A Design Pattern Detection Technique that Aids Reverse Engineering

Hakjin Lee, Hyunsang Youn, Eunseok Lee
School of Information and Communication Engineering, Sungkyunkwan University
{hakjinlee, wizehack, eslee}@ece.skku.ac.kr

Abstract

If software design-patterns could be captured and reused then this would be very helpful to reverse-engineering often practiced by those who develop and who maintain software. The ad-hoc nature and informality of this reverse-engineering process however, makes the discovery of these patterns not straightforward. Moreover, a high false positive rate results from trying to detect these design-patterns. Although several static and dynamic analysis approaches have been proposed to overcome these difficulties, each technique cannot be used separately because of different reasons. And, even if this were possible, each technique in isolation cannot address detection of all of the important patterns. We propose a new taxonomy of GoF design patterns that can guide the reverse-engineering process. This new approach not only combines static analysis with dynamic analysis but also adds what we call the implementation-specific analysis. Using it we demonstrate that the reverse engineering process is faster and more accurate.

1. Introduction

The need for software maintenance has never been as high as lately, particularly for discovering bug, adding function, and so on. The lack of documentation leads to high costs of software maintenance. This has prompted numerous research efforts at the reverse engineering of software [1][3][6][7].

A design-pattern [2], generally speaking, is a reusable solution to a commonly occurring problem in software design. The term has become common parlance to software designers, requirement engineers, and software coders alike. The language of design patterns is now necessary for an understanding and manipulation of software as it helps us to understand the design-intent of the pre-developed software. Hence, of reverse engineering and of the re-engineering of software [1][3][4][5][6][7][11].

The aim of static analysis [4] is to elucidate the structural aspects of software without the program execution such as the relationship between the many software elements that construct a software module. Dynamic analysis, on the other hand, elucidates dynamical aspects of software during the program execution such as the passing of messages between objects in a configured software module.

Many attempts have been made to detect objects by the reverse engineering technique using both methods. Usually, this follows a two-step approach. Static analysis is used in the first step to obtain candidate patterns and dynamic analysis as a second step to discover which of these candidates is the relevant pattern. This two-step approach assists in discovering what the intention of the software module is. Never the less, this approach suffers from serious drawbacks to its practical implementation: phenomenal time, search space complexity, and inaccurate detection (false alarms).

This paper presents a novel algorithm; it is more of a methodology for detection of design patterns that is rooted in a reverse engineering method. The paper also reclassifies the GoF pattern. The GoF(Gang of Four) [2] pattern is well known to be very helpful to the design-pattern detection with the reverse engineering method. In this paper, we propose the GoF pattern detection technique.

Section 2 discusses existing research and techniques that are able to detect a design-pattern in software. Section 3 introduces the novel design-pattern taxonomy that better supports the reverse engineering process. This section also presents our algorithms and processes to detect design-patterns. Section 4 evaluates our new technique. Section 5 contains the conclusions and the pros and cons of our new approach and what remains to be done in terms of future research.

2. Related Work

The main purpose of detecting design patterns from the view of reverse engineering is to obtain advantages in being able to understand many parts of the program. However, difficulties in detecting practical design patterns arise from the following:

(1) Non-formalization of pattern: GoF patterns are classified in a pattern catalogue represented by natural language and object-oriented graphical notations. However, the representation of such way can be ambiguous and often mislead to understand and apply each pattern. Hence, there should be a need for formal means of accurately describing patterns to allow users to know precisely when and how to use them. This is one of the main difficulties that make pattern detect hard.

(2) Large base of source code: Even if we barely use pattern specification or formalization, it places a large burden on the maintainer since source codes to comprehend are too large.

(3) Inconsistency between design structure and intent: Even if the detected pattern instance corresponds to their structure, often design intents are entirely different, such as the strategy pattern and the state pattern.

(4) High False-positive Rate: Due to the above reasons, even the detected pattern is not entirely accurate.

Thus, we propose a new approach combining static and dynamic analysis as well as implementation-specific analysis to resolve such difficulties.

Generally, design patterns are composed of two parts, structural and behavioral. Thus, we can easily understand pattern intent, considering the two aspects. Most existing approaches [4][5][6][7] consider such aspects of patterns and use static or dynamic analysis or a combination to detect each design-pattern. The approach using only static analysis is very difficult to distinguish among the patterns having similar structure with high false-positive rate. Differently, the approach using only dynamic analysis needs too many searching space to read source code and requires the well-arranged testing environment. The approach combining only both static and dynamic analysis is time-consuming. A different approach is employed to catch design intent [3]. In spite of the efforts from various approaches, they still have very high false-positive rate for design patterns detected.

