International Journal of Hybrid Information Technology
Vol.8, No. 5 (2015), pp. 63-72
http://dx.doi.org/10.14257/ijhit.2015.8.5.07

Research on Detecting Design Pattern Variants from Source Code Based on Constraints

Li Wen-Jin[1], Pan Ju-long[2] and Wang Kang-Jian[3]

[1]*College of Modern Science and Technology; China Jiliang University; Zhejiang Hangzhou*
[2,3]*College of Information Engineering; China Jiliang University; Zhejiang Hangzhou*
[1]*liwenjin@cjlu.edu.cn*, [2]*pjl@cjlu.edu.cn*, [3]*Kangjian.wang@cjlu.edu.cn*

Abstract

Identifying design patterns from source code is one of the most promising methods for improving software maintainability, reusing experience and facilitating software refactoring In the process of design pattern application, different methods of instantiation usually lead to generation of pattern variants. The detection of these variants from source code is a key point and a challenge of reverse engineering. In this paper, we propose an approach of detecting design pattern variants from source code based on constraints. More specifically, we first propose the method to describe the design pattern variants based on constraints, input the constraint conditions of design patterns into the library of pattern features, conduct static analysis on source code including analysis of control flow and data flow to obtain representations of pattern participants in the source code, conduct matching with the predicate matching tool, and then merge the instances by clustering analysis and obtain the final collection of pattern instances. Finally, a tool of extracting design pattern variants from source code is implemented and the feasibility of our approach is verified through the results of running it on three open source software.

Keywords: *design pattern, variants, constraints, clustering analysis*

1. Introduction

Design pattern [1] is a high-level abstraction of object-oriented design. From the perspectives of programmed understandability and software maintenance, a design pattern provides the role information of all classes in the pattern structure as well as the information about the relations among constituent elements of the pattern and the relations between these elements with other parts in the system. Therefore, the extraction of design patterns from the source code remains a key issue in reverse engineering. Since GOF offers only the purposes and basic design methods of each design pattern, software engineers tend to change the basic implementation methods while ensuring consistent purposes which we called design pattern variants to improve the applicability. How to effectively identify these pattern variants is of crucial importance for the understanding of design of legacy system and a difficulty of detecting design patterns from the source code.

In the pattern detecting process, software engineers often choose to enhance the constraints of identification to improve recognition accuracy. However, the enhancement will increase the omission rate, which means that some candidates of pattern instances may be missed out due to the enhanced constraints. In order to reduce such omission rate, weakened constraints are usually chosen, which adversely reduces the recognition accuracy. Some instances that are not candidates for design pattern will also be identified. Therefore, how to achieve a balance between the omission rate and the recognition accuracy is a rather difficult task to be tackled [2].

ISSN: 1738-9968 IJHIT

This paper proposes a constraint-based method to identify variants of design pattern in the source code by decomposing the constraints of design pattern into basic constraints and variant constraints. The basic constraints represent prerequisites for design pattern instances, while the variant constraints vary based on different implementation methods of design pattern. In this paper, we attempt to respectively match basic constraints and variant constraints with the source code representation, summarize the matching results, and the candidates which satisfy all of the basic constraints and one of the variant constraint is design pattern instances. Compared with the method that use relaxed constraints to improve the recognition rate, the approach proposed in this paper is an attempt to improve the recognition accuracy of design pattern without affecting the omission rate.

The structure of this paper is organized in the following sequence. Related work is first introduced; forms and implementation methods of variants are then illustrated by taking Adapter and Composite as examples; meta model of design pattern is defined; formal methods of design pattern is then explained by giving examples of structural design patterns; matching algorithm for the design pattern is presented; and finally the paper is concluded with experimental results, conclusions and future work.

2. Related Work

Design pattern detection as an important branch of design discovery, has been attracting widespread attention in the academic community. Related work has been discussed in [3-4], so there is no need to repeat here. This paper focuses on discussing researches of detecting design pattern variants.

