PAPER *Special Section on Knowledge-Based Software Engineering*

Design Pattern Detection by Using Meta Patterns

Shinpei HAYASHI[†a)], **Junya KATADA**[†*], **Ryota SAKAMOTO**[†], ***Nonmembers***, **Takashi KOBAYASHI**[††], ***and*** **Motoshi SAEKI**[†], ***Members***

SUMMARY One of the approaches to improve program understanding is to extract what kinds of design pattern are used in existing object-oriented software. This paper proposes a technique for efficiently and accurately detecting occurrences of design patterns included in source codes. We use both static and dynamic analyses to achieve the detection with high accuracy. Moreover, to reduce computation and maintenance costs, detection conditions are hierarchically specified based on Pree's meta patterns as common structures of design patterns. The usage of Prolog to represent the detection conditions enables us to easily add and modify them. Finally, we have implemented an automated tool as an Eclipse plug-in and conducted experiments with Java programs. The experimental results show the effectiveness of our approach.

key words: *design patterns, program understanding, meta patterns, dynamic analysis, Prolog*

1. Introduction

Design patterns [1] are increasingly being applied in object-oriented software design processes. Design patterns are abstract structures of artifacts that recur as successful practice in software design so that they can be reused. The usage of design patterns allows us to improve the quality of software designs, e.g. reusability, extensibility, and maintainability. In addition, since design patterns reflect the designers' intents on the reasons why the designers adopt a design, the identification of design patterns in an existing source code helps us to understand it by grasping the designers' intents.

Investigating the documentation of source codes is one of the approaches to identify the occurrences of design patterns. However, it frequently happens that the document has not been completely developed or has not been updated to the newest version. Therefore, the documents cannot always include the accurate information on which patterns were used in which parts of the code. Additionally, and developers spend large amounts of effort on conjecturing pattern occurrences by investigating the incomplete document and the source code.

There are several researches to (semi-)automatically detect the occurrences of design patterns in source codes, in order to support various activities of software process under the situation where the used patterns are not explicitly specified in the document of the source code [2]–[19]. Detecting occurrences of design patterns is a prerequire for many applications:

Program understanding – Developers can guess a program's behavior by obtaining what classes and methods in the program are corresponding with pattern roles.

Design recovery – Developers can confirm whether patterns in a program are correctly used by continuously recovering design information of the program.

Pattern mining – An automated pattern detection mechanism leads to exploring unrecognized program structures that are frequently used.

The usage of design recovery involves strict time efficiency because the detection is incrementally performed at the software development process. The pattern mining approach also involves fast detection in order to apply the detection to many projects and larger-scale software. Additionally, accurate results are important in high-frequency and/or larger-scale pattern detection. We then have to consider how to improve the accuracy and time efficiency of pattern detection, and of course to reduce the cost of human effort.

This paper proposes a technique to accurately and efficiently detect occurrences of design patterns in existing source codes. Although there are varieties of software patterns, we focus only on the design patterns. For example, Gang-of-Four (GoF) patterns [1] or J2EE patterns [20] are well-known and widely used design patterns in industry. In our approach, we adopt both static and dynamic analyses so that we can obtain more accurate results by capturing the dynamic properties of a program as well as the static structure. To improve time efficiency of our automated detection, and to maintain detection conditions more efficiently, we hierarchically compose the detection conditions to decide whether a part of a program follows a design pattern.

Since design patterns have common properties, we then use Pree's meta patterns [21] to represent these common properties as a part of the detection conditions. More concretely, if a design pattern includes meta patterns, it can be specified with the combination of meta patterns. For this benefit, we take a two-stage pattern detection. First, we detect the occurrences of meta patterns in a source code. Next,

Manuscript received July 2, 2007.

Manuscript revised October 17, 2007.

[†]The authors are with the Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.

[††]The author is with the Center for Embedded Computing Systems, Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8601 Japan.

[*]Presently, with NTT COMWARE CORPORATION.

a) E-mail: hayashi@se.cs.titech.ac.jp

DOI: 10.1093/ietisy/e91–d.4.933

we check if the pre-defined conditions of the design pattern, including the combination conditions of the meta patterns, hold or not. If it fails in this stage, we move to other patterns that have the same meta patterns. By using this technique, we can reduce any redundant repetition of checking the common properties of design patterns, i.e. the detection of meta patterns. Furthermore, we use Prolog to define detection conditions so that we can easily add and/or modify the conditions. It is not a new idea to use Prolog or the first order predicate logic (FOPL) for reasoning about structural properties of an object-oriented program, including static method invocation property [7], [8], [13]–[15], [17]. Additionally, for design pattern detection, reclassification of GoF patterns is also proposed [16]. However, none of them used any hierarchical technique to reduce computation and maintenance costs.

In summary, the main contributions of this paper are considered as follows:

Improved accuracy – By using both static and dynamic analyses, we can extract the candidates of design pattern instances more accurately than only using static analysis.

Reduced computation and maintenance costs – We use a hierarchy of conditions by using Pree's meta patterns [21]. Dynamic analysis causes slow detection because it produces large amounts of data to be analyzed. Furthermore, comprehensibility of pattern descriptions may decrease by introducing dynamic analysis. Hierarchical categorization of common conditions for each of the design patterns leads to their improvements in time efficiency and maintainability.