We also recognize structural information for design patterns using static analysis, and behavioral information through dynamic analysis. However, the process and algorithm for

design-pattern detection are entirely different from that of other research. Our novel approach can reduce the overload from unnecessary testing environment for every pattern since we do not always need to execute the program to detect each pattern. Accordingly, it can obtain the economy of both time and space effectively. The proposed method can achieve better performance and greater accuracy in detecting design patterns.

3. Proposed Approach

Originally, Gang of Four (GoF) patterns are classified into three types, according to purpose of their usage. There are creational patterns, concerned with the process of object creation, structural patterns, related to relation of interaction among classes, and behavioral patterns, related to program behavior. This classification definitely supports forward engineering.

However when we examine the relevance for reverse engineering from the view of this classification, with all 23 GoF design patterns, this does not directly support reverse engineering. Some research also proved this claim [3]. Thus, we reclassify existing taxonomy into new three types of pattern so that can use for supporting reverse engineering. Table 1 addresses the new taxonomy.

(1) Static Structural Patterns: These patterns are detected by the relationship among classes. The three types of relationship may be generalization, realization, association and aggregation. These patterns can be formalized by the UML class diagram representing each relation. It is possible to detect this by using static analysis.

(2) Dynamic Behavioral Patterns: The patterns are implemented through interaction between various instances of objects for class implementation. Even though they are structurally similar to some other patterns, the patterns are made for their own design intent or communication method. It is possible to detect these patterns by combining static and dynamic analysis.

(3) Program-Specific Patterns: When these patterns are implemented into source code during software development, they use specific keywords or coding styles. These patterns cannot be detected by both static and dynamic analyses. So, this type of patterns makes it possible to detect by checking their specific keywords or coding styles from source code.

Table 1. Pattern Taxonomy for Reverse Engineering

Pattern Type	GoF Design Pattern
Static Structural Pattern	Proxy, Adapter, Façade, Bridge, Visitor, Composite, Template Method
Dynamic Behavioral Pattern	Decorator, Chain of Responsibility, Factory Method, Mediator, Builder, Observer, State, Strategy, Abstract Factory
Implementation-Specific Pattern	Singleton, Flyweight, Iterator, Command, Interpreter, Prototype, Memento

This Fig.1 shows the overall process to detect design patterns based on predefined taxonomy. It can help detect all GoF design patterns in the taxonomy. We apply only static analysis to detect if it is a static structural pattern, both static and dynamic analysis if it is a dynamic behavioral pattern, and implementation-specific analysis for other cases. Here we addresses more detailed steps to detect all patterns.

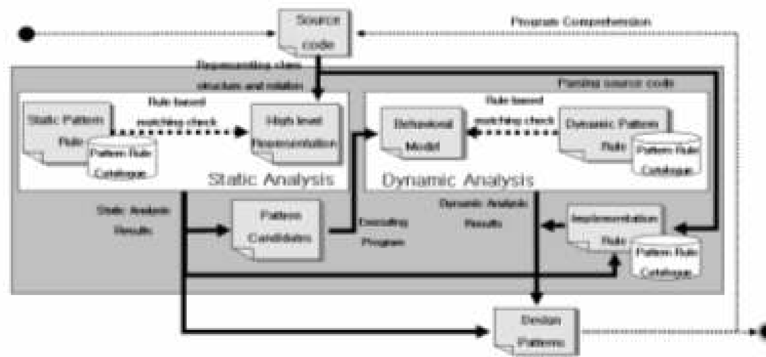


Fig 1. Design pattern Detection Process

[Step 1] In this step, we represent program source code extraction into class structural relation, with high-level representation. The high-level representation can be expressed by various ways such as AST(Abstract Syntax Tree) [4], or ASG(Abstract Syntax Graph) [5].

[Step 2] Then, in the next step, with the static analysis algorithm defining each pattern, we check if the parts of high-level code representation from the previous step are correct. If satisfied, it can be stored as a result of static analysis and can be detected as a design pattern; otherwise it would be removed from the pattern instance candidate class. We detect design patterns by using our XMI(XML Metadata Interchange) [8] parser to perform the matching check.

