Java Source Code Static Check Eclipse Plug-in Based on Common Design Pattern

Zhang Haotian, Liu Shu
School of Software
Harbin Institute of Technology
Harbin, China
socool.king@gmail.com, sliu@hit.edu.cn

***Abstract*—Design patterns represent the best practices used by experienced java developers. Design patterns abstract reusable object-oriented software designs which are solutions to general problems that software developers faced during software development. Each pattern solves design problems that occur in every day development. The detection of design patterns during the process of software reverse engineering can provide a better understanding of the software system. The most recent approaches in design pattern detection rely on the abstract syntax tree representation of the source code for fact extraction. This paper proposes a new way to detect design pattern in Java applications by analyzing the java source code. This paper discussed about the relations both between classes and objects and established the derivative design pattern about model property. Using the roles, responsibilities and collaboration information of each design pattern define static definitions. The static definitions are used to find candidate instances during the static analysis by using BCEL. After the static analysis the found candidate instances are validated by using the dynamic behavior of design patterns. We instrument the Java byte code of the application we are analyzing with additional code fragments and extract method traces from the running application. These method traces represent the dynamic facts of an application. The several restrictions are presented to define design patterns dynamically. Four common patterns of original GoF design patterns are tested and the results are analyzed which give better results in detecting design patterns comparing with other tools. In addition a Java source code static checking tool, an Eclipse plug-in, is implemented. Finally several java applications source codes were analyzed by using the plug-in, and verified the feasibility of the approach.**

Keywords-design pattern; Eclipse Plug-in; BCEL;

I. INTRODUCTION

As software system becomes more complicated and larger, the difficulty of analyzing and understanding its design and architecture also grows accordingly. Design pattern has been regarded as one of the most useful and universal tool to document existing designs. Design patterns capture solutions which can introduce simple and elegant solutions to specific problems in object-oriented software design [1]. For engineers, design patterns are fundamental abstractions to get a better overview of the system without having to know about the source code in detail. Moreover, the detection of design patterns leads to the improvement and construction of well-structured, maintainable, and reusable software systems.

Due to intangibility, identifying the architecture of software system might require a lot of efforts and experience of software developing. The myriad variations with which design pattern implements makes them difficult found in source code [2]. This paper provides the solution to solve the problem of finding all pattern realizations used in the program and then suggest developers applying more suitable pattern to improve code quality. So the time and energy needed by developers to understand a software system decrease because developers have access to the detailed information about the architecture of the application. The automation of this task can contribute to detect improve the software architecture. This paper introduces a novel system, called Design Pattern Automatic Detection (DPAD), which addresses the problem of automatically retrieving the exploiting design patterns in system.

Although design pattern can be implemented in variable forms, most of design patterns can be formalized and described in unified format [3]. So it enables us to analysis system architecture in abstraction level and detect patterns through modeling comparison. In the proposed methodology, system analyzer would prefer to describe design pattern in terms of graph and utilize graph similarity algorithm to calculate the similarity between system architecture [4]. In this paper, we replace graph with relational calculus to simplify the approach of recording and comparing design pattern.

One of the greatest challenges in pattern detection is to excavate the relations between classes. All elements found in detection process can be divided into two parts: roles and responsibilities separately [5]. Role is commonly represented by a class and the responsibilities are coded in the classes with attributes and methods. In class level, the responsibilities depict the relations between classes. To handle this issue, the program deploys a Java bytecode manipulation framework (i.e. BCEL) to provide detailed information concerning the static structure of the system.

Another difficulty in pattern detection is to establish the unified form of design pattern. Gang-of-Four (GoF) firstly introduced 23 design patterns in the book Design Patterns: Elements of Reusable Object-Oriented Software. So how to distinguish these patterns and how to resolve different combinations of these design patterns in one system are both critical problems [6]. According to the relation among classes, we can define these design patterns by applying relational calculus to depict the relationships between classes. And then, composing the formula of the entire system may introduce some difficulty especially when the

978-1-4799-2883-5/13 $31.00 © 2013 IEEE
DOI 10.1109/WCSE.2013.30

analyzed system applies multiple design patterns [7]. So we partition the whole system into clusters of hierarchies and match the pattern in the scope of subsystem instead of entire system. This paper tests DPAD with 4 common patterns of original GoF design patterns.