The rest of the paper is organized as follows. In the next section, we outline our approach. Section 3 clarifies what information is extracted from a source code. Sections 4 and 5 present the hierarchical structure of detection conditions and their representations with Prolog, respectively. In Sect. 6, we discuss our supporting tool and experiments to show the benefits of our approach. Section 7 discusses the most relevant work, and Sect. 8 concludes with our contribution and discusses future work.

2. Overview of Our Approach

In our approach, we extract the candidates of design patterns from a source code. Figure 1 shows the overview of it.

First of all, we perform both static and dynamic analyses to a source code to be analyzed. The static analysis extracts the information on the inheritance relations of classes, the methods defined in the classes, and so on. The extracted information is translated into a set of facts in Prolog. Since object-oriented programs have the mechanism of dynamic binding, we cannot obtain all of the necessary information without executing them. For example, consider that an object receives a method invocation and calls its corresponding method. However, by applying static analysis, which only investigates the text of the source code, we cannot identify

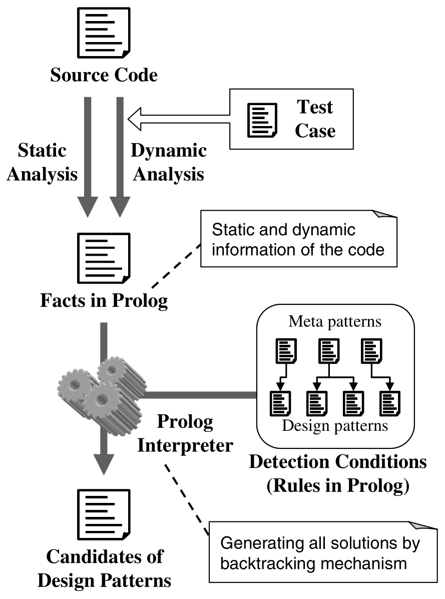

Source Code
Test Case
Static Analysis
Dynamic Analysis
Static and dynamic information of the code
Facts in Prolog
Meta patterns
Design patterns
Prolog Interpreter
Detection Conditions (Rules in Prolog)
Generating all solutions by backtracking mechanism
Candidates of Design Patterns

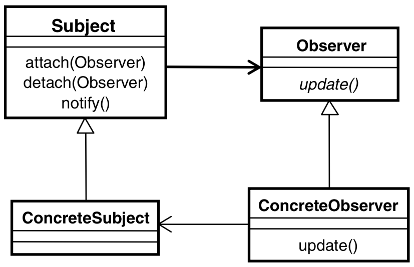
Fig. 1 An overview of our approach.

to which class the created object actually belongs because of the dynamic binding mechanism. To detect the occurrences of design patterns, the information on interactions among classes is necessary. In addition, only with static analysis, we cannot identify which method was actually executed, in the case where the method is overridden by a method having the same name defined in the subclasses. The dynamic analysis, where we actually execute the code, complements the static one to extract the information such as the methods that were actually executed or the classes of the objects that were created during the execution. This information is also represented with facts in Prolog.

Using extracted information as facts in Prolog, the detection conditions, described as Prolog programs, are executed to infer the existence of the class structures satisfying the conditions of design patterns. For example, the predicate `observer(ClassRoleList, MethodRoleList)` for detecting Observer pattern is defined as a Prolog program and executed as a query on the facts denoting the extracted information of the source code.[†] After finishing the execution, the classes and the methods recognized as constituents of Observer are bound to the variables `ClassRoleList` and `MethodRoleList`, respectively, as a solution, i.e. an occurrence of Observer. One of the benefits to use Prolog is that it can perform exhaustive search for solutions with backtracking mechanism during its execution. The Prolog program can detect all occurrences of Observer by backtracking execution if existed in the source code.

Pattern description should have functionality to represent fuzzy conditions. It is difficult to completely decide whether a set of classes is actually an instance of a design pattern because of the variations of design pattern instances.

[†]We use sans-serifs for pattern names such as Observer, italics for roles in some patterns such as *Observer*, and teletypes for parts of Prolog/Java code such as `observer` for avoiding ambiguity.

(a)


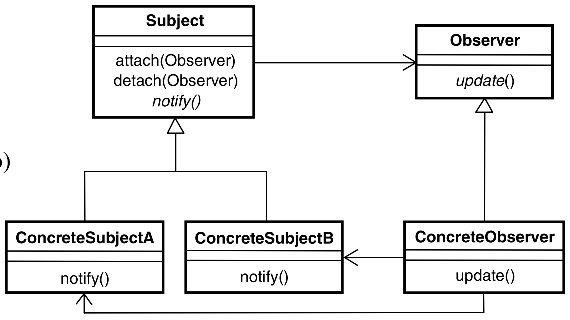
(b)


Fig. 2 Variations in Observer.

For example, Fig. 2 illustrates two variations in Observer. Based on the description in GoF's book [1], *notify*() in *Subject* is not overridden in its subclasses, i.e. *ConcreteSubject* (Fig. 2-a). However, sometimes we modify how to notify to observers. Then *notify*() may be overridden by *ConcreteSubject*s (Fig. 2-b). To deal with these variations in a unified way, our detection conditions do not simply return true or false value, but the degree of certainty that a code includes the design pattern. That is to say, we embed the calculation of *fuzziness* to our facts in Prolog. The description about our fuzzy conditions is included in Sect. 5.

