40] Y.-G. Guéhéneuc, G. Antoniol, Demima: A multilayered approach for design pattern identification, IEEE Transactions on Software Engineering 34 (5) (2008) 667–684. doi:10.1109/TSE.2008.48.

[41] H. Zhu, I. Bayley, L. Shan, R. Amphlett, Tool support for design pattern recognition at model level, in: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, Vol. 1, 2009, pp. 228–233. doi:10.1109/COMPSAC.2009.37.

[42] A. Wierda, E. Dortmans, L. Somers, Pattern detection in object-oriented source code, in: Software and Data Technologies, Springer, 2009, pp. 141–158. doi:10.1007/978-3-540-88655-6_11.

[43] K. Mens, T. Tourwé, Delving source code with formal concept analysis, Computer Languages, Systems & Structures 31 (3-4) (2005) 183–197. doi:10.1016/j.cl.2004.11.004.

[44] P. Tonella, G. Antoniol, Object oriented design pattern inference, in: Proceedings of the IEEE International Conference on Software Maintenance, 1999, pp. 230–238. doi:10.1109/ICSM.1999.792619.

[45] D. Beyer, C. Lewerentz, Crocopat: efficient pattern analysis in object-oriented programs, in: Proceedings of the 11th IEEE International Workshop on Program Comprehension, 2003, pp. 294–295. doi:10.1109/WPC.2003.1199220.

[46] J. Smith, D. Stotts, Spqr: flexible automated design pattern extraction from source code, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 215–224. doi:10.1109/ASE.2003.1240309.

[47] A. Blewitt, A. Bundy, I. Stark, Automatic verification of design patterns in java, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, 2005, pp. 224–232. doi:10.1145/1101908.1101943.

[48] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Design pattern recovery by visual language parsing, in: Proceedings of the 9th European Conference on Software Maintenance and Reengineering, 2005, pp. 102–111. doi:10.1109/CSMR.2005.23.

[49] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Behavioral pattern identification through visual language parsing and code instrumentation, in: Proceedings of the 13th European Conference on Software Maintenance and Reengineering, 2009, pp. 99–108. doi:10.1109/CSMR.2009.29.

[50] A. D. Lucia, V. Deufemia, C. Gravino, M. Risi, Design pattern recovery through visual language parsing and source code analysis, Journal of Systems and Software 82 (7) (2009) 1177 – 1193. doi:http://dx.doi.org/10.1016/j.jss.2009.02.012.

[51] I. Philippow, D. Streitferdt, M. Riebisch, S. Naumann, An approach for reverse engineering of design patterns, Software & Systems Modeling 4 (1) (2004) 55–70. doi:10.1007/s10270-004-0059-9.

[52] M. L. Bernardi, M. Cimitile, G. D. Ruvo, G. A. D. Lucca, A. Santone, Improving design patterns finder precision using a model checking approach, in: Proceedings of the CAiSE 2015 Forum at the 27th International Conference on Advanced Information Systems Engineering co-located with 27th International Conference on Advanced Information Systems Engineering, 2015, pp. 113–120.

[53] M. L. Bernardi, M. Cimitile, G. D. Ruvo, G. A. D. Lucca, A. Santone, Model checking to improve precision of design pattern instances identification in OO systems, in: Proceedings of the 10th International Conference on Software Paradigm Trends, 2015, pp. 53–63.

[54] S. Romano, G. Scanniello, M. Risi, C. Gravino, Clustering and lexical information support for the recovery of design pattern in source code, in: Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 500–503. doi:10.1109/ICSM.2011.6080818.

[55] G. Rasool, D. Streitferdt, A survey on design pattern recovery techniques, International Journal of Computer Science Issues 8 (2) (2011) 251–260. doi:10.1.1.403.3851.

[56] A. D. Lucia, V. Deufemia, C. Gravino, M. Risi, Towards automating dynamic analysis for behavioral design pattern detection, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 161–170. doi:10.1109/ICSM.2015.7332462.

