Towards the Identification of Cross-Cutting Concerns: A Comprehensive Dynamic Approach Based on Execution Relations

Article *in* IEICE Transactions on Information and Systems · May 2014

DOI: 10.1587/transinf.E97.D.1235

3 authors, including:

Dongjin Yu
Hangzhou Dianzi University
71 PUBLICATIONS **381** CITATIONS
SEE PROFILE

Mu Yunlei
Hangzhou Dianzi University
1 PUBLICATION **3** CITATIONS
SEE PROFILE

The Journal of Systems and Software 103 (2015) 1–16

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures

Dongjin Yu*, Yanyan Zhang, Zhenli Chen

School of Computer, Hangzhou Dianzi University, Hangzhou, China

ARTICLE INFO

Article history:
Received 20 April 2014
Revised 30 November 2014
Accepted 10 January 2015
Available online 15 January 2015

Keywords:
Design pattern recovery
Design patterns
Sub-patterns
Graph mining
Method signature

ABSTRACT

Design patterns are formalized best practices that address concerns related to high-level structures for applications being developed. The efficient recovery of design pattern instances significantly facilitates program comprehension and software reengineering. However, the recovery of design pattern instances is not a straightforward task. In this paper, we present a novel comprehensive approach to the recovery of instances of 23 *GoF* design patterns from source codes. The key point of the approach lies in that we consider different design pattern instances consist of some commonly recurring sub-patterns that are easier to be detected. In addition, we focus not only on the class relationship, but also on the characteristics of underlying method signatures in classes. We first transform the source codes and predefined *GoF* patterns into graphs, with the classes as nodes and the relationships as edges. We then identify the instances of sub-patterns that would be the possible constituents of pattern instances by means of subgraph discovery. The sub-pattern instances are further merged by the joint classes to see if the collective matches one of the predefined patterns. Finally, we compare the behavioral characteristics of method invocation with the predefined method signature templates of *GoF* patterns to obtain the final pattern instances directly. Compared with existing approaches, we integrate and improve some of the previous ideas and put forward a comprehensive and elaborative approach also based on our own ideas. We detect sub-patterns via graph isomorphism based on prime number composition and the joint classes to reduce the search space. Meanwhile, we employ the method signatures to investigate the behavioral features to avoid choosing the test cases with full code coverage. The results of the extensive experiments on recovering pattern instances from nine open source software systems demonstrate that our approach obtains the balanced high precision and recall.

1. Introduction

Design patterns are extensively used in software development to provide reusable and documented solutions to common design problems. In the mid-1990s, the idea of patterns was adopted by object-oriented software developers, among whom the so-called *GoF* (Gang of Four, Gamma, Helms, Johnson and Vlisides) cataloged 23 design patterns aimed at meeting some commonly-recurring object-oriented design needs (Gamma et al., 1995). However, although the majority of current software systems embed instances of design patterns in source codes, the pattern-related knowledge is generally no longer available after patterns are applied and implemented. Recovering the instances of design patterns from the source codes of software systems can therefore assist the understanding of the systems and the process of re-engineering them. More importantly, it also helps to trace back to the original design decisions, which are typically missing in legacy systems (Dong et al., 2009a; Pettersson et al., 2010).

Nevertheless, the recovery of design pattern instances is not a straightforward task. The lack of documentation, the ad-hoc nature of programming and the possible variants of pattern instances often lead to the low accuracy of pattern occurrence detection. There have been a number of different approaches proposed to solve this problem in the literature (Dong et al., 2009b; Fontana et al., 2011b; Niere et al., 2002; Rasool and Mäder, 2011; Tsantalis et al., 2006). Unfortunately, many of them focus on just one category of design patterns (creational, structural and behavioral), or even some specific patterns. Meanwhile, the recovering process is usually lack of the accurate and elaborative explanation. Some just present the illustrative examples, but not the formal approaches. Moreover, the extensive results of experiments with precisions and recalls are seldom reported.

As a matter of fact, different patterns may contain some common sub-components that share the same internal structures, such as one class inheriting another class that further inherits the third, and one

* Corresponding author. Tel.: +86 13357108288.
E-mail address: yudj@hdu.edu.cn (D. Yu).

http://dx.doi.org/10.1016/j.jss.2015.01.019
0164-1212/

class inheriting and containing another class. We name these sub-components of patterns as the sub-patterns.

In addition, some researches detect design pattern instances by means of dynamic behavioral analyzing (De Lucia et al., 2010, 2011; Heuzeroth et al., 2003a; Wendehals, 2011). The effectiveness of these approaches, however, relies heavily on the test case chosen. The more codes the test case covers, the more pattern instances might be identified. Nevertheless, it is usually very difficult or even impossible to choose the test cases that cover the majority of source codes.

In this paper, we try to integrate and improve the previous ideas and put forward a comprehensive and elaborative approach to the recovering *GoF* design pattern instances from source codes also based on our own ideas. Unlike other approaches, it makes good use of the intermediate results, or sub-patterns of patterns, while detecting several kinds of pattern instances at the same time. In other words, the detection of somewhat complicated pattern instances is transformed into the detection of the simpler sub-pattern instances, which can then be merged into different kinds of pattern instances. In addition, it extracts the method signatures in classes, which are then compared with predefined templates. In this way, it can obtain the balanced high precision and recall without collecting execution traces for behavioral analyzing.

This study focuses on *GoF* patterns although it does not suggest that *GoF* patterns are better practices than other design patterns. However, the interest that *GoF* patterns have attracted from both academia and industry justifies the scope of this study. We believe what we have proposed in this paper is also applicable for other design pattern catalogs.

The rest of the paper is organized as follows. In Section 2, we introduce the definitions of sub-patterns and structural feature models, as well as the directed graphs used to represent the structural characteristics of source codes. Afterward, in Section 3, we describe the process of recovering design pattern instances which covers four steps, namely system modeling, sub-pattern detecting, sub-pattern combining and method-signature analyzing. Section 4 presents the experimental results in detail, whereas Section 5 discusses the threats that could affect the validity of our study. Following the related works given in Section 6, the last section concludes the paper and outlines the future work. The interested readers may refer the Appendix which presents all the structural feature models that correspond to 23 *GoF* design patterns.

2. Definitions

Design patterns are concerned with how classes are composed to form larger structures. No matter whether they are creational, structural and behavioral ones, they usually share some similarities, especially in their participants and collaborations. This is so probably because design patterns rely on the same small set of language mechanisms for structuring code and objects (Gamma et al., 1995). In this section, therefore, we first introduce the concept of sub-patterns that represent the frequent occurred constituents in the well known *GoF* design patterns. Afterward, we describe how to represent the pattern in the so-called *Structural Feature Model*, which is the combination of different sub-patterns. Finally, we show the graphic and matrix representation of both sub-patterns and patterns, i.e., *Class-Relationship Directed Graph* and *Class-Relationship Matrix*, in detail.

2.1. Sub-patterns

Definition 0. A directed-relationship from class c_i and c_j is denoted by $r(c_i, c_j) = \{inherit|agg|ass|dep\}$, in which:

(1) $r(c_i, c_j) = \{inherit\}$ represents that class c_j inherits class c_i.
(2) $r(c_i, c_j) = \{ass\}$ represents that classes c_i and c_j have an association relationship, where class c_i has an attribute that is a type of class c_j, or class c_i has a method which returns a c_j object.
(3) $r(c_i, c_j) = \{agg\}$ represents that classes c_i and c_j have an aggregation relationship, where class c_i has an attribute that is a type of class c_j. In the context of this paper, we regard the aggregation as a special kind of association relationship, in which class c_i is the whole class and class c_j is the partial class.
(4) $r(c_i, c_j) = \{dep\}$ represents that class c_i depends on class c_j, which can be further classified into three cases: (i) The instance of class c_i calls a static method in class c_j. (ii) An instance of class c_j is declared in class c_i as a local variable. (iii) An instance of class c_j is used as the parameter passed to a method in class c_i.

Definition 1. A sub-pattern represents a set of classes and the relationship between them, which is denoted as a 2-tuple: $SP = (< c_0, \ldots, c_{k-1} >, R)$, where $c_0, \ldots, c_{k-1}$ represent a set of classes and R represents a set of directed-relationships among them, i.e., $R = \{r(c_i, c_j)|r(c_i, c_j) = \{inherit|agg|ass|dep\}\}$.

For simplicity, we use $ClassSet_{SP}$ to denote the classes that make up the pattern SP.

Using the above definition, we introduce the following 15 sub-patterns, which can be combined to form different design patterns under different circumstances.

Definition 2.1. ICA (Inheritance Child Association) indicates the inheritance and association relationship among three classes, in which two classes have an inheritance relationship and the child class has an association relationship with another child class.

$ICA = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_2, c_3) = \{ass\}\}$.

Definition 2.2. CI (Common Inheritance) indicates that two classes inherit the same parent class.

$CI = (< c_1, c_2, c_3 >, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_1, c_3) = \{inherit\}\}$.

Definition 2.3. IAGG (Inheritance AGGregation) indicates that two classes have not only an inheritance relationship but also an aggregation relationship.

$IAGG = (< c_1, c_2 >, R)$ where $R = \{r(c_1, c_2) = \{inherit\},\ r(c_2, c_1) = \{agg\}\}$.

Definition 2.4. IPAG (Inheritance Parent AGgregation) indicates the inheritance and aggregation relationship between three classes, where two classes have an inheritance relationship and the parent class has an aggregation relationship with the other class.

$IPAG = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_1, c_3) = \{agg\}\}$.

Definition 2.5. MLI (Multi-Level Inheritance) indicates the inheritance relationship between three classes, where the first class is inherited by the second, which is further inherited by the third one.

$MLI = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_2, c_3) = \{inherit\}\}$.

Definition 2.6. IASS (Inheritance ASSociation) indicates that two classes have not only an inheritance relationship but also an association relationship.

$IASS = (< c_1, c_2 >, R)$ where $R = \{r(c_1, c_2) = \{inherit\},\ r(c_2, c_1) = \{ass\}\}$.

Definition 2.7. SAGG (Self-Aggregation) indicates one class has an aggregation relationship with itself.

$SAGG = (< c_1 >, R)$ where $R = \{r(c_1, c_1) = \{agg\}\}$.

Definition 2.8. IIAGG (Indirect Inheritance AGGregation) indicates the inheritance and aggregation relationship among three classes, where the first class is aggregated by the third one and inherited from the second one, which is further inherited from the third one.

$IIAGG = (< c_1, c_2, c_3 >, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_2, c_3) = \{inherit\}, r(c_3, c_1) = \{agg\}\}$.

Definition 2.9. SASS (Self-ASSociation) indicates that one class has an association relationship with itself.

$SASS = (< c_1 >, R)$ where $R = \{r(c_1, c_1) = \{ass\}\}$.

Definition 2.10. ICD (Inheritance Child Dependency) indicates the inheritance and dependency relationships among three classes, where two classes have an inheritance relationship and the child class depends on the other class.

$ICD = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_2, c_3) = \{dep\}\}$.

Definition 2.11. DCI (Dependency Child Inheritance) indicates the dependency and inheritance relationship among three classes, where two classes have an inheritance relationship and the other class depends on the child class.

$DCI = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_3, c_2) = \{dep\}\}$.

Definition 2.12. IPAS (Inheritance Parent ASsociation) indicates the inheritance and association relationship among three classes, where two classes have an inheritance relationship and the parent class has an association relationship with the other class.

$IPAS = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_1, c_3) = \{ass\}\}$.

Definition 2.13. AGPI (AGgregation Parent Inherited) indicates the aggregation and inheritance relationship among three classes, where two classes have an inheritance relationship and the other class has an aggregation relationship with the parent class.

$AGPI = (< c_1, c_2, c_3 >, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_3, c_1) = \{agg\}\}$.

Definition 2.14. IPD (Inheritance Parent Dependency) indicates the inheritance and dependency relationship among three classes, where two classes have an inheritance relationship and the parent class depends on the other class.

$IPD = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_1, c_3) = \{dep\}\}$.

Definition 2.15. DPI (Dependency Parent Inherited) indicates the dependency and inheritance relationship among three classes, where two classes have an inheritance relationship and the other class depends on the parent class.

$DPI = (<c_1, c_2, c_3>, R)$ where $R = \{r(c_1, c_2) = \{inherit\}, r(c_3, c_1) = \{dep\}\}$.

Fig. 1 illustrates the class diagrams of the above mentioned 15 sub-patterns.

2.2. Structural feature models

The design patterns can be regarded as the combinations of certain sub-patterns as defined above. Here, we use the concept of *Structural Feature Model* to represent the structural features of design patterns.

Definition 3. A Structural Feature Model is a set of relevant sub-patterns representing the structural characteristics of one specific design pattern, which is denoted as: $SFM = SFM_0|\ldots|SFM_i|\ldots|SFM_{n-1}$, where SFM_i is the combination of two sub-patterns connected by the joint classes in *jClassSet*, or $SFM_i = SP_{i_1}\&\&SP_{i_2}$, $jClassSet = ClassSet_{SP_{i_1}} \cap ClassSet_{SP_{i_2}}$.

