Detecting Design Patterns Using Source Code of Before Applying Design Patterns

Conference Paper · January 2009

DOI: 10.1109/ICIS.2009.209 · Source: DBLP

4 authors, including:

Hironori Washizaki
Waseda University
293 PUBLICATIONS **1,438** CITATIONS
SEE PROFILE

Atsuto Kubo
A.O.M. Corporation
28 PUBLICATIONS **212** CITATIONS
SEE PROFILE

Yoshiaki Fukazawa
Waseda University
311 PUBLICATIONS **1,207** CITATIONS
SEE PROFILE

Detecting Design Patterns Using Source Code of Before Applying Design Patterns

Hironori Washizaki†‡, Kazuhiro Fukaya†, Atsuto Kubo†, Yoshiaki Fukazawa†
†Department of Computer Science and Engineering, Waseda University
3-4-1 Ohkubo, Shinjuku-ku, Tokyo 169-8555, Japan
washizaki@waseda.jp, {hiro220, a.kubo}@fuka.info.waseda.ac.jp,
fukazawa@waseda.jp
‡GRACE Center, National Institute of Infomatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract

Detecting design patterns from object-oriented program source-code can help maintainers understand the design of the program. However, the detection precision of conventional approaches based on the structural aspects of patterns is low due to the fact that there are several patterns with the same structure. To solve this problem, we propose an approach of design pattern detection using source-code of before the application of the design pattern. Our approach is able to distinguish different design patterns with similar structures, and help maintainers understand the design of the program more accurately. Moreover, our technique reveals when and where the target pattern has been applied in an ordered series of revisions of the target program. Our technique is useful to assess what kinds of patterns increase what kinds of quality characteristics such as the maintainability.

1. Introduction

A design pattern is an abstracted repeatable solution to a commonly occurring software design problem under a certain context. Among a large number of design patterns extracted and reported from well-designed software, 23 Gang-of-Four (GoF) design patterns [1] are particularly used in object-oriented design.

Detecting (recovering) such GoF design patterns (later, simply denoted as "design patterns") from object-oriented program source-code can help maintainers understand the design of the program. Most of the conventional approaches for detecting design patterns from programs analyze the structural aspects of programs, especially inter-class relationships [2, 3, 4, 5]. Since these approaches are based on structural aspects, it is impossible to distinguish different design patterns with the same structure, such as the `State` design pattern[1] (Figure 1) and the `Strategy` design pattern[1] (Figure 2). These patterns have different aims and deal with different problems; however, structures provided by these patterns' solutions are quite similar.

To solve this problem, there are several advanced approaches. Wendehals et al. combined static and dynamic analysis[6]; however, the result of dynamic analysis depends on the representativeness of the execution sequences. Moreover, dynamic analysis techniques require executable programs beforehand. Shi and Olsson applied static program analysis techniques (such as the data-flow analysis and control-flow analysis) to the abstract syntax tree (AST) in method bodies[7]. The approach distinguishes the patterns that are structurally identical but differ in behavior. However the approach only deals with the problem-solved designs (patterns' solutions) and not the problems themselves; therefore it seems to difficult to detect distorted implementations of patterns. As an another approach, Hahsler has detected design patterns by using comments stored in version archives[8]; however such approach might overlook design pattern applications that are not intended by committers.

Thus, this paper proposes a novel technique for design patterns detection based on static analysis using source-code of before the application of the design pattern. By utilizing the source-code of before applying design patterns, our approach is able to distinguish design patterns that are structurally identical but differ in their dealing problems. This feature helps maintainers understand the design of the program more accurately.