Nikolas Tsantalis [5] adopted similarity scoring method to detect instances of design patterns. This method is designed to represent information of target system and design pattern, including association and inheritance relations between classes, abstract classes, object creations, abstract method invocations etc. into matrices as the first step. Then, the similarity scoring algorithm is used to calculate similarity between subsystems and corresponding pattern matrices. Finally, a collection of instances of a design pattern is constituted by subsystem classes that have the highest similarity with pattern roles. The characteristic of this method lies in its support for detecting pattern variants from the target system. Since it only relies on static analysis to identify pattern instances and provides only a collection of instance candidates for the behavioral design patterns, we should continue to combine it with dynamic analysis to verify whether these instances have been accurately identified.

Ghulam Rasool [6-7] analyzed the purposes of Abstract Factory, Decorator, Adapter and Proxy and possible variants that may arise during the process of implementation; used feature types to define these variants and Java language to implement every type of variant; and finally adopted the mainstream design pattern detection tool to test the identification of these pattern variants. The work in the paper focuses more on defining variants of design pattern than on identifying these variants.

K. Stencel and P. W. egrzynowicz [8-9] analyzed the possible pattern variants of three patterns, singleton, factor method and abstract Factory; offered the detecting standards of these three patterns in the form of predicate; and combined the structural analysis, control flow analysis and data flow analysis to implement the process of identifying pattern instances.

Alexander Binunv [10-11] improved design pattern description and put forward two models: Super type forwarder and Decuples. Super type forwarders include design patterns of Decorator, Proxy and Chains of Responsibility that forward invocations to an identical “parent”. While Decuple contains the design patterns of Observer, Composite, Bridge, State and Strategy that forward an invocation to all dependent objects. The methodology discussed some constraints that may lead to faulty instances such as

attribute maintenance and status change, etc. and effectively combined the structural constraints with behavioral constraints when searching for pattern instances to improve the recognition accuracy of patterns. In addition, the methodology also discussed several reasons that effect the recall rate. For example, some relevant role players are not identified because they are situated among codes that have not been analyzed; the attribute values of some role players are too strict; and some relational closures are ignored due to insufficient analysis etc. To improve the recall rate of pattern recognition, some stricter constraints are relaxed in the process of searching for pattern instances.

In comparison with the foregoing work, the work in this paper mainly is embodied in the following aspects:

(1)It proposes a general method to define variants of design pattern and offers the meta model for design pattern definition.

(2)It combines control flow analysis and data flow analysis to analyze the source code and acquires more behavioral information that is crucial to search for pattern variants.

(3)It uses first-order logic tools to match the information library of source code with the library of pattern features, provides instances collection of each design pattern, takes the standard that instances of homologous constraints are similar and merges the instances by clustering analysis, which improves the recognition accuracy.

3. Examples of Design Pattern Variants

The existence of design pattern variants is an important factor affecting the recognition accuracy and omission rate. The principle of constraint relaxation is to remove some constraints on pattern "variants" and retain the most essential features of the pattern, which leads to the decline of recognition accuracy. This paper distinguishes the most essential features of a pattern from the features associated with variants, that is, to divide the forms of a design pattern into basic constraints and variant constraints. The recognition standards of a pattern instance are to satisfy the most essential features as well as features of a certain variant. The most essential features here are in fact equivalent to the features after constraint relaxation. With variant features as constraints, we can filter instances that satisfy essential features and thus significantly improve the recognition accuracy without increasing the omission rate.

We take Adpater pattern and Composite pattern as examples to introduce instances of pattern variants in the following:

Example 1: Adapter Pattern

Two common design methods of Adapter are generic class adapter and object adapter. The former uses multiple inheritances to implement the process of adaptation. The advantages of this implementation method lies in that Adapter call method of Adoptee conveniently, while the disadvantage lies in that for languages without private inheritance mechanism such as Java, using multiple inheritance will expose the interface of Adoptee to the Client. Therefore, this implementation method usually appears in the design implemented in C++ language. The latter method designs Adapter and Adoptee into an association relation to implement the adaptation process, commonly used in object-oriented languages. The advantage of this implementation method is the decoupling of the relationship between Adapter and Adapee, while the disadvantage is that the calling is less convenient than the former method.

