STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 1, 2008

A SEARCH BASED APPROACH FOR IDENTIFYING DESIGN PATTERNS

GABRIELA ŞERBAN AND ISTVÁN GERGELY CZIBULA

Abstract. *Software design patterns* are well-known and frequently reused micro-architectures: they provide proved solutions to design recurring problems with certain contexts. In restructuring legacy code it is useful to introduce a design pattern in order to add clarity to the system and thus facilitate further program evolution. That is why the problem of design patterns identification is very important. Automating the detection of design pattern instances could be of significant help to the process of reverse engineering large software systems. The aim of this paper is to introduce a new search based approach for identifying instances of design patterns in a software system. We provide an experimental evaluation of our approach, emphasizing its advantages.

Key words: Design patterns, Pattern recognition, Clustering.

1. Introduction

Design patterns have attracted significant attention in software engineering in the last period. An important reason behind this is that design patterns are potentially useful in both development of new, and comprehension of existing object-oriented design, especially for large legacy systems without sufficient documentation. The design patterns introduced by Gamma et al. [3] capture solutions that have developed and evolved over time. Each design pattern indicates a high level abstraction, encompasses expert design knowledge, and represents a solution to a common design problem. A pattern can be reused as a building block for better software construction and designer communication.

In restructuring legacy code is useful to introduce a design pattern in order to add clarity to the system and thus facilitate further program evolution. That is why the problem of design patterns identification is very important.

Received by the editors: December 4, 2007.

2000 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement –*Restructuring, reverse engineering, and reengineering*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

Automating the detection of design pattern instances could be of significant help to the process of reverse engineering large software systems.

From a program understanding and reverse engineering perspective, extracting information from a design or source code is very important, the complexity of this operation being essential. Localizing instances of design patterns in existing software can improve the maintainability of software. Automatic detection of design pattern instances is probably a useful aid for maintenance purposes, for quickly finding places where extensions and changes are most easily applied.

It would be useful to find instances of design patterns especially in designs where they were not used explicitly or where their use is not documented. This could improve the maintainability of software, because larger chunks could be understood as a whole.

The presence of patterns in a design should be reflected also in the corresponding code: being able to extract pattern information from both design and code is fundamental in identifying traceability links between different documents, explaining the rationale of the chosen solution in a given system and thus simplifying the activity of building its conceptual model.

The main contributions of this paper are:

- To introduce an original search based approach for identifying instances of design patterns in a software system.
- To emphasize the advantages of the proposed approach in comparison with existing approaches.

The rest of the paper is structured as follows. Section 2 presents some existing approaches in the field of automatic design patterns identification. The theoretical model of our search based approach for identifying design patterns is introduced in section 3. Section 4 presents our approach and Section 5 contains an experimental evaluation of it. An analysis of the proposed approach is made in Section 6. Section 7 presents some conclusions of the paper and future research directions.

2. Related Work

We will briefly present, in the following, the most significant results obtained in the literature in the field of automatic design patterns identification.

Different approaches, exploiting software metrics, were used in previous works to automatically detect design concepts and function clones [6, 7] in large software systems. An approach for extracting design information directly from C++ header files and for storing them in a repository is proposed in [7]. The patterns are expressed as PROLOG rules and the design information is translated into facts. A single Prolog query is then used to search for all

patterns. The disadvantage of this approach is that handles a small number of design patterns (only the structural design patterns – Adapter, Bridge, Composite, Decorator and Proxy) and the precision obtained in recognition is small (40%).

A multi-stage approach using OO software metrics and structural properties to extract structural design patterns from object oriented artifacts, design, or code, is introduced in [1]. The drawback of this approach is that only few pattern families (the structural design patterns - Adapter, Bridge, Composite, Decorator and Proxy) are considered.

In [8] the authors have developed an iterative semiautomatic approach to design recovery using static analysis on the source code level of a system. The approach facilitates a rule-based recognition of design pattern instances. It is a highly scalable process which can be applied to large real world applications with more than 100,000 LOC. The approach has been enhanced by fuzzyfied rules to provide the reverse engineer with accuracy information about the analysis results [9]. Fuzzyfied rules have a credibility value expressing how much design pattern candidates identified by the rule are real design pattern instances. The reverse engineer adapts the accuracy values of the results which are then used to calibrate credibility values of the rules [4].

