2013 IEEE 37th Annual Computer Software and Applications Conference

From Sub-patterns to Patterns: an Approach to the Detection of Structural Design Pattern Instances by Subgraph Mining and Merging

Dongjin Yu, Yanyan Zhang, Jianlin Ge
School of Computer
Hangzhou Dianzi University
Hangzhou, China
yudj@hdu.edu.cn

Wei Wu
Zhejiang Provincial Key Laboratory of Network Technology and Information Security
Hangzhou, China
ww@topcheer.cn

***Abstract*—Structural design patterns address concerns related to high-level structures for applications being developed. Accurately recovered instances of structural design patterns support development related tasks like program comprehension and reengineering. However, the detection of structural design pattern instances is not always 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 detection. In this paper, we present an approach to the detection of instances of structural design patterns using source codes. We first transform the source codes and predefined patterns into graphs, with the classes as nodes and the relations 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 joint classes to see if the collective matches one of the predefined patterns. Compared with existing approaches, our approach focuses on simple sub-patterns, not complicated patterns. In this way, it can not only simplify the detection process, but also detect multiple pattern instances at a time. The results of the experiments on detecting pattern instances of Adapter, Bridge, Composite, Decorator and Proxy from 4 open source software systems demonstrate that our approach obtains better precision than the existing approaches.**

Keywords-design pattern detection; structural design patterns; sub-patterns; subgraph mining; graph merging;

I. INTRODUCTION

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. There are many types of design patterns, among which structural design patterns are design patterns that ease the design process by identifying simple ways to realize relationships between entities. Examples of structural patterns include adapter patterns, proxy patterns, decorator patterns, composite patterns, bridge patterns etc.

As demonstrated solutions to recurring problems, design patterns, including structural ones, help to reuse expert experience in software system design. During the past decade, they have been extensively applied in the software industry. 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. Detecting the instances of structural 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 [1][2].

However, the detection of structural design pattern instances is not always 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. Many approaches and tools that exploit different techniques have been proposed in the literature for structural design pattern detection. To the best of our knowledge, however, the existing approaches generally identify patterns sequentially, and few take advantage of the sub-components of design patterns to improve the accuracy and efficiency of pattern detection. 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 inherits the third, and one class inheriting and containing another class. We name these sub-components of patterns as the sub-patterns.

In order to further improve accuracy and efficiency, this paper proposes a novel approach to detecting structural design pattern instances from source codes. Unlike other approaches, it makes good use of the intermediate results, or sub-patterns of patterns, while detecting several kinds of structural 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.

The rest of the paper is organized as follows. After introducing the definitions of sub-patterns and structural feature models as background knowledge in Sections 2, we show how to represent the structural characteristics of source codes by directed graphs in Section 3. Section 4 describes the process of detecting structural design pattern instances in detail, covering four phases, namely system modelling, sub-pattern detecting, sub-pattern merging and behavioral analysis. The experimental results are given in

0730-3157/13 $26.00 © 2013 IEEE
DOI 10.1109/COMPSAC.2013.92

Section 5, followed by the related works in Section 6. Finally, the last section concludes the paper.

II. SUB-PATTERNS AND PATTERNS

Structural patterns are concerned with how classes and objects are composed to form larger structures. Although there are many kinds of structural patterns such as Adapter, Proxy and Bridge, they usually share some similarities, especially in their participants and collaborations. This is so probably because structural patterns rely on the same small set of language mechanisms for structuring code and objects: single and multiple inheritance for class-based patterns, and object composition for object patterns [3]. Therefore, we first detect these similar components of patterns, or *sub-patterns*, and then try to merge these components for the possible structural pattern instances. In this way, we avoid the direct pattern search that is far more complicated.

Definition 1 A *sub-pattern* represents a set of classes and the relationship between them, which is denoted as a 2-tuple: $SP = (< C_0, \dots, C_{k-1} >, R)$, where $C_0, \dots, C_{k-1}$ represents a set of classes and R represents a set of relationships among them, such as inheritance, association and aggregation.

$R = \{R_0, \dots, R_n, \dots, R_{r-1}\}$, and $R_n = \{r(C_i, C_j) = \{inherit|agg|ass\}\}$, where

$r(C_i, C_j) = \{inherit\}$ represents that class C_j inherits class C_i.

$r(C_i, C_j) = \{agg\}$ represents that classes C_i and C_j have an aggregation relationship, where class C_i is the whole class, and class C_j is the partial class.

$r(C_i, C_j) = \{ass\}$ 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.

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

Using the above definition, we introduce the following 8 sub-patterns, which can be combined to form different structural design patterns.

ICA $= (< C_1, C_2, C_3 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_2, C_3) = \{ass\}\}$

CI $= (< C_1, C_2, C_3 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_1, C_3) = \{inherit\}\}$

IAGG $= (< C_1, C_2 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_2, C_1) = \{ass\}\}$

IPA $= (< C_1, C_2, C_3 >, R)$ where $R = \{r(C_1, C_3) = \{inherit\}, r(C_1, C_2) = \{agg\}\}$

MLI $= (< C_1, C_2, C_3 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_2, C_3) = \{inherit\}\}$

IASS $= (< C_1, C_2 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_2, C_1) = \{ass\}\}$

SAGG $= (< C_1 >, R)$ where $R = \{r(C_1, C_1) = \{agg\}\}$

IIAGG $= (< C_1, C_2, C_3 >, R)$ where $R = \{r(C_1, C_2) = \{inherit\}, r(C_2, C_3) = \{inherit\}, r(C_3, C_1) = \{agg\}\}$

The structural 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 structural design patterns.