DPAD process can be divided into structural information extraction part and static information refining part. The first part focus on the refining the basic system structure and extracting information like inheritance, fields and methods. As for static part, it focus on analyzing the detailed interactions between classes. The connections among the classes are call, delegation or inheritance relations [8].

As objects are created at runtime, relations among objects are dynamic by nature [9]. However, dynamic analysis cannot predict all branches and loops at run time, especially when the program requires user interactions. Therefore, it would be hard to fetch all the implemented design pattern in source code by dynamic analysis. In most cases, we are able to determine whether the user apply specified design pattern depending on the source code structure. Hence, only analyzing the static structure and interact behavior of the system is enough to detect and extract some basic forms of design pattern.

Although design pattern detection can help engineers to improve code readability and speed up development process to some extent, its benefit would not be that remarkable without being intergraded with development environment. In order to increase efficiency and productivity of Java programming, this paper designs and implements Eclipse plug-in combined with DPAD to solve this problem. After being integrated with Eclipse platform, programmer can utilize DPAD to analysis the design pattern of current building project. However, there isn't a design pattern detection tool can be used directly with the Eclipse platform now. Therefore, this paper illustrates the method to design DPAD and combine it with the Eclipse plug-in.

II. DESIGN PATTERN DETECTION

A. Overview

The decomposition and analysis of design pattern instance is defined by a tuple of program elements such as classes, methods and attributes, which represent the roles and rules of a certain design pattern. This chapter will use the Adapter design pattern as a running example to present all features of our approach.

We define roles of a design pattern according to the distinguished elements in program. Note that a single class can fulfill more than one role in more than one pattern (e.g., Target, Adapter and Adaptee are roles of classes in the Adapter Pattern) [10]. The rules represent the actions among roles such as Adapter could call the functionality implemented in Adaptee without implementing this method itself.

For the understanding of a system's architecture and the detecting of design patterns, the operation to perform object-oriented analysis is essential and prior to other analysis. UML diagram has been widely used as a mature form to represent the fine granularity class relationships [11]. However, we need to filter the candidates of classes according to their roles in specific design pattern and abstract the high level communication pattern inside components.

Fig. 1 illustrates the conversion from overall UML diagram to the standard structure of design pattern. The left graph shows all the classes and their relations in UML model. The right graph depicts the UML diagram of the Adapter design pattern. After DPAD process, the classes without roles will be excluded and the classes that match design pattern instances will be presented.

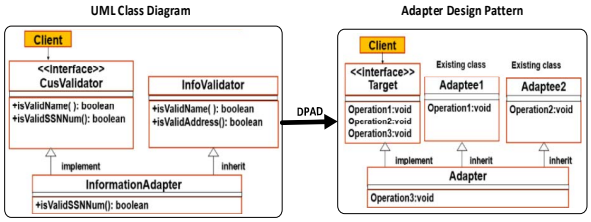

Figure 1: UML for Adapter pattern detection

As mentioned above, determining the applied design pattern not only requires to extract the roles in the system but also to verify the correctness of the rules among these roles. The intent of Adapter pattern is to covert the interface of a class into another interface clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces [12]. In general, an adapter makes one interface (Adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces. A class adapter accomplishes this by inheriting privately from an Adaptee class. The adapter then expresses its interface in terms of the Adaptee's. The Adapter design pattern consists of four roles:

- Target: defines the domain-specific interface that Client uses.
- Client: collaborates with objects conforming to the Target interface.
- Adaptee: defines an existing interface that needs adapting.
- Adapter: adapts the interface of Adaptee to the Target interface.