3. Extracted Information

3.1 Static Information

Our approach first extracts static information by source code analysis. Table 1 lists up the types of the extracted information. For example, for a class, we extract the information on inheritance relationship (its superclass), visibility (public or private), whether it is an abstract or concrete class, the list of methods that it defines, and fields that it has (instance variable declarations). The left side of Fig. 3 illustrates an example of extracting the static structure of a source code as a fact in Prolog. In the figure, the line starting with the predicate `class` represents the static information of class `IntNumber` as a fact. The six parameters of the predicate denote its class name (`'IntNumber'`), superclass (`'Number'`), methods (`['IntNumber.intValue()']`), instance variables (`[]`: a null list because it has no fields), the information on whether it is an abstract or concrete class (`'false'`: abstract class), and its visibility (`'public'`), respectively.

Table 1 Static information extracted from a source code.

Information on a Class
(1) Inheritance relationship
(2) Visibility
(3) Abstract class or concrete class
(4) Methods defined in a class
(5) Fields (Instance Variables, Attributes) defined in a class
Information on a Field (Instance Variable, Attribute)
(1) Data type of a field
(2) Visibility
(3) Static field (class variable) or not
Information on a Method
(1) Input parameters (data types)
(2) Visibility
(3) Abstract method or not
(4) Static method (class method) or not
(5) Constructor or not
(6) Return value (data type)

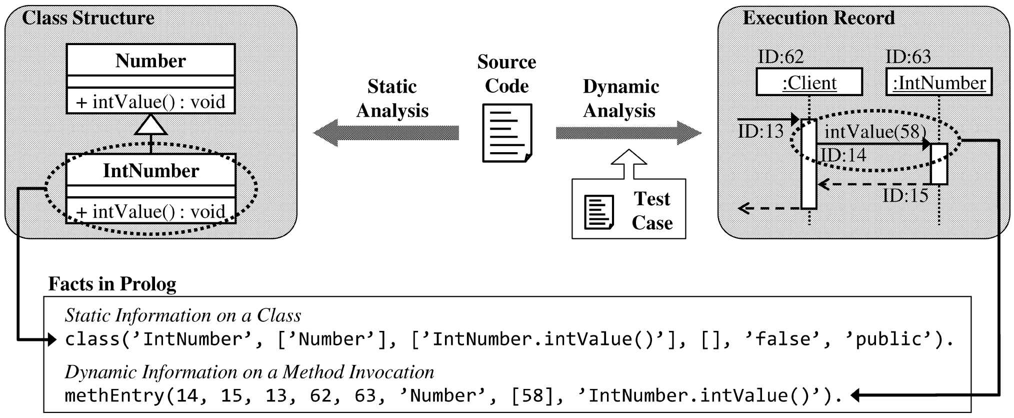
3.2 Dynamic Information

For dynamic analysis, we execute the source code and extract, from the execution records, the occurrences of method invocations, including constructors' invocations to create instances. The information that we extracted is listed up in Table 2.

The right side of Fig. 3 illustrates an example of extracting the dynamic execution record as a fact in Prolog. A part of the facts is shown in the bottom line of the figure, and it begins with the name of the predicate, `methEntry`. This fact denotes an invocation of the method `intValue()` in `IntNumber` class during the execution. We model the behavioral aspect of method invocation with a pair of events; the event of the method being called and the event of the method being returned. Each event is assigned a number corresponding the order in which it was invoked. We will refer to it as the *ordinal execution number* of the event. The first parameter `14` in the predicate `methEntry` stands for the ordinal execution number of the event of the method just being called in the analyzed execution record. After finishing the execution of the method and returning to the caller object, the 15th (second parameter of the predicate) event occurred as an event of the method to be returned. It means that the method invocation `intValue()` is modeled with a pair of events, 14 and 15, as shown in Fig. 3. The identifier `13` as the third parameter shows the parent method invocation event that caused this invocation. The caller (sender of the method invocation) and the callee (receiver) object are referenced by `62` and `63`, respectively. The class `Number` is a superclass of `IntNumber`, and this method invocation was done on the object whose class is `Number` from syntactic view. For example, the code of this method invocation can be written as follows:

```
Number obj = new IntNumber();
obj.intValue(obj1);
```

Note that the receiver object is bound to the type of `Number`, not `IntNumber`, and therefore the sixth parameter of the



```
class('IntNumber', ['Number'], ['IntNumber.intValue()'], [], 'false', 'public').
methEntry(14, 15, 13, 62, 63, 'Number', [58], 'IntNumber.intValue()').
```

Fig. 3 Representing static and dynamic information with Prolog.

Table 2 Dynamic information extracted from execution records.

Information on an Occurrence of Method Invocation
(1) Method that was actually invoked
(2) Ordinal execution number in the analyzed record
(3) Method that sent the invocation
(4) Object that sent the invocation
(5) Object that received the invocation
(6) Class to which the invocation was done
(7) Objects that were bound to the input parameters
(8) Object that was returned as the result of executing the invocation
Information on a Created Object
(1) Class that the object actually belongs to

above predicate is `Number`. The seventh parameter `[58]` shows that the method invocation had only one input parameter and that it is the object with the ID number 58.

4. Detection Condition by Meta Patterns

4.1 Meta Patterns

In many cases, customizable parts of design patterns, so-called hot spots, are implemented with employing subclasses of an abstract class and overriding its abstract methods by concrete ones in the subclasses. The designers can fill the hot spots with the concrete methods that override the abstract one. The method overridden by a concrete method in a subclass is called *hook method*, and a *template method* includes one or more hook methods to implement its functions.