The example of class adapter (C++)：

```
Class Adapter: public Target, private Adaptee{
        Public void request ()
        {
            ….
            Specific Request () ;}…}
```

The example of object adapter (Java) ：

```
Class Adapter extends Target
 {
        Private Adaptee adaptee;
        Public void request ()
        {
            Adaptee. specific Request()}
```

Example 2: Composite Pattern

The variants of Composite pattern are mainly manifested in whether to put the "Child" operation into the "Component". During the process of design, we choose to put it in the Component or in the Composite according to different scenarios. The advantage of putting Child operation in the Component lies in that Clients can treat Leaf and Composite transparently and have no need of concerning whether each Component is "Leaf" or "Composite". However, the disadvantage is that the Child operation will be taken to the implementation process of Leaf classes, so in the implementation of Leaf classes it should be considered that how to handle the situation when add/remove operation is invoked. If we put the Child operation in the Composite, the advantages and disadvantages are just the opposite.

Example of "parent-based" Composite：

```
Public abstract class Component {
        Private Set<Component> children;
        Public void add (Component c)
  {
        Children.add(c);
  }
  Public void remove (Component c) {
        Children.remove(c);
  }
  Public abstract void operation ();
}
```

Example of "child-based" Composite：

```
Public interface Component {
   Void operation () ;}
Public class Composite implements Component {
   Private Set<Component> children;
   Public void add (Component c) {
        Children. Add(c) ;}
        Public void remove (Component c) {
        Children. Remove(c);
   }
```

In the process of actual software development, variants of each pattern are all possible to show up. If we can identify the variants as many as possible and at the same time decide which variant it belongs to, it will be of crucial importance to the understanding of design.

4. Definition and Detection of Variants

To detect variants, we need to describe pattern and its variants, and then analyze the source code to obtain the internal representations, and finally match the internal representations with the formalized results of patterns and conclude the final results.

The patterns and their variants is formalized in the following two steps: 1) define the meta model which describes a collection of relations among pattern participants; 2) analyze the features of each pattern based on the meta model, find out which features are essential to satisfy and which features belong only to a certain variant. Formalize the essential features and relevant features of variants into the form of predicate.

The variants of design patterns are extracted from source code in the following steps: firstly analyze the source code and represent the information of source code into an information library of participants, match the information library with pattern constraint predicates, then filter and screen on the matching results.

The entire process of defining and detecting design pattern variants is shown in Figure 1.

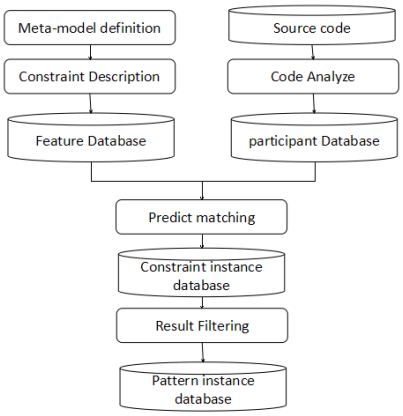

Figure 1. The Pattern Detection Process

4.1. Metal Model Definition

As the basis for the representation of design pattern and pattern variants, we describe features associated with the recognition of design patterns by the Meta model and describe them respectively according to the relations among class, method, attribute, method parameter.

1. Constraint of class relationship

(1) Inherit (class A × class B): Class A inherits Class B, which is manifested by the inheritance relationship with Class B in the declaration of Class A;

(2) Association (class A × class B): Class A associates with Class B, manifested by Class A possessing one attribute of Class B;

(3) Aggregation (class A × class B): Class A aggregates with Class B, manifested by Class A being associated with Class B and Class A having created the instances of Class B at the initial stage;

(4) Delegate (class A × class B): Class A delegates Class B, manifested by all public methods of Class A containing the invocation of one or several methods of Class B;

2. Constraint of method relationship

Invoke (method A × method B): method A invokes method B.