B. Relation Calculus Analysis

In order to simplify the process of pattern detection, it is necessary to standardizing and formalizing the expression of the architecture of both the system being checked and the specified design patterns. Although the class relationship graphs of software architectures can provide us visual presentation, analyzing and matching corresponding graphs requires lots of graph knowledge and complicates the detection process. We found that using relational calculus to model the relations among classes in object-oriented design is feasible. The critical principle is that the class diagram can be straightforwardly refined into relational calculus. On top of that we can define rules which can represent specific relationship.

All these design patterns we discussed cannot not be simply retrieved without resolving the internal relationships among classes. On the class level, we distinguish two relationships: inheritance and contain. On the object level the

relationships that we have chosen to represent includes: create, call and reference. Based on these relationships, we can define the common used design pattern with unified formulas. Unifying Theories of Programming (UTP) deals with program semantics and explains a wide variety of paradigms [13]. Following lists some basic UTP expressions:

R :: = BR (basic predicates)
| R & R (and)
| R || R (or)
| R => R (implication)
| R ; R (composition)
| R <b> R (conditional)
| !R (negation)

This paper extends this language to include basic Java class and object relations. Following illustrates the extended UTP formulas:

BR :: = Inh(X, Y) (inheritance)
| Imp(X, Y) (implementation)
| Use(X, Y) (use)
| Con(X, Y) (containment)
| Cal(X, Y) (call)
| Cre(X, Y) (creation)
| Ref(X, Y) (reference)

As shown in the table below, Inh(X, Y) and Imp(X, Y) depicts the relationship between super class or interface and its children class. Con(X, Y) represents that X contains Y in the static level. Cal(X, Y) means that X would call a method in class Y. Ref(X, Y) indicates that X quote Y.

C. Design Pattern Plug-in Architecture

Our approach in detecting design patterns is divided into static code parse part and class relationship analysis part. Fig. 2 presents the different stages of our approach.

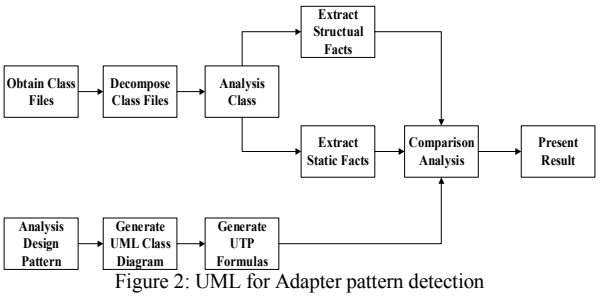

Figure 2: UML for Adapter pattern detection

DPAD plug-in can be divided into two main layers. The key logical layer is mainly implemented by functionalities provided by DPAD. It is the engine of the entire project whose is to control the detection request, response and data flow throughout the execution. Its main purpose is to detect project architecture and match the given design pattern with the analyzed project. The second layer is composed by Eclipse plug-in components which provide graphical presentation of the analyzed project. This layer is dependent on the Eclipse development environment and this project utilizes suitable extend point and components to achieve tracing and presentation functions.

For DPAD part, in the first stage we need to locate and obtain all the class files of selected application project. The second stage is static analysis. In this phase, on the basis of UML class diagrams which depict static definitions of the design pattern, we build the corresponding relational calculus. Afterwards we instrument the bytecode of these class files by utilizing BCEL. The results include class names and attributes of the classes which will be processed to find possible candidate instances that match the UML notation of design pattern. In the third stage we refine the dynamic relationships among classes to create more accurate facts and establish corresponding relational calculus formulas of extracted static facts. In the last stage, we process the facts from the candidate instances from the static analysis together with the dynamic definition. And then verify the candidate instances if they match the dynamic definition.

III. TECHNOLOGIES USED TO IMPLEMENT DPAD

A. BCEL

The Byte Code Engineering Library (BCEL) is a project sponsored by the Apache Foundation previously under their Jakarta charter to provide a simple API for decomposing, modifying, and recomposing binary Java classes (i.e. bytecode).