For example, consider Observer of GoF patterns shown in the bottom of Fig. 4. The method *update*() of the abstract class *Observer* has no implementation but defines its interface, i.e. its name and parameters. When a designer uses Observer, he/she composes concrete subclasses of *Observer* class and customizes the pattern by overriding the abstract method *update*() with a concrete method, i.e. a method having an implementation, in each subclass in order to implement the functions that he/she wants. The method *update*() is a hook method, and the method *notify*() in *Subject* class which calls *update*() is a template method. The instance variable *observers* in *Subject* class holds all instances

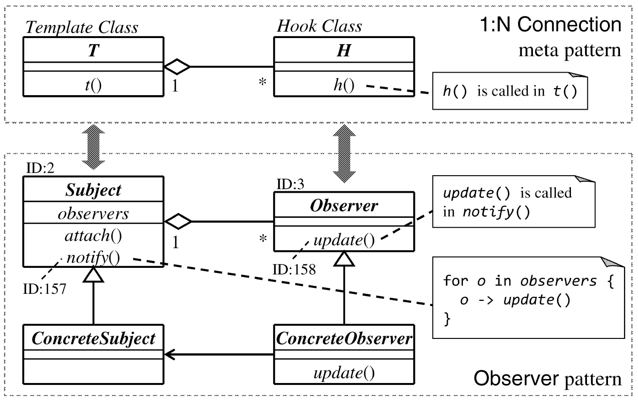


Fig. 4 Observer pattern and 1:N Connection meta pattern.

of *ConcreteObserver*, and the invocation of *notify*() leads to sending update messages to all of the *ConcreteObserver* instances. By using dynamic binding and overriding mechanisms, the execution of update varies on its implementation in *ConcreteObserver* subclass. Note that the ID numbers such as ID:157 attached to the syntactic elements in Fig. 4 will be used to explain a detection condition in Sect. 5.2.

Many of design patterns include common structures. Pree classified hot spots of design patterns into seven categories from the viewpoint of associations between the class having hook methods (called a hook class) and a class having template methods (template class). He defined these categories as meta patterns [21]. Meta patterns can be considered as a basis of design patterns and in fact. For example, most of GoF patterns include meta patterns. Figure 5 depicts the class diagrams of the Pree's meta patterns. For example, as shown in Fig. 4, Observer includes 1:N Connection meta pattern. In 1:N Connection, there is no inheritance relationship between a template class and a hook class, but they are connected through an aggregation relationship whose cardinality is one-to-many. In Fig. 4, *Subject* class and *Observer* class correspond to a template class and a hook class, respectively.

We analyzed 23 GoF design patterns and investigated which meta patterns are included in them. Table 3 shows the

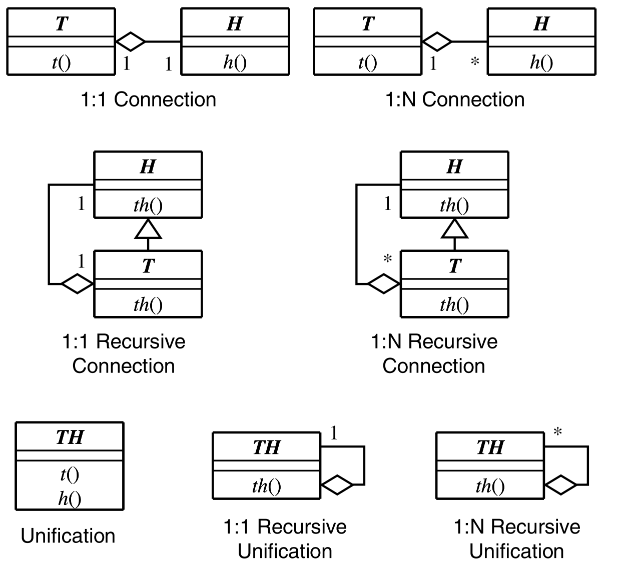


Fig. 5 Meta patterns. $T, H, TH, t(), h(), th()$ denote a template class, a hook class, a class having both roles of template and hook class, a template method, a hook method, and a method having both roles of template and hook method, respectively.

Table 3 Classification of GoF design patterns by meta patterns.

Meta Pattern	GoF Design Pattern
1:1 Connection	Bridge, Builder, Mediator, State, Strategy, Visitor*
1:N Connection	AbstractFactory, Adapter, Command, Flyweight, Iterator*, Observer, Prototype, Proxy, Visitor*
1:1 Recursive Connection	Decorator
1:N Recursive Connection	Composite, Interpreter
Unification	Factory Method, Template Method
1:1 Recursive Unification	Chain of Responsibility
1:N Recursive Unification	None
No meta patterns included	Facade, Memento, Singleton

*: patterns including more than one meta pattern

classification of GoF patterns by the included meta patterns. 20 of the 23 patterns include a meta pattern, except for Iterator that includes two occurrences of 1:N Connection meta pattern and Visitor that includes both 1:N Connection and 1:1 Connection meta patterns.

4.2 Hierarchy of Detection Conditions

We define the hierarchical structure of GoF patterns based on the inclusion of meta patterns. In the case where several design patterns have the *common properties* for detecting them, we can attain efficient detection by avoiding the redundant repetition of checking these common properties. In our approach, these common properties are specified with meta patterns.