For a precise design pattern recognition, a static analysis is not sufficient. The behavioral aspects of a pattern are an important factor. Dynamic analyses can be used to analyze the runtime behavior of a system. A sole dynamic analysis is not feasible since the amount of data gathered during runtime is too big. A combination of static and dynamic analysis techniques is proposed in [13]. The static analysis identifies candidates for design pattern instances. These candidates form a significantly reduced search space for a subsequent dynamic analysis that confirms or weakens the results from static analysis.

3. Theoretical model

Let $S = \{s_1, s_2, ..., s_n\}$ be a software system, where $s_i, 1 \leq i \leq n$ may be an application class, a class method or a class attribute. We will refer an element of the software system S as an *entity*.

Let us consider that:

- $Class(S) = \{C_1, C_2, \ldots, C_l\}$, $Class(S) \subset S$, is the set of applications classes in the initial structure of the software system S.
- Each application class C_i $(1 \leq i \leq l)$ is a set of methods and attributes, i.e., $C_i = \{m_{i1}, m_{i2}, \ldots, m_{ip_i}, a_{i1}, a_{i2}, \ldots, a_{ir_i}\}$, $1 \leq p_i \leq n$, $1 \leq r_i \leq n$, where m_{ij} $(\forall j,\ 1 \leq j \leq p_i)$ are methods and a_{ik} $(\forall k,\ 1 \leq k \leq r_i)$ are attributes from C_i.

- $Meth(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{p_i} m_{ij}$, $Meth(S) \subset S$, is the set of methods from all the application classes of the software system S.
- $Attr(S) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{r_i} a_{ij}$, $Attr(S) \subset S$, is the set of attributes from the application classes of the software system S.

Based on the above notations, the software system S is defined as follows:

$$S = Class(S) \bigcup Meth(S) \bigcup Attr(S). \tag{1}$$

A given *design pattern* p from the software system S can be viewed as a pair $p = (\mathcal{C}_p, \mathcal{R}_p)$, where

- $\mathcal{C}_p = \{C_1^p, C_2^p, \ldots, C_{nc_p}^p\}$ is a subset from $Class(S)$, $\mathcal{C}_p \subset Class(S)$, and represents the set of classes that are components of the design pattern p. nc_p represents the number of classes from the pattern p.
- $\mathcal{R}_p = \{r_1^p, r_2^p, \ldots, r_{nr_p}^p\}$ is a set of constraints (relations) existing among the classes from $\mathcal{C}_p$, constraints that characterize the design pattern p. Consequently, each constraint $r_i^p, \forall 1 \leq i \leq nr_p$ from $\mathcal{R}_p$ is a relation defined on a subset of classes from $\mathcal{C}_p$, and nr_p represents the number of constraints characterizing the design pattern p.

We mention that all the constraints from $\mathcal{R}_p$ can be expressed as binary constraints (there are two classes involved in the constraint). That is why, in the following, we will assume, without loosing generality, that all the constraints from $\mathcal{R}_p$ are binary.

Let us denote by *cmin* the minimum number of binary constraints from $\mathcal{R}_p$ that a class from $\mathcal{C}_p$ can satisfy, as indicated by equality (2).

$$cmin = min_{i=1,nc_p} |\{j,\ |1 \leq j \leq nr_p,\ \exists 1 \leq k \leq nc_p, k \neq i \text{ s.t. } C_i^p\ r_j^p\ C_k^p \vee C_k^p\ r_j^p\ C_i^p\}| \tag{2}$$

4. Our search-based approach

In this section we are focusing on identifying all the instances of a given design pattern p in a given design (software system).

Based on the theoretical model defined in Section 3, it can be easily seen that the problem of identifying all instances of the design pattern p in the software system S is a *constraint satisfaction problem* [10], i.e., the problem of searching for all possible combinations of nc_p classes from S such that all the constraints from $\mathcal{R}_p$ to be satisfied.

It is obvious that a brute force approach for solving this problem would lead to a worst case time complexity of $O(l^{nc_p})$. The main goal of the search based approach that we propose in this section in order to find all instances of a design pattern p is to reduce the time complexity of the process of solving the analyzed problem.