Figs. 9–11 in the Appendix present all the structural feature models that correspond to all 23 *GoF* design patterns. As an example, the *Structural Feature Model* of *Observer* pattern, in Fig. 11(g), is composed of the sub-patterns of *AGPI* and *ICD*. In other words, $SFM_{Observer} = APGI \;\&\&\; ICD = (<Subject, Observer, ConcreteObserver>, R_{AGPI})\&\&(<Observer, ConcreteObserver, ConcreteSubject>, R_{ICD})$. In addition, to create the *Observer* pattern, *AGPI* and *ICD* should be connected with the classes of *Observer* and *ConcreteObserver*. Therefore,

$jClassSet = \{Observer, ConcreteObserver\}$.

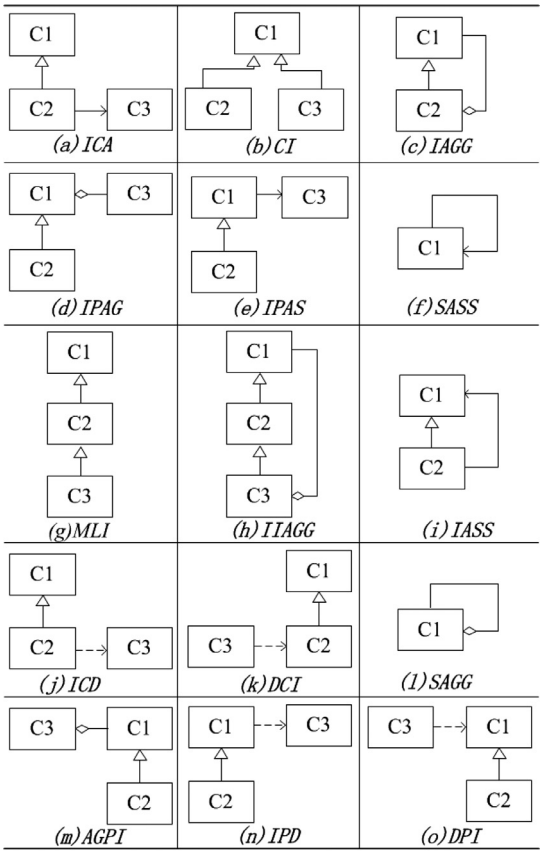


Fig. 1. The class diagrams of 15 sub-patterns.

2.3. Class-relationship directed graph

Definition 4. Class-Relationship Directed Graph, or GCDR is a directed and weighted graph that represents a set of classes and the relationships between them, denoted as a 3-tuple $GCDR = (V, E, W)$ where $V = \{v_0, v_1, \ldots, v_n\}$ is the set of vertices which represent classes, $E = \{e(v_i, v_j)\} \subset V \times V$ is the set of directed edges which represent the directed-relationship between classes, $W : E \to W_E$ is the function that assigns weights to the edges, or the relationships.

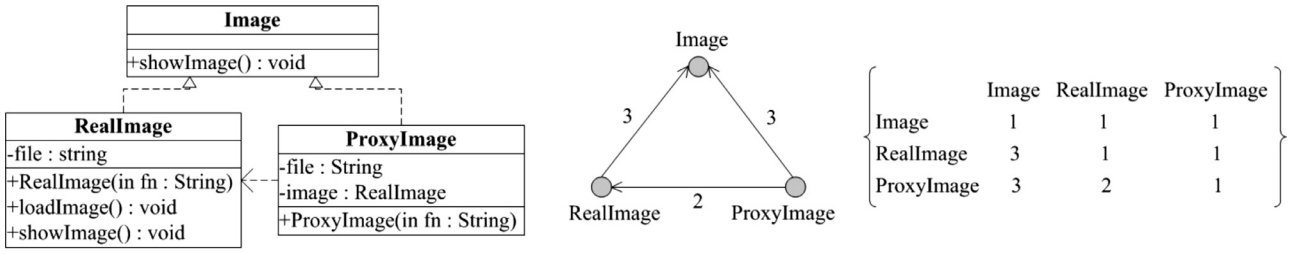
We use the prime number of 2, 3, 5 and 7 to represent the weights of the directed-relationship of association, inheritance, aggregation and dependency respectively. More specifically, if $r(c_i, c_j) = \{ass\}$ then $w(v_i, v_j) = 2$, if $r(c_i, c_j) = \{inherit\}$ then $w(v_i, v_j) = 3$, if $r(c_i, c_j) = \{agg\}$ then $w(v_i, v_j) = 5$ and if $r(c_i, c_j) = \{dep\}$ then $w(v_i, v_j) = 7$. Here, c_i and c_j are the classes corresponding to the vertices v_i and v_j.

If two classes hold multiple relationships, we simply multiply the corresponding prime numbers. On the other hand, given a composite number that represents the relationship of two classes, we can uniquely decompose it into the product of several prime numbers. In this way, the specific relationships two classes actually hold can be easily revealed based on the underlying prime numbers.

Definition 5. Let $GCDR = (V_1, E_1, W_1)$, $GCDR^{ksub} = (V_2, E_2, W_2)$. $GCDR^{ksub}$ is called the *k-SubGraph of Class-Relationship Directed Graph*, or *k-SubGraph of GCDR*, if $V_2 \subseteq V_1$, $|V_2| = k$, $E_2 = E_1 \cap (V_2 \times V_2)$, and $w_2(e) = w_1(e)$ for all $e \in E_2$. Here, k is called the size of $GCDR^{ksub}$.

Definition 6. Let $GCDR = (V, E, W)$, $|V| = n$, $M = (m_{i,j})_{n\times n}$ is called the *Class-Relationship Matrix* of *GCDR*, or *MCR*, where

$$m_{i,j} = \begin{cases} w(v_i, v_j), & e(v_i, v_j) \in E \\ 1, & e(v_i, v_j) \notin E \end{cases} \quad (1)$$



(a) the example of class diagrams (b) the example of Class-Relationship Directed Graph (c) the example of Class-Relationship Matrix

Fig. 2. The example of the class diagram and its corresponding class-relationship directed graph and class-relationship matrix.

For simplicity, the *Class-Relationship Directed Graph* corresponding to *Class-Relationship Matrix* m is denoted by $GCDR_m$.

Definition 7. Let $GCDR = (V, E, W)$, $|V| = n$, $v_i \in V$, for the *Class-Relationship Matrix* of $M = (m_{i,j})_{n\times n}$, the *Outbound Composite Weight* of v_i is denoted as $CW_{out}(v_i)$, where $CW_{out}(v_i) = \prod_{j=1}^{n} m_{i,j}$.

Definition 8. Let $GCDR = (V, E, W)$, $|V| = n$, $v_i \in V$, for the *Class-Relationship Matrix* of $M = (m_{i,j})_{n\times n}$, the *Inbound Composite Weight* of $v_i \in V$ is denoted as $CW_{in}(v_i)$, where $CW_{in}(v_i) = \prod_{j=1}^{n} m_{j,i}$.

Fig. 2 gives an example of class diagrams and their corresponding *Class-Relationship Matrix*, in which *ProxyImage* and *RealImage* inherit *Image* and *ProxyImage* refers *RealImage*, or in other words, $r(C_{Image}, C_{RealImage}) = \{inherit\}$, $r(C_{Image}, C_{ProxyImage}) = \{inherit\}$ and $r(C_{ProxyImage}, C_{RealImage}) = \{ass\}$. According to Definitions 7 and 8, we can simple find that $CW_{out}(V_{Image}) = 1*1*1 = 1$, $CW_{out}(V_{RealImage}) = 3*1*1 = 3$, $CW_{out}(V_{ProxyImage}) = 3*2*1 = 6$. $CW_{in}(V_{Image}) = 1*3*3 = 9$, $CW_{in}(V_{RealImage}) = 1*1*2 = 2$ and $CW_{in}(V_{ProxyImage}) = 1*1*1 = 1$.

3. Recovering design pattern instances

The problem can be defined as follows: given the source codes of one certain system, recover all the instances of *GoF* design pattern instances that the system contains. To resolve this problem, we developed an approach that consists of the following four steps.

(1) Modeling the source codes
The source codes of the system are scanned and transformed into a series of *Class-Relationship Directed Graphs*, in which the vertices represent classes and the edges and their weights represent relationships between classes.
(2) Detecting the sub-pattern instances
The sub-graphs of the *Class-Relationship Directed Graphs* obtained in the previous step are checked according to the *Class-Relationship Directed Graphs* of the predefined sub-patterns. The matched ones are considered as the instances of the sub-patterns.
(3) Combining the sub-pattern instances to make the candidate pattern instances
The relevant sub-patterns are combined based on the joint classes and compared with the predefined *Structural Feature Models*. Those matched ones are considered as the *candidate* instances of design patterns.
(4) Analyzing method signatures to obtain the final recovered pattern instances
In this step for behavioral checking, the method signatures of the classes that are part of the *candidate* pattern instances are further investigated and compared with the predefined templates of standard patterns given in Table 3. Those matched ones are regarded as the *final* recovered pattern instances.

Above four steps and their corresponding outputs are illustrated in Fig. 3, and are discussed in detail in the following.

3.1. Modeling source codes

In this pre-process step, the structural information, such as the classes, the method signatures and the relationship among classes, is extracted from source codes for the detection of pattern instances. We first use Enterprise Architect, a visual platform for modeling software systems which can be downloaded from http://www.sparxsystems.com.au, to parse the source codes of each class. Afterward, we produce class diagrams and XML-based Metadata Interchange files, or XMI files, which represent the structural information of system for further analysis. We then generate the *Class-Relationship Matrix* of the system represented by the obtained XMI files.

Fig. 4 shows a fragment of XMI file, which describes the structural information inside and among classes of Image, RealImage and ProxyImage in Fig. 2.

3.2. Detecting sub-patterns

Since both the systems and predefined sub-patterns can be represented by *Class-Relationship Directed Graphs*, the problem of detecting

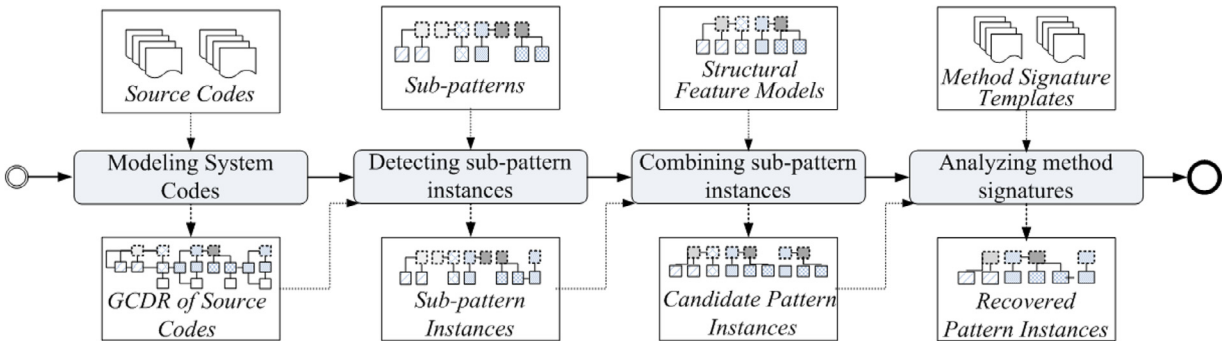


Fig. 3. Process of recovering design pattern instances.

```
<?xml version="1.0" encoding="windows-1252"?>
<!DOCTYPE XMI SYSTEM "UML_EA.dtd">
<XMI xmi.version="1.1" xmlns:UML="omg.org/UML1.3" timestamp="2014-04-03 13:20:27">
    <XMI.header>
    <XMI.content>
        <UML:Model name="EA Model" xmi.id="MX_EAID_828A0FDD_151A_458f_AE42_FD218DD68573">
            <UML:Namespace.ownedElement>
                <UML:Class name="EARootClass" xmi.id="EAID_11111111_5487_4080_A7F4_41526CB0AA00" isRoot="true" isLeaf="false" isAbstract="false"/>
                <UML:Package name="patterns" xmi.id="EAPK_828A0FDD_151A_458f_AE42_FD218DD68573" isRoot="false" isLeaf="false" isAbstract="false" visibility="public">
                    <UML:Namespace.ownedElement>
                        <UML:Class name="ProxyImage" xmi.id="..." visibility="public" namespace=".." isRoot="false" isLeaf="false" isAbstract="false" isActive="false">
                            <UML:Attribute name="file" changeable="none" visibility="private" ownerScope="instance" targetScope="instance">
                            <UML:Attribute name="image" changeable="none" visibility="private" ownerScope="instance" targetScope="instance">
                            <UML:Operation name="ProxyImage" visibility="public" ownerScope="instance" isQuery="false" concurrency="sequential">
                            <UML:Operation name="showImage" visibility="public" ownerScope="instance" isQuery="false" concurrency="sequential">
                        </UML:Class>
                        <UML:Dependency client="..." supplier="..." xmi.id="..." visibility="public">
                        <UML:Association xmi.id="..." visibility="public" isRoot="false" isLeaf="false" isAbstract="false">
                        <UML:Interface name="Image" xmi.id="..." visibility="public" namespace="..." isRoot="false" isLeaf="false" isAbstract="true">
                            <UML:Operation name="showImage" visibility="public" ownerScope="instance" isQuery="false" concurrency="sequential">
                        </UML:Interface>
                        <UML:Dependency client="..." supplier="..." xmi.id="..." visibility="public">
                        <UML:Class name="RealImage" xmi.id="..." visibility="public" namespace="..." isRoot="false" isLeaf="false" isAbstract="false" isActive="false">
                            <UML:Attribute name="file" changeable="none" visibility="private" ownerScope="instance" targetScope="instance">
                            <UML:Operation name="loadImage" visibility="private" ownerScope="instance" isQuery="false" concurrency="sequential">
                            <UML:Operation name="RealImage" visibility="public" ownerScope="instance" isQuery="false" concurrency="sequential">
                            <UML:Operation name="showImage" visibility="public" ownerScope="instance" isQuery="false" concurrency="sequential">
                        </UML:Class>
                    </UML:Namespace.ownedElement>
                </UML:Package>
            </UML:Namespace.ownedElement>
        </UML:Model>
    </XMI.content>
    <XMI.difference/>
    <XMI.extensions xmi.extender="Enterprise Architect 2.5"/>
</XMI>
```

Fig. 4. The example of XMI file fragment (due to the limited space, only the essential part is given).