As an example, consider the detection of Builder and Bridge. As shown in Table 3, both of them include 1:1 Connection meta pattern. Therefore, the Builder detection conditions can consist of the conditions for 1:1 Connection and the conditions specific to Builder. Similarly, the conditions for Bridge are the combination of 1:1 Connection conditions and Bridge-specific ones. If we try to detect each pattern individually, we have to check 1:1 Connection conditions twice on the whole source code to be analyzed which brings inefficiency. In our approach, as shown in Fig. 6, we organize the hierarchical structure where 1:1 Connection conditions are in the upper layer and where Builder-specific conditions and Bridge-specific ones are in the lower layer. To detect the occurrences of GoF design patterns, we visit nodes in this hierarchical structure from the root node and check, in descending order, the conditions which the visited nodes have. In the example, we first find out the parts of the source code that hold the 1:1 Connection conditions and then apply the Builder- and Bridge-specific conditions to them. Since we check the 1:1 Connection conditions only once, we can attain more efficient detection. The hierarchical structure is organized based on logical implication relationship among the detection conditions. For example, 1:1 Connection can be considered as a logically stronger version of 1:N Connection, because the constraint that the cardinality should be one-to-one is additionally imposed to 1:N Connection condition. Therefore we check the 1:N Connection condition, and then check 1:1 Connection-specific condition, i.e. one-to-one cardinality if 1:N Connection is satisfied. In this sense, the hierarchical structure represents the order of checking the detection conditions. We start with checking the Common Conditions for Meta Patterns, which all meta patterns have. This common condition specifies the existence of a method that is overridden as a hook method and is called from somewhere. Common Conditions for Connection, the definition of an aggregation between the corresponding classes, is common to all of connection types of meta patterns. Proposed hierarchical structure is used for both static and dynamic analyses. As shown in Fig. 6, we can define hierarchical relationships not only between a meta pattern and a design pattern but also between two meta patterns and between two design patterns. Factory Method is a special case of Template Method, we then check the condition of Factory Method after that of Template Method is satisfied.

By constructing the hierarchy of conditions, we can also reduce the size of detection conditions. In our technique, we develop detection conditions efficiently by using a reusing mechanism as well as the inheritance mechanism of object-oriented programming. If a target pattern (e.g. P) includes other pattern or meta patterns (e.g. P_i), we can consider P is inherited from P_i, and P_i is regarded as the superclasses of P. After declaring the relationship between P and P_i, we can just focus on writing the condition specific to P. For example, to describe the detection condition of Factory Method, all we have to do is declare parent pattern Template Method and describe Factory Method-specific conditions, e.g. a constraint of result type. Furthermore, if a modification of detection conditions is required, e.g. adding debugging information or introducing stricter conditions, we can easily find where to modify because the conditions in common and those specific to the target pattern are separated.

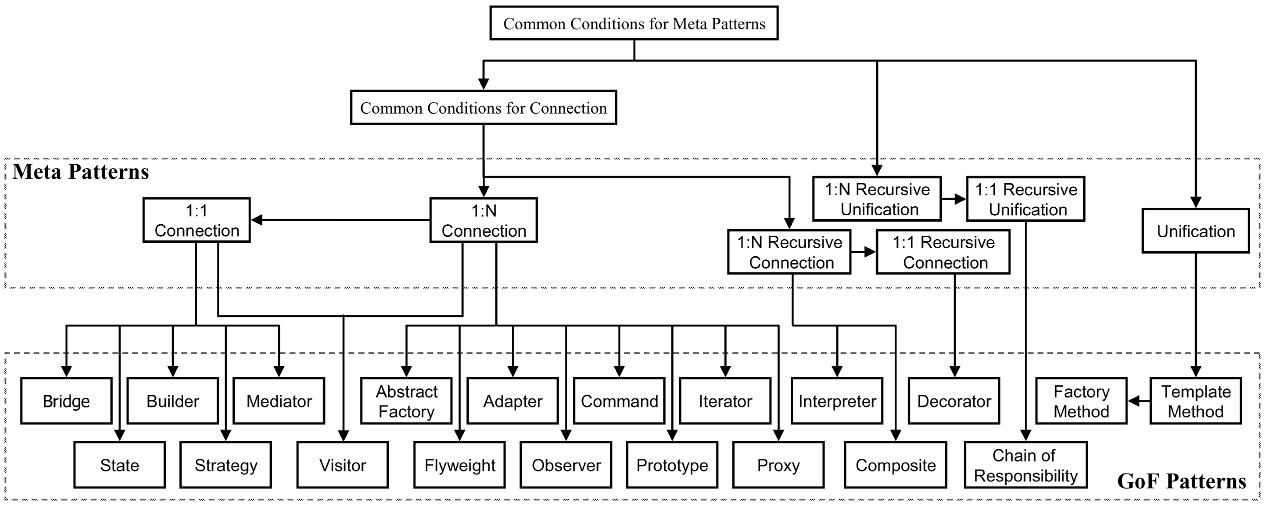


Fig. 6 Hierarchical structure of detection conditions.

Compared with detection conditions that do not use any hierarchical structure, we can acquire >30% reduction by meta patterns, furthermore >60% reduction by meta patterns and common conditions for meta patterns. A well-balanced hierarchical structure of GoF patterns and Pree's meta patterns contributes an effective reduction of whole description of detection conditions.