[Step 3] The results are stored through the matching check performed in the previous step, to define static structural patterns or candidates of the dynamic behavioral pattern. A purpose of static analysis during the recognition of dynamic behavioral patterns is to reduce search space for the next dynamic analysis, by detecting the pattern candidate. In time, these pattern candidates represented as the results of static analysis later become the input of dynamic analysis. However, static analysis for detecting dynamic behavioral patterns is followed by the previous steps, using such candidates as inputs.

[Step 4] The program is executed. Then, we collect and monitor their method call tracing. The method call information can be obtained through the JDI(Java Debug Interface) [12]. During this work, we conduct a matching check if methods are followed with predefined pattern behavior rules. If this check is not satisfied, it will be removed from pattern candidates, and will be detected as a pattern instance.

[Step 5] In this step, the implementation-specific pattern is detected. We have information of specific keywords or coding styles for each pattern implementation in the pattern catalogue.

Through following this detection process for each design-pattern, software maintainers can understand the program much better than previously. In addition, it assists in later software re-engineering work.

According to the above detection process, we adapt our pattern detection algorithms. To address these algorithms, we examine the detection of the Builder pattern, which is one of the dynamic behavioral patterns and the Prototype pattern, which is one of the implementation-specific patterns. As mentioned before, the dynamic behavioral pattern can be detected by combining both static and dynamic analysis algorithms. And the implementation-specific pattern can be detected by parsing source code. Fig 2 shows the general structure of the Builder pattern and Fig 3 shows the behavior of the pattern.

3.1 Builder pattern Detection

This pattern separates the construction of a complex object from its representation so that the same construction process can create different representations [2]. Fig.2 and Fig.3 are UML representation of Builder pattern. First, we perform static analysis to detect the structure of this pattern.

i) Check structural relation among classes. To do this, the following tagged value in standard XMI representation of class diagram is retrieved from source code.

```
<UML:Class … IsAbstract = 'false' …>
<UML:Generalization …> …
<UML:Association …> …
<UML:Aggregation …> …
```

ii) Check that the class performing the role of Director (one of the participants of Builder pattern) has the member field of 'Builder' type. To do this, the following tagged value is retrieved from the XMI representation of the class diagram.

```
<UML:DataType xmi.id = … name = 'Builder'…>
```

iii) For dynamic analysis, execute the program, then perform method call tracing.

iv) Check if construct() method of the candidate classes plays the role of Director, calling methods of the Builder class.

v) Finally it is detected as a Builder pattern instance if above all conditions are satisfied.

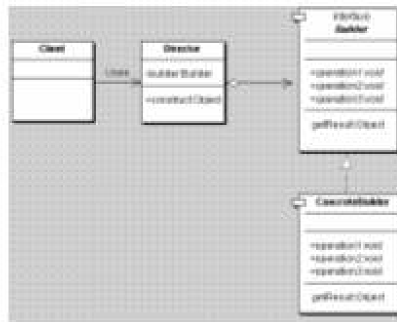


Fig. 2. Structure of Builder Pattern

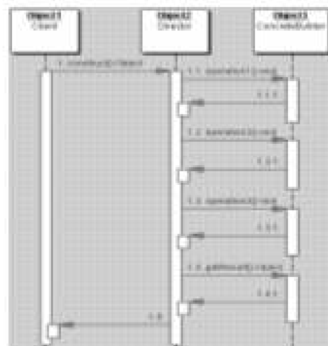


Fig. 3. Behavior of Builder Pattern

3.2 Prototype pattern Detection

This pattern specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype [2]. A representative implementation for this pattern uses the clone method which is specified in Java language specification. This method can copy only for the classes which implement Cloneable interface.

i) Check structural relation among classes. Check if the class that operates as Prototype role in Prototype Pattern, extends java.lang.Cloneable Interface.

ii) Parse the source code with a specific keyword, clone method.

iii) Find the method from source code. If the class is cloned by other class, analyze the method that contains the clone() method.

4. Evaluation

To evaluate our approach in this section, we examine our design pattern detection technique on some practical programs. We apply our pattern detection approach into different programs to assure its validity and test our algorithms.

Firstly, we intend to detect pattern instances of a multi-agent system development tool, which is called PURE (Pattern-based Ubiquitous agent development and Running Environment) toolkit [10] as a target program. This tool is an output of former research area performed by us. The tool was implemented with 527 classes and approximately about 3500 methods. During the development of this tool in the past, we designed and implemented to

use the GoF design patterns actively. Thus, it is made up of different design patterns, such as Iterator, Singleton, Builder, Command, Prototype pattern, etc.