Definition 2 A *Structural Feature Model* is a set of two relevant sub-patterns representing the structural characteristics of one structural design pattern, which is denoted as: $SFT = SFT_0 || \dots || SFT_i || \dots || SFT_{n-1}$, where SFT_i is the combination of two Sub-patterns connected by the joint class $jClass$, or $SFT_i = SP_1 \,\&\&\, SP_2, jClass \in ClassSet_{SP_1}, jClass \in ClassSet_{SP_2}$.

The following presents five structural feature models, corresponding to five *GoF* structural design patterns [3].

A. The Adapter Pattern

The Adapter pattern converts the interface of a class into another interface that clients expect. It has the sub-pattern of ICA, but no sub-pattern of CI. More specifically,

$\text{SFM}_{Adapter} = \text{ICA} \,\&\&\, (!\, CI)$
$= (< Target, Adapter, Adaptee >, R_{ICA})$
$\&\&\, !\, (< Target, Adapter, Adaptee >, R_{CI})$

where, as Fig. 1 illustrates,

$R_{ICA} = \{r(Target, Adapter) = \{inherit\}, r(Adapter, Adaptee) = \{ass\}\}$

$R_{CI} = \{r(Target, Adapter) = \{inherit\}, r(Target, Adaptee) = \{inherit\}\}$

$\text{jClassSet} = \{\text{Target}, \text{Adapter}, \text{Adaptee}\}$

B. The Proxy Pattern

The Proxy pattern provides a surrogate or placeholder for another object to control access to it. At the lower level, it is composed of the sub-patterns of CI and ICA, or CI and IASS. More specifically,

$\text{SFM}_{Proxy} = (\text{CI}\&\&ICA)||(\text{CI}\&\&IASS)$
$= (< Subject, RealSubject, Proxy >, R_{CI})$
$\&\& (< Subject, RealSubject, Proxy, R_{ICA})$
$||(< Subject, RealSubject, Proxy >, R_{CI})$
$\&\&(< Subject, RealSubject, Proxy >, R_{IASS})$

where, as Fig. 2 illustrates,

$R_{CI} = \{r(Subject, RealSubject) = \{inherit\}, r(\text{Subject}, \text{Proxy}) = \{\text{inherit}\}\}$

$R_{\text{ICA}} = \{r(\text{Subject}, \text{RealSubject}) = \{\text{inherit}\}, r(RealSubject, Proxy) = \{ass\}\}$

$R_{\text{IASS}} = \{r(\text{Subject}, \text{Proxy}) = \{\text{inherit}\}, r(Subject, Proxy) = \{ass\}\}$

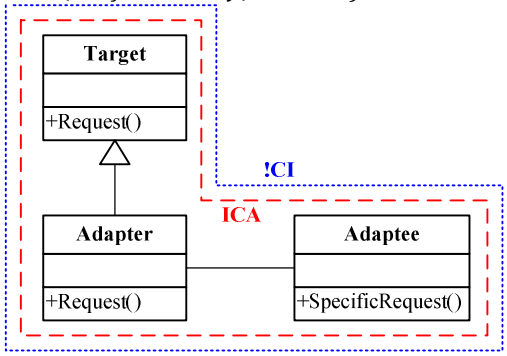
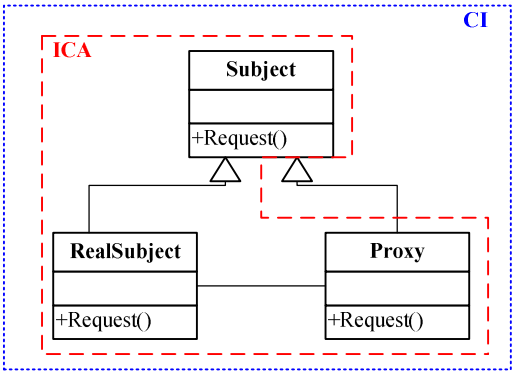
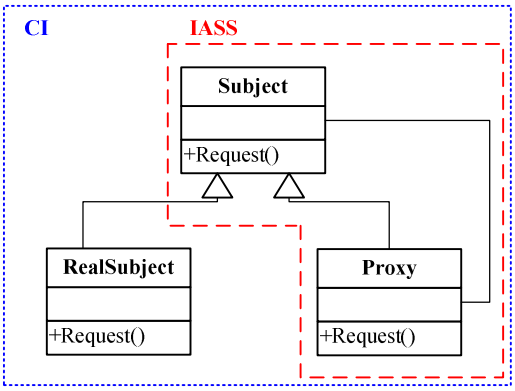


Figure 1. Structural Feature Model of Adapter pattern



(a) Proxy Pattern with sub-pattern of CI and ICA



(b) Proxy Pattern with sub-pattern of CI and IASS

Figure 2. Structural Feature Model of Proxy pattern

$\text{jClassSet} = \{\text{Subject}, \text{Proxy}\}$

C. The Decorator Pattern

The Decorator pattern attaches additional responsibilities to an object dynamically. At the lower level, it is composed of the sub-patterns of CI and IAGG, or CI and IAGG and MLI. More specifically,

$$\begin{aligned}\text{SFM}_{Decorator} &= (\text{CI}\&\&IAGG)||(\text{CI}\&\&\text{IAGG}\&\&\text{MLI}) \\ &= (< Comp, ConcreteComp, Decorator >, R_{CI}) \\ &\&\&(< Comp, Decorator >, R_{IAGG}) \\ &||(< Comp, ConcreteComp, Decorator >, R_{CI}) \\ &\&\&(< Comp, Decorator >, R_{IAGG}) \\ &\&\&(< Comp, Decorator, ComcreteDecrator >, \\ &\quad R_{MLI})\end{aligned}$$

where, as Fig. 3 illustrates,

$R_{CI} = \{r(Comp, ConcreteComp) = \{inherit\},$
$\text{r(Comp, Decorator)} = \{\text{inherit}\}\}$
$R_{IAGG} = \{r(Comp, Decorator) = \{inherit\},$
$\text{r(Comp, Decorator)} = \{\text{agg}\}\}$
$R_{MLI} = \{r(Comp, Decorator) = \{inherit\},$
$\text{r(Decorator, ConcreteDecorator)} = \{\text{inherit}\}\}$
$\text{jClassSet} = \{\text{Comp}, \text{Decorator}\}$