This project utilizes BCEL API, formerly known as Java Class, which is a toolkit for the static analysis and dynamic creation or transformation of Java class files. It enables developers to extract some useful internal details of the Java class file. BCEL is written entirely in Java and freely available under the terms of GNU Library Public License (LGPL).

B. Eclipse Plug-in

Eclipse is an open-source software framework written in Java. In its default form it is a Java Integrated Development Environment (IDE), comprised of the Java Development Toolkit (JDT) and compiler (ECJ). Fig. 3 shows that users can easily extend its capabilities by writing their own plug-in modules. We also use an intermediate graphical user interface layer called JFace, which simplifies the construction of applications based on SWT.

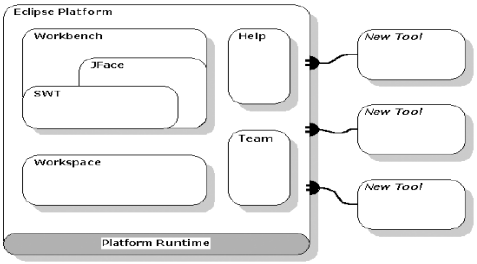

Figure 3: Eclipse Plug-in Platform Framework

JFace is a UI toolkit with classes for handling many common UI programming tasks. This project highly depends on its interesting features - actions and viewers. The action mechanism allows user commands to be defined independently from their exact in the UI. Viewers are model based adapters for certain SWT widgets, simplifying the presentation of application data structured as lists, tables or trees.

In order to provide a concise Eclipse plug-in, this project only adds Viewer and PopMenu components to compose plug-in. Moreover, for catching the user's action, we add listener to monitor the user activities.

IV. THE IMPLEMENTATION OF DPAD

A. Static Code Analysis

The first step in our detection process involves structural information extraction. For this part, tool BCEL can be utilized to extract static facts from Java classes. This constructs a fact base that contains information about the class hierarchies, interface hierarchies, attributes and methods of classes.

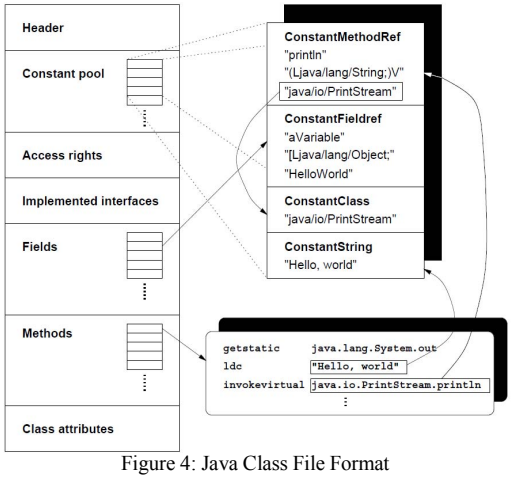

Figure 4: Java Class File Format

As Fig. 4 shown, the byte code instruction consists of different class information. We list several groups of information needed in static analysis as follows:

- Field access: The value of field access illustrates the instances of other classes the analyzed class contain or refer to.
- Implemented interfaces: This filed indicates all the interfaces which the analyzed class has implemented.
- Method invocation: Methods may be called via static references of other class. It accelerates the process to tracing the invocation of static function.
- Object allocation: Class instances are allocated with the new instruction. These new objects will describe the creation or call relationships among classes.
- Conversion and type checking: There exist casting operations which converts Super object into Child object. This field would help us to understand the inheritance hierarchy of given class.

B. Static Fact Refining

After parsing structural facts and gather the following relationships from class. The parenthesis below shows the abbreviations of these information.

- class extends class
- class implements interface
- class overrides method
- class references class
- class contains method
- class contains attribute
- method calls method
- method creates class

In general, five basic class relationships among classes and objects should be parsed: inheritance, use, create, call and reference. In general, inheritance and use can be retrieved by static parsing, while for the other three relations, they can be detected with the execution of class files. However, as mentioned above, we neglect the dynamic tracing of the execution process and take all the class relationship facts into account.