The main idea of our approach is to obtain a set of possible pattern candidates (by applying a preprocessing step on the set $Class(S)$) and then to apply a hierarchical clustering algorithm in order to obtain all instances of the design pattern p.

Our search-based approach for identifying instances of design patterns in a software system consists of the following steps:

- **Data collection**: The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them.
- **Preprocessing**: From the set of all classes from S we eliminate all the classes that can not be part of an instance of pattern p. This preprocessing step will be explained later.
- **Grouping**: The set of classes obtained after the **Preprocessing** step are grouped in clusters using a hierarchical clustering algorithm. The aim is to obtain clusters with the instances of p (each cluster containing an instance) and clusters containing classes that do not represent instances of p.
- **Design pattern instances recovery**: The clusters obtained at the previous step are filtered in order to obtain only the clusters that represent instances of the design pattern p.

In the following we will give a descriptions of the above enumerated steps.

4.1. **Data collection.** During this step, the existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them. In order to verify the constraints $\mathcal{R}_p$ of the design pattern p, we need to collect from the system information such as: all interfaces implemented by a class, the base class of each class, all methods invoked by a class, all possible concrete types for a formal parameter of a method, etc.

In order to express the dissimilarity degree between any two classes relating to the considered design pattern p, we will consider the distance $d(C_i, C_j)$ between two classes C_i and C_j from S given by the number of binary constraints from $\mathcal{R}_p$ that are not satisfied by classes C_i and C_j. It is obvious that as smaller the distance d between two classes is, as it is more likely that the two classes are in an instance of the design pattern p. The distance is expressed as in formula (3).

(3)
$$d(C_i, C_j) = \begin{cases} 1 + |\{k \,|1 \leq k \leq nr_p \; s.t. \; \neg(C_i \; r_k^p \; C_j \vee C_j \; r_k^p \; C_i)\}| & i \neq j \\ 0 & i = j \end{cases}.$$

Based on the definition of d given above it can be simply proved that d is a semimetric function.

4.2. **Preprocessing.** After the **Data collection** step was performed and the needed data was collected from the software system in order to compute the *distances* between the classes (3), a preprocessing step is performed in order to reduce the search space, i.e, the set of possible pattern candidates.

In order to significantly reduce the search space, we eliminate from the set of all classes $Class(S)$ those classes that certainly can not be part of an instance of the design pattern p. By applying this filtering, we will obtain a set of possible pattern candidates, denoted by $PatCand(S)$. More specifically, the following filtering step is performed:

- We eliminate from the set of all classes those classes that satisfy less than *cmin* binary constraints from $\mathcal{R}_p$. The idea is that based on the definition of *cmin* given by (2), in order to be in an instance of design pattern p, a class has to satisfy at least *cmin* constraints from $\mathcal{R}_p$. After the filtering step, the set of pattern candidates becomes:

$$PatCand(S) = Class(S) - \{C_j | 1 \leq j \leq l \; s.t. \sum_{i=1, i \neq j}^{l} (1 + nr_p - d(C_j, C_i)) < cmin\}$$

After applying the previous filtering step, the set $PatCand(S)$ of possible pattern candidates is significantly reduced in comparison with the set of all classes from S. Let us denote by nc the number of possible pattern candidates, i.e, the cardinality of the set $PatCand(S)$.

4.3. **Grouping.** After the grouping step we aim to obtain a partition $\mathcal{K} = \{K_1, K_2, ..., K_v\}$ of the set $PatCand(S)$ such that each instance of the design pattern p to form a cluster. Based only on the distance semimetric d (3), it is possible that two classes would seem to be in an instance of the design pattern (a so called "false positive" decision), even if they are not cohesive enough in order to take this decision. That is why we need a measure in order to decide how cohesive are two classes.

We will adapt the generic cohesion measure introduced in [12] that is connected with the theory of similarity and dissimilarity. In our view, this cohesion measure is the most appropriate to our goal. We will consider the dissimilarity degree (from the cohesion point of view) between any two classes

from the software system S. Consequently, we will consider the dissimilarity $diss(C_i, C_j)$ between classes C_i and C_j as expressed in (4).

$$diss(C_i, C_j) = \begin{cases} 1 - \frac{|p(C_i) \cap p(C_j)|}{|p(C_i) \cup p(C_j)|} & if \quad p(C_i) \cap p(C_j) \neq \emptyset \\ \infty & otherwise \end{cases}, \tag{4}$$

where $p(C)$ defines a set of relevant properties of class C and it consists of the application class itself, all attributes and methods defined in the class C, all interfaces implemented by C and the base class of C.