Table 1
Algorithm of detecting sub-pattern instances.

Input:
$GCDR^m =< V_m, E_m, W_m >$ //*Class-Relationship Directed Graph* of sub-pattern
$GCDR^s =< V_s, E_s, W_s >$ //*Class-Relationship Directed Graph* of system
Output:
IdentifiedSubPatternSet //Set of *Class-Relationship Directed Graphs* of identified sub-pattern instances

1 IdentifiedSubPatternSet = ∅
2 //generate *Candidate Vertex Sets*
3 $k = |V_m|$
4 for each $c_i (1 \le i \le k)$ in V_m {
5 $CVS(c_i) = \{c | c \in V_s$
6 $\wedge (CW_{out}(c_i) | CW_{out}(c))$ //check the *Outbound Composite Weight*
7 $\wedge (CW_{in}(c_i) | CW_{in}(c))\}$ //check the *Inbound Composite Weight*
8 }
9 for each m_1 in $CVS(c_1), \ldots, m_i$ in $CVS(c_i), \ldots, m_j$ in $CVS(c_j), \ldots, m_k$ in $CVS(c_k)$ {
10 if exists m_i, m_j such that $NOT(w(c_i, c_j) | w(m_i, m_j))$ break
11 generate $GCDR^{s-ksub} =< V_{s-ksub}, E_{s-ksub}, W_{s-ksub} >$, where
12 $V_{s-ksub} = \{m_1, m_2, \ldots, m_k\}$,//pick one class from each *Candidate Class Set*
13 $E_{s-ksub} = E_s \cap (V_{s-ksub} \times V_{s-ksub})$,
14 $W_{s-ksub}(e) = W_s(e)$, for all $e \in E_{s-ksub}$,
15 $IdentifiedSubPatternSet = IdentifiedSubPatternSet \cup GCDR^{s-ksub}$
16 }
17 return IdentifiedSubPatternSet

sub-patterns can be converted into searching for sub-graphs in *Class-Relationship Directed Graphs* of the system, which are isomorphic to those of the sub-patterns.

The algorithm for detecting sub-patterns is illustrated in Table 1, and can be divided into the following two steps.

(1) Search the candidate vertices in $GCDR^s$ of the system (from line 4 to line 8)
For each vertex v_i in $GCDR^m$, select the vertices in $GCDR^s$ whose *Outbound Composite Weights* and *Inbound Composite Weights* in $GCDR^s$ can be divided with no remainder by those of v_i in $GCDR^m$. These selected vertices for c_i constitute the *Candidate Vertex Set* of v_i.
(2) Combine the candidate vertices to generate *k-subGraphs* (from line 9 to line 16)
The *k-subGraph* of $GCDR^s$, or $GCDR^{s-ksub}$, is generated by choosing each vertex in every *Candidate Vertex Set*, i.e., a sequence of candidate vertices. The sequence of candidate vertices are ignored first if the weight of any connecting edge in $GCDR^s$ cannot be divided with no remainder by the corresponding one in $GCDR^m$ (line 10).

In this way, the sub-graphs of the *Class-Relationship Direct Graph* are checked that if the corresponding classes hold the correct relationship with others according to the underlying prime numbers assigned.

3.3. Combining sub-pattern instances to make candidate pattern instances

After all sub-patterns are identified, we then combine the relevant ones and compare those collectives with the *Structural Feature Models* of *GoF* patterns. Those that match are then picked up as the candidate pattern instances for further behavioral analysis.

Since a pattern is composed of several sub-patterns with joint classes, only those connected sub-pattern instances are considered for the creation of pattern instances. The algorithm for combining sub-pattern instances to pattern instances is illustrated in Table 2. For each possible *Structural Feature Model* for a specific pattern, we pick two different sub-pattern instances that are of relevant types and check if they are connected with the joint classes (line 6). The pattern instances finally kept in *PatternInstanceSet* are considered as the candidate pattern instances found after the structural checking.

3.4. Analyzing method signatures to obtain final recovered pattern instances

The candidate pattern instances recovered in the above step only match the standard pattern structures on the class level. Most design patterns, however, have their own unique behavioral features related with method invocation. Therefore, we further investigate the method signatures inside the involved classes of the candidate pattern instances. In this way, we can filter out a large number of false ones.

In order to characterize the features of method invocation, we define the template of method signatures for each *GoF* design pattern, which describes how one certain method inside the class with one certain role is invoked and what it returns, as Table 3 indicates.

Once the candidate pattern instances are obtained, the roles that the related classes play are determined. We examine the method signatures inside the classes with the specific roles. Only those instances in which the classes with certain roles contain the method

Table 2
Algorithm of combining sub-patterns.

Input:
SFM // *Structural Feature Model* of design pattern instance to be detected
SubPatternInstanceSet //Set of *Class-Relationship Directed Graphs* of identified sub-pattern instances
Output:
PatternInstanceSet //Set of *Class-Relationship Directed Graphs* of identified pattern instances conformed to *SFM*

```
1   PatternInstanceSet = ∅
2   for each SFM_i = SP_i in SFM{ //for all possible Structural Feature Models
3     Suppose SFM_i = SP_{i_1} && SP_{i_2} //the constitutes of the Structural Feature Model
4     for each sub-pattern instance SP_a in SubPatternInstanceSet that is type of SP_{i_1}{
5       for each sub-pattern instance SP_b in SubPatternInstanceSet that is type of SP_{i_2}{
6         if(ClassSet_{SP_{i_1}} ∩ ClassSetSP_{i_2}) ⊆ (ClassSet_{SP_a} ∩ ClassSet_{SP_b}){
7           GCDR^combined = GCDR^{SP_a} ∪ GCDR^{SP_b}
8           PatternInstanceSet = PatternInstanceSet ∪ GCDR^combined
9         }// end if
10      } //end for each sub-pattern instance SP_b
11    } //end for each sub-pattern instance SP_a
12  }//end for each possible Structural Feature Model
13  return PatternInstanceSet
```

Table 3
Templates of method signatures of 23 *GoF* design patterns.

Type	Patterns	Templates of method signatures
Structural design patterns	Adapter	*Target*.[*Method*] = *Adapter*.[*Method*] → *Adaptee*.[*Method*]
	Bridge	*Abstraction*.[*Method*] = *RefinedAbstraction*.[*Method*] → *Implementor*.[*Method*] = *ConImplementor*.[*Method*]
	Composite	*Component*.[*Method*] = *Composite*.[*Method*] → *Component*.[*Method*]
	Proxy	*Subject*.[*Method*] = *Proxy*.[*Method*] → *RealSubject*.[*Method*]
	Decorator	(1) *Component*.[*Method*] = *Decorator*.[*Method*] → *Component*.[*Method*] (2) *Component*.[*Method*] = *Decorator*.[*Method*] = *ConcreteDecorator*.[*Method*] → *Component*.[*Method*]
	Flyweight	*FlyweightFactory*.[*Method*] ⇒ *Flyweight*, *FlyweightFactory*.[*Method*] → *ConcreteFlyweight*.[*ConstructionMethod*]
	Facade	*Facade*.[*Method*] = *ConcreteFacade*.[*Method*] → *SubSystem*.[*Method*]
Creational design patterns	Abstract factory	*AbstractFactory*.[*Method*] = *ConFactory*.[*Method*] ⇒ *Product* *ConFactory*.[*Method*] → *ConProduct*.[*ConstrucationMethod*]
	Builder	*Builder*.[*Method*] = *ConBuilder*.[*Method*] ⇒ *Product* *Director*.[*Method*] → (*Builder*.[*Method*] = *ConBuilder*.[*Method*])
	Factory method	*Creator*.[*Mehtod*] = *ConcreteCreator*.[*Mehod*] ⇒ *Product*
	Prototype	*ConcretePrototype* ∼ {*Cloneable*}
	Singleton	(1) *Singleton*.[*ConstructionMethod*] ◊ "*private*" (2) *Singleton*.[*ConstructionMethod*] ◊ "*protected*" (3) *Singleton*.[*ConstructionMethod*] ◊ "*public*", *Singleton*.[*ConstrcutionMethod*] ◊ "*static*"
Behavioral design patterns	Chain of responsibility	*Handler*.[*Method*] = *ConcreteHandlerB*.[*Method*] = *ConcreteHandlerA*.[*Mehtod*] → *Handler*.[*Method*]; *Handler*.[*Method*] = *ConcreteHandlerA*.[*Method*] = *ConcreteHandlerB*.[*Method*] → *Handler*.[*Method*]
	Command	*Commander*.[*Method*] = *ConcreteCommand*.[*Method*] → *Receiver*.[*Mehtod*] *Invoker*.[*Method*] → *Command*.[*Method*] = *ConcreteCommand*.[*Method*]
	Interpreter	*AbstractExpression*.[*Method*] = *TerminalExpression*.[*Method*] = *NonerminalExpression*.[*Method*] → *Receiver*.[*Method*]
	Iterator	*ConcreteIterator*.[*Method*] ⇒ *ConcreteAggregate*.*Collection*.*Object*
	Mediator	*Colleague*.[*Method*] ⇒ *Mediator*, *Mediator*.[*Method*] = *ConMediator*.[*Method*] → *Colleague*.[*Method*]
	Memento	*Caretaker*.[*Method*] ⇒ *Memento*, *Original*.[*Method*] → *ConMemento*.[*Method*]
	Observer	*Subject*.[*Method*] → *Observer*.[*Method*] = *ConObserver*.[*Method*]
	State	*State*.[*Method*] = *ConStateA*.[*Method*] → *ConStateB*.[*ConstructionMethod*] *State*.[*Method*] = *ConStateB*.[*Method*] → *ConstateA*.[*ConstructionMethod*]
	Strategy	*Context*.[*Method*] → *Strategy*.[*Method*] = *ConStrategyA*.[*Method*] *Context*.[*Method*] → *Strategy*.[*Method*] = *ConStrategyB*.[*Method*]
	Template	*AbstractClass*.[*Method*] → *AbstractClass*.[*Method*] = *ConcreteClass*.[*Method*]
	Visitor	*Element*.[*Method*] = *ConElement*.[*Method*] → *Visitor*.[*Method*]

Notations and their descriptions:

- *Role*.[*Method*] : any method in the class with the role of *Role*.
- *Role*.[*ConstructionMethod*] : any construction method in the class with the role of *Role*.
- *Role*1.[*Method*] = *Role*2.[*Method*] : the class of *Role1* and the class of *Role2* define the same of *Method*.
- *Role*1.[*Mehtod*] → *Role*2.[*Method*] : a method in the class of *Role1* invokes a method in the class of *Role2*.
- *Role*1.[*Method*] ⇒ *Role*2 : a method in the class of *Role1* returns a class of *Role2*.
- *Role*1 ∼ *Role*2 : the class of *Role1* or one of its ancestors implements the interface of *Role2*.
- *Role*.[*Method*] ◊ "*Modifier*" : the class of *Role* defines a method with the modifier of *Modifier*.