3. Constraints of method-class relationship

(1) has Method (class A × method B): method B is a method of class A;

(2) Has Param (method A × class B): method A contains parameters of class B;

(3) Return Type (method A× class B): the return type of method A is class B.

4. Constraints of attribute-class relationship

(1) Is Type (attribute A × class B): the class type of attribute A is class B;

(2) Has Attr (class A × attribute B): class A contains attribute B.

4.2. Pattern Description

Definition 1: The function Participant (V) returns the collection of all participants appeared in a constraint predicate V.

Firstly, we define a “complete collection” of participants: that is, all participants mentioned in the feature description of a pattern are the elements of the complete collection”. Then, we describe the predicates of basic constraints according to the basic features of the pattern, and also describe the predicates of variant constraints according to the features of each variant,

thus forming a complete set of pattern description. On this basis, we use the triplet <M, N, K> to describe a pattern.

.M represents the collection of all participants, including type, method, attribute *etc*.;

.N represents the basic constraints of a pattern which describes the necessary conditions that the pattern must satisfy and satisfies participant (N) ⊆ M;

.K represents the collection of variant constraints of a pattern which describes the necessary conditions that each variant should satisfy and satisfies $\forall$T∈ K: participant (T) ⊆ M. The number of elements in the collection should be more than one, namely, the pattern should have at least one variant.

There are some differences between the variant mentioned in this paper and the variant mentioned in GOF [1]: the pattern variant mentioned in GOF [1] is defined as another form of representation of a pattern; while for the convenience of identification, all forms of representation of a pattern are all regarded as variants of the pattern in this paper. This leads to no difference in terms of output results, only that we hope to treat the forms of representation without discrimination for the convenience of description and identification. Below we take Adapter and Composite as examples to illustrate the description method of patterns.

1. Adapter Pattern

The operation of Adapter is completed by Adaptee. Therefore, Adapter and Adaptee should satisfy a delegate relationship, and Adapter should inherit from Target. These are the two basic constraints of Adapter pattern. Considering previous implementations of variants, Adapter should inherit from Adaptee for class Adapter; while Adapter and Adaptee are in association relationship for object Adapter. Therefore, the basic constraints and variant constraints of Adapter are as follows.

The participant collection :{ Adapter, Target, Adaptee, attribute A}

The basic constraints: inherit (Adapter × Target) $\wedge$ delegate (Adapter × Adapteee）

The variant constraints for class Adapter: inherit (Adapter × Adaptee)

The variant constraints for object Adapter:$\exists$attribute A: has Attr (Adapter × attribute A) $\wedge$ is Type (attribute A × A daptee)

2. Composite Pattern

Composite constructs a “tree” structure through the relations about Component, Leaf and Composite. Therefore, the inheritance relationships between Component and Leaf and between Component and Composite are necessary. When the operation of child nodes is implemented in the Component class (known as the “parent-based” Composite), there is a aggregation relationship between Component and Component. When the operation of child nodes is implemented in the Composite class (known as “child-based” composite), there is a aggregation relationship between Composite and Component. Therefore, basic constraints and variant constraints of Composite are shown as below.

The participant collection :{ Leaf, Component, add Child, remove Child, and get Child}

The basic constraints: inherit (Leaf × Component) $\wedge$ inherit (Composite × Component)

The variant constraints for “parent-based” Composite: aggregation (Component × Component) $\wedge\exists$add Child: has Method (Component × add Child) $\wedge$ has Param (add Child × Component) $\wedge\exists$remove Child: has Method (Component × remove Child) $\wedge$ has Param (remove Child × Component) $\wedge\exists$ get Child: has Method (Component × get Child) $\wedge$ return Type (get Child × Component)

The variant constraints for “child-based” Composite: aggregation (Composite × Component) $\wedge\exists$add Child: has Method (Composite × add Child) $\wedge$ has Param (add Child × Component) $\wedge\exists$remove Child: has Method (Composite × remove Child) $\wedge$ has Param

(remove Child × Component) $\wedge\exists$ get Child: has Method (Composite × get Child) $\wedge$ return Type (get Child × Component)

4.3. Pattern Matching

The methodology of pattern matching in this paper is to input the features of participants, basic constraints and variant constraints of the pattern to be identified into a library of pattern features, then analyzes the source code to generate the representations of pattern participants, use predicate matching tool to carry out the matching work, and finally filter the matching results and get the final collection of pattern instances.