2. Detecting design patterns by code of before applying patterns

Source-code sometimes has parts to where design patterns can be applied, and we use these parts to detect design

978-0-7695-3641-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ICIS.2009.209

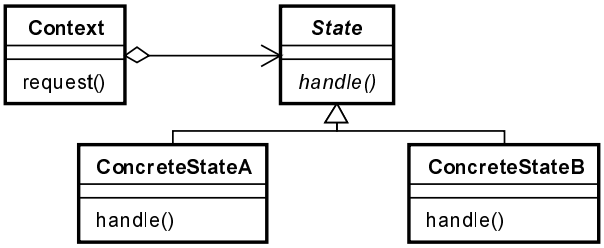

Figure 1. `State` **design pattern**

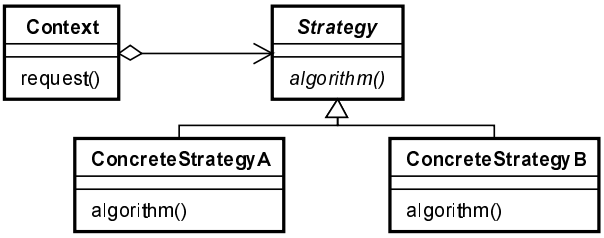

Figure 2. `Strategy` **design pattern**

patterns. We define characteristics and limitations that are commonly found in those parts as "conditions of smells" for each design pattern. Usually, the conditions of smells can be derived by seeing the following sections of each pattern document: *Motivation, Context, Problem, Forces, Solution* and *Examples*. Many of pattern documents are described in a common format (known as the "Canonical" format[9]) including these sections.

It is not guaranteed that design patterns can be applied to any program that satisfies the conditions of smells. We use the conditions of smells and the conditions used in conventional techniques (we call them "conditions of pattern specifications") together for design pattern detection. Moreover we assume that design patterns are tend to be applied in design improvement activities (such as the refactoring activities) and not applied from the beginning. Such usage is sometimes recommended, such as in [10].

2.1 Detection procedure

Our technique requires a pair of source-code of before applying design patterns and that of after applying them. In our technique, users (usually software maintainers) check the target source-code whether it satisfies the conditions of smells and/or pattern specifications of a certain design pattern. After that, the users examine the correspondence relationship between the pair of source-code regarding roles of the design pattern. If and only if the correspondence relationship is proper, the users judge that the design pattern has been applied.

We give two detection procedures: "forward method" shown in Figure 3 and "backward method" shown in Figure 4. The backward method is useful when the target design pattern to be detected is given (i.e. obvious). On the other hand, the forward method enables users to identify a possibility that the design pattern might have been applied, which conventional approaches failed to detect.

In below, we describe these procedures in detail with the version control systems.

Forward method consists of the following steps:

1. Users check whether each source-code satisfies the conditions of smells of a pattern P from the oldest to the newest version by using our technique until the source-code does not satisfy the conditions of smells. Here we assume that the version that has not satisfy the conditions of smells was Ver.K.

2. In this case, the users check whether the source-code of both Ver.$K-1$ and Ver.K satisfy the conditions of pattern specifications of P by using conventional techniques or manual review.

3. (a) If only the source-code of Ver.K satisfies the conditions of pattern specifications, the users examine the correspondence relationship between the source-code of Ver.$K-1$ and that of Ver.K regarding the design pattern roles. And if there are clear correspondence relationship between them, it is recognized that the pattern P has been newly applied in Ver.K.

   (b) Otherwise, if both of Ver.$K-1$ and Ver.K do not satisfy the conditions of pattern specifications, the users should check manually whether the target design pattern has been applied in Ver.K.

Backward method consists of the following steps:

1. The users check whether each source-code satisfies the conditions of pattern specifications of a pattern P from the newest to the oldest version by using conventional techniques or manual review until the source-code does not satisfy the conditions of pattern specifications. Here we assume that the version that has not satisfy the conditions of pattern specifications was Ver.L.

2. In this case, the users check whether the source-code of both Ver.L and Ver.$L+1$ satisfy the conditions of smells of P by using our technique.

3. If only the source-code of Ver.L satisfies the conditions of smells, the users examine the correspondence relationship between these versions. And if there are

clear correspondence relationship between them, it is recognized that the pattern P has been newly applied in Ver.$L+1$.

2.2 Example of detection

We show an example of detecting a design pattern from Java program source-code in [10]. Regarding the design shown in Figure 5 (taken from [10]), conventional techniques based on the structural aspects of patterns (such as the Tsantalis's technique[4]) judge that `State` or `Strategy` has been applied; however they cannot distinguish these two patterns[2, 4]. Moreover, it is assumed that the previous version of the design in Figure 5 has been implemented as shown in Figure 6 (also taken from [10]).