[57] B. Gallagher, Matching structure and semantics: A survey on graph-based pattern matching, in: Proceedings of the Conference of the American Association for Artificial Intelligence, AAAI'06, 2006, pp. 45–53. doi:10.1.1.465.7805.

[58] Visual paradigm, http://www.visual-paradigm.com, accessed: 2016-01-15.

[59] H. Zhu, I. Bayley, An algebra of design patterns, ACM Transactions on Software Engineering and Methodology 22 (3) (2013) 23:1–23:35. doi:10.1145/2491509.2491517.

[60] N. Pettersson, W. Lowe, J. Nivre, Evaluation of accuracy in design pattern occurrence detection, IEEE Transactions on Software Engineering 36 (4) (2010) 575–590. doi:10.1109/TSE.2009.92.

[61] G. Rasool, I. Ahmad, M. Atif, A comparative study on results of design patterns recovery tools, World Applied Sciences Journal 28 (9) (2013) 1316–1321. doi:10.5829/idosi.wasj.2013.28.09.1284.

[62] F. A. Fontana, A. Caracciolo, M. Zanoni, Dpb: A benchmark for design pattern detection tools, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR), 2012, pp. 235–244. doi:10.1109/CSMR.2012.32.

Appendix A. Classification of the design pattern detection approaches

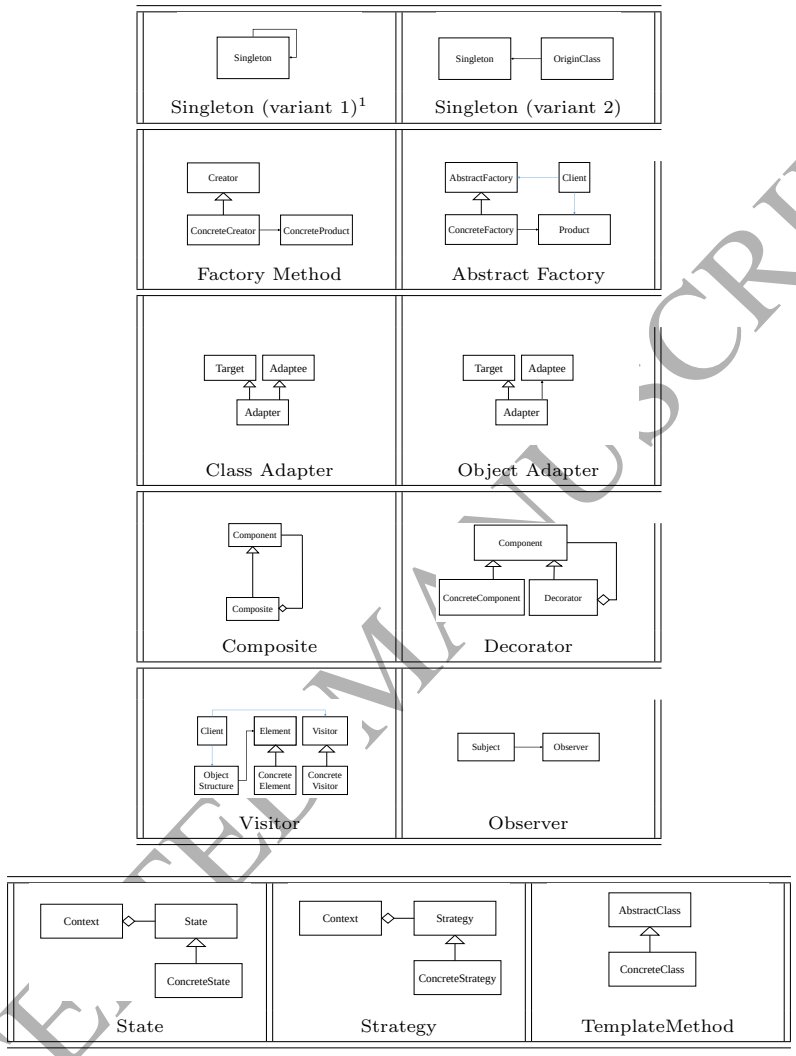
Table A

Category	Ref	Structural		Behavioral				Limitation(s)
				Static		Dynamic		
		Syntactic	Semantic	Syntactic	Semantic	Syntactic	Semantic	
Quantitative approaches	[16]	✓	×	×	×	×	×	The detection process focus only on five of the GoF structural design patterns.
	[17]	✓	×	×	×	×	×	Low recall.
	[18]	✓	×	×	×	×	×	The detection process is semi-automatic and user should verify candidate instances manually.
	[19]	✓	×	×	×	×	×	The validation of the detection process is applied on a small system only.
	[20]	✓	✓	×	×	×	×	The semantic metric which is used for the detection process, exploits the semantics encoded in the names of the design patterns elements, but it is not guaranteed that the names of the elements in the design motifs is chosen in a meaningful way.
	[21]	✓	×	×	×	×	×	This study is applicable only for those patterns with four roles.
Query-based approaches	[22]	✓	×	×	✓	×	×	The detection process is semi-automatic and user should put annotations on the source code manually.
	[23]	✓	×	×	×	×	×	This approach is language-dependent (C++) and can recover five design patterns only.
	[24]	✓	×	×	✓	×	×	This study is applicable only for GoF design patterns.
	[25]	×	✓	×	✓	×	×	This study is applicable only for GoF design patterns. The detection mechanism cannot discriminate some patterns, for example, the Observer design pattern from the Bridge pattern.
	[26]	×	✓	×	×	×	×	The validation of the detection process is applied on some small test cases only.
	[27]	✓	✓	×	×	✓	×	The detection process is not validated and the approach is discussed only for the Observer design pattern on a Java system.