D. The Composite Pattern

The Composite pattern composes objects into tree structures to represent part-whole hierarchies. At the lower lever, it is composed of the sub-patterns of CI and IAGG, or the sub-pattern of SAGG, or the sub-patterns of CI and IIAGG. More specifically,

$$\begin{aligned}\text{SFM}_{Composite} &= (\text{CI}\&\&IAGG)||SAGG||(CI\&\&IIAGG) \\ &= (< Comp, ConcreteComp, Composite >, R_{CI}) \\ &\&\&(< Comp, Composite >, R_{IAGG}), \\ &||(< Comp >, R_{SAGG}) \\ &||(< Comp, ConcreteComp, Composite >, R_{CI}) \\ &\&\&(< Comp, Composite, Composite1 >, R_{IIAGG})\end{aligned}$$

where, as Fig. 4 illustrates,

$R_{CI} = \{r(\text{Comp}, \text{ConcreteComp}) = \{\text{inherit}\},$
$\text{r(Comp, Composite)} = \{\text{inherit}\}\}$
$R_{IAGG} = \{r(\text{Comp}, \text{Composite}) = \{\text{inherit}\},$
$\text{r(Comp, Composite)} = \{\text{ass}\}\}$
$R_{SAGG} = \{r(Comp, Comp) = \{agg\}\}$
$R_{IIAGG} = \{r(\text{Comp}, \text{Composite}) = \{\text{inherit}\},$
$\text{r(Composite, Composite1)} = \{\text{inherit}\},$
$\text{r(Comp, Composite1)} = \{\text{agg}\}\}$
$jClassSet = \{(Comp, Composite), Comp\}$

E. The Bridge Pattern

The Bridge pattern decouples an abstraction from its implementation so that the two can vary independently. At the lower level, it is composed of the sub-patterns of CI and IPA. More specifically,

$$\begin{aligned}\text{SFM}_{Bridge} &= CI \;\&\&\; IPA \\ &= (< Implementor, ConcreteImplementorA, \\ &\quad ConcreteImplementorB >, R_{CI}) \\ &\quad \&\&(< Abstraction, RefinedAbstraction, \\ &\quad Implementor >, R_{IPA})\end{aligned}$$

where, as Fig. 5 illustrates,

$$\begin{aligned}R_{CI} &= \{r(Implementor, ConcreteImplementorA) \\ &\quad = \{inherit\}, \\ &\quad r(Implementor, ConcreteImplementorB) \\ &\quad = \{inherit\}\}\end{aligned}$$

$R_{IPA} = \{r(Abstraction, RefinedAbstraction)$
$= \{inherit\},$

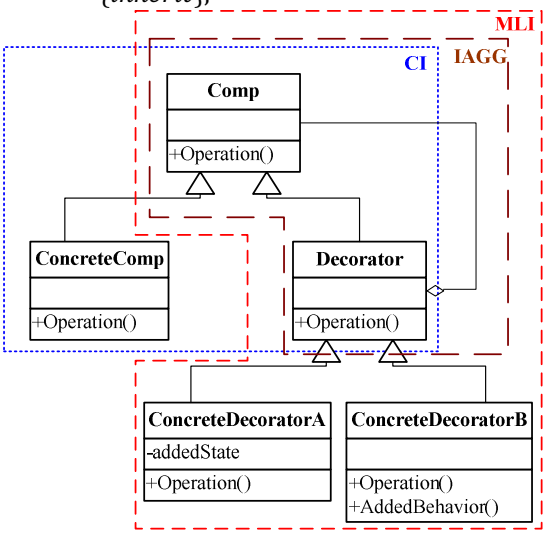
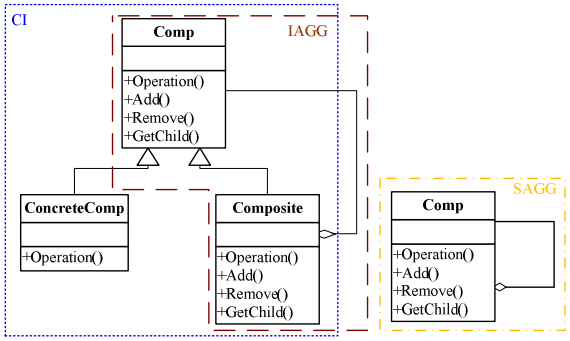
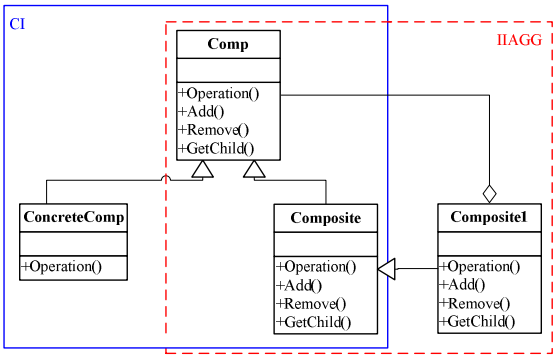


Figure 3. Structural Feature Model of Decorator pattern



(a) Composite Pattern with sub-patterns of CI and IAGG *(b) Proxy Pattern with sub-pattern of SAGG*