Agreement: each pattern, each constraint and each variant constraint is assigned with a unique identifier to identify it.

Definition 2: The function dpid(N) returns the unique identifier of the pattern that constraint N belongs to.

We use the two-tuple <vid, insts> to represent an instance that satisfies the specified constraints. Among them, vid represents the unique identifier of the constraint; while insts represents a collection of the two tuple <pv,ps>,which describe the mapping relation between constraint participants and participant instances. pv represents the participant in the constraint, while ps represents the instance corresponding to the pv participant.

The matching begins from constraints (including basic constraint and variant constraint), and we need to gradually merge the instances that satisfy the constraints according to the clustering approach and finally deduce the pattern instances. Therefore, we should know which constraint instances are "relevant" and give the following definition.

Definition 3 Homologous Constraint instance: The constraint instances A and B are homologous; and mark A$\backsim$B only when dpid (A)=dpid (B) $\wedge$ $\forall$<pvi,psi>$\in$A.insts,<pvj,psj>$\in$ B.insts: pvi=pvj $\rightarrow$ psi=ps.

Because homologous constraint instances have no transitivity (namely, A$\backsim$B and B$\backsim$C cannot reach the conclusion A$\backsim$C), we extend the definition of multiple homogenous instances: R1, R2……and Rn are homologous, when and only when $\forall$1≤i, j≥n: Ri$\backsim$Rj. According to the definition, two and multiple homogenous constraint instances can serve as the basis for deducing pattern instances.

Pattern detecting steps:

1. Preparation Step: analyze the participants of the pattern to be identified, define the predicates of basic constraints and variant constraints according to known variants of the pattern, assign a unique identifier to each basic constraint and each variant constraint, and then input the information into the library of pattern features. Data structures recorded in the library of pattern features are described by domain model as shown in Figure 2.

2. Step of Source Code Analysis: analyze the source code, ignore all information irrelevant to identification, identify the participants in the source code and the relationship among participants, input them into the information library of source code, and use the domain model to describe the organization mode of information as shown in Figure 3. Dynamic analysis needs more complete testing cases, which is a difficult issue for the majority of software system. In this paper, we use the method of static analysis to analyze the control flow and data flow and thus extract the data of method invocation. Specific methods have been discussed in [4], so we will not explain them again here.

3. Matching step: take the constraint predicate as a unit, use first-order logic tools to match the information in the library of source code with the information in the library of pattern features, get a collection of instances of each constraint. In the matching step, we do not care whether the constraint itself is a basic pattern constraint or a variant constraint, but care that which participants in the information library of source code are matched with the constraints in the library of pattern features.

4. Filtering step: use the clustering method to merge the pattern candidates based on the standard that homologous constraint instances are similar. There are several clusters as follows:

1) Contain a basic constraint + a variant constraint: merge the participants of two instances, get a complete collection of instances of pattern participants, and output the information according the method <unique pattern identifier, unique variant identifier, pattern instance >

2) Contain a basic constraint + several variant constraints: namely, both two variants satisfy the constraint conditions, which mean that the definition of pattern variant constraint has ambiguity; print a warning of ambiguity.

3) Contain 0 basic constraints + one to multiple variant constraints: do not satisfy the pattern conditions, throw them away.

4) Contain only one basic constraint: they may be variants that are not defined in the library of pattern features, and may not be the instances of the pattern. Under this circumstance, the cluster is handed over to the software engineer for judgment.