This paper unifies the fact base and reduces the facts to inherit and use relations on class level. These are the only facts we use for the static analysis. These facts represent the structure of the program in a simplified way that relates to the UML notation of class diagrams. We lift all facts about attributes and methods to class level and then store these facts in inherits or uses relation.

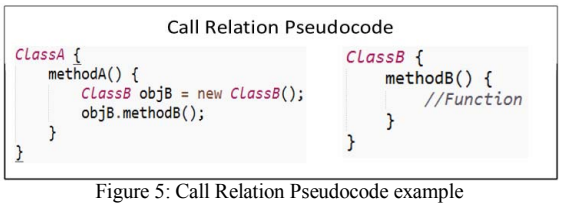

Figure 5: Call Relation Pseudocode example

As shown in Fig. 5, the pseudocode example illustrates method in ClassA calls method in ClassB. Lifting these facts to class level means we will store the information that ClassA calls ClassB in our final compare process. Only by converting the method calls between classes to class level, can this project detect the design pattern.

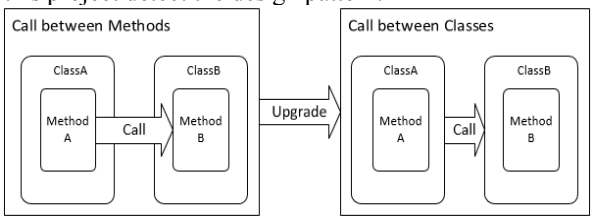

Figure 6: Lifting facts to class level

In our case lifting relations is illustrated in Fig. 6. Lifting these three facts to class level means we will store the information that Calls ClassA ClassB as our result. Besides the method call between different classes, the other forms of method calls would also be classified to class level.

Figure 7: Static Relations Standard Format

Fig. 7 shows uses and inherits relationships among classes and interfaces. In the left part of this figure, Uses ClassA ClassB declares that ClassA has a method call to ClassB, or in another example, Inherits ClassA ClassC indicates that ClassA inherits from ClassC. After normalization, these two relation are transformed to relational calculus and finally presented in the right form.

Inheritance: Although we can easily get the inheritance relation from the source code of classes, we should mention the fact that many classes have the identical father class - java.lang.obect. On top of that, many of these classes would also possibly inherit from java systematic class or other class provided by third part jar. Most of these inheritance relationships cannot represent the actual class hierarchy defined by user, especially for the four design pattern we tested. So class pool is built in order to record the user created classes. Moreover, we also filter the inheritance hierarchy by comparing their super classes with the classes in pool. So that this paper only processes the class whose super class is also created by programmer.

Moreover, Java requests that class only can inherit from one super class. Nevertheless, Java class can implement different interfaces, so that this project also analysis the count of interfaces and retrieve all interfaces it implements. And then filtering the interfaces defined by user.

Use: As for the contain relation, we also applied the class pool to retrieve the classes who satisfy such relation. However, there is one point where use relation is different from inheritance. Class can be declared as generics type or non-generics type, this project distinguishes these two kinds of contain type by using different methods.

(1) and (2) show what potential facts we store in the uses and inherits relation between class A and class B separately:

$$Use(A,B) = Call(A, B) \cup Ref(A, B) \cup Cre(A, B) \quad (1)$$

$$Inh(A,B) = Ext(A, B) \cup Imp(A, B) \cup Override(A, B) \quad (2)$$

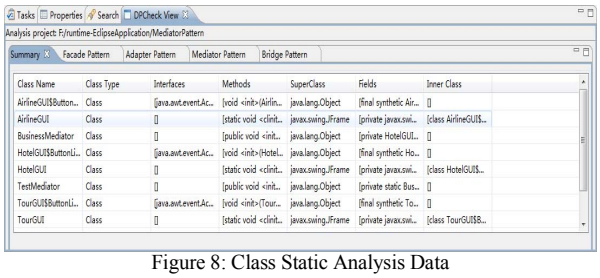

Figure 8: Class Static Analysis Data

Fig. 8 illustrates that the structural class information retrieved by DPAD structural part. Inner class column displays the list of inner class which records some classes and their inner classes separately. Although inner class belongs to the class who contain it, after compiling, they are compiled into different class files. So it is necessary to distinguish inner class from its ship class.