(c) Composite Pattern with sub-patterns of CI and IIAGG

Figure 4. Structural Feature Model of Compoiste pattern

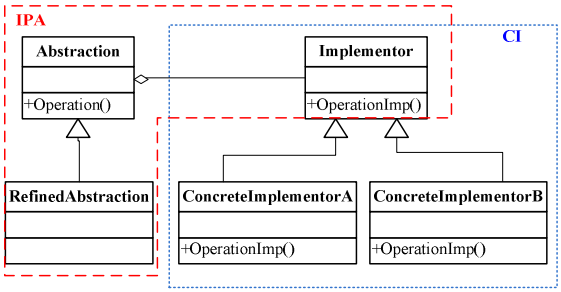


Figure 5. Structural Feature Model of Bridge pattern

$$r(Abstraction, Implementor) = \{agg\}\}$$
$$jClassSet = \{Implementor\}$$

III. CLASS-RELATIONSHIP DIRECTED GRAPH

Definition 3 *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 $GCDR = (V, E, v)$, where V is the set of vertices which represent classes, $E \subset V \times V$ is the set of directed edges which represent the relationship between classes, and $v: E \to W_E$ is the function that assigns weights to the edges, or the relationships.

$e(v_i, v_j) \in E$ means that the class represented by v_i inherits or contains or refers the class represented by v_j. In addition, we use the prime numbers of 2, 3, 5 and their products as the weights of edges or relationships, where 2 means the association relationship, 3 means the inheritance relationship and 5 means the aggregation relationship. If a compound relationship exists between two classes, we use the product of the weights of individual relationships instead. For instance, we use 6 as the weight of a compound relationship comprising the association and inheritance relationships, and 15 as the weight of a compound relationship made up of the inheritance and aggregation relationship.

Definition 4 Let $GCDR = (V_1, E_1, v_1)$, $GCDR^{ksub} = (V_2, E_2, v_2)$, $GCDR^{ksub}$ 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 $v_2(e) = v_1(e)$ for all $e \in E_2$. Here, k is called the size of $GCDR^{ksub}$.

Definition 5 Let $GCDR = (V, E, v)$, $|V| = n$, $M = (m_{i,j})_{n \times n}$ *is* called the *Class-Relationship Matrix* of GCDR, *or* MCR, where

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

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

Definition 6 Let $GCDR = (V, E, v)$, $|V| = n$, $V_i \in V$, 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 7 Let $GCDR = (V, E, v)$, $|V| = n$, $V_i \in V$, for the *Class-Relationship Matrix* of $M = (m_{i,j})_{n \times n}$, the *Inbound Composite Weight* of V_i is denoted as $CW_{in}(V_i)$, where $CW_{in}(V_i) = \prod_{j=1}^{n} m_{j,i}$.

Definition 8 Let $GCDR^1 = (V_1, E_1, v_1)$, $GCDR^2 = (V_2, E_2, v_2)$, if a bijective function $f: V_1 \to V_2$ exists such that each $a \in V_1$, $b \in V_1$, $(a, b) \in E_1$, if and only if $f(a) \in V_2$, $f(b) \in V_2$, $(f(a), f(b)) \in E_2$, $v_1((a, b)) = v_2((f(a), f(b)))$, then $GCDR^1$ and $GCDR^2$ are *Isomorphic Class-Relationship-Graphs*, donated as $GCDR^1 \cong GCDR^2$

Definition 9 A $n \times n$ matrix $p = (p_{ij})$ is called a *Permutation Matrix* where $p_{ij} \in \{0,1\}$, $\sum_{i=1}^{n} p_{ij} = 1$ and $\sum_{j=1}^{n} p_{ij} = 1$ for $i = 1, \ldots, n$, $j = 1, \ldots, n$.

Theorem 1 Let $GCDR_{M'} = (V', E', v')$, $GCDR_{M''} = (V'', E'', v'')$, $|V'| = |V''| = n$, $M' = (m_{i,j})_{n \times n}$, $M'' = (m_{i,j})_{n \times n}$ are *Class-Relationship Matrixes* of $GCDR_{M'}$ and $GCDR_{M''}$ respectively. If a Permutation Matrix P exists such that $M' = PM''P^T$, then $GCDR_{M'} \cong GCDR_{M''}$.

Proof Suppose $P = P_1 P_2 \ldots P_i \ldots P_{m-1} P_m$ where P_i denotes the Row-switching Elementary Matrix which is obtained by swapping two rows of the identity matrix. Consequently,

$$\begin{aligned} M' &= PM''P^T \\ &= (P_1 P_2 \ldots P_{m-1} P_m) M'' (P_1 P_2 \ldots P_{m-1} P_m)^T \\ &= P_1 P_2 \ldots P_{m-1} P_m M'' P_m^T P_{m-1}^T \ldots P_2^T P_1^T \end{aligned}$$

As $P_i = P_i^T$, then

$$M' = P_1 P_2 \ldots P_{m-1} (P_m M'' P_m) P_{m-1} \ldots P_2 P_1.$$

Let $M_m = P_m M'' P_m$, then

$M' = P_1P_2 \dots (P_{m-1}M_mP_{m-1}) \dots P_2P_1$.

As M_m is obtained by swapping two rows and two columns of M'', the *Class-Relationship Directed Graph* corresponding to M_m is isomorphic to that corresponding to M'', or $GCDR_{M_m} \cong GCDR_{M''}$.

Let $M_{m-1} = P_{m-1}M_mP_{m-1}$, then

$M' = P_1P_2 \dots (P_{m-2}M_{m-1}P_{m-2}) \dots P_2P_1$.