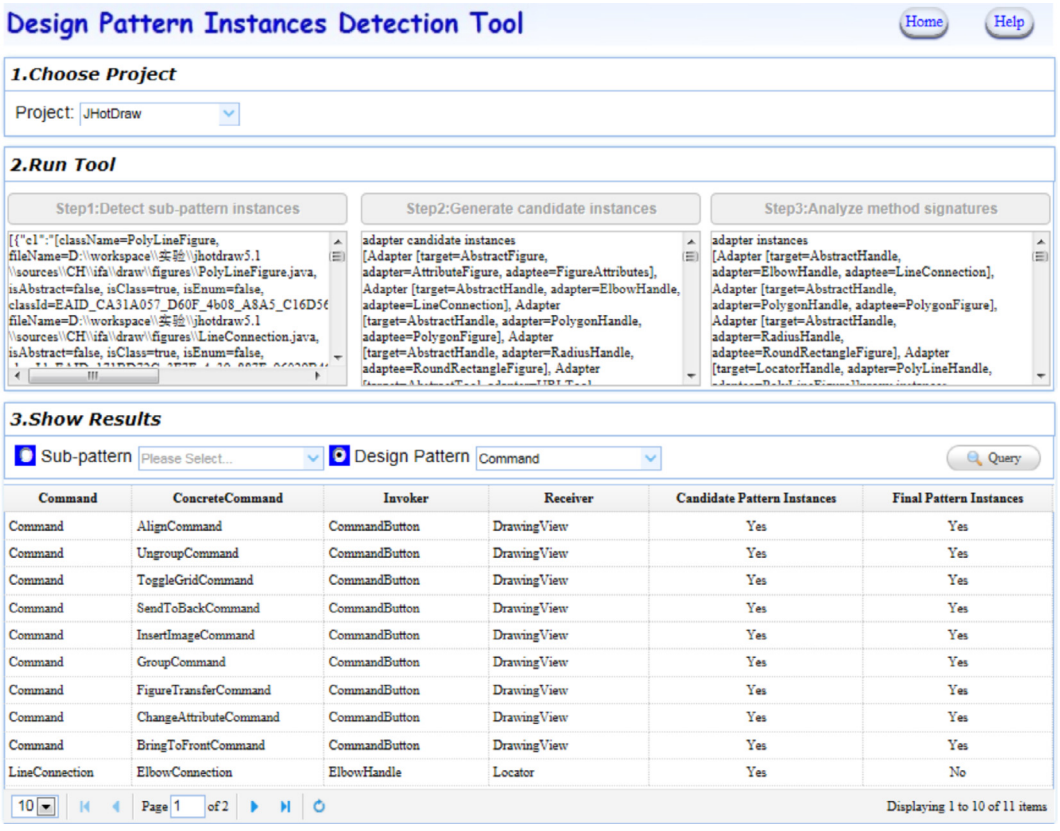


Fig. 5. The screenshot of design pattern instances detecting tool.

Table 4
Characteristics of the software systems used in experiment.

Project	Category	Version	#Classes	KLOC
JavaAWT	Graphics user interface	5.0	345	56
JHotDraw	Graphics user interface	5.1	218	10
JUnit	Unit testing	3.8	56	4
Dom4j	Java XML library	1.6.1	179	18
Lizzy	Communications	1.1.1	197	13
Hodoku	Game	2.1.1	170	54
Barcode4J	Business and enterprise	2.1.0	135	11
RstpProxy	Audio and video	3.0	158	13
Teamcenter	Business and enterprise	6.02.199	143	13
Total			1601	192

signatures conforming to the corresponding method signature templates are considered as the final recovered pattern instances.

More specifically, we parse the XMI file obtained in Step A and investigate if one certain method signature has the correct features, such as '=', '→', '⟹', '◊' and '∼' given in Table 3. Since the roles of each class are already determined through Step C, the analysis of method signatures is therefore straightforward.

4. Experiments

In order to evaluate the effectiveness of our approach, we developed the tool called *Design Pattern Instances Detecting Tool*, as Fig. 5 shows.

We applied the approach on nine open source software systems that are widely used as the benchmarks of design pattern occurrence detection. Table 4 shows the characteristics of the nine software systems used in the experiment. We ran the experiments on Windows 7 with Intel Core2 Duo CPU E7500. The numbers of identified sub-pattern instances in these nine software systems are given in Table 5.

Table 5
Numbers of sub-patterns instances recovered.

	JavaAWT	JHotDraw	JUnit	Dom4j	Lizzy	Hodoku	Barcode4J	Rtsp	Teamcenter
AGPI	622	15	2	224	52	29	73	287	65
CI	550	265	11	102	358	108	116	374	153
DCI	499	178	3	218	298	14	26	73	44
DPI	1645	294	62	700	376	37	139	96	101
IAGG	42	2	2	15	6	0	1	0	2
IASS	42	2	2	15	6	0	1	0	2
ICA	727	154	33	169	254	49	65	103	82
ICD	1230	260	31	216	308	66	76	54	55
IIAGG	4	6	0	0	1	0	0	1	0
IPAG	408	30	1	69	145	58	81	30	22
IPAS	429	171	7	284	46	1	24	32	16
IPD	1042	271	24	251	270	47	87	46	6
MLI	143	117	10	93	54	7	53	61	39
SAGG	41	2	0	9	8	17	9	2	17
SASS	70	3	0	20	41	37	10	20	16
Total	7494	1770	188	2385	2223	470	761	1179	620

We also publish all the results including the recovered sub-pattern instances, candidate pattern instances and final pattern instances of nine software systems at http://dbsi.hdu.edu.cn/dpidp/index.jsp for reference.

We evaluate the performance of our approach by using two widely-adopted metrics, namely precision and recall. However, a higher precision usually leads to a lower recall. On the other hand, if we try to increase the recall, the precision would be inevitable decreased. Therefore, we also report the F-measure, defined as the harmonic mean of precision and recall. We scanned the source codes and identified the pattern instances using the toolkit based on our proposed approach. The identified pattern instances were then manually examined to see whether they are correct. Because manual identification of all true pattern instances in large unfamiliar applications is extremely challenging, we used the repository of *Percerons*, which contains more than 4500 pattern instances extracted from 141 Java open source projects (Ampatzoglou et al., 2013b), to obtain the recall in a more realistic way. In other words, we marked the pattern instance in *Percerons* as the true ones and the recall is then calculated as percentage of how many pattern instances recovered among those recovered in *Percerons*. Thus, the precision, recall and F-measure can be calculated as follows.

$$Precision = \frac{|TPIR|}{|APIR|}\% \tag{2}$$

$$Recall = \frac{|TPIR \cap TPIRIP|}{|TPIRIP|}\% \tag{3}$$

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

where *TPIR* represents *True Pattern Instances Recovered*, and *APIR* represents *All Pattern Instances Recovered*, and *TPIRIP* represents *True Pattern Instances Recovered in Percerons*.

Table 6 shows the number of pattern instance recovered after the structural analysis and after behavioral analysis (method signature analysis), together with the precision, recall and F-measure, for 23 *GoF* patterns for all nine systems.

Fig. 6 illustrates the total precision, recall and F-measure of our approach to the recovery of *GoF* design pattern instances for the nine open source systems. As indicated, our approach achieves high precision and recall, particularly on JavaAWT, JhotDraw, Junit and Dom4j which seem to be better structured than the other five. In summary, the precisions range from 68% to 100%, whereas the recalls range from 73% to 100%.

In order to get a clear idea of the performance of our approach, we also measure the execution time for the sub-pattern instance recovery, structural analysis and behavioral analysis, as shown in Figs. 7 and 8.

As Fig. 8 indicates, it takes the most time to conduct the structural analysis for the recovery of the instances of *Memento*. The reason for this is the numbers of *AGPI* and *DPI* sub-pattern instances are extremely large as Table 5 shows. Moreover, Fig. 8 shows it takes the most time to conduct the behavioral analysis for the recovery of *Chain of Responsibility* instances. The reason lies in that its behavioral analysis filters out a large number of false ones for the recovery of *Chain of Responsibility* instances.

As Figs. 7 and 8 indicate, the execution time depends on not only the number of classes involved, but also the number of pattern instances recovered and the complexity of pattern instances. Generally speaking, the time spent on behavioral analysis or method signature analysis is much longer than the time spent on structural analysis. Besides, as the sub-pattern instances are recovered simultaneously, the time spent on recovering sub-pattern instances is regarded as the same no matter which pattern instance needs to be recovered.

Since there are not many quantitative results reviewed in the literature, we just compare our results with that of *SSA* (Tsantalis et al., 2006), DeMIMA (Guéhenéuc and Antoniol, 2008) and *Sempatrec* (Alnusair et al., 2014). As Table 7 shows, our approach recovers more true instances than *DeMIMA*, *SSA* and *Sempatrec* do. In other words, our approach wins the precision. As for the recall, *DeMIMA* achieves the highest one when identifying design pattern instances, or strictly design motif according to Guéhenéuc and Antoniol (2008). However, our approach achieves the highest F-measure if considering both precision and recall.

We also found that our approach missed some pattern instances, which *Percerons* claimed to recover, due to some controversial variants of design patterns. For example, six Facade pattern instances recovered by *Percerons* in *Barcode4J* actually lack the roles of *ConFacade*. However, we argue that there exists the class of *ConFacade* that extends *Facade*. Another example is the *Template* pattern. In *Barcode4J*, we recovered one instance of *Template* pattern, and missed the other that *Percerons* claimed to recover. However, we found the missed one contains only one concrete subclass of *AbstractClass*. Actually, we consider that *AbstractClass* should have at least two concrete subclasses, i.e., *ConClassA* and *ConClassB*. The same happens on the case of the *Mediator* pattern. In fact, the pattern instances are recovered based on the definition of the pattern itself which might be varied under different circumstances. In this paper, we introduce the classical *GoF* definitions of creational, structural and behavioral patterns. Obviously, a more strict definition may lead to less recovered instances and vice versa. Moreover, we just take the recovered instances in *Percerons* as the oracle when calculating the recall, although the recovered instances in *Percerons* are not always the true ones, or at least some of them are controversial.

In addition, our approach is open to the new variants, which can be easily incorporated into the *Design Pattern Instances Detecting Tool* through the following two steps: (1) determine the sub-patterns that constitute the new variant, and (2) determine its feature templates of method signature. Because the sub-patterns and feature templates can be configured in the configuration file in the tool, the new variant can be resolved without changing the tool itself.

5. Threats to validity

Construct validity refers to the validity of inferences that observations or measurement tools actually represent or measure the construct being investigated. In the context of our study, this is mainly due to how the precision and recall are measured. Since the numbers of recovered pattern instances are generally no more than several dozens, we had one master student to manually check if the recovered ones are true positives. However, we cannot exclude that such manual analysis could have missed some false positives. In addition, we took the recovered instances in *Percerons* as the oracle when calculating the recall. Although the recovered instances in *Percerons* are the samples of all true instances, they are of course not equal to the all.

Threats to internal validity concern factors that could have influenced the results. In our study, this is mainly due to the variants of defined *Structural Feature Models* and *Method Signature Templates* for different patterns. In fact, the pattern instances are recovered based on the definition of the pattern itself which might be varied under different circumstances. In this paper, we introduce the classical *GoF* definitions of creational, structural and behavioral patterns. Obviously, a more strict definition may lead to less recovered instances and vice versa.

Threats to external validity concern the generalization of the results. Our approach only deals with 23 *GoF* patterns, while there might be many more left uncovered. In addition, we conducted the evaluation on nine Java projects. It could be worthwhile to replicate the evaluation on other projects having different language constructs (e.g., systems programmed in C++, which allow multi-inheritance).

Table 6
The experimental results for recovering 23 *GoF* pattern instances from nine open source software systems.