$$Adapter = Use(C, Adr) \,\&\, Imp(Adr, T) \,\&\, Use(Adr, Ade) \quad (3)$$

The Adapter design pattern has four roles: the Client, the Target interface, the Adapter class and the Adaptee class. As shown in (3), C, T, Adr and Ade stand for Client, Target, Adapter and Adaptee separately. We can find that complicated relational calculus can exhibit class relationships more clearly after transformation from UML class diagram. Statically we can express, that the Client uses the Adapter, that the Adapter class implements the target interface and that the Adapter uses the Adaptee. These are all static facts that make up the Adapter pattern. We use these relations and create a set of possible candidate instances that match this definition.

More detailed analyzing the use relationships among classes is especially critical for distinguishing some similar patterns. E.g. Facade design pattern and Mediator design pattern.

1) Facade:

a) Know which subsystem classes are responsible for a request.

b) Delegate client requests to appropriate subsystem objects.

2) Facade Subsystem classes:

a) Implement subsystem functionality.

b) Handle work assigned by the Facade object.

c) Have no knowledge of the facade; that is, they keep no references to it.

3) Mediator:

a) Abstract arbitrary communication between colleague objects.

b) Centralize functionality.

4) Mediator Subsystem classes:

a) Implement subsystem functionality.

b) Keep references of Mediator.

c) Communicate with the mediator.

It is obvious that Facade and Mediator design pattern have a lot of similarities and the structures of both patterns are all star-like. Both of them have a central control class, which is facade class and mediator class separately. From perspective of their design architecture, the difference between these two kinds of patterns is that the relationships between these two central class and their subsystem classes are different. As for Mediator Pattern, Mediator would create subsystem objects and call their functions and these subsystem objects would also use Mediator while executing. However, Facade class will keep an instance of each class and invoke their methods when needed. From the analysis above, we can conclude that it would be easy to make a distinction between these two patterns.

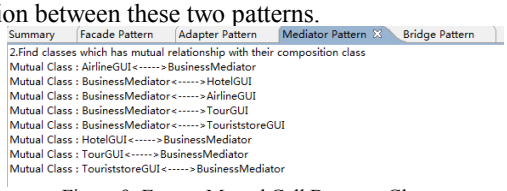

Figure 9: Extract Mutual Call Between Classes

From Fig. 9, for that BusinessMediator keep an instance of every class and each class also would call BusinessMediator, DPAD could retrieve this relation easily and conclude that this project satisfies Mediator pattern.

The relational calculus of Facade pattern is listed in (4):

$$Facade = Use(C, F) \,\&\, Use(F, Sub1) \,\&\, Use(F, Sub2) \quad (4)$$

C, F, Sub1 and Sub2 stand for Client, Facade, Subsystem1 and Subsystem2 separately.

The relational calculus of Mediator pattern is nearly the same with Facade pattern.

$$\text{Mediator} = \text{Use}(C, M) \,\&\, \text{Use}(M, \text{Sub}) \,\&\, \text{Use}(\text{Sub}, M) \quad (5)$$

In (5), C, M and Sub stand for Client, Mediator and Subsystem separately. Facade differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa. In contrast, Mediator enables cooperative behavior that colleague objects don't or can't provide, and the protocol is multidirectional.

Finally, by comparing relational patterns of specific design pattern and analyzed project, we can find the design pattern applied in analyzed project.