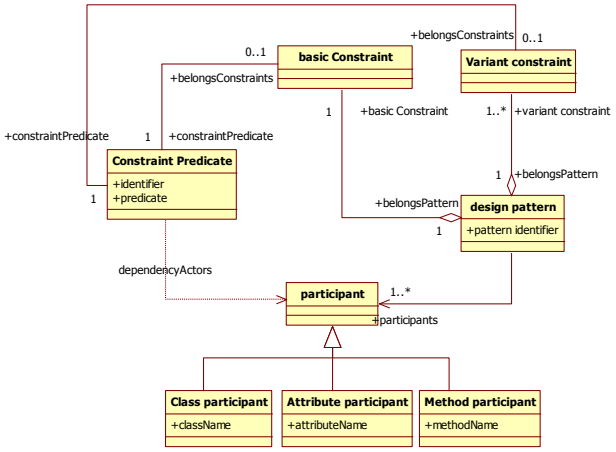

Figure 2. Domain Model for the Library o Pattern Features

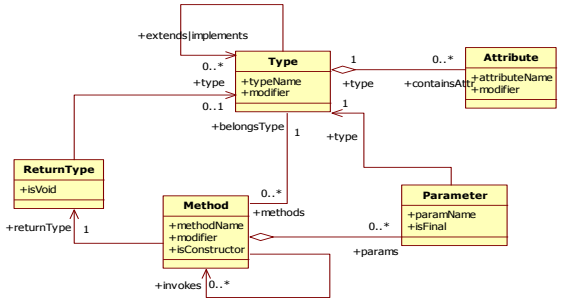

Figure 3. Domain Model for the Information Library of Source Code

5. Tool Development and Experimental Results

We have developed an extraction tool DPET4V (Design Pattern Extract Tool for Variant) according to the approach described in this paper. The extraction tool uses the JJ Tree[12] of Java to analyze Java source codes, analyzes and builds the information library of source code on the basis of AST; represents the information library of source code and the library of pattern features into the form of predicate; uses the open source tool Drools [13] to perform rule matching; and

finally uses the Apache Mahout[14] to merge the results into clusters and generates the result report.

Experimental results are measured according to following standards:

1) True Positive (tp for short): the number of instances that are manually verified as true in the result report;

2)Recall (rec for short): the number of instances that are manually verified as true in the result report/the number of instances actually existing in the software system;

3) Total (tot for short): the number of instances contained in the result report;

4) Precision (pre for short): the number of instances that are manually verified as true in the result report/the number of instances in the result report;

5)Total Precision (tpre for short):for a concrete pattern, the number of its instances manually verified as true/the number of its instances in the final report;

6)Total Recall (trec for shot): for a concrete pattern, the number of its instances manually verified as true /the number of its instances actually existing in the software system.

To obtain the analytical results, we choose the Composite pattern and Adapter pattern as examples, detect them from the JavaIO, AWT in the open source library JDK5.0, and a open source modeling tool ArgoUML. The results are shown as Table1.

Table 1. Experiment Result 1

	JavaIO				AWT				ArgoUML			
	tot	tp	pre	rec	tot	tp	pre	rec	tot	tp	pre	rec
Composite	0	0	100	100	2	2	100	100	3	2	67	100
Adapter	2	2	100	100	10	2	11	100	50	6	12	100

Take the Total Precision and Recall as evaluation criteria, and compare the identifying results of DPET4V with PINOT [15], PTIDEJ [16], SSA [5] and DPJF [9]. The results are shown as Table2.

Table 2. Experiment Result 2

	PINOT		PTIDEJ		SSA		DPJF		DPET4V	
	tpre	trec	tpre	trec	tpre	trec	tpre	trec	tpre	trec
Composite	20	75	--	--	100	25	100	50	80	100
Adapter	15	100	13	100	12	100	14	100	16	100

For the identification of Composite pattern, DPET4V maintains 80% recognition accuracy under the condition of 100% Recall and shows some advantages compared with other tools. DPJF uses appropriate formal method for Composite pattern and achieves 100% recognition accuracy, but there is still a pattern instance missed out.