5. Specifying Detection Conditions with Prolog

5.1 Detecting Patterns with Prolog

The overview of a detection process by using Prolog is as follows. We first represent the extracted information from the source code as facts in Prolog and *assert* (registration of facts) them into a Prolog database, which is a collection of facts. We then check the detection conditions in the order following the hierarchical structure from its root shown in Fig. 6. The candidates of design pattern instances are obtained as solutions satisfying the detection conditions, and they are asserted into the database in order to utilize them for checking the conditions of the lower level.

5.2 Detection Condition

The detection conditions are defined as rules in Prolog. The rules specify the properties that the design patterns should have with a series of predicates so as to evaluate the properties to the facts of the source code in Prolog. Figure 7 shows a part of the detection conditions for Observer which has the 1:N Connection condition in the upper layer. Observer and its correspondence to 1:N Connection are shown in Fig. 4. According to Fig. 6, we have to start with the check of 1:N Connection conditions, and the predicate `meta1_Nconnection` appears in the right side of the rule. The results from evaluating this predicate, i.e. the parts of the source code satisfying it, are bound to the variables `NotifySN` as the template method *t*(), `UpdateSNList` as

```
observer(ClassRoleList, MethodRoleList) :-
    meta1_Nconnection(NotifySN, UpdateSNList,
       Subject, Observer),
    methEntry(NotifySN, _, _, _,
       ConcreteSubjectObj, _, _, NotifyMethID),      1
    method(NotifyMethID, _, NotifyDeclClass,
       _, _, _, _, _, _),
    obj(ConcreteSubjectObj, ConcreteSubject),
    inHierarchy(Subject, ConcreteSubject),           2
    makeConcreteObserverList(UpdateSNList, [],
       ConcreteObserverList, [],                     3
       ConcreteObserverObjList),
    findAttach(UpdateSNList, NotifyMethID,
       NotifyDeclClass, [], AttachSNList,            4
       [], AttachMethIDList),
    ...

score(observer, ClassRoleList, MethodRoleList, -40) :-
    member(['ConcreteSubject', _, Id], ClassRoleList),   5
    class(Id, _, _, _, 'false').
```

Fig. 7 A part of the detection condition for Observer.

the list of the hook methods *h*(), `Subject` as the template class *T*, and `Observer` as the hook class *H*. The results are asserted as facts in Prolog, e.g. `meta1_Nconnection(157, [158], 2, 3)`, into the database so that we can use them to check further the Observer specific conditions.

The predicates following `methEntry` predicate are for the Observer specific conditions:

1. The method *notify*() should be called by the instance of *ConcreteSubject* class. The candidates of *notify*() method, which is calculated in `meta1_Nconnection`, is passed through the variable `NotifySN`, and its actual invocation from an instance of the *ConcreteSubject* in the execution record is bound to `NotifyMethID`.
2. The *ConcreteSubject* class should be a subclass of *Subject* class.
3. The *ConcreteObserver* classes should have the method *update*(). The candidates of *update*() methods, concrete classes, and their instances are bound to the vari-

ables `UpdateSNList`, `ConcreteObserverList`, and `ConcreteObserverObjList`, respectively.

4. `NotifyDeclClass`, the classes having method *notify*(), should have the method *attach*().

The following Prolog goal is for detecting the occurrences of Observer in a source code.

```
observer(ClassRoleList, MethodRoleList).
```

The first parameter `ClassRoleList` in the goal has the triples consisting of role names in Observer, their corresponding detected class names, and the instances of the classes. On the other hand, a list of triples regarding the detected methods is bound to `MethodRoleList`, and each triple contains a role name in Observer, its corresponding method signature, and an identifier of its method invocation included in the execution record. A concrete example of the detection result is shown in Fig. 8. It is the result from applying the condition in Fig. 7 to a source code which includes the Observer shown in Fig. 4. The first four lines in Fig. 8 show the value of the first parameter `ClassRoleList` as the result regarding the detected classes. More concretely, the class `Node` corresponds to *Subject* class in Observer, but there are no instances of `Node` because it is an abstract class, hence the `null`. The class `File` plays a role of *ConcreteSubject* in Observer and an instance of the identifier `57` was created during the execution. In the fifth line of the Fig. 8, the results from detecting methods are shown. For example, `notifyObservers()` method, defined in `Node`, corresponds to the role of *notify*() in Observer, and its method invocation has the identifier `157`.

Pattern candidates are associated with its fuzziness score, which is represented using the predicate `score` shown in Fig. 7–5. The rule represents, "for Observer, if *Subject* is not an abstract class, then the score should have 40 subtracted from it." The second and third parameters are directly passed from predicate `observer`. The fourth parameter is the score that should be added to the default score 100 if the following conditions are satisfied. In this case, a negative score 40 is used because the following conditions may not be suitable for Observer. As explained above, since the definitions of fuzziness can be described declaratively, we then can easily add, remove, and modify the estimation rules without changing the detection rules.

```
observer([['Subject', 'Node', 'null'],
   ['ConcreteSubject', 'File', 57],
   ['Observer', 'Observer', 'null'],
   ['ConcreteObserver', ['SizeObserver'], [61]]],
  [['notify', 'Node.notifyObservers()', 157],
   ['update', ['SizeObserver.update(Subject)'], [158]],
   ['attach', ['Node.addObserver(Observer)'], [22]]]).
```

Fig. 8 An example of detection results.