Similarity scoring approaches	[13]	✓	×	×	×	×	×	The detection mechanism cannot discriminate between behavioral patterns.
	[28]	✓	×	×	×	×	×	The detection mechanism cannot discriminate between behavioral patterns.
	[30]	✓	×	✓	✓	×	×	The study focused on recovering only four design patterns in the experiments.
	[31]	✓	×	✓	✓	×	×	The approach can detect four patterns only.
	[32]	✓	×	✓	×	×	×	The approach only deals with 23 GoF patterns and cannot detect the patterns variants.
	[33]	✓	×	✓	×	×	×	Low precision.

Table A (Cont.)

Category	Ref	Structural		Behavioral				Limitation(s)
				Static		Dynamic		
		Syntactic	Semantic	Syntactic	Semantic	Syntactic	Semantic	
Reasoning-based approaches	[34]	✓	×	×	×	×	×	This approach is language-dependent (C++). The detection process is semi-automatic. It is difficult to find behavioral patterns.
	[35]	✓	×	×	×	×	×	This approach is language-dependent (Smalltalk) and the detection process is not validated.
	[36]	✓	×	✓	×	✓	×	This study reports low precision and the case study which used is pre-limited.
	[37]	✓	×	✓	×	×	×	The detection process is semi-automatic and is applicable only for GoF design patterns.
Constraint satisfaction approaches	[38]	✓	×	✓	×	×	×	This approach is language-dependent (Java).
	[39]	✓	×	✓	×	×	×	This approach is language-dependent (Java).
	[40]	✓	×	×	×	×	×	This method cannot distinguish between patterns which their structures are identical.
Formal approaches	[41]	✓	✓	×	×	✓	×	The detection process can not recognize pattern variants.
	[42]	✓	×	×	×	×	×	The detection mechanism takes a lot of computing time and cannot discriminate between behavioral patterns.
	[43]	✓	×	✓	×	×	×	Low precision and recall. The approach is not appropriate for large systems, because the time of computation theoretically increases in a non-linear way with the system size.
	[44]	✓	×	✓	×	×	×	This approach is limited to structural patterns.
	[45]	✓	×	×	×	×	×	This approach is limited to structural patterns.
	[46]	✓	×	✓	×	×	×	The detection process is not validated on large systems.
	[47]	×	✓	×	✓	×	×	This approach is language-dependent (Java).
Parsing-based approaches	[48]	✓	×	×	×	×	×	This approach is limited to structural patterns.
	[49]	✓	×	×	×	✓	×	The effectiveness of the approach depends on the selection of test cases that cover the behavior of the pattern candidates.
	[14]	✓	×	×	×	✓	×	The effectiveness of the approach depends on the selection of test cases that cover the behavior of the pattern candidates
	[50]	✓	×	✓	×	×	×	This approach is limited to structural patterns.
Miscellaneous approaches	[51]	✓	×	✓	×	×	×	The detection process is applicable only for GoF design patterns.
	[6]	✓	×	✓	✓	×	×	The detection process is applicable only for GoF design patterns.
	[52]	✓	×	✓	×	×	×	The detection process is not validated on large systems.
	[53]	✓	×	✓	×	×	×	The detection process is not validated on large systems.
	[54]	✓	×	×	×	×	×	Low recall.