As M_{m-1} is obtained by swapping two rows and two columns of M_m, the *Class-Relationship Directed Graph* corresponding to M_{m-1} is isomorphic to that corresponding to M_m, or $GCDR_{M_{m-1}} \cong GCDR_{M_m}$.

Repeat the above transformations until $M' = P_1M_2P_1$, $GCDR_{M'} \cong GCDR_{M_2}$, where $M_2 = P_2M_3P_2$.Thus,

$GCDR_{M'} \cong GCDR_{M_2} \cong \cdots \cong GCDR_{M_m} \cong GCDR_{M''}$, or $GCDR_{M'} \cong GCDR_{M''}$.

IV. DETECTION OF STRUCTURAL DESIGN PATTERN INSTANCES

The problem can be defined as follows: given the source codes of one certain system, detect all the instances of structural design patterns that the system contains. To resolve this problem, we developed an approach that consists of the following 4 phases.

1) Modelling the system 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 phase are compared with the *Class-Relationship Directed Graphs* of the predefined sub-patterns. The matched ones are considered as the instances of the sub-patterns.

3) Obtaining the candidate pattern instances

An effort is made to combine relevant sub-patterns as the predefined *Structural Feature Models* indicate. The collectives which have specific joint classes are considered as the candidate instances of structural design patterns.

4) Analyzing behavioral characteristics

The behavioral characteristics of candidate pattern instances are identified and matched with those of standard patterns. The pattern instances are finally obtained by filtering out false candidates.

The four phases and their corresponding outputs are illustrated in Fig. 6, and are discussed in detail in the following sections.

A. Modelling system codes

As the pre-process phase, the process of modelling system codes extracts structural information, such as classes and the relationship among classes, from source codes for detection of pattern instances. We first use Enterprise Architect to analysis source codes, and then produce class diagrams and XML-based Metadata Interchange files, or XMI files, which represent the structural information of system.

Enterprise Architect is a visual platform for designing and constructing software systems for generalized modelling purposes, which can be downloaded from http://www.sparxsystems.com.au.

We then generate the *Class-Relationship Matrix* of the system represented by obtained XMI files. Fig. 7 gives an example of class diagrams and their corresponding *Class-Relationship Matrix,* in which *ProxyImage* and *RealImage* inherit *Image* and *ProxyImage* refers *RealImage*.

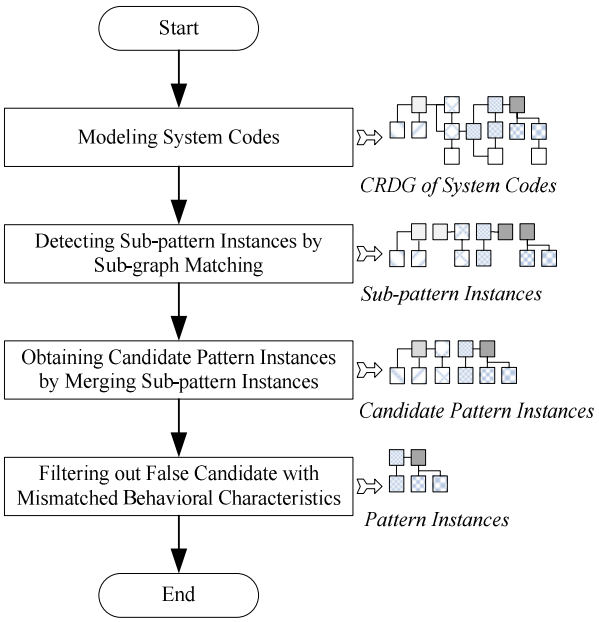
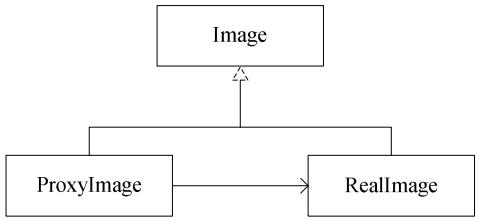


Figure 6. Process of detection of structural design pattern instances



(a) example of class diagrams

	Image	RealImage	ProxyImage
Image	1	1	1
RealImage	3	1	1
ProxyImage	3	2	1

(b) example of Class-Relationship Matrix

Figure 7. Example of class diagrams and their corresponding Class-Relationship Matrix

B. Detecting sub-patterns

Since both the systems and predefined sub-patterns can be represented by *Class-Relationship Directed Graphs*, the problem of detecting sub-patterns is 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 I, and can be divided into the following 2 steps.

1) Search the candidate classes in GCDR^s of the system (from Line 3 to Line 7)

For each class c_i in GCDR^m, select the classes in GCDR^s whose *Outbound Composite Weights* and *Inbound Composite Weights* in GCDR^s can be divided with no remainder by those of c_i in GCDR^m. These selected classes for c_i constitute the *Candidate Class Set* of c_i.

2) Combine the candidate classes to generate *k-subGraphs* and determine if they are isomorphic to GCDR^m of the sub-patterns (from Line 8 to Line 16)

The *k-subGraph* of GCDR^s, or GCDR^{s-ksub}, is generated by choosing each vertex in every *Candidate Class Set*, and then compared with GCDR^m of sub-patterns. If GCDR^{s-ksub} is isomorphic to GCDR^m, then GCDR^{s-ksub} is regarded as corresponding to the specific sub-pattern.