In this case, the design patterns to be distinguished are obvious; so we can apply the backward method of our detection technique to the example.

1. **Conditions of smells for the `State` pattern**

   First, we derive the conditions of smells of `State` as follows:

   - C_1: A method in the `Context` role class has two or more conditional expressions that depend on the same field in `Context`.
   - C_2: The field's value is modified when `Context` changes its state.
   - C_3: All of the possible values of the field are given by constants.

2. **Conditions of smells for the `Strategy` pattern**

   Second, we derive the conditions of smells of `Strategy` as follows. Among the following two conditions, C_4 is exactly the same as that of the `State` pattern (C_1); however C_5 is different from any condition of the `State` pattern.

   - $C_4 (= C_1)$: A method in the `Context` class has two or more conditional expressions that depend on the same field in `Context`.
   - C_5: The field's value is not modified by `Context` itself.

3. **Distinction of the State pattern and the Strategy pattern**

   Third, we check whether the source-code of before applying the design pattern (figure 6) satisfies the conditions of smells of `State` and `Strategy`. This step is currently automated by preparing the smell detection tool; the tool has been implemented by JavaML[11] and XPath[12] for program static analysis (excerpt shown in Figure 7).

```
public class SystemPermission ... {
 public void claimedBy(SystemAdmin
   admin) {
  if (state == REQUESTED) {
   state = CLAIMED;
  } else if (state == UNIX_REQUESTED){
   state = UNIX_CLAIMED;
  }
  ...
 }

 public void grantedBy(SystemAdmin
   admin) {
 if(profile.isUnixPermissionRequired()
   && state == UNIX_CLAIMED) {
   isUnixPermissionGranted = true;
  } else if (profile.
   isUnixPermissionRequired() &&
   !isUnixPermissionGranted()) {
   state = UNIX_REQUESTED;
notifyUnixAdminsOfPermissionRequest();
   return;
  }
  ...
 }
 ...
}
```

Figure 6. Previous version of Figure 5 (taken from [10])

Consequently, the program of Figure 6 satisfies the conditions of smells of `State` ($C_1 \sim C_3$) as below; however, the program does not satisfy one of the conditions of smells of `Strategy` (C_5).

- C_1: the method `claimedBy` in the class `SystemPermission` has multiple conditional expressions that depend on the same field `state`. According to Figure 5, `SystemPermission` has the role of `Context` of the `State` pattern.
- C_2: The value of the field `state` is modified in these expressions.
- C_3: `state`'s values are given by constants (`CLAIMED`, `REQUESTED`, `UNIX_REQUESTED` and `UNIX_CLAIMED`).

Thus, we judge that `State` has been applied in the design of Figure 5.