Appendix B. The main body of some of the GoF design patterns



[1] Note that in the main body of the singleton design pattern we have an association link which connects OriginClass to the Singleton. Sometimes these two classes merge and make one Singleton class. Therefore, we consider two cases for this pattern.

Appendix C. The signature of some of the GoF design patterns

The Abstract Factory Signature

	Abstract Factory
Participants	AbstractFactory, ConcreteFactory, Product, Client ∈ Classes
Methods	creator ∈ opers(AbstractFactory)
Constraints	(1) creator is overridden ¬ isLeaf(creator) (2) Class ConcreteFactory creates an instance of Product which is returned by an operation overriding creator (ConcreteFactory ↦ Product)∈ creates ∧ returns(red(creator, ConcreteFactory), Product) (3) Client has an attribute of class AbstractFactory ∃ p∈ attrs(Client). type(p)=AbstractFactory

The Composite Signature

	Composite
Participants	Component, Composite ∈ Classes
Methods	Operation ∈ opers(Component)
Constraints	(1) The association is from Composite to Component with multiplicity * type(Parent)=Composite ∧ type(Children)=Component ∧ multiplicity(Children)=* (2) Operation is overridden in the Composite class and called by it ¬ isLeaf(Operation) ∧ (red(Operation, Composite)↦ Operation)∈ calls

The Decorator Signature

	Decorator
Participants	Component, ConcreteComponent, Decorator∈ Classes
Methods	Operation ∈ opers(Component)
Constraints	(1) Operation is overridden in Decorator by an operation that call it ¬ isLeaf(Operation) ∧ (red(Operation, Decorator)↦ Operation ∈ calls) (2) Operation is overridden in ConcreteComponent ¬ isLeaf(Operation) ∧ red(Operation,ConcreteComponent)

The Factory Method Signature

	Factory Method
Participants	Creator, ConcreteProduct, ConcreteCreator ∈ Classes
Methods	factoryMetod ∈ opers(Creator)
Constraints	(1) factoryMethod is overridden ¬ isLeaf(factoryMethod) (2) Class ConcreteCreator creates an instance of ConcreteProduct which is returned by an operation overriding factoryMethod (ConcreteCreator ↦ ConcreteProduct)∈ creates ∧ returns(red(factoryMethod, Concrete-Creator), ConcreteProduct)

The Template Method Signature

	Template Method
Participants	AbstractClass, ConcreteClass ∈ Classes
Methods	TemplateMethod, PrimitiveOperation ∈ opers(AbstractClass)
Constraints	(1) Operation PrimitiveOperation is not a leaf operation and is overridden in ConcreteClass ¬ isLeaf(PrimitiveOperation) ∧ red(PrimitiveOperation, ConcreteClass) (2) TemplateMethod calls PrimitiveOperation (TemplateMethod ↦ PrimitiveOperation)∈ calls