We have chosen the dissimilarity between two classes as expressed in (4) because it emphasizes the idea of cohesion. As illustrated in [2], "*Cohesion refers to the degree to which module components belong together*". The dissimilarity measure defined in Equation (4) highlights the concept of cohesion, i.e., classes with low dissimilarities are cohesive, whereas classes with higher distances are less cohesive.

Based on the definition of *diss* (4), it can be easily proved that *diss* is a semimetric function.

In the original paper [11] a theoretical validation of the *semimetric* dissimilarity function *diss* is given. It is proved that *diss* highlights the concept of cohesion, i.e., classes with low distances are cohesive, whereas classes with higher distances are less cohesive.

Consequently, the dissimilarity semimetric *diss* can be used in order to decide how cohesive are two classes. We will use *diss* in the **Grouping** step of our approach in order to decide if two classes are cohesive enough in order to be part of an instance of the design pattern.

In order to obtain the desired partition $\mathcal{K}$, we introduce a *hierarchical agglomerative clustering algorithm* (HAC).

In our approach the objects to be clustered are the classes from the set $PatCand(S)$ and the distance function between the objects is given by the semimetric d (3). We use *complete-link* [5] as linkage metric between the clusters, because it is the most appropriate linkage metric to our goal. Consequently, the distance $dist(k, k')$ between two clusters $k \in \mathcal{K}$ and $k' \in \mathcal{K}$ $(k \neq k')$ is given as in (5).

$$dist(k, k') = max_{e \in k, e' \in k'} d(e, e') \tag{5}$$

In the hierarchical clustering process, the dissimilarity semimetric *diss* will be used in order to decide how cohesive are two classes, i.e, if they will be merged or not in the same cluster. That is why we will consider the *dissimilarity* between two clusters $k \in \mathcal{K}$ and $k' \in \mathcal{K}$ (denoted by $dissimilarity(k, k')$) as the maximum dissimilarity between the objects from the clusters:

$$dissimilarity(k, k') = max_{e \in k, e' \in k'} diss(e, e') \tag{6}$$

The main steps of HAC algorithm are:

- Each class from $PatCand(S)$ is put in its own cluster (singleton).
- The following steps are repeated until the partition of classes remains unchanged (no more clusters can be selected for merging):
  - Select the two most similar clusters from the current partition, i.e, the pair of clusters that minimize the distance from (5). If this selection is nondeterministic (there are several pair of clusters with the same minimum distance between them), we will choose the pair (K_i, K_j) that has the minimum associated dissimilarity value ($dissimilarity(K_i, K_j)$). Let us denote by $dmin$ the distance between the most similar clusters K_i and K_j.
  - If $dmin < 1 + nr_p$ (nr_p is the number of constraints imposed by the design pattern p), then clusters K_i and K_j will be merged, otherwise the partition remains unchanged. The idea of this step is that two clusters will not be merged if their most distant classes can not be part of an instance of the design pattern p (they invalidate all the constraints that must hold).

We give next HAC algorithm.

`Algorithm HAC is`
Input: `-` `the set of possible pattern candidates` $PatCand(S)$`,`
`-` `the semimetric` d`,`
`-` `the semimetric` $diss$`,`
`-` `the number` nr_p `of imposed constraints.`
Precondition: `-` $l \geq 2$.
Output: `-` `the partition` $\mathcal{K} = \{K_1, K_2, ..., K_v\}$.
Begin
$v \leftarrow |PatCand(S)|$ `//the number of possible pattern candidates`
`For each` $C \in PatCand(S)$ `do`
$K_i \leftarrow \{C\}$ `//each possible candidate is put in its own cluster`
`endfor`
$\mathcal{K} \leftarrow \{K_1, \ldots, K_v\}$ `//the initial partition`
$change \leftarrow$ `true`
`While` $change$ `do //while` $\mathcal{K}$ `changes`
`//the most similar clusters are chosen for merging`
$dmin \leftarrow \infty$ `//the minimum distance between clusters`
$dissmin \leftarrow \infty$ `//the minimum dissimilarity between clusters`
`For` $i^* \leftarrow$ `1 to` v`-1 do //the most similar clusters are chosen`

```
      For j* ← i* + 1 to v do
        d ← dist(K_i*, K_j*) //the distance between the clusters
        If d < dmin then
          dmin ← d
          i ← i*
          j ← j*
        else
          If d = dmin then
            dss ← dissimilarity(K_i*, K_j*) //the dissimilarity between the clusters
            If dss < dissmin then
              dissmin ← dss
              i ← i*
              j ← j*
            endif
          endif
        endif
      endfor
    endfor
    If dmin < 1 + nr_p then
      K_new ← K_i ∪ K_j
      K ← (K \ {K_i, K_j}) ∪ {K_new}
      v ← v − 1
    else
      change ← false //the partition remains unchanged
    endif
  endwhile
  //K = {K_1, K_2, ..., K_v} is the output partition
End.
```