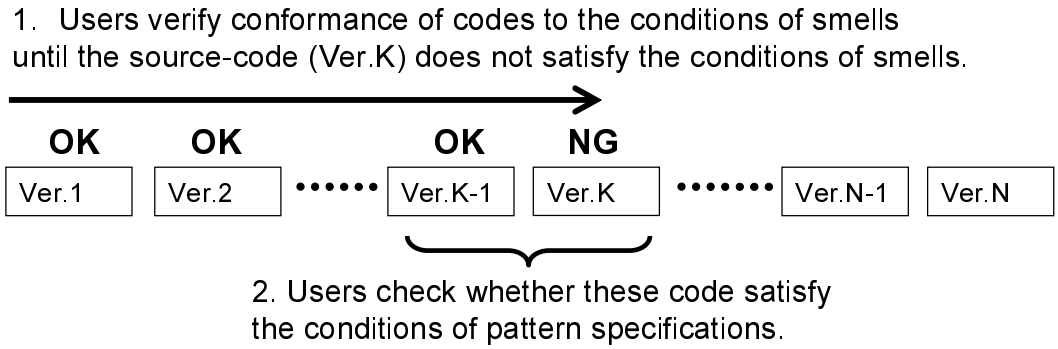

Figure 3. Forward method

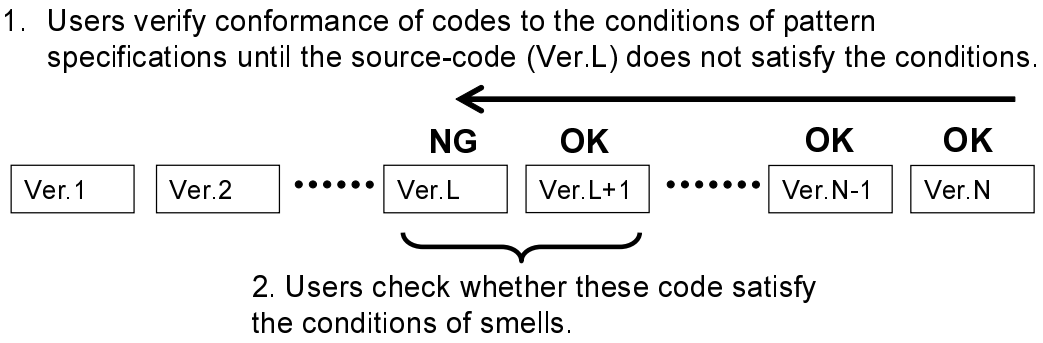

Figure 4. Backward method

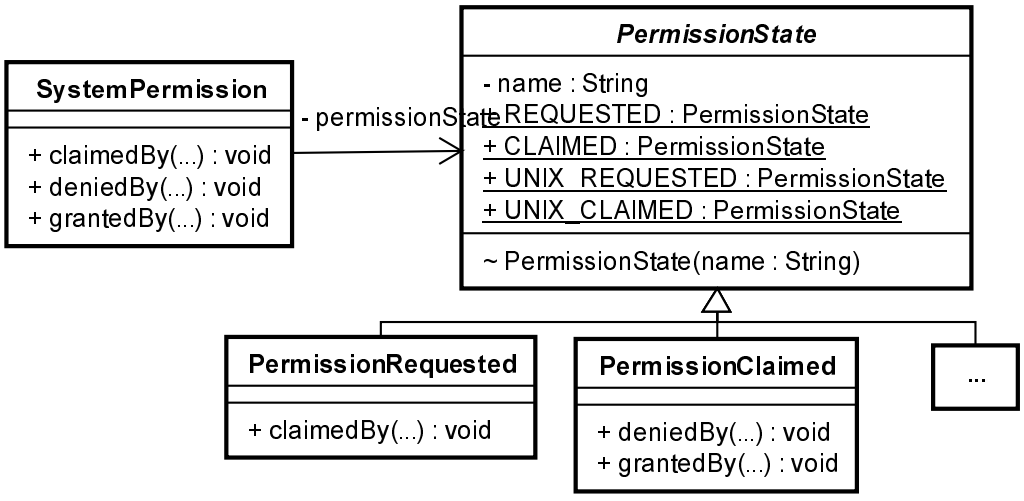

Figure 5. Example of applied the `State` pattern (taken from [10])

```
...
String path = "test/binary-expr"; // Specifying conditional expressions
NodeIterator nb = XPathAPI.selectNodeIterator(nodeIf, path);
while ((nodeb = nb.nextNode()) != null) {
  if(((Element)nodeb).getAttribute("op").equals("==")) {
    // Checking whether the comparison operator is ==
    path = "var-ref[1]"; // Extracting the variable for comparison
    ...
  }
  ...
}
...
```

Figure 7. Implementation of the smell detection tool by JavaML and XPath (excerpt)

3. Experimental evaluations

To evaluate the usefulness of our technique, we conducted an experiment of detecting design patterns from several programs. We prepared four pairs of Java programs of before and after applying the `State` pattern (totally eight programs taken from [10, 13, 14, 15]) as the detection targets.

By using the four pairs, we compared the following two approaches to evaluate whether the target approach can detect the `State` pattern successfully: (a) applying only Tsantalis's technique[4] as the representative of the available conventional techniques, and (b) applying both of Tsantalis's technique and our technique in the backward method.

Table 1 shows the numbers of program pairs where only `State` has been judged to be applied, `State` or `Strategy` has been applied but not clearly distinguished which one has been applied, and no pattern has been applied.