C. Obtaining candidate pattern instances

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

Since a pattern is composed of several sub-patterns with joint classes, only the connected sub-patterns are considered for the creation of pattern instances. The algorithm for detecting Bridge pattern instances is illustrated in Table II, as an example of obtaining candidate pattern instances. We compare the instances of CI sub-patterns with the instances of IPA sub-patterns (Line 4). If the parent class of one CI sub-pattern instance, or $\mathrm{V}_{CI}.C1$ in Table II, is identical to the partial class of one IPA sub-pattern instance, or $\mathrm{V}_{IPA}.C3$ in Table II, these two sub-pattern instances are then merged into the Bridge pattern instance (from Line 6 to Line 13). Other structural pattern instances such as Adapter, Bridge, Composite, Decorator and Proxy can be obtained from sub-pattern instances in similar ways.

D. Analyzing behavioral characteristic

There may be falsely positive candidates obtained by the structure analysis because structural analysis concentrates only on the structural aspect of design patterns. Most design patterns have their own unique behavioral features in addition to the structural ones. Analyzing the behavior of these candidates can filter out some false ones. Due to space limitation, we do not however proceed with the detail about behavioral checking. Interested readers may refer to [4].

TABLE I. ALGORITHM OF DETECTING SUB-PATTERNS

Input:
$\mathrm{GCDR}^m =< \mathrm{V}_m, \mathrm{E}_m, \mathrm{v}_m >$ //*Class-Relationship Directed Graph* of sub-pattern
$\mathrm{GCDR}^s =< \mathrm{V}_s, \mathrm{E}_s, \mathrm{v}_s >$ // *Class-Relationship Directed Graph* of system
Output:
IdentifiedSubPatterSet //Set of *Class-Relationship Directed Graphs* of Identified sub-patterns

```
1  IdentifiedSubPatterSet = ∅
2  //generate Candidate Class Sets
3  for each c_i (1 ≤ i ≤ k) in V_m {
4     CCS(c_i) = {c|c ∈ V_s
5        ∩ (CW_out(c_i)|CW_out(c))  //check the Outbound Composite Weight
6        ∩ (CW_in(c_i)|CW_in(c))}  //check the Inbound Composite Weight
7  }
8  for each m_1 in CCS(c_1), m_2 in CCS(c_2), …, m_k in CCS(c_k) {
9     generate GCDR^{s-ksub} =< V_{s-ksub}, E_{s-ksub}, v_{s-ksub} >, where
10       V_{s-ksub} = {m_1, m_2, … , m_k},  //pick one class from each Candidate Class Set
11       E_{s-ksub} = E_s ∩ (V_{s-ksub} × V_{s-ksub}),
12       v_{s-ksub}(e) = v_s(e), for all e ∈ E_{s-ksub}
13     //check if GCDR^{s-ksub} and GCDR^m are isomorphic or not
14    if IsIsomorphic(GCDR^{s-ksub}, GCDR^m)
15       //if GCDR of k selected classes is isomorphic to that of GCDR^m
16       IdentifiedSubPatterSet = IdentifiedSubPatterSet ∪ GCDR^{s-ksub}
17 }
18 return IdentifiedSubPatterSet
```

TABLE II. ALGORITHM OF OBTAINING BRIDGE PATTERNS

Input:
$CISet$ //Set of *Class-Relationship Directed Graphs* of identified *CI* sub-pattern
$IPASet$ //Set of *Class-Relationship Directed Graphs* of identified *IPA* sub-pattern
Output:
$BridgeSet$ //Set of *Class-Relationship Directed Graphs* of identified *Bridge* patterns

```
1  BridgeSet = ∅
2  for each GCDR^CI =< V_CI, E_CI, v_CI > in CISet {
3     for each GCDR^IPA =< V_IPA, E_IPA, v_IPA > in IPASet {
4        if (V_CI.C1 = V_IPA.C3) {
5           //having the joint class of implementer
6           generate GCDR^combined =< V_combined, E_combined, v_combined >, where
7              V_combined = {V_CI.C1, V_CI.C2, V_CI.C3, V_IPA.C1, V_IPA.C2},
8              E_combined = {e1 = (V_CI.C1, V_CI.C2), e2 = (V_CI.C1, V_CI.C3),
9                 e3 = (V_IPA.C1, V_IPA.C2), e4 = (V_IPA.C1, V_CI.C1)}
10             v_combined(e1) = 3 //inheritance relationship
11             v_combined(e2) = 3 //inheritance relationship
12             v_combined(e3) = 3 //inheritance relationship
13             v_combined(e4) = 5 //aggregation relationship
14          BridgeSet = BridgeSet ∪ GCDR^combined
15       } // end of if
16    } //end of inner for
17 } // end of outer for
18 return BridgeSet
```

V. EXPERIMENTS

We applied the approach on four open source software systems to validate its effectiveness. These four systems were JHotDraw, a Java GUI framework for technical and structured graphics, JReFactory, a refactoring tool for the Java programming language, JEdit, a programmer's text editor written in Java, and Dom4j, a Java library for working with XML, XPath and XSLT. We scanned the source codes and identified the structural pattern instances using the toolkit based on our proposed approach. The identified pattern instances were then manually examined to see whether they are correct. The numbers of eight identified sub-pattern instances in these four software systems are given in Table III.

Table IV gives the numbers of identified pattern instances. Here, *Num. of TP* denotes the number of pattern instances that were identified and really existed in the system, while *Num. of FP* denotes the number of pattern instances that were found, but not implemented in the system. The *precision* in the last column is defined as follows.

$$Precision = \frac{Num.of\ TP}{Num.of\ TP + Num.of\ FP} \quad (1)$$

The small numbers of False Positives in Table IV shows that our approach achieves high precision. Since there are not many quantitative results reviewed in the literature, we just compare our results with that in [4]. As Table IV shows, our approach detects more true instances of *Adapters* and *Bridges* in JHotDraw 6.0.