The Observer Signature

	Observer
Participants	Subject, Observer ∈ Classes
Methods	Update ∈ opers(Observer), Notify ∈ opers(Subject)
Constraints	(1) The association from Subject to Observer has multiplicity * multiplicity(Observer)=* (2) Operation Notify in class Subject calls operation Update in class Observer (Notify ↦ Update)∈ calls

The Visitor Signature

	Visitor
Participants	Visitor, Element, ConcreteElement, ConcreteVisitor, ObjectStructure∈ Classes
Methods	Accept ∈ opers(Element), Visit∈ opers(Visitor)
Constraints	(1) The association from ObjectStructure to Element has multiplicity * . multiplicity(Element)=* (2) Client depends only on Visitor and ObjectStructure access(Visitor, ObjectStructure, Element ∪ ConcreteElement ∪ ConcreteVisitor) (3) Accept has a parameter of class Visitor ∃ p ∈ params(Accept).type(p)=Visitor (4) Visit is called by the operation overridding Accept (red(Accept, ConcreteElement)↦ Visit)∈ calls

The Strategy Signature

	Strategy
Participants	Context, Strategy, ConcreteStrategy∈ Classes
Methods	ContextInterface ∈ opers(Context), AlgorithmInterface ∈ opers(Strategy)
Constraints	(1) Operation ContextInterface calls operation AlgorithmInterface (ContextInterface ↦ AlgorithmInterface)∈ calls (2) AlgorithmInterface is overridden in ConcreteStrategy ¬ isLeaf(AlgorithmInterface) ∧ red(AlgorithmInterface, ConcreteStrategy)

The State Signature

	State
Participants	Context, State, ConcreteState∈ Classes
Methods	Request ∈ opers(Context), Handle ∈ opers(State)
Constraints	(1) Operation Request calls operation Handle (Request ↦ Handle)∈ calls (2) Handle is overridden in ConcreteState ¬ isLeaf(Handle) ∧ red(Handle, ConcreteState)

The Singleton Signature

	Singleton (Variant 1)
Participants	Singleton, OrigiClass ∈ Classes
Methods	ConstructioMethod ∈ opers(Singleton), instance ∈ opers(OriginClass)
Constraints	(1) The construction method of class Singleton is not public type(ConstructionMethod)=Singleton ∧ ¬ isPublic(ConstructionMethod) (2) Operation instance of OriginClass is not private and returns an instance of Singleton returns(instance, Singleton) ∧ ¬ isPrivate(instance) (3) There is a conditional statement cc in the body of the instance operation (instance ↦ cc)∈ contains

	Singleton (Variant 2)
Participants	Singleton ∈ Classes
Methods	ConstructioMethod, instance ∈ opers(Singleton)
Constraints	(1) The construction method of class Singleton is not public type(ConstructionMethod)=Singleton ∧ ¬ isPublic(ConstructionMethod) (2) Operation instance of Singleton is static and public and returns an instance of Singleton returns(instance, Singleton) ∧ isStatic(instance) ∧ isPublic(instance)

The Adapter Signature

	Object Adapter
Participants	Target, Adapter, Adaptee ∈ Classes
Methods	Request ∈ opers(Target), SpecificRequest ∈ opers(Adaptee)
Constraints	(1) Request's redefinition in Adapter calls SpecificRequest (red(Request, Adapter)↦ SpecificRequest)∈ calls

	Class Adapter
Participants	Target, Adapter, Adaptee ∈ Classes
Methods	Request ∈ opers(Target), SpecificRequest ∈ opers(Adaptee)
Constraints	(1) Request's redefinition in Adapter calls SpecificRequest (red(Request, Adapter)↦ SpecificRequest)∈ calls