For the identification of Adapter pattern, even though DPET4V shows a slim advantage in recognition accuracy, the rate reaches only a relatively low level of 16%. It is because that the pattern features of Adapter, Mediator and Bridge are delegate relation and the instances that satisfy the delegate relation far outnumber the actual existing instances. How to filter these instances or how to define these patterns more accurately is a topic worthy of further studies.

6. Conclusion and Future Work

This paper presents an approach to extract the design pattern variants based on constraints. This approach improves the recognition accuracy of design pattern instances while maintaining the omission rate unchanged and recognizes which patterns or variants the design pattern instances belong to, which is of crucial importance to design recovery of software system. Meanwhile, the work of this paper also shows certain significance for extracting anti-patterns and restructuring software architectures.

For the work in this paper, further researches need to be done: 1) due to different ways of implementation, there are many varieties of patterns, so it is difficult for software engineers to enumerate them one by one, thus leading to the result that some recognition still needs human judgment. How to reduce or even remove human intervention is a topic in need of further discussion. 2) How to improve the formal methods of design patterns and enable them to represent various variants of patterns more flexibly is also an issue for further research.

Acknowledgment

This research was supported by a grant of the Fund of the National Natural Science Foundation of China (No.6110016), the Project of Education Department of Zhejiang Province (No. Y201018837). The authors would like to express their sincere appreciation to these supports.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object- Oriented Software", Addison W- esley, Menlo Park, CA, **(1995)**.
[2] F. A. Fontana, M. Zanoni, A. Caracciolo, "A benchmark platform for design pattern detection", // PATT ERNS 2010, The Second International Conferences on Pervasive Patterns and Applications,**(2010)**, pp. 42-47.
[3] F. Tie, L. Wen-jin, Z. Jia-Chen, "Research on Design Pattern Detection Technology Towards Java", Computer Engineering and Applications, vol. 41, no. 25, **(2005)**, pp. 28-33.
[4] L. Wen-jin and W. Kang-jian, "Research on detecting and validating design pattern instances from source code", Application Research of Computers, vol. 29, no. 11, **(2012)**, pp. 4199 -4205.
[5] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides and S. T. Halkidis, "Design pattern detection using similarity scoring", IEEE TSE, vol. 32, no. 11, **(2006)**, pp. 896–909.
[6] G. Rasool and H. Akhtar, "Discovering variants of design patterns", Journal of Basic and Applied Scientific Research, vol. 3, no. 1, **(2013)**, pp. 139-147.
[7] G. Rasool and P. Mader, "Flexible Design pattern detection based on feature types", //Automated Software Engineering (ASE), 26th IEEE/ACM International Conference, **(2011)**, pp. 243 - 252.
[8] K. Stencel and P. Wegrzynowicz, "Detection of diverse design pattern variants", // Pro. of the 2008 15th Asia-Pacific Software Engineering Conference, Washington, DC, USA: IEEE Computer Society, **(2008)**, pp. 25–32.
[9] K. Stencel and P. Wegrzynowicz, "Implementation variants of the singleton design pattern", On the Move to Meaningful Internet Systems: OTM 2008 Workshops, **(2008)**, pp. 396-406.
[10] A. Binun, G. Kniesel, "Joining forces for higher precision and recall of design pattern detection", CS Department III, Uni. Bonn, Germany, Technical report IAI-TR-2012-01, **(2012)**.
[11] A. Binu and G. Kniesel, "DPJF - Design Pattern Detection with High Accuracy", CSMR **(2012)**, pp. 245-254.
[12] JavaCC. https://javacc.java.net/
[13] Jboss Drools. http://www.jboss.org/drools/
[14] The Apache Mahout Project.http://lucene.apache.org/mahout/.
[15] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from java source code", in ASE'06. Washington, USA: IEEE Computer Society, **(2006)**, pp. 123–134.
[16] Y. G. Guéhéneuc, "A reverse engineering tool for precise class diagrams", in CASCON'04, IBM Press, **(2004)**, pp. 28–41.