		Structural pattern							Creational pattern					Behavioral pattern										
		ADPT	PRXY	DCRT	CMPT	BRGE	FLWT	FCD	AF	BLD	FM	PRTT	SGLT	COR	CMD	ITPT	ITRT	MDT	MMT	OBSV	STT	STTG	TPLT	VSTR
JA	SA	348	727	49	104	155	45	179	194	94	33	0	70	58	85	30	11	211	23	190	134	134	273	8
	BA	46	52	20	4	7	9	49	7	1	22	0	65	5	13	2	0	8	2	21	0	0	17	5
	P	100%	100%	100%	100%	100%	89%	100%	100%	100%	100%	N/A	100%	100%	100%	100%	N/A	100%	100%	100%	N/A	N/A	100%	100%
	R	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
	F	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
JD	SA	31	75	27	27	0	3	21	37	12	34	2	3	0	11	1	0	51	2	51	4	4	116	2
	BA	4	1	1	1	0	1	12	0	0	11	1	3	0	9	0	0	0	0	3	0	3	9	2
	P	100%	100%	100%	100%	N/A	100%	100%	N/A	N/A	100%	100%	100%	N/A	100%	N/A	N/A	N/A	N/A	100%	100%	100%	100%	100%
	R	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	F	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
JT	SA	19	12	4	4	0	0	11	0	0	2	0	0	0	4	1	0	2	0	12	0	0	1	0
	BA	9	0	2	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	1	0	0	1	0
	P	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A	N/A	100%	N/A
	R	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	50%	N/A	N/A	100%	N/A
	F	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	N/A	67%	N/A	N/A	100%	N/A
DJ	SA	51	74	9	51	38	7	54	0	44	41	13	20	40	14	5	1	75	7	113	30	30	35	8
	BA	14	6	2	6	3	1	21	0	3	12	3	12	1	4	0	0	3	1	26	0	12	6	4
	P	100%	100%	100%	100%	100%	100%	100%	N/A	100%	100%	100%	100%	100%	100%	N/A	N/A	100%	100%	100%	N/A	100%	100%	100%
	R	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
	F	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
LZ	SA	221	254	3	20	15	14	63	9	30	29	0	8	3	286	29	0	36	9	16	13	13	30	7
	BA	6	28	3	3	3	1	20	0	0	1	0	6	3	6	0	0	0	0	0	0	2	8	3
	P	100%	100%	100%	100%	100%	100%	100%	N/A	N/A	100%	N/A	67%	100%	100%	N/A	N/A	N/A	N/A	N/A	N/A	100%	100%	33%
	R	67%	N/A	N/A	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	100%	100%	67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	100%
	F	80%	N/A	N/A	100%	N/A	100%	N/A	N/A	N/A	N/A	N/A	80%	100%	80%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	50%
HDK	SA	49	49	0	0	1	15	17	0	42	0	1	17	0	57	0	0	0	2	3	5	5	100	0
	BA	2	13	0	0	0	0	3	0	14	0	1	17	0	4	0	0	0	0	3	0	3	3	0
	P	100%	100%	N/A	N/A	N/A	N/A	67%	N/A	100%	N/A	100%	71%	N/A	50%	N/A	N/A	N/A	N/A	67%	N/A	33%	100%	N/A
	R	100%	N/A	N/A	N/A	N/A	N/A	100%	N/A	N/A	N/A	0%	100%	N/A	100%	N/A	N/A	N/A	N/A	100%	N/A	50%	100%	N/A
	F	100%	N/A	N/A	N/A	N/A	N/A	80%	N/A	N/A	N/A	50%	83%	N/A	67%	N/A	N/A	N/A	N/A	80%	N/A	40%	100%	N/A
BC	SA	65	65	0	3	24	14	24	96	29	20	0	10	0	109	1	0	13	3	2	13	13	101	0
	BA	5	5	0	0	0	0	13	12	7	4	0	10	0	3	0	0	7	0	2	0	4	7	0
	P	100%	100%	N/A	N/A	N/A	N/A	77%	100%	100%	100%	N/A	80%	N/A	100%	N/A	N/A	100%	N/A	100%	N/A	33%	100%	0%
	R	100%	100%	N/A	N/A	N/A	N/A	29%	100%	N/A	100%	N/A	100%	N/A	100%	N/A	N/A	18%	N/A	N/A	N/A	100%	50%	0%
	F	100%	100%	N/A	N/A	N/A	N/A	41%	100%	N/A	100%	N/A	89%	N/A	100%	N/A	N/A	31%	N/A	N/A	N/A	50%	67%	0%
RP	SA	91	103	0	6	20	38	7	231	48	14	13	20	0	213	0	2	8	4	17	20	20	264	16
	BA	17	15	0	0	2	3	5	3	0	2	13	19	0	2	0	2	1	3	1	0	0	13	2
	P	100%	100%	N/A	N/A	100%	100%	20%	100%	N/A	100%	100%	11%	N/A	100%	N/A	100%	100%	100%	100%	N/A	N/A	100%	100%
	R	100%	100%	N/A	N/A	N/A	N/A	100%	100%	N/A	100%	100%	100%	N/A	22%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
	F	100%	100%	N/A	N/A	N/A	N/A	33%	100%	N/A	100%	100%	20%	N/A	36%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	100%	N/A
TC	SA	81	82	4	9	13	13	17	195	11	0	0	16	0	9	0	0	7	2	5	12	12	134	0
	BA	11	1	4	0	0	0	5	0	0	0	0	15	0	0	0	0	0	0	0	0	4	5	0
	P	100%	100%	100%	N/A	N/A	N/A	40%	N/A	N/A	N/A	N/A	80%	N/A	0%	N/A	N/A	N/A	N/A	N/A	N/A	25%	100%	N/A
	R	100%	100%	100%	N/A	N/A	N/A	100%	N/A	N/A	N/A	N/A	100%	N/A	0%	N/A	N/A	N/A	N/A	N/A	N/A	100%	100%	N/A
	F	100%	100%	100%	N/A	N/A	N/A	57%	N/A	N/A	N/A	N/A	89%	N/A	0%	N/A	N/A	N/A	N/A	N/A	N/A	40%	100%	N/A
Total	SA	956	1441	96	224	266	149	393	762	310	173	30	163	101	788	67	14	403	52	409	231	231	1054	41
	BA	114	121	32	14	15	15	128	22	25	54	18	147	9	41	2	2	19	6	57	0	28	69	16
	P	100%	100%	100%	100%	100%	100%	76%	100%	100%	100%	100%	88%	100%	81%	100%	100%	100%	100%	84%	N/A	58%	100%	72%
	R	93%	100%	100%	100%	N/A	100%	76%	100%	N/A	100%	50%	100%	100%	58%	N/A	N/A	18%	N/A	75%	N/A	83%	94%	50%
	F	96%	100%	100%	100%	N/A	100%	76%	100%	N/A	100%	67%	94%	100%	63%	N/A	N/A	31%	N/A	79%	N/A	68%	97%	59%

Note: ADPT=Adapter, PRXY=Proxy, DCRT=Decorator, CMPT=Composite, BRGE=Bridge, FLWT=Flyweight, FCD=Facade, AF=Abstract Factory, BLD=Builder, FM=Factory Method, PRTT=Prototype, SGLT=Singleton, COR=Chain Of Responsibility, CMD=Command, ITPT=Interpreter, ITRT=Iterator, MDT=Mediator, MMT=Memento, OBSV=Observer, STT=State, STTG=Strategy, TPLT=Template Method, VSTR=Visitor.
SA: (After) Structural Analysis, BA: (After) Behavioral Analysis, P: Precision, R=Recall, F: F-measure.
JA=JavaAWT, JD=JHotDraw, JT=JUnit, DJ=Dom4j, LZ=Lizzy, HDK=Hodoku, BC=Barcode4J, RP=Rtsp, TC=Teamcenter.
N/A: not acceptable due to no instances recovered by either *Percerons* or our approach.

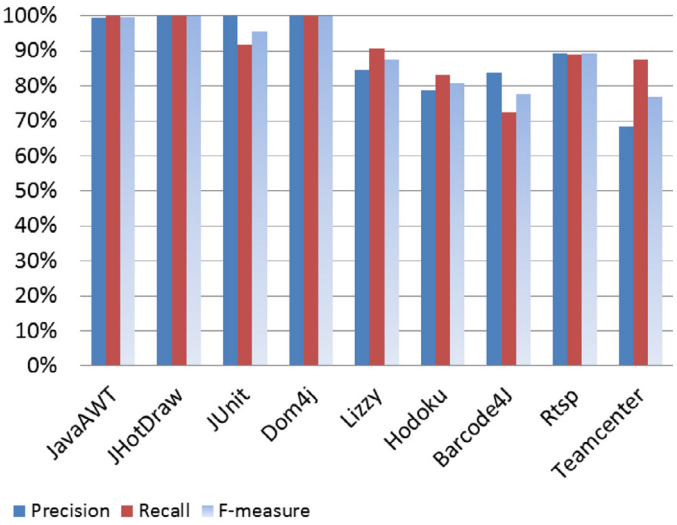


Fig. 6. The comparison of total precision, recall and F-measure for nine open source systems.

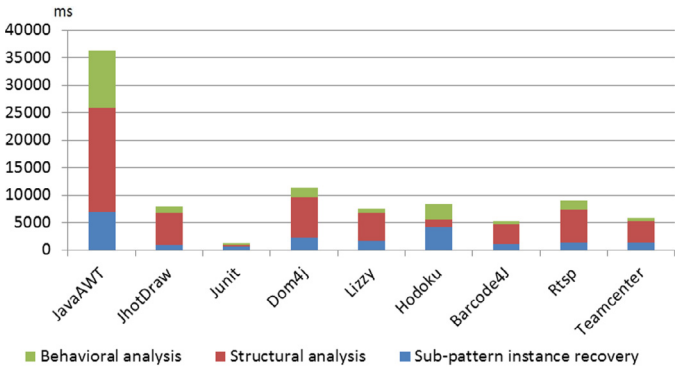


Fig. 7. The execution time for sub-pattern instance recovery, structural analysis and behavioral analysis(method signature analysis) for nine open source systems in milliseconds.

6. Related work

In recent years, *GoF* design patterns have attracted the attention of researchers and they are now considered a respectable part of software engineering research and practice. As Apostolos Ampatzoglou suggests, the most popular subtopics of design pattern research is design pattern detection (Ampatzoglou et al., 2013a). There have been a number of approaches for recovering design patterns presented in the literature. The following outlines the previous works and also gives the comparison between our work and the previous ones.

6.1. Structural analysis

Since most of design patterns have their unique structural characteristics, nearly all approaches focus on the structural aspects of source codes, including classes, attributes, methods and the relationships between classes, such as generalization, association and aggregation. Rasool and Mäder (2011) propose variable pattern definitions composed of reusable feature types. Each feature type is assigned to one of the multiple search techniques that is best fitting for its detection. On the other hand, Fontana et al. introduce four types of microtructures, i.e., elemental design patterns, design pattern clues, micro patterns and sub-patterns, which are regarded as the building blocks of design patterns (Fontana et al., 2011a, 2011b). They evaluate whether the detection of these building blocks is relevant for the detection of occurrences of the design patterns. They also find that the detection of some design patterns can be performed through the detection of a combined set of micro-structures. Niere et al. (2002) define patterns and sub-patterns with respect to the abstract syntax graph (ASG) representation of a program, which can be produced by the JavaCC source code parser (JCC). Besides, Posnett et al. (2010) introduce four kinds of meta-patterns rooted in two structural roles: TEMPLATE and HOOK. Meanwhile, Dong and Zhao (2007) categorize different characteristics of each design pattern as its traits in form of predicates. They also classify different predicates into groups and levels. In this way, the significant characteristics of each design pattern are explicitly specified in predicates that can be used for design pattern recovery and evolution analysis.

Although many concepts such as clues, elemental design patterns, feature types, micro-patterns, meta-patterns, traits and even sub-patterns are proposed to describe the constitutes of different design patterns, the concept of sub-pattern presented here is yet different to the above mentioned ones. Clues (MARPLE Group, 2009) and micro patterns (Gil and Maman, 2005) are defined starting directly from source code, and considering various detail levels. In the work of Niere et al. (2002), patterns and sub-patterns describe the detail of inner structure of a class including the behavioral characteristics based on *Abstract Syntax Graph*. In contrast, the sub-patterns defined here are starting from design patterns by considering their shared parts, and thus closer to the objectives of design pattern detection. In addition, they also reveal the relationships among classes. Meanwhile, *Elemental Design Patterns* (Smith, 2002) place themselves on a coarser-grained level of detail concentrating on the behavioral aspects, and provide too general hints for design pattern detection. On the other hand, the sub-patterns, given by Fontana et al. (2011a), are defined on top of some others and therefore would become too complicated to deal with. Concerning metapatterns introduced by Posnett et al. (2010), the *GoF* design patterns are not always the instances of metapatterns or combinations of multiple metapatterns. Therefore, they claim that THEX they present is just a metapattern detector, but not a design pattern detector. Finally, the pattern trait (Dong and Zhao, 2007) and the feature type (Rasool and Mäder, 2011) are quite different to the graph-based sub-pattern presented here. The former specifies the significant characteristics of each design pattern in predicates, whereas the latter characterizes the recurring sub-structures by a search technology and others.

6.2. Behavioral checking

In addition to inspecting the code structures, some approaches also investigated the behavioral aspects. De Lucia et al. (2011) present a pattern recovery approach that analyzes the behavior of pattern instances both statically and dynamically. In particular, their proposed approach exploits model checking to statically verify the behavioral aspects of design pattern instances. Meanwhile, Rasool et al. present a design pattern recovery approach based on annotations, regular expressions and database queries (Rasool et al., 2010). They define the varying features of patterns and apply rules to match these features with the source code elements. The novelty of this approach lies in the introduction of appropriate semantics of annotations from large legacy systems, which reduces the search space and time for detecting pattern instances. De Lucia et al. (2009) employ the visual language to describe the relationship among classes. Particularly, they introduce checks to validate the role of classes participating to the pattern instances and also verify method declaration. The templates for method signatures we summarize in our paper are somewhat similar to the visual sentences in De Lucia et al. (2009). However, they are clearer and thus easier to be updated for the variants simply because they focus only on the method signatures for behavioral checking.