In Table 1, regarding three pairs, the conventional technique could not distinguish two design patterns because of the structural similarity between them; our technique together with the conventional one has successfully distinguished them and detected that only the `State` pattern has been applied. On the other hand, as well as the conventional technique only, our technique together with the conventional one could not detect the `State` pattern regarding one pair. This is because we applied our technique in the backward method. In such case where the conventional technique cannot detect any pattern for the program, our technique should be applied in the forward method.

As a result of the above-mentioned experiment, we confirmed that our technique is useful for detecting design patterns precisely when used together with conventional techniques.

Table 1. Comparisons of the number of program pairs where the following pattern has been judged to be applied

Approach	Detected pattern		
	`State`	`State` or `Strategy`	None
(a) Conventional only	0	3	1
(b) Conventional with our technique	3	0	1

4. Discussion from the viewpoint of quality

Our technique reveals when and where the design pattern has been applied in an ordered series of versions of the target program. Such capability is useful to assess what kinds of design patterns increase what kinds of quality characteristics. Table 2 shows measurement results of applying the following design metrics (mainly related to the maintainability) on the source-code of before and after applying `State` in [10].

- Lack of Cohesion in Methods (LCOM)[16]: LCOM measures the correlation between methods and instance fields (instance variables) in the same class. High value indicates low cohesion of the target class; it could be subdivided into two or more classes with high cohesion.

- Weighted Methods per Class (WMC)[16]: WMC measures the number of weighted methods in each class. High value indicates high complexity of the target class. In our experiment, we simply counted the number of methods (i.e. we did not set any weight for any method).

- McCabe Cyclomatic Complexity (MCC)[17]: MCC measures the number of independent paths in the program control flow. High value indicates high complexity of the target method (or entire program). In our experiment, we calculated the average of MCC values of all methods.

In the result of source-code of after applying the `State` pattern, all of three measurement values have decreased compared with the source-code of before applying the pattern. These results suggest that our technique has a capability of identifying designs before applying design patterns as poor ones from the viewpoint of quality (especially the maintainability in this case).

Table 2. Effects on quality measurements

Metric	Before	After
LCOM	0.37	0.14
WMC	8.75	4.18
MCC	1.67	1.15

5. Conclusion and future work

This paper proposed a technique of design pattern detection using source-code of before applying the design pattern. Our approach is able to distinguish different design patterns with similar structures, and help maintainers understand the design of the program more accurately. Moreover, our technique reveals when and where the design pattern has been applied in an ordered series of versions of the target program, which is useful to assess what kinds of design patterns increase what kinds of quality characteristics such as the maintainability.

As our future work, we need more detection experiments with a collection of source-code examples of before and after applying design patterns, to confirm the validity and usefulness of our technique.

References

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

[2] J. Niere, W. Schafer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. 24th International Conference on Software Engineering*, 2002.

[3] A. Blewitt, A. Bundy, and L. Stark. Automatic verification of design patterns in Java. In *Proceedings of the 20th International Conference on Automated Software Engineering*, 2005.

[4] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11), 2006.

[5] J. Dong, Y. Sun, and Y. Zhao. Design pattern detection by template matching. In *Proc. ACM Symposium on Applied Computing*, 2008.

[6] L. Wendehals and A. Orso. Recognizing behavioral patterns at runtime using finite automata. In *Proc. ICSE Workshop on Dynamic Analysis*, 2006.

[7] N. Shi and R.A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[8] M. Hahsler. A quantitative study of the adoption of design patterns by open source software developers. In *Free/Open Source Software Development*. Idea Group, 2005.

[9] The portland pattern repository. Canonical form. `http://c2.com/cgi/wiki?CanonicalForm`

[10] J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[11] G. Badros. JavaML: a markup language for Java source code. In *Proceedings of the 9th international World Wide Web conference on Computer Networks*, 2000.

[12] The World Wide Web Consortium (W3C) Recommendation. Xml path language (XPath) version 1.0, 1999. `http://www.w3.org/TR/xpath/`

[13] E. Freeman, E. Freeman, B. Bates, and K. Sierra. Head First Design Patterns. O'Reilly Media, 2004.

[14] R. Martin. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, 2002.

[15] S. Metsker. Design Patterns Java Workbook. Addison Wesley Professional, 2002.

[16] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476-493, 1994.

[17] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308-320, 1976.