6. Experimentation

6.1 Automated Tool

6.1.1 Design Pattern Detector

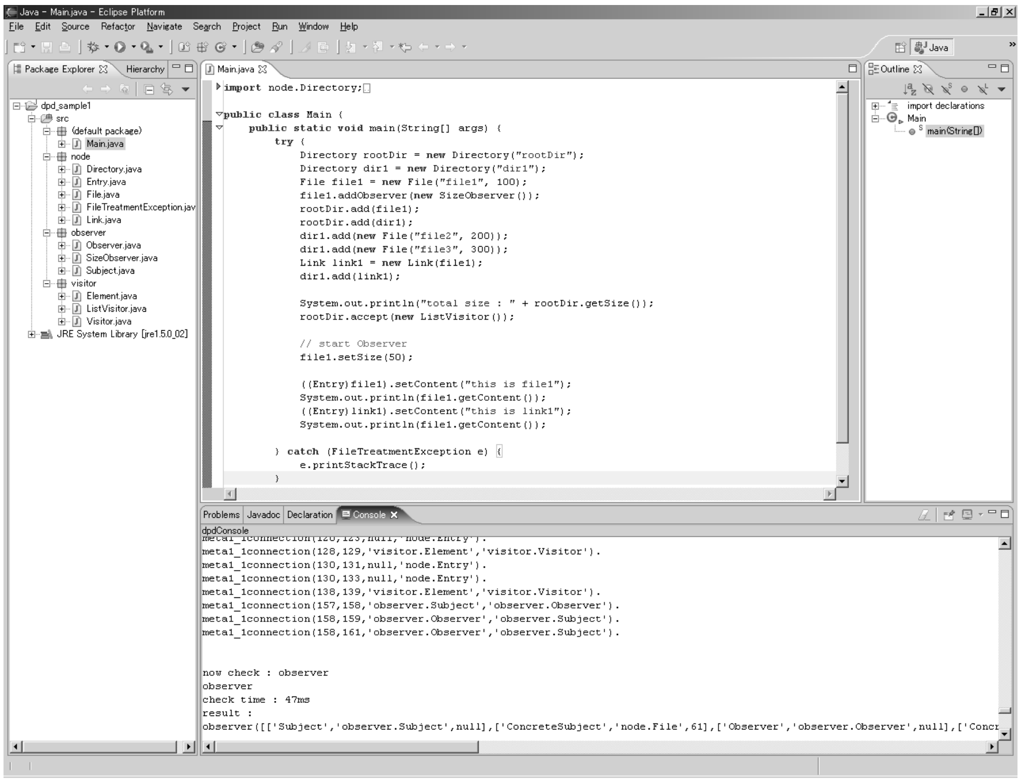
We implemented a tool to automate the detection of design patterns based on our approach as a plug-in of Eclipse IDE [22]. As suggested in Fig. 1, our automated tool consists of two parts: 1) an analysis module for extracting static and dynamic information from a Java source code (called static analyzer and dynamic analyzer, respectively), and 2) a detection module for generating facts in Prolog from the extracted information and checking detection conditions against them. The static analyzer has been implemented by using Abstract Syntax Tree (AST) parser of the Eclipse JDT plug-in [23], and the dynamic analyzer uses Java Debug Interface (JDI) to obtain, from Java Virtual Machine, the events of method calls and method returns that occur during the execution. We have used tuProlog [24], [25], that can be operated with Java platform, to check detection conditions written in Prolog.

Figure 9 is the screenshot of the automated tool on Eclipse IDE. The console view part (the bottom part of the figure) displays the result from detecting Observer in the form of Prolog predicates. After recognizing the several occurrences of 1:1 Connection, the tool detects an occurrence of Observer. The user identifies what parts of the program are organized as design patterns from detection results. Our tool is under the prototyping, so only preliminary functionalities are available. As an empirical viewpoint, detection results should be provided not only by textual but also more graphical information. To improve the legibility of detection results, a tool to visualize pattern candidates and a combination of multiple patterns is desired.

6.1.2 Test Case Generator

Dynamic analysis requires test cases to execute the target program with the regular calling order in an effort to extract amounts of dynamic information enough to detect design patterns. Our detection conditions represent what kind of methods are invoked in what order, so execution sequences against the order do not satisfy the conditions. To reduce computation time and human cost, avoiding unnecessary executions is useful. However, some techniques for improving coverage requirements often generate many ineffective test cases.