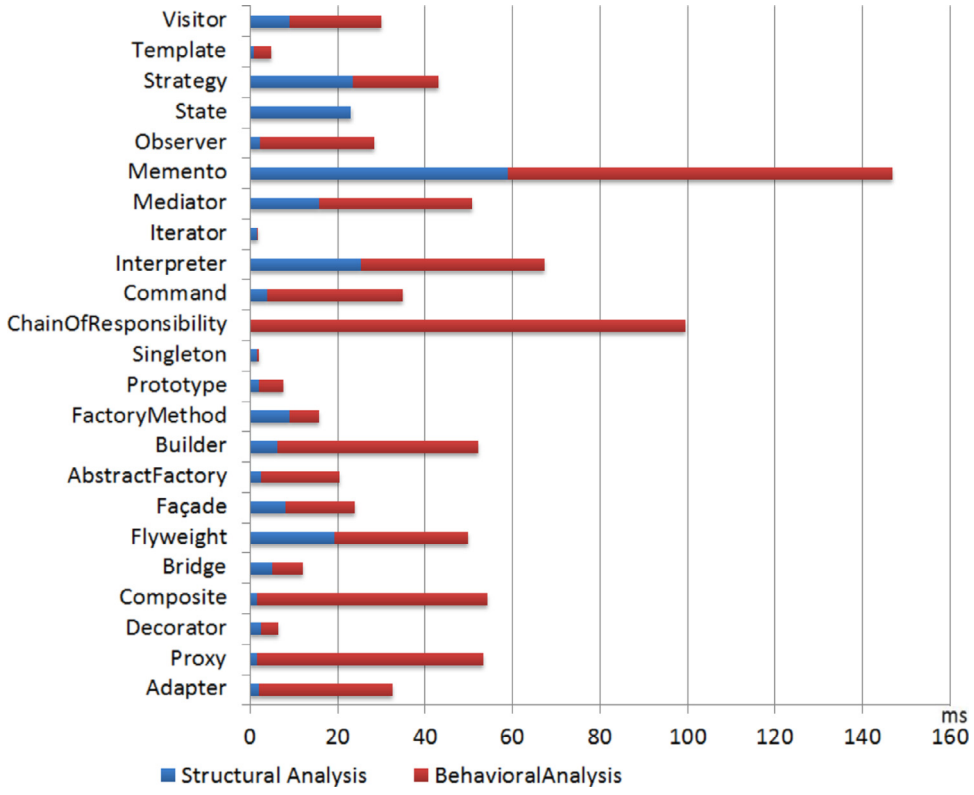


Fig. 8. The execution time for structural analysis and behavioral analysis (method signature analysis) for recovering 23 *GoF* pattern instances in milliseconds. The execution time for sub-pattern instance recovery is omitted since they are recovered together.

Table 7
Comparison of the results of our approach and that of other approaches.

		Our approach			DeMIMA (Guéhenéuc and Antoniol, 2008)			SSA (Tsantalis et al., 2006)			Sempatrec (Alnusair et al., 2014)		
		P	R	F	P	R	F	P	R	F	P	R	F
ADPT	JD	100%			4%	100%	8%	44%	100%	61%	45%		
	JT	100%	100%	100%	0%			17%	100%	29%	100%	100%	100%
DCRT	JD	100%			8%	100%	15%	33%	33%	33%	50%	33%	40%
	JT	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
CMPT	JD	100%			33%	100%	50%	100%	100%	100%	100%	100%	100%
	JT	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
FM	JD	100%			2%	100%	4%	100%	67%	80%	100%	100%	100%
	JT	100%	100%	100%		100%							
SGLT	JD	100%			100%	100%	100%	100%	100%	100%	100%	100%	100%
	JT	100%	100%	100%									
OBSV	JD	100%			25%	100%	40%	50%	40%	44%	50%	40%	44%
	JT	100%	50%	67%	25%	100%	40%	100%	100%	100%	100%	100%	100%
TPLT	JD	100%	100%	100%	7%	100%	13%	20%	100%	33%	50%	100%	67%
	JT	100%	100%	100%	0%			100%	100%	100%	100%	100%	100%
VSTR	JD	100%				100%		100%	100%	100%			
	JT					100%							
Total		100%	94%	96%	30%	100%	40%	78%	89%	79%	79%	77%	86%

Note: ADPT=Adapter, DCRT=Decorator, CMPT=Composite, FM=FactoryMethod, SGLT=Singleton, OBSV=Observer TPLT=TemplateMethod, VSTR=Visitor. JD: JHotDraw, JT: JUnit, P: Precision R: Recall, F: F-measure.
We only compare the results that DeMIMA, SSA or Sempatrec revealed.

6.3. Intermediate expressions

One of the challenges comes from the intermediate expression for the source codes in order to detect the design pattern instances efficiently. Possible solutions include predicates, matrices, vectors, ontology and so on. Heuzeroth et al. specify the static and dynamic aspects of patterns as predicates, and represent legacy codes by predicates that encode their attributed abstract syntax trees (Heuzeroth et al., 2003b). Dong et al. present an approach to the recovery of design patterns based on matrices and weights. They encode both the systems and the design patterns into matrices and weights. The formal specification rigorously defines the structural, behavioral and semantic analyses of their approach (Dong et al., 2009b). Kaczor et al. express the problem of design pattern identification with operations on finite sets of bit-vectors. They use the inherent parallelism of bit-wise operations to derive an efficient bit-vector algorithm that finds exact and approximate occurrences of design patterns in a program (Kaczor et al., 2006). Besides, Alnusair et al. use ontology formalism to represent the conceptual knowledge of the source code and semantic rules to capture the structures and behaviors of the design patterns in the libraries (Alnusair et al., 2014). Graphs and sub-graphs are another frequently used expression for source codes. For instance, Niere

et al. exploit context knowledge given by a special form of an annotated abstract syntax graph to overcome the scalability problems caused by the variants of design pattern instances (Niere et al., 2002). Pande et al. apply graph decomposition and graph isomorphism techniques for design pattern detection (Pande et al., 2010). Tsantalis et al. calculate the similarities between two vertices for pattern detection (Tsantalis et al., 2006). Qiu and Jiang introduce a state space graph to avoid the search space explosion and reduce the opportunity of detecting subgraph isomorphism (Qiu and Jiang, 2010). Other representations include modeling language (Elaasar et al., 2013) and XML (Bouassida and Ben-Abdallah, 2010).

Our approach employs mainly the graph to illustrate the relationship of different classes. We transform the system, design pattern instances and sub-pattern instances into the graphs which are represented by matrix. Different from previous works based on graph, we reduce the search space by considering the inbound and outbound composite weights during graph matching, and determine the roles of classes by considering the joint classes during graph merging. The performance can be therefore improved to the satisfactory level as Figs. 7 and 8 indicate.

6.4. Variants and performance

The identification of modified pattern versions also brings about challenges for pattern detection and performance. Tsantalis et al. (2006) propose a design pattern detection methodology based on similarity scoring between graph vertices. It has the ability to recognize patterns that are modified from their standard representation. Meanwhile, Ferenc et al. adopt machine learning to enhance pattern mining by filtering out as many false hits as possible. They distinguish similar design patterns such as state and strategy with the help of a learning database created by manually tagging a large C++ system (Ferenc et al., 2005).

Maurice and Vassilios, on the other hand, introduce an approach that complements existing detection methods by utilizing finely grained static information contained in the software system (Maurice and Vassilios, 2012). It filters a large number of false positives by utilizing finely grained rules that describe the static structure of a design pattern. *DeMIMA*, presented in Guéhenéuc and Antoniol (2008), consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify design patterns in the abstract model. Through the use of explanation-based constraint programming, *DeMIMA* ensures 100% recall on five systems. Romano et al. (2011) propose an approach that leverages lexical information and fuzzy clustering to reduce the number of the false positives while preserving those correctly identified.

In our approach, we consider the cases of variants in the *Structural Feature Models* as Table 8 in the Appendix indicates. For example, we consider two variants for *Composite*, one variant for *Proxy*, one variant for *Decorator*, one variant for *Prototype*, etc. Moreover, the definition of both *Structural Feature Models* and *Templates of Method Signatures* can be also updated to adapt to new variants. However, we think that the variants may vary from software to software and many of them are still not generally recognized.

6.5. Toolkits and experiments

Many toolkits have been developed to support the detection of pattern instances. Khatoon et al. propose a framework that extracts a large variety of pattern instances from source codes and finds locations where the extracted patterns are violated (Khatoon et al., 2011). It also helps in code reusing by suggesting to the programmer how to write API code to facilitate rapid software development. *DPJF*, presented by Alexander and Günter (2012), achieves high detection quality through a well-balanced combination of structural and behavioral analysis techniques. *PINOT*, developed by Shi and Olsson, uses lightweight static program analysis techniques to capture program intent (Shi and Olsson, 2006). *PINOT* detects all the *GoF* patterns that have concrete definitions driven by code structure or system behavior. Fontana and Zanoni (2011) propose an Eclipse plug-in called *MARPLE*, which supports both the detection of design patterns and software architecture reconstruction activities through the use of basic elements and metrics that are mechanically extracted from the source code. Another Eclipse plug-in called *ePAD*, presented in De Lucia et al. (2010), is able to recover design pattern instances through a structural analysis performed on a data model extracted from source code, and a behavioral analysis performed through the instrumentation and the monitoring of the software system. Other notable toolkits include *DP-Miner* (Dong et al., 2009b) , *FUJABA* (Niere et al., 2002), and *Columbus* (Ferenc et al., 2005).

6.6. Benchmarks

Currently, there are several different methods used in the community to evaluate accuracy and recall. Java-based open source systems, such as *JHotDraw*, *JUnit* and *JavaAWT*, are widely used as the benchmarks. Pettersson et al. propose a benchmark suite and also a set of fine-grained metrics to ensure the comparability of various approaches (Pettersson et al., 2010). Ampatzoglou et al. introduce a web repository of design patterns instances, called *Percerons*, which has been used in open source projects. Currently, over 100 open source projects have been considered and more than 4000 pattern instances have been found and recorded in the database of the repository (Ampatzoglou et al., 2013b). Since *Percerons* shows a number of recovered pattern instances in detail, we use it as the oracle in our experiment. However, generally speaking, there are still no standard benchmarks available that facilitate the comparison of pattern detectors.

7. Conclusions

Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Detecting instances of design patterns from source codes can help to understand and trace back to the original design decisions and reengineer the systems.

In this paper, we present a new approach to the detection of instances of design patterns from source codes. We first transform the source codes and the predefined patterns into the *Class-Relationship Directed Graphs*. We then identify instances of sub-patterns that would be the possible constituents of pattern instances based on subgraph isomorphism. Afterward, the identified sub-pattern instances are combined by joint classes to see if the collective matches with one of the predefined patterns. The method signatures of those matched ones are further compared with the predefined templates of the standard patterns. In this way, we can recover instances of all *GoF* design patterns with the balanced high precision and recall. As a comparison, our previous approach, presented in Yu et al. (2013b) and Yu et al. (2013a), can only recover the structural pattern instances. Moreover, it does not investigate the method signature, which therefore leads to a number of false instances.

The contributions of this paper are summarized as follows. (1) We integrate and improve some of the previous ideas and put forward a comprehensive and elaborative approach also based on our own ideas. In contrast, many of previous works focus on just one type of design patterns, or even some specific patterns. Meanwhile, their recovering processes are sometimes lack of the accurate and elaborative explanation. Many just present some illustrative examples. (2) We present 15 sub-patterns that directly originate from the concept of design pattern and thus facilitate pattern recovering. Meanwhile, we detect sub-patterns via graph isomorphism based on prime number composition

and the joint classes, and employ the method signatures to investigate the behavioral features. The former reduces the search space, whereas the latter avoids choosing the test cases with full code coverage. (3) We conducted the extensive experiments which cover nine open source systems and 23 *GoF* design patterns. To the best of our knowledge, the result showed in this paper is the most exhaustive one.

For our approach, much time is spent on graph mining and matching for recovering pattern instances. In future, we plan to filter out certain sub-pattern instances before merging in order to accelerate the execution speed of the approach when dealing with software systems of large scales. On the other hand, we will introduce the technique of model checking to determine automatically if the recovered instances are true positives because current validation process still requires some manpower. Last but not least, we will adapt our approach to more pattern variants although variants may vary from software to software and many of them are still controversial and not generally recognized.

Acknowledgments

The work is supported by National Natural Science Foundation of China (no. 61100043), Natural Science Foundation of Zhejiang Province (no. LY12F02003), the Key Science and Technology Project of Zhejiang (no. 2012C11026-3). The authors would also like to thank anonymous reviewers and the colleagues in UC Santa Barbara who made valuable suggestions to improve the quality of the paper.

Appendix

Table 8
The definitions of *Structural Feature Models* that correspond to 23 *GoF* design patterns.