4.4. **Design pattern instances recovery.** The partition $\mathcal{K}$ obtained after the **Grouping** step will be filtered in order to obtain only the clusters that represent instances of the design pattern p. A cluster k from the partition $\mathcal{K}$ is consider an instance of design pattern p iff the classes from k verify all the constraints from $\mathcal{R}_p$ (the set of constraints imposed by the design pattern p).

5. Experimental Evaluation

In our experiment, we are focusing on identifying instances of *Proxy* design pattern using the search based approach that we have introduced in the previous section.

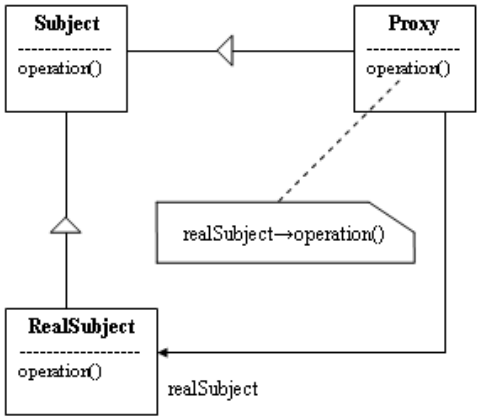

FIGURE 1. *Proxy* design pattern.

5.1. **The *Proxy* design pattern.** The class diagram of the *Proxy* [3] design pattern is given in Figure 1.

A Proxy pattern constitutes use of proxy objects during object interaction. A proxy object acts as a substitute for the actual object Provide a surrogate or placeholder for another object to control access to it

Proxy is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. Use of proxy objects is prevalent in remote object interaction protocols (*Remote proxy*): a local object needs to communicate with a remote process but we want to hide the details about the remote process location or the communication protocol. The *proxy* object allows to access remote services with the same interface of local processes. In fact, when an *Operation* is required to the proxy object, it delegates the implementation of the required operation to the *RealSubject* object. Being both *Proxy* and *RealSubject* subclasses of *Subject*, this guarantees that they export the same interface for *Operation*. To be able to call *RealSubject* methods, *Proxy* needs an association to it.

According to the considerations from Section 3, the design pattern *proxy* can be defined as the pair $proxy = (\mathcal{C}_{proxy}, \mathcal{R}_{proxy})$, where:

- $\mathcal{C}_{proxy} = \{C_1, C_2, C_3\}$, and $nc_{proxy} = 3$ (the number of classes involved in the design pattern *proxy* is 3).
- $\mathcal{R}_{proxy} = \{r_1, r_2, r_3\}$, $nr_{proxy} = 3$ (the number of constraints imposed by the design pattern *proxy* is 3), and the constraints are:

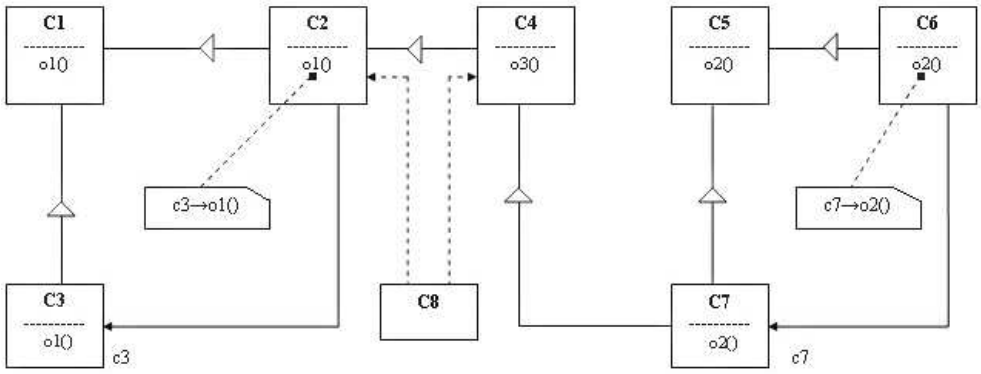

FIGURE 2. The example design S.

- $r_1(C_1, C_2)$ represents the relation "C_2 extends C_1".
- $r_1(C_1, C_3)$ represents the relation "C_3 extends C_1".
- $r_1(C_2, C_3)$ represents the relation "C_2 delegates any method inherited from a class C to C_3, where both C_2 and C_3 extend C".

Considering the above, the minimum number of binary constraints *cmin* from $\mathcal{R}_p$ that a class from $\mathcal{C}_p$ can satisfy (as indicated in (2)) is 2.

5.2. **Example.** Let us consider as a case study the simple design illustrated in Figure 2.

For the analyzed design S, the set of classes is $Class(S) = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$ and the number of classes is $l = 8$.

After performing the **Data collection** step from our approach, the matrix $D(8, 8)$ (where a line i corresponds to class C_i and a column j corresponds to class C_j) of distances between the classes from $Class(S)$ is:

$$D = \begin{pmatrix} 0 & 3 & 3 & 4 & 4 & 4 & 4 & 4 \\ 3 & 0 & 3 & 3 & 4 & 4 & 4 & 4 \\ 3 & 3 & 0 & 4 & 4 & 4 & 4 & 4 \\ 4 & 3 & 4 & 0 & 4 & 4 & 3 & 4 \\ 4 & 4 & 4 & 4 & 0 & 3 & 3 & 4 \\ 4 & 4 & 4 & 4 & 3 & 0 & 3 & 4 \\ 4 & 4 & 4 & 3 & 3 & 3 & 0 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 0 \end{pmatrix} \tag{7}$$

After the **Preprocessing** step, the set of possible pattern candidates is computed, $PatCand(S) = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$ and the number of possible pattern candidates is $nc = 7$. At this step, class C_8 is eliminated because it does not satisfy any constraint from the set of all constraints imposed by the *Proxy* design pattern.

Now the **Grouping** step will be performed and first the matrix $DISS(7,7)$ (where a line i corresponds to class C_i and a column j corresponds to class C_j) of dissimilarities between the classes from $PatCand(S)$ will be computed:

$$DISS = \begin{pmatrix} 0 & 0.87 & 0.83 & \infty & \infty & \infty & \infty \\ 0.87 & 0 & 0.75 & 0.88 & \infty & \infty & \infty \\ 0.83 & 0.75 & 0 & \infty & \infty & \infty & \infty \\ \infty & 0.88 & \infty & 0 & \infty & \infty & 0.87 \\ \infty & \infty & \infty & \infty & 0 & 0.87 & 0.85 \\ \infty & \infty & \infty & \infty & 0.87 & 0 & 0.77 \\ \infty & \infty & \infty & 0.87 & 0.85 & 0.77 & 0 \end{pmatrix} \tag{8}$$

After applying *HAC* clustering algorithm, the obtained partition of $PatCand(S)$ is $\mathcal{K} = \{K_1, K_2, K_3\}$, where $K_1 = \{C_4\}$, $K_2 = \{C_3, C_1, C_2\}$ and $K_3 = \{C_5, C_6, C_7\}$.

We mention that without using the dissimilarity matrix $DISS$, the class C_7 would have been grouped with the class C_4, instead of being grouped with classes C_5 and C_6 and an instance of the design pattern *Proxy* would have been missed.

Now we analyze the obtained partition $\mathcal{K}$ in order to identify instances of *Proxy* design pattern, and the identified instances are correctly reported: K_2 and K_3.

6. Analysis of our approach

In the following we will make a time complexity analysis of our search based approach for identifying instances of design patterns in a given software system S. Let us consider that n is the number of entities from S and l is the application classes from S.