Table V. presents all 5 Adapter instances identified from JReFactory, showing their *Targets*, *Adapters* and *Adaptees* respectively.

VI. RELATED WORKS

Detecting instances of design patterns from system source codes can help to understand and trace back to the original design decisions and reengineer the systems. There have been a number of different approaches proposed to solve this problem in the literature, among which the approaches to the detection of structural design pattern instances have drawn the most interest.

TABLE III. NUMBERS OF SUB-PATERNS DETECTED

Sub-pattern	JHotDraw 6.0	JReFactory 2.9.19	JEdit 4.2	Dom4j 1.6.1
IASS	6	8	7	3
IAGG	1	318	284	96
ICA	271	348	182	82
IPA	329	701	136	174
IIAGG	1	116	86	39
CI	5984	22158	3270	7864
MLI	230	619	86	121
SAGG	2	2	0	0

TABLE IV. NUMBER OF PATTERN INSTANCES DETECTED

Design Pattern	Num. of TP	Num. of FP	Precision
JHotDraw 6.0			
Adapter	7(4)	0(0)	100%
Composite	2(0)	0(0)	100%
Decorator	4	0	100%
Bridge	62(53)	5(5)	93%
Proxy	5	0	100%
JReFactory 2.9.19			
Adapter	5	0	100%
Composite	3	0	100%
Decorator	0	0	N/A
Bridge	32	1	97%
Proxy	0	0	N/A
JEdit 4.2			
Adapter	4	1	80%
Composite	0	0	N/A
Decorator	12	0	100%
Bridge	2	0	100%
Proxy	64	2	97%
Dom4j 1.6.1			
Adapter	7	0	100%
Composite	1	0	100%
Decorator	9	0	100%
Bridge	11	0	100%
Proxy	10	0	100%

Note: the numbers in brackets are the counts of recovered instances obtained in [4].

TABLE V. ADAPTER INSTANCES DETECTED IN JREFACTORY

No.	Role	Class
1	Target	org.acm.seguin.io.DirectoryTreeTraversal
	Adapter	org.acm.seguin.tools.builder.PrettyPrinter
	Adaptee	org.acm.seguin.pretty Pretty.PrintFile
2	Target	org.acm.seguin.io.DirectoryTreeTraversal
	Adapter	org.acm.seguin.tools.stub.StubGenTraversal
	Adaptee	org.acm.seguin.tools.stub.StubFile
3	Target	org.acm.seguin.summary.load.LoadStatus
	Adapter	org.acm.seguin.summary.load.SwingLoadStatus
	Adaptee	org.acm.seguin.summary.load.RefactoryStorage
4	Target	org.acm.seguin.pmd.Rule
	Adapter	org.acm.seguin.pmd.AbstractRule
	Adaptee	org.acm.seguin.pmd.RuleProperties
5	Target	org.acm.seguin.pmd.Rule
	Adapter	test.net.sourceforge.pmd.MockRule
	Adaptee	org.acm.seguin.pmd.RuleProperties

Since structural design patterns each 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. In [5], Rasool and Mäder 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, Arcelli Fontana *et al.* introduce micro-structures which are regarded as the building blocks of design patterns [6][7]. 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. In order to reduce the complexity of design pattern detection, Daryl Posnett *et al.* introduce three kinds of meta-patterns [8]. According to their definitions, a meta-pattern is part of a design pattern which contains structural and behavioral features of that design pattern. In addition to inspecting the code structures, some approaches also investigated the behavioral aspects. In [9], Andrea *et al.* present a pattern recovery approach that analyzes the behavior of pattern instances both statically and dynamically. In particular, the proposed approach exploits model checking to statically verify the behavioral aspects of design pattern instances.

One of challenges comes from the intermediate expression for the source codes. Possible solutions include predicates, matrices, vectors 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 [10]. Jing 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 [4]. 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 [11]. 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 [12]. Akshara Pande *et al.* apply graph decomposition and graph isomorphism techniques for design pattern detection [13]. Tsantalis *et al.* calculate the similarities between two vertices for pattern detection [14]. Qiu *et al.* introduce a state space graph to avoid the search space explosion and reduce the opportunity of detecting subgraph isomorphism [15]. Other representations include modeling language [16] and XML [17].

The identification of modified pattern versions also brings about challenges for pattern detection. In [14], Tsantalis *et al.* 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 [18]. Lebon Maurice and Tzerpos Vassilio, on the other hand, introduce an approach that complements existing detection methods by utilizing finely grained static information contained in the software system [19]. It filters a large number of false positives by utilizing finely grained rules that describe the static structure of a design pattern.

Many toolkits have been developed to support the detection of pattern instances. Shaheen 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. It also helps in code reusing by suggesting to the programmer how to write API code to facilitate rapid software development [20]. DPJF, presented by Binun Alexander and Kniesel Günter in [21], achieves high detection quality through a well-balanced combination of structural and behavioral analysis techniques. PINOT, developed by Shi *et al.*, uses lightweight static program analysis techniques to capture program intent. PINOT detects all the GoF patterns that have concrete definitions driven by code structure or system behavior [22]. In [23], Arcelli Fontana and Zanoni 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 [24], 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 [4], FUJABA [12] and Columbus [18].

Though much progress was made, there is still a lack of accuracy and flexibility when detecting pattern instances. To the best of our knowledge, the existing approaches generally concentrate on the structural characteristics of the whole pattern directly. However, the different patterns in fact always share common internal structures that are easier to detect. Therefore, our approach detects the sub-patterns first and then tries to merge these sub-patterns into all possible patterns. Moreover, the sub-patterns defined in this paper are different from the micro-structures defined in [6] and [7], or meta-patterns in [8], which contain both structural and behavioral features. Because behavioral features change frequently and are hard to detect, the sub-patterns presented in this paper contain only classes and the relationships between them. Finally, compared with the approaches based on graph isomorphism like [13], our approach does not need to decompose the system graph and design pattern graph. Its complexity is thus reduced.