Type	Patterns	Structural Feature Models
Structural Design Patterns	Adapter	$SFM_{Adapter} = ICA\&\&(!CI) = (< Target, Adapter, Adaptee >, R_{ICA})\&\&!(< Target, Adapter, Adaptee >, R_{CI})$ $jClassSet = \{Target, Adapter, Adaptee\}$
	Bridge	$SFM_{Bridge} = CI\&\&IPAG = (< Implementor, ConImplementorA, ConImplementorB >, R_{CI})$ $\&\&(< Abstraction, RefinedAbstraction, Implementor >, R_{IPAG})\ jClassSet = \{Implementor\}$
	Composite	$SFM_{Composite} = (CI\&\&IAGG) \parallel SAGG \parallel (CI\&\&IIAGG) = (< Comp, ConComp, Composite >, R_{CI})$ $\&\&(< Comp, Composite >, R_{IAGG}) \parallel (< Comp >, R_{SAGG}) \parallel (< Comp, ConComp, Composite >, R_{CI})$ $\&\&(< Comp, Composite, Composite1 >, R_{IIAGG})\ jClassSet = \{(Comp, Composite), Comp\}$
	Proxy	$SFM_{Proxy} = (CI\&\&ICA) \parallel (CI\&\&IASS) = (< Subject, RealSubject, Proxy >, R_{CI})$ $\&\&(< Subject, RealSubject, Proxy, R_{ICA} >) \parallel (< Subject, RealSubject, Proxy >, R_{CI})\ \&\&(< Subject, RealSubject, Proxy >, R_{IASS})$ $jClassSet = \{Subject, Proxy\}$
	Decorator	$SFM_{Decorator} = (CI\&\&IAGG) \parallel (CI\&\&IAGG\&\&MLI) = (< Comp, ConComp, Decorator >, R_{CI})$ $\&\&(< Comp, Decorator >, R_{IAGG}) \parallel (< Comp, ConComp, Decorator >, R_{CI})\&\&$ $(< Comp, Decorator >, R_{IAGG})\&\&(< Comp, Decorator, ConDecorator >, R_{MLI})\ jClassSet = \{Comp, Decorator\}$
	Flyweight	$SFM_{Flyweight} = AGPI\&\&CI = (< FlyweightFactory, Flyweight, ConFlyweight >, R_{AGPI})$ $\&\&(< Flyweight, ConFlyweight, UnsharedConFlyweight >, R_{CI})\ jClassSet = \{Flyweight, ConFlyweight\}$
	Facade	$SFM_{Facade} = ICD = (< Facade, ConFacade, Subsystem >, R_{ICD})\ jClassSet = \{Facade, ConFacade\}$
Creational Design Patterns	Abstract Factory	$SFM_{AbstractFactory} = ICD\&\&CI\&\&DCI = (< AbstractFactory, ConFactory, ConProductA >, R_{ICD})$ $\&\&(< AbstractProduct, ConProductA, ConProductB >, R_{CI})\ \&\&(< ConFactory, AbstractProduct, ConProductA >, R_{DCI})$ $\|\|(< ConFactory, AbstractProduct, ConProductB >, R_{DCI})$ $jClassSet = \{(ConFactory, AbstractProduct, ConProductA)\|\|(ConFactory, AbstractProduct, ConProductB)\}$
	Builder	$SFM_{Builder} = AGPI\&\&ICA = (< Director, Builder, ConBuilder >, R_{AGPI})\&\& (< Bulider, ConBuilder, Product >, R_{ICA})$ $jClassSet = \{Builder, ConBuilder\}$
	Factory Method	$SFM_{FactoryMethod} = ICD\&\&DCI = (< Creator, ConCreator, ConProduct >, R_{ICD})$ $\&\&(< ConCreator, ConProduct, Product >, R_{DCI})\ jClassSet = \{ConCreator, ConProduct\}$
	Prototype	$SFM_{Prototype} = (CI\&\&AGPI) \parallel AGPI = (< Prototype, ConPrototypeA, ConPrototypeB >, R_{CI})$ $\&\&(< Client, Prototype, ConPrototypeA >, R_{AGPI}) \parallel (< Clent, Prototype, ConPrototypeB >, R_{AGPI})$ $jClassSet = \{(Prototype, ConPrototypeA)\|\|(Prototype, ConPrototypeB)\}$
	Singleton	$SFM_{Singleton} = SASS = (< Singleton, Singleton >, R_{SASS})\ jClassSet = \{Singleton\}$
Behavioral Design Patterns	Chain of Responsibility	$SFM_{ChainofResponsibility} = SASS\&\&CI = (< Handler >, R_{SAS})\&\&(< Handler, ConHandlerA, ConHandlerB >, R_{CI})$ $jClassSet = \{Handler\}$
	Command	$SFM_{Command} = AGPI\&\&ICA = (< Invoker, Command, ConCommand >, R_{AGPI})\ \&\&(< Command, ConCommand, Receiver >, R_{ICA})$ $jClassSet = \{Command, ConCommand\}$
	Interpreter	$SFM_{Interpreter} = CI\&\&IAGG\&\&IPD = (< AbstractExpression, TerminalExpression, NonterminalExpression >, R_{CI})$ $\&\&(< AbstractExpression, NonterminalExpression >, R_{IAGG})\ \&\&(AbstractExpression, TerminalExpression, Context >, R_{IPD})$ $jCllassSet = \{AbstractExpression\}$
	Iterator	$SFM_{Iterator} = (ICA\&\&ICD) \parallel (ICA\&\&DCI) = (< Iterator, ConIterator, ConAggregate >, R_{ICA})$ $\&\&(< Aggregate, ConAggregate, ConIterator >, R_{ICD}) \parallel$ $(< Iterator, ConIterator, ConAggregate >, R_{ICA})\&\&(< Iterator, ConIterator, ConAggregate >, R_{DCI})$ $jClassSet = \{ConIterator, ConAggregation\}$
	Mediator	$SFM_{Mediator} = CI\&\&IPAS\&\&ICA = (< Colleague, ConColleagueA, ConColleagueB >, R_{CI})$ $\&\&(< Colleague, ConColleagueA, Mediator >, R_{IPAS})\|\|(< Colleague, ConColleagueB, Mediator >, R_{IPAS})$ $\&\&(< Mediator, ConMediator, ConColleagueA >, R_{ICA})\|\|(< Mediator, ConMediator, ConColleagueB >, R_{ICA})$ $jClassSet = \{Mediator, Colleague\}$
	Memento	$SFM_{Memento} = AGPI\&\&DPI = (< Cakertaker, Memento, MementoImp >, R_{AGPI})$ $\&\&(< Memento, MementoImp, Originator >, R_{DPI})\ jClassSet = \{Memeto, MementoImp\}$
	Observer	$SFM_{Observer} = AGPI\&\&ICD = (< Subject, Observer, ConObserver >), R_{AGPI})\ \&\&(< Observer, ConObserver, ConSubject >, R_{ICD})$ $jClassSet = \{Observer, ConObserver\}$
	State	$SFM_{State} = AGPI\&\&CI = (< Context, State, ConStateA >, R_{AGPI})\|\|(< Context, State, ConStateB >, R_{AGPI})$ $\&\&(< State, ConStateA, ConStateB >, R_{CI})\ jClassSet = \{(State, ConStateA)\|\|(State, ConStateB)\}$
	Strategy	$SFM_{Strategy} = AGPI\&\&CI = (< Context, Strategy, ConStrategyA >, R_{AGPI})\|\|(< Context, Strategy, ConStrategyB >, R_{AGPI})$ $\&\&(< Strategy, ConStrategyA, ConStrategyB >, R_{CI})\ jClassSet = \{(Strategy, ConStrategyA)\|\|(Strategy, ConStrategyB)\}$
	Template	$SFM_{TemplateMethod} = CI = (< AbstractClass, ConClassA, ConClassB >, R_{CI})\ jClassSet = \{AbstractClass\}$
	Visitor	$SFM_{Visitor} = AGPI\&\&DPI\&\&ICD = (< ObjectStructure, Element, ConElement >, R_{AGPI})$ $\&\&(< Element, ConElement, Visitor >, R_{DPI})\&\&(< Visitor, ConVisitor, ConElement >, R_{ICD})\ jClassSet = \{Element, Visitor\}$

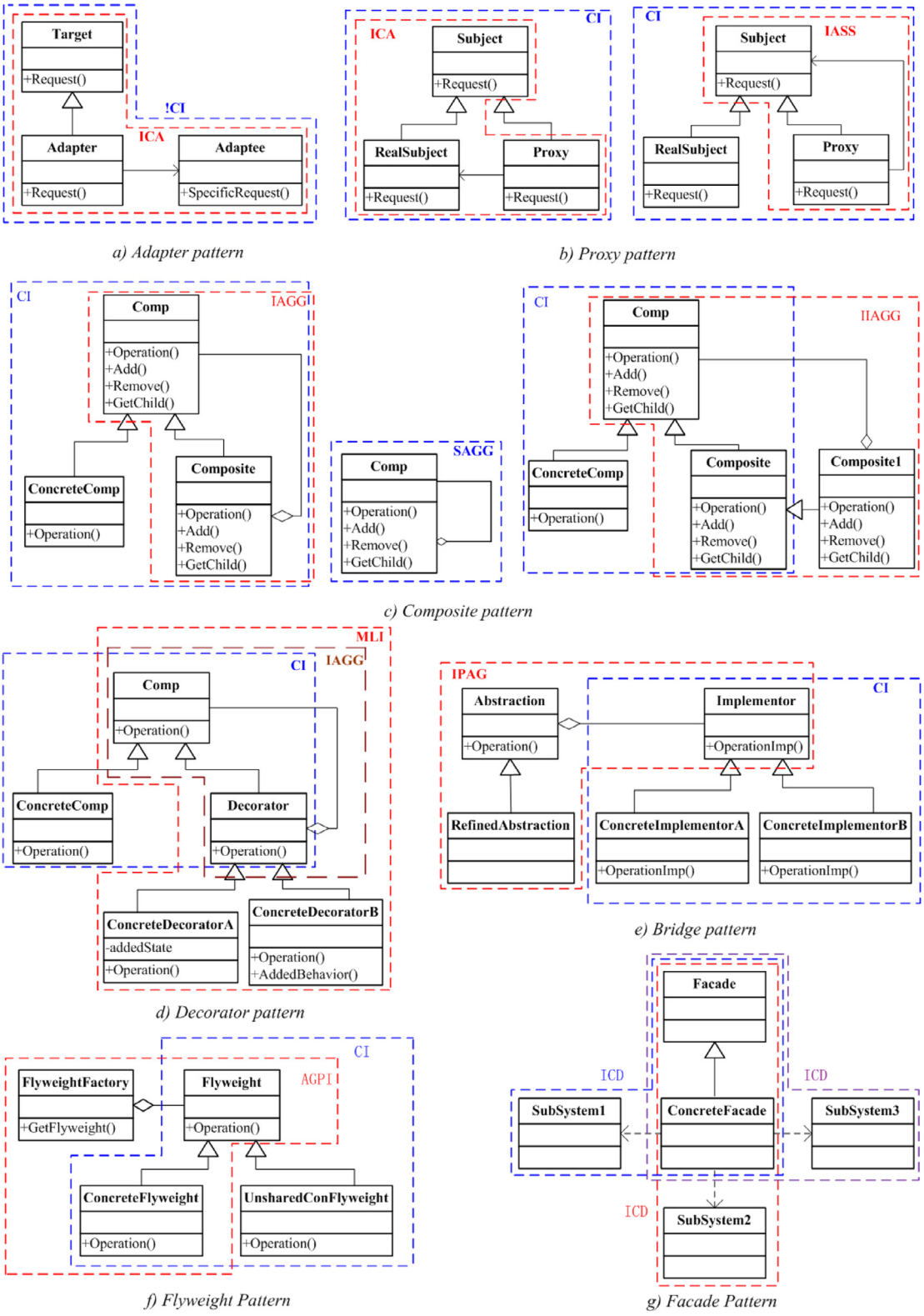


Fig. 9. The graphical *Structural Feature Models* of *GoF* structural design patterns.