V. CONCLUSION

Practice has fully shown that DPAD can correctly detect design pattern in Eclipse platform and make relevant suggestions for programmer to improve their code quality. It has the following advantages:

The advantages of this approach are:

- Combined with Eclipse platform: Programmer can easily analysis the design pattern of specified project while programming, and moreover, in some cases, it may result in the optimization of software architecture and reduction of bugs or internal errors.
- Detection based on relational calculus: Programmers and computer scientists with mathematic background are intuitively more familiar with relational calculus than other forms of model.
- No need to access source file: Instrumenting the bytecode rather than the source code directly has many advantages. Although analyzing the source code may be more efficient for us to retrieve the detailed information about the class and object, we shouldn't ignore the fact that these classes may not pass the compilation of JVM. In addition, there might exist the possibility that some classes or methods share the same name so that it is difficult to fetch the actual relationship between different objects.
- Project-oriented: This tool would analysis the selected project instead of a standalone class file. In most of real projects, there are different hierarchies of files so that only analyzing one of these files may be not enough to get an overall grasp of whole project. This rule especially applies for analyzing the relationship between different packages or classes. And Eclipse plug-in can provide corresponding report based on project according to the user's selection.

For future work we definitely see potential to improve the process of dynamic detection. For example, currently, DPAD can only detect four basic kinds of Java design pattern, so expanding DPAD to include more design patterns can satisfy the growing needs from programmers. Moreover, if there are design patterns used in an application but they are never called during run-time, we will not be able to detect them via DPAD. Afterwards, we need to use run-time tool to analysis the call process.

ACKNOWLEDGMENT

We would like to thank Ting He and Yushan Sun from Harbin Institute of Technology, China for their collaborations and useful hints.

REFERENCES

[1] Vlissides, John, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," Reading: Addison-Wesley, 1995.

[2] Smith, Jason McC, and David Stotts, "Elemental design patterns: A logical inference system and theorem prover support for flexible discovery of design patterns," In In Proceedings of the 9th European Conference on Arti Intelligence, 2002.

[3] Andres J. Ramirez and Betty H. C. Cheng, "Design patterns for developing dynamically adaptive systems," Proc. 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '10), ACM, New York, 2010, pp. 49-58.

[4] Tsantalis, N.; Chatzigeorgiou, A.; Stephanides, G.; Halkidis, S.T., "Design Pattern Detection Using Similarity Scoring," Software Engineering, IEEE Transactions, vol. 32, no. 11, Nov. 2006, pp. 896–909.

[5] N. Shi and R.A. Olsson, "Reverse Engineering of Design Patterns from Java Source Code," Proc. 21st IEEE/ACM, l Conf. Automated Software Eng, Sept. 2006, pp. 123-134.

[6] Babaoglu, Ozalp, Geoffrey Canright, Andreas Deutsch, et al, "Design patterns from biology for distributed computing," ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1, no. 1, 2006, pp. 26-66.

[7] Zhu, Hong, Ian Bayley, Lijun Shan, and Richard Amphlett, "Tool support for design pattern recognition at model level," In Computer Software and Applications Conference, COMPSAC'09, 33rd Annual IEEE International, vol. 1, 2009, pp. 228-233.

[8] Cheon, Yoonsik, and Ashaveena Perumandla, "Specifying and checking method call sequences of Java programs," Software Quality Journal 15, no. 1, 2007, pp. 7-25.

[9] Pettersson, Niklas, "Measuring precision for static and dynamic design pattern recognition as a function of coverage," ACM SIGSOFT Software Engineering Notes 30, no. 4 , 2005, pp. 1-7.

[10] Smith, J.M.; Stotts, D., "SPQR: flexible automated design pattern extraction from source code," Automated Software Engineering, 2003. Proceedings, 18th IEEE International Conference, Oct. 2003, pp. 215-224.

[11] Porras, Gerardo Cepeda, and Yann-Gaël Guéhéneuc, "An empirical study on the efficiency of different design pattern representations in UML class diagrams," Empirical Software Engineering 15, no. 5, 2010, pp. 493-522.

[12] Shuai, Jiang, and Mu Huaxin, "Design patterns in object oriented analysis and design," In Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference, 2011, pp. 326-329.

[13] Cavalcanti, Ana, and Marie-Claude Gaudel, "A note on traces refinement and the conf relation in the Unifying Theories of Programming," In Unifying Theories of Programming, Springer Berlin Heidelberg, 2010, pp. 42-61.