VII. CONCLUSIONS

Patterns are formalized best practices that the programmer must implement themselves in the application. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

In this paper, we present a new approach to the detection of instances of structural 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. The identified sub-pattern instances are further merged by joint classes to see if the collective matches with one of the predefined patterns. The results of the experiments demonstrate that our approach obtains better precision than the existing approaches.

However, our approach currently can only detect instances of structural design patterns. In the future, we will upgrade it to detect creational and behavioral design pattern instances as well. In addition, we plan to filter out certain candidate sub-patterns before merging in order to accelerate the execution speed of the approach when dealing with software systems of large scales.

ACKNOWLEDGMENT

The work is supported by the Natural Science Foundation of Zhejiang (No.LY12F02003), the Key Science and Technology Project of Zhejiang (No. 2012C11026-3, No. 2008C11099-1) and the open project foundation of Zhejiang Provincial Key Laboratory of Network Technology and Information Security. The authors would also like to thank anonymous reviewers who made valuable suggestions to improve the quality of the paper.

REFERENCES

[1] N. Pettersson, W. Lowe and J. Nivre, "Evaluation of Accuracy in Design Pattern Occurrence Detection", IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 575-590, 2010.

[2] J. Dong, Y. Zhao and T. Peng, “A Review of Design Pattern Mining Techniques”, The International Journal of Software Engineering and Knowledge Engineering (IJSEKE), World Scientific Publishing, vol. 19, pp. 823–855, 2009.

[3] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994

[4] J. Dong, Y. Zhao and Y. Sun, “A Matrix-Based Approach to Recovering Design Patterns”, IEEE Transactions on Systems, Man and Cybernetics – Part A: Systems and Humans, vol. 39 no. 6, pp. 1271–1282, 2009.

[5] G. Rasool and P. Mäder, “Flexible design pattern detection based on feature types”, 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 243-252, 2011.

[6] F. Arcelli Fontana, S. Maggioni and C. Raibulet, “Understanding the relevance of micro-structures for design patterns detection”, Journal of Systems and Software 84, pp. 2334–2347, 2011.

[7] F. Arcelli Fontana, S. Maggioni and C. Raibulet, “Design patterns: a survey on their micro-structures”, Journal of Software Maintenance and Evolution 33(8), pp. 1–25, 2011.

[8] D. Posnett, C. Bird and P. T. Devanbu, “THEX: Mining meta-patterns from java”, 7th IEEE Working Conference on Mining Software Repositories, pp. 122–125, 2010.

[9] A. De Lucia, V. Deufemia, C. Gravino and M. Risi, “Improving behavioral design pattern detection through model checking”, European Conference on Software Maintenance and Reengineering, CSMR, pp. 176-185, 2011.

[10] D. Heuzeroth, S. Mandel and W. Lowe, “Generating design pattern detectors from pattern specifications”, 18th IEEE International Conference on Automated Software Engineering, pp.245-248, 2003.

[11] O. Kaczor, Y. Guéhéneuc and S. Hamel, “Efficient identification of design patterns with bit-vector algorithm”, 10th European Conference on Software Maintenance and Reengineering, pp. 175-184, 2006.

[12] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals and J. Welsh. “Towards pattern-based design recovery”, 24rd International Conference on Software Engineering, pp. 338–348, 2002.

[13] A. Pande, M. Gupta and A. K. Tripath, “A New Approach for Detecting Design Patterns by Graph Decomposition and Graph Isomorphism”, Communications in Computer and Information Science, pp. 95, 108–119, 2010.

[14] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and S. T. Halkidis, “Design Pattern Detection Using Similarity Scoring”, IEEE Transactions on Software Engineering 32(11), pp. 896–909, 2006.

[15] M. Qiu and Q. Jiang, “Detecting Design Pattern Using Subgraph Discovery”, Lecture Notes in Computer Science, vol. 5990, pp. 350–359, 2010.

[16] M. Elaasar, L. C. Briand and Y. Labiche, “VPML: an approach to detect design patterns of MOF-based modeling languages”, Software and Systems Modeling, pp. 1-30, 2013.

[17] N. Bouassida and H. Ben-Abdallah, “A New Approach for Pattern Problem Detection”, B. Pernici (Ed.): CAiSE 2010, LNCS 6051, pp. 150–164, 2010.

[18] R. Ferenc, A. Beszedes, L. Fulop and J. Lele, “Design pattern mining enhanced by machine learning”, 21st IEEE International Conference on Software Maintenance, pp. 295 – 304, 2005.

[19] L. Maurice and T. Vassilios, “Fine-grained design pattern detection”, 36th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2012, pp. 267-272.

[20] S. Khatoon, G. Li and R. Muhammad Ashfaq, “A Framework for automatically mining source code”, Journal of Software Engineering, vol. 5 no. 2, pp. 64-77, 2011.

[21] Binun Alexander and Kniesel Günter, “DPJF - Design pattern detection with high accuracy”, European Conference on Software Maintenance and Reengineering, pp. 245-254, 2012.

[22] N. Shi and R. Olsson, “Reverse Engineering of Design Patterns from Java Source Code”, 21st IEEE International Conference on Automated Software Engineering, 2006, pp. 245-248.

[23] F. Arcelli Fontana and M. Zanoni, “A tool for design pattern detection and software architecture reconstruction”, Information Sciences, vol. 181 no. 7, pp. 1306-1324, April 2011.

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