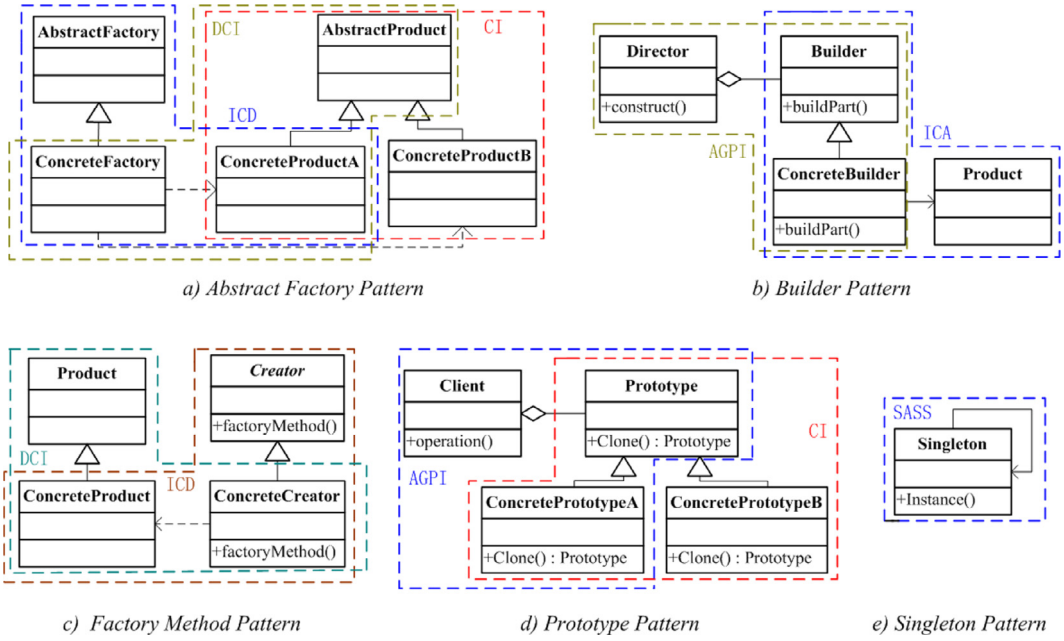


Fig. 10. The graphical *Structural Feature Models* of *GoF* creational design patterns.

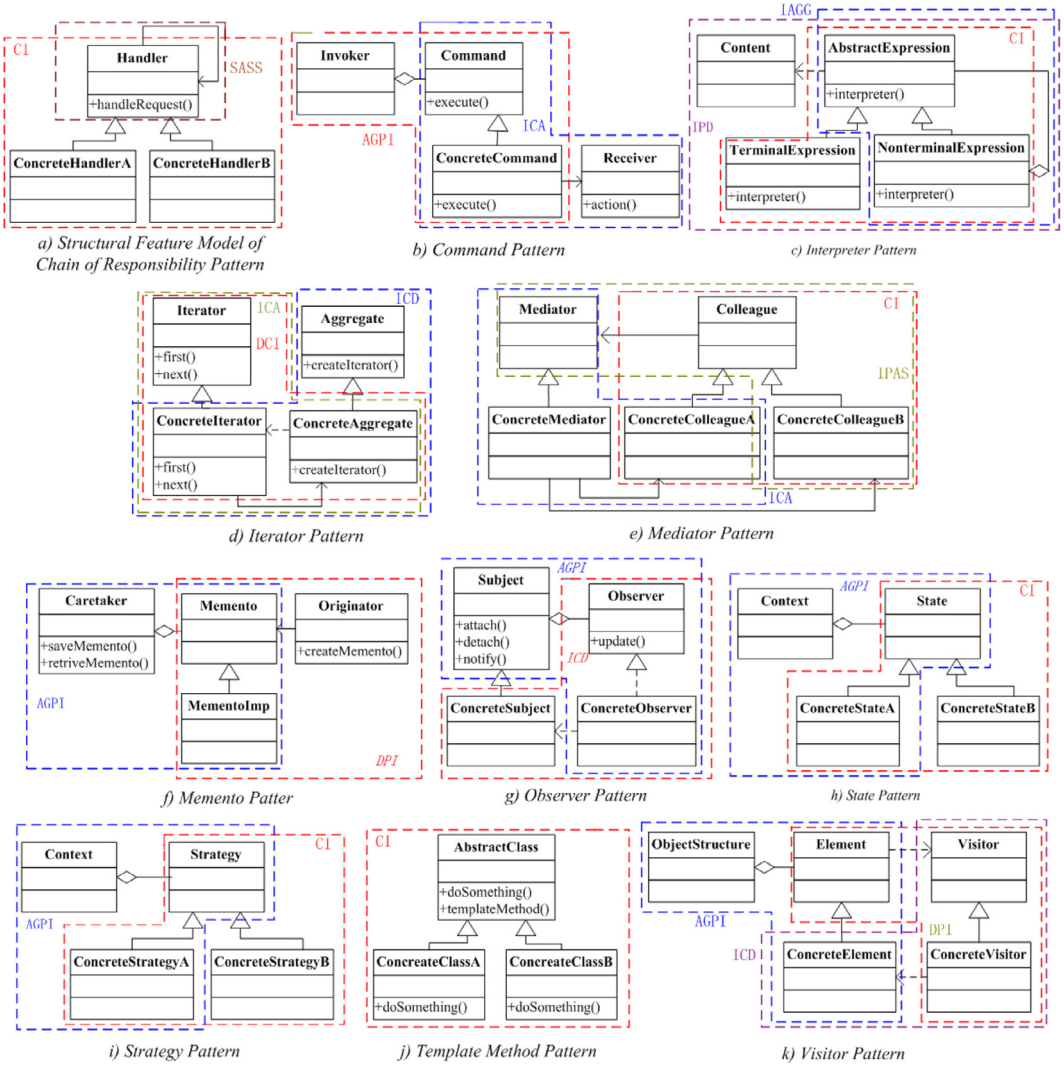


Fig. 11. The graphical *Structural Feature Models* of *GoF* behavioral design patterns.

References

Alexander, B., Günter, K., 2012. DPJF – design pattern detection with high accuracy. In: European Conference on Software Maintenance and Reengineering, pp. 245–254.

Alnusair, A., Zhao, T., Yan, G., 2014. Rule-based detection of design patterns in program code. Int. J. Softw. Tools Technol. Trans. 16 (3), 315–334.

Ampatzoglou, A., Charalampidou, S., Stamelos, I., 2013a. Research state of the art on *GoF* design patterns: a mapping study. J. Syst. Softw. 86 (7), 1945–1964.

Ampatzoglou, A., Michou, O., Stamelos, I., 2013b. Building and mining a repository of design pattern instances: practical and research benefits. Entertainment Comput. 4, 131–142.

Bouassida, N., Ben-Abdallah, H., 2010. A new approach for pattern problem detection. In: Pernici, B. (Ed.), Advanced Information Systems Engineering, CAiSE 2010, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, vol. 6051, pp. 150–164.

De Lucia, A., Deufemia, V., Gravino, C., Risi, M., 2009. Design pattern recovery through visual language parsing and source code analysis. J. Syst. Softw. 82 (7), 1177–1193.

De Lucia, A., Deufemia, V., Gravino, C., Risi, M., 2010. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In: IEEE International Conference on Software Maintenance, ICSM, pp. 1–6.

De Lucia, A., Deufemia, V., Gravino, C., Risi, M., 2011. Improving behavioral design pattern detection through model checking. In: European Conference on Software Maintenance and Reengineering, CSMR, pp. 176–185.

Dong, J., Zhao, Y., 2007. Classification of design pattern traits. In: Proceedings of Nineteenth International Conference on Software Engineering and Knowledge (SEKE), Boston, USA, pp. 473–476.

Dong, J., Zhao, Y., Peng, T., 2009a. A review of design pattern mining techniques. Int. J. Software Eng. Knowl. Eng. 19, 823–855.

Dong, J., Zhao, Y., Sun, Y., 2009b. A matrix-based approach to recovering design patterns. IEEE Trans. Syst. Man Cybern. Part A: Syst. Humans 39 (6), 1271–1282.

Elaasar, M., Briand, L.C., Labiche, Y., 2013. VPML: an approach to detect design patterns of MOF-based modeling languages. Softw. Syst. Model., 1–30.

Ferenc, R., Beszedes, A., Fulop, L., Lele, J., 2005. Design pattern mining enhanced by machine learning. In: 21st IEEE International Conference on Software Maintenance, pp. 295–304.

Fontana, F.A., Maggioni, S., Raibulet, C., 2011a. Design patterns: a survey on their micro-structures. J. Softw. Maint. Evol. 33 (8), 1–25.

Fontana, F.A., Maggioni, S., Raibulet, C., 2011b. Understanding the relevance of micro-structures for design patterns detection. J. Syst. Softw. 84, 2334–2347.

Fontana, F.A., Zanoni, M., 2011. A tool for design pattern detection and software architecture reconstruction. Inf. Sci. 181 (7), 1306–1324.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

Gil, J., Maman, I., 2005. Micro patterns in Java code. In: Proceedings of the 20th Conference on Object-Oriented Programming Systems Languages and Applications, ACM, New York, NY, pp. 97–116.

Guéhenéuc, Y.G., Antoniol, G., 2008. Demima: a multilayered approach for design pattern identification. IEEE Trans. Softw. Eng. 34 (5), 667–684.

Heuzeroth, D., Holl, T., Hogstrom, G., Lowe, W., 2003a. Automatic design pattern detection. In: Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003), pp. 94–103.

Heuzeroth, D., Mandel, S., Lowe, W., 2003b. Generating design pattern detectors from pattern specifications. In: 18th IEEE International Conference on Automated Software Engineering, pp. 245–248.

Kaczor, O., Guéhéneuc, Y., Hamel, S., 2006. Efficient identification of design patterns with bit-vector algorithm. In: 10th European Conference on Software Maintenance and Reengineering, pp. 175–184.

Khatoon, S., Li, G., Ashfaq, R.M., 2011. A framework for automatically mining source code. J. Softw. Eng. 5 (2), 64–77.

MARPLE Group, 2009. A design pattern detection catalog. http://essere.disco.unimib.it/reverse/marple.

Maurice, L., Vassilios, T., 2012. Fine-grained design pattern detection. In: 36th Annual IEEE International Computer Software and Applications Conference, COMPSAC, pp. 267–272.

Niere, J., Schafer, W., Wadsack, J.P., Wendehals, L., Welsh, J., 2002. Towards pattern-based design recovery. In: 24th International Conference on Software Engineering, pp. 338–348.

Pande, A., Gupta, M., Tripathi, A.K., 2010. A new approach for detecting design patterns by graph decomposition and graph isomorphism. Commun. Comput. Inf. Sci. 95, 108–119.

Pettersson, N., Lowe, W., Nivre, J., 2010. Evaluation of accuracy in design pattern occurrence detection. IEEE Trans. Softw. Eng. 36 (4), 575–590.

Posnett, D., Bird, C., Devanbu, P.T., 2010. THEX: mining meta-patterns from Java. In: 7th IEEE Working Conference on Mining Software Repositories, pp. 122–125.

Qiu, M., Jiang, Q., 2010. Detecting design pattern using subgraph discovery. Lect. Notes Comput. Sci. 5990, 350–359.

Rasool, G., Mäder, P., 2011. Flexible design pattern detection based on feature types. In: 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 243–252

Rasool, G., Philippow, I., Mäder, P., 2010. Design pattern recovery based on annotations. Adv. Eng. Software 41, 519–526.

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

Shi, N., Olsson, R., 2006. Reverse engineering of design patterns from java source code. In: 21st IEEE International Conference on Automated Software Engineering, pp. 245–248.

Smith, J.M., 2002. An elemental design pattern catalog (Technical report TR 02-040). University of North Carolina. Chapel Hill, NC.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T., 2006. Design pattern detection using similarity scoring. IEEE Trans. Softw. Eng. 32 (11), 896–909.

Wendehals, L., 2011. Improving design pattern instance recognition by dynamic analysis. In: Proceedings of the ICSE workshop on Dynamic Analysis, pp. 29–32.

Yu, D., Liu, Z., Ge, J., 2013a. Mining Instances of structural design patterns from class diagrams based on sub-patterns. In: 13th International Conference on Software Reuse, ICSR, Lecture Notes in Computer Science, Springer, Berlin, vol. 7925, pp. 255–266.

Yu, D., Zhang, Y., Ge, J., Wu, W., 2013b. From sub-patterns to patterns: an approach to the detection of structural design pattern instances by subgraph mining and merging. In: 37th Annual IEEE International Computer Software and Applications Conference, COMPSAC, pp. 579–588.

Dongjin Yu is currently a professor at Hangzhou Dianzi University and a visiting scholar of University of California, Santa Barbara. He received his BS and MS in Computer Applications from Zhejiang University in China, and PhD in Management from Zhejiang Gongshang University in China. His current research efforts include program comprehension, service computing and cloud computing. He is especially interested in the novel approaches to constructing large enterprise information systems effectively and efficiently by emerging advanced information technologies. He is the vice director of Institute of Intelligent and Software Technology of Hangzhou Dianzi University. He is a member of ACM and IEEE, and a senior member of China Computer Federation (CCF). He is also a member of Technical Committee of Software Engineering CCF (TCSE CCF) and a member of Technical Committee of Service Computing CCF (TCSC CCF).

Yanyan Zhang received her master degree in computer science from Hangzhou Dianzi University in China. She has published a number of high-quality papers related with program comprehension. Her current research interests mainly include program comprehension and design pattern mining.

Zhenli Chen was born in 1991. He is currently a postgraduate at Hangzhou Dianzi University, China. He has participated in some government funded projects related with software engineering. His current research interests mainly include program comprehension and information retrieval.