Usually, the number nr_p of constraints in a design pattern p is a small constant (as 3 for the *Proxy* design pattern), that is why we will ignore it in the worst time complexity asymptotic analysis.

Analyzing the steps performed in order to identify the instances of design patterns in a given software system (as indicated in Section 4) we can compute their worst time complexity. The results are given in Table 1.

Step	Worst time complexity
Data collection	$O(n)$
Preprocessing	$O(l^2)$
Grouping	$O(l^3)$
Design pattern instances recovery	$O(l^3)$

TABLE 1. Complexity asymptotic analysis.

Based on the results from Table 1, we can conclude that the overall worst time complexity of our approach is $O(max\{n, l^3\})$. As, in a large real software system, usually $l^3 > n$, the overall complexity is $O(l^3)$.

As a conclusion, we can summarize the advantages of the search based approach proposed in this paper in comparison with existing approaches:

- The overall worst time complexity ($O(l^3)$) of our approach is reduced in comparison with the worst time complexity of a brute force approach ($O(l^{nc_p})$) (as the number of classes nc_p of classes contained in a design pattern p is greater or equal to 3).
- Our approach is not dependent on a particular design pattern. It may be used to identify instances of various design patterns, as any design pattern can be described according to the theoretical model introduced in Section 3.
- Our approach may be used to identify both *structural* and *behavioral* design patterns, as the constraints can express both structural and behavioral aspects of the application classes from the analyzed software system.

7. Conclusions and future work

We have introduced in this paper a search based approach for identifying instances of design patterns in existing software systems. We have emphasized the advantages of our approach in comparison with existing approaches in the field.

Further work can be done in the following directions:

- Improving the **Preprocessing** and **Grouping** steps from our approach.
- Applying the proposed approach on real software systems.
- Extending the proposed approach towards identifying several design patterns.
- Extending the proposed approach towards introducing design patterns in existing software systems.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti, *Using metrics to identify design patterns in object-oriented software*, Proc. of the Fifth International Symposium on Software Metrics - METRICS'98, 1998, pp. 23–34.

[2] James M. Bieman and Byung-Kyoo Kang, *Measuring design-level cohesion*, Software Engineering **24** (1998), no. 2, 111–124.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object oriented software*, Addison-Wesley Publishing Company, USA, 1995.

[4] M. Meyer, J. Niere, and L. Wendehals, *User-driven adaption in rule-based pattern recognition*, Technical Report TR-RI-04-249 (2004), University of Paderborn, Paderborn, Germany.

[5] A. K. Jain, M. N. Murty, and P. J. Flynn, *Data clustering: a review*, ACM Computing Surveys **31** (1999), no. 3, 264–323.

[6] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and Morris Bernstein, *Pattern matching for clone and concept detection*, Automated Software Engineering **3** (1996), no. 1/2, 77–108.

[7] Christian Kramer and Lutz Prechelt, *Design recovery by automated search for structural design patterns in object-oriented software*, WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96), 1996, pp. 208–215.

[8] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh, *Towards pattern-based design recovery*, ICSE '02: Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 338–348.

[9] Jörg Niere, Jörg P. Wadsack, and Lothar Wendehals, *Handling large search space in pattern-based reverse engineering*, IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension, 2003, pp. 274.

[10] Elaine Rich and Kevin Knight, *Artificial intelligence*, 2nd ed., McGraw Hill, New York, 1991.

[11] G. Serban and I.G. Czibula, *On evaluating software systems design*, Studia Universitatis "Babes-Bolyai", Informatica **LII** (2007), no. 1, 55–66.

[12] Frank Simon, Silvio Loffler, and Claus Lewerentz, *Distance based Cohesion Measuring*, Proceedings of the 2nd European Software Measurement Conference (FESMA), 1999, pp. 69–83.

[13] L. Wendehals, *Improving Design Pattern Instance Recognition by Dynamic Analysis*, Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), 2003, pp. 29–32.

Department of Computer Science, Babeş-Bolyai University 1, M. Kogălniceanu Street, 400084, Cluj-Napoca, Romania,
E-mail address: gabis@cs.ubbcluj.ro

Department of Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu Street, 400084, Cluj-Napoca, Romania,
E-mail address: istvanc@cs.ubbcluj.ro