Table 2. Experimental Objects

The Object for Detection	The Num. of Classes	The Num. of Methods (Approximately)
PURE Toolkit	527	3500
JINI-based Home Appliance System	950	6500
Project Management Supporting Tool	288	1500
MP3 Player	52	250

Next, we had a design pattern course in our university. The term-project results1 of the course work are used to test our detection approach. We divided into three project group and assign four persons per each group. Each group submitted their final source code and documents that specified design patterns used in design phase. The first group implemented a Jini-based Home Appliance Controller system. The second group implemented a Project Management Tool. The last group implemented simple MP3 player. All of these objects are implemented with the Java language. The following Table 2 represents the number of classes and methods from our experimental objects. Finally, we will compare our approach with existing detecting tool which are PINOT [3], and Fujaba [9].

For measuring the correctness of pattern detection, we consider the false positive rate (FPR) and the false negative rate (FNR) for each pattern detected. A false positive is a candidate which is not designed as a pattern instance, but is 'detected' falsely as an instance. A false negative is a candidate which is designed as an instance, but is not detected as one. We use this below formula [3] for the FPR and the FNR:

$$\textit{False Positive Rate (FPR)} = \frac{\textit{Number of Incorrect Pattern Instances}}{\textit{Number of All Detected Pattern Instances}} \times 100\%$$

$$\textit{False Negative Rate (FNR)} = \frac{\textit{Number of Undetected Correct Pattern Instances}}{\textit{Number of Correct Pattern Instances}} \times 100\%$$

We show all our experimental results. We exclude the duplicated design patterns during detection for each program. Table 3, shows the result that we obtain the output from the first experiment. The following Table 4, table 5 and table 6 represent detected pattern instances results from next three experiments. From the results of the experiments, (such as PURE, JINI-based home appliance system, MP3 Player), we found some drawbacks that the Implementation-specific pattern detection we did not consider many different ways for implementation included very high error rate. Next, Table 5 describes the comparison

[1] This website is available - http://selab.skku.ac.kr/s9832033/undergraduate/term_project_2005/results/

between the experimental results of our approach and the results of other approaches for same objects.

Table 3. The Output from PURE toolkit

GoF Design Pattern	FPR	FNR
Iterator	25.0%	39.0%
Singleton	12.6%	7.0%
Builder	22.0%	24.0%
Command	53.3%	76.0%
Memento	29.0%	11.7%
Prototype	4.5%	2.8%
Chain of Responsibility	6.0%	17.5%

Table 4. JINI-based Home Appliance System

GoF Pattern	FPR	FNR
Façade	12.0%	2.0%
Mediator	1.9%	4.7%
Observer	5.5%	6.0%
Abstract Factory	11.0%	7.5%

Table 5. Project Management Supporting Tool

GoF Pattern	FPR	FNR
Flyweight	46.8%	30.0%
Bridge	15.5%	8.0%
Visitor	14.1%	23.2%
Strategy	6.0%	3.7%

Table 6. MP3 Player

GoF Pattern	FPR	FNR
Adapter	13.3%	3.2%
Composite	7.2%	24.0%
Decorator	11.2%	29.5%
State	8.0%	23.8%
Proxy	12.5%	15.0%

Table 7. The Comparison between Pattern Detection tools

GoF Design Pattern	PINOT	FUJABA	Our Approach
Proxy	O		O
Adapter	O	X	O
Façade	O	O	O
Bridge	O	O	O
Visitor	O		O
Composite	O	X	O
Template Method	X	O	
Decorator	O	X	O
Chain of Responsibility	O	X	O
Factory Method	X	X	O
Mediator	O	X	O
Observer	O	X	O
State	O		O
Strategy	O	O	O
Abstract Factory	X	X	O
Singleton	O	O	O
Flyweight	O	X	O
Iterator		X	O
Command			O
Interpreter			
Prototype			O
Builder			O
Memento		X	

O: The tool provides detection for the pattern correctly identifies it.
X: The tool provides detection for the pattern fails to identify it.
Blank: The tool does not provide detection for the pattern

5 Conclusion

In this paper, we reported a contribution to area of software maintenance and reverse engineering by using our new taxonomy and a novel pattern detection technique. Our approach supports following two main ideas. Firstly, we reclassified all 23 GoF design-patterns to support reverse engineering. The original classification from GoF book only supports forward engineering. However, this did not help design-pattern detection in reverse engineering. But, our taxonomy was utilized as a basis of supporting this. Secondly, we proposed flexible design-pattern detection process. And also, we created detection algorithms for each design pattern. The process will help software maintainers who need to reengineer existing program or legacy system during the maintenance time. By using this, we can obtain

some advantages for comprehending the program faster and more correct than before. We also evaluated our detection techniques by adapting into various programs and comparing the results with other pattern detection tools.

However, our detection results still showed high error rate, especially in most implementation-specific patterns because we did not consider so many specific implementation methods for such type of design-patterns. Also, we could not compare process performance for our technique with other approaches because of the lack of supporting toolkit, applying our approach. As our future work, firstly, we will improve our algorithm including implementation-specific pattern to obtain better correctness of pattern detection. Secondly, we will support an appropriate toolkit so that maintainers can accomplish their tasks easily. Finally, we will extend our approach so that can detect many different design-patterns beyond the GoF pattern.

6. References

[1] G. Antoniol, R. Fiutem and L. Cristoforetti, "Design Pattern Recovery in Object-Oriented Software", In 6th International Workshop on Program Comprehension, 1998

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

[3] N. Shi, and R. Olsson, "Reverse Engineering of Design Patterns from Java Source Code", The 21st IEEE/ACM International Conference on Automated Software Engineering, 2006

[4] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe, "Automatic Design Pattern Detection", Proc. of the 11th IEEE International Workshop on Program Comprehension, 2003

[5] L. Wendehals, "Improving Design Pattern Instance Recognition by Dynamic Analysis" , Proc. of the ICSE 2003 Workshop on Dynamic Analysis, 2003

[6] J. Niere, J. Wadsack, and L. Wendehals, "Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic", Technical Report, Univ. of Paderborn, 2001

[7] Brown, K., "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk", Master Thesis, University of Illinois at Urbana-Champaign, 1997, http://www.omg.org/technology/xml/index.htm

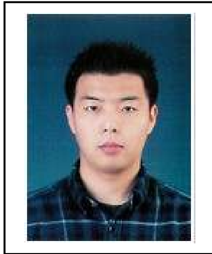
[9] FUJABA - http://wwwcs.uni-paderborn.de/cs/fujaba/index.html

[10] Hyunsang Youn, et.al, "Next Generation Agent Development Supporting Tool: Case Study. SERA, 2005

[11] N. Pettersson, "Measuring precision for static and dynamic design pattern recognition as a function of coverage", ICSE 2005 Workshop on Dynamic Analysis, 2005

[12] JDI - http://download.java.net/jdk6/docs/jdk/api/jpda/jdi/overview-summary.html

Authors



Hakjin Lee received the MSc degree in computer engineering at the Sungkyunkwan University, South Korea, in 2007. He previously worked for intelligent context aware agent platform development in the Ubiquitous Computing Research Institute. His research interests include multi-agent based system development and software reverse engineering. Currently, he is a programmer for cycle logic of air-conditioner in LG Electronics.

Hyunsang Youn is currently pursuing the PhD degree at Sungkyunkwan University. received M.S. degree in computer engineering at Sungkyunkwan University, Korea, in 2006. He obtained the B.S. degree in Information Communication Engineering at Seokyeong University, Korea, in 2004, He is a member of the Software Engineering Lab. His research interests the area of model based software performance prediction and analysis, Autonomic Computing System for Self-management, Model Driven Architecture. Currently, He works for intelligent context aware agent platform development in the Ubiquitous Computing Research Institute.

Eunseok Lee received the Ph.D. and M.S. degrees in Department of Information Engineering at Tohoku University, Japan, in 1992 and 1988, respectively, and his B.S. degree in Electronic Engineering at Sungkyunkwan University, Korea, in 1985. He is a Professor of the Department of Computer Engineering of Sungkyunkwan University, Korea. From 1994 to 1995, he was an assistance professor of the Department of In-formation Engineering of Tohoku University, Japan. He was a research scientist in In-formation and Electronics Laboratory of Mitsubishi Electric Corporation, Japan, from 1992 to 1994. His areas of research include Software Engineering, Autonomic/Ubiquitous Computing and Agent-oriented Intelligence System.