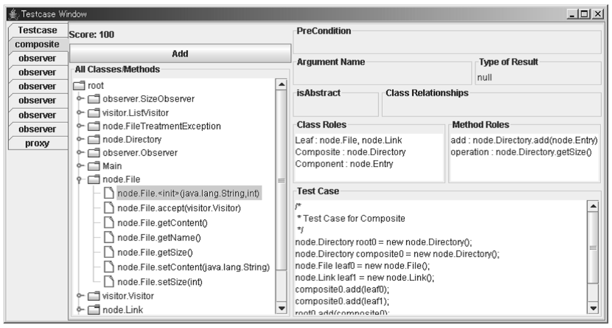
We also implemented a tool for supporting test case construction by using test case templates and pattern candidates detected by static analysis. Figure 10 is the screenshot of the supporting tool. The tool lists the candidates of design patterns detected by static analysis (left side of tabs). When the user selects a candidate, the tool shows a template which is suitable for design pattern detection. For example, the template for Observer is shown in Fig. 11. For each part



```
import node.Directory;

public class Main {
    public static void main(String[] args) {
        try {
            Directory rootDir = new Directory("rootDir");
            Directory dir1 = new Directory("dir1");
            File file1 = new File("file1", 100);
            file1.addObserver(new SizeObserver());
            rootDir.add(file1);
            rootDir.add(dir1);
            dir1.add(new File("file2", 200));
            dir1.add(new File("file3", 300));
            Link link1 = new Link(file1);
            dir1.add(link1);

            System.out.println("total size : " + rootDir.getSize());
            rootDir.accept(new ListVisitor());

            // start Observer
            file1.setSize(50);

            ((Entry)file1).setContent("this is file1");
            System.out.println(file1.getContent());
            ((Entry)link1).setContent("this is link1");
            System.out.println(file1.getContent());

        } catch (FileTreatmentException e) {
            e.printStackTrace();
        }
```

```
meta1_1connection(120,123,null,'node.Entry').
meta1_1connection(128,129,'visitor.Element','visitor.Visitor').
meta1_1connection(130,131,null,'node.Entry').
meta1_1connection(130,133,null,'node.Entry').
meta1_1connection(138,139,'visitor.Element','visitor.Visitor').
meta1_1connection(157,158,'observer.Subject','observer.Observer').
meta1_1connection(158,159,'observer.Observer','observer.Subject').
meta1_1connection(158,161,'observer.Observer','observer.Subject').


now check : observer
observer
check time : 47ms
result :
observer([['Subject','observer.Subject',null],['ConcreteSubject','node.File',61],['Observer','observer.Observer',null],['Concr
```

Fig. 9 A screenshot of the tool for design pattern detection.



Score: 100

PreCondition

Argument Name

Type of Result: null

isAbstract

Class Relationships

Class Roles:
Leaf : node.File, node.Link
Composite : node.Directory
Component : node.Entry

Method Roles:
add : node.Directory.add(node.Entry)
operation : node.Directory.getSize()

Test Case:

```
/*
 * Test Case for Composite
 */
node.Directory root0 = new node.Directory();
node.Directory composite0 = new node.Directory();
node.File leaf0 = new node.File();
node.Link leaf1 = new node.Link();
composite0.add(leaf0);
composite0.add(leaf1);
root0.add(composite0);
```

Fig. 10 A screenshot of the tool for test case generation.

```
ConcreteSubject concrete_subject = new ConcreteSubject();
ConcreteObserver concrete_observer = new ConcreteObserver();
concrete_subject.attach(concrete_observer);
concrete_subject.notify();
```

Fig. 11 A template of test case generation for Observer.

of the pattern candidate, the tool also shows information on its names, types, and pre-conditions described in JML [26]. A user implements test cases by using this supporting information.

6.2 Experimental Method

In this subsection, we introduce the experiments that we performed to assess our approach and tool. In particular, the aim is to assess the accuracy and time efficiency of our approach. We picked up two Java programs to detect design patterns from: 1) the modified version of a sample program which appears in the textbook of design pattern written by Vlissides [27] (we call it program #1) and 2) the dynamic analyzer of our supporting tool itself (called program #2). The reason why we selected them was that we had known which design patterns were actually used in which parts of the programs in advance. We then could assess the accuracy of our detection by comparing the results produced by the tool with the real usages of design patterns in the programs.

Table 4 Sizes of the programs.

	# Classes	# Method invocations
Program #1	12	88
Program #2	56	1439

Table 4 shows the size of the programs #1 and #2. For example, the program #2 includes 56 classes, and 1,439 method invocations (excluding the invocations within Java libraries such as Swing) were done during the execution.

We have designed the detection conditions for 12 GoF design patterns; Abstract Factory, Builder, Chain of Responsibility, Composite, Factory Method, Observer, Proxy, State, Strategy, Template Method, Visitor, and Singleton. Note that Singleton is not included in the hierarchical structure of Fig. 6 because it is too simple and has no meta patterns.

When performing dynamic analysis, what methods are actually executed depends on the input data that will be sup-

Table 5 Results of test case generation.

Thr.	# Cand.	# Tests	Time	Found patterns
100	11	8	20	$\left\{\begin{array}{c}\textsf{Composite, Observer,}\\ \textsf{Proxy, Strategy,}\\ \textsf{Singleton}\end{array}\right\} (= P)$
80	14	11	41	$P \cup \left\{\begin{array}{c}\textsf{Template Method,}\\ \textsf{Visitor}\end{array}\right\}$

plied to the program during the execution, so we cannot get the information on the methods that were not executed. We also added test cases to extract additional information by using our test case generator for program #1. The results are shown in Table 5. In the table, the columns "Thr.", "# Cand.", "# Tests", "Time", and "Found patterns" represent the threshold of the detection conditions, the number of automatically generated test cases, the number of manually constructed test cases from candidates with a tool support, time spent on construction in minutes, and pattern names that the instances of design patterns in constructed test cases belong to, respectively. At the end we additionally obtained 11 test cases, which are enough to find seven types of patterns. Three test cases were obtained by the candidates by using lower threshold of 80, which in turn rescuing informal candidates lead to lower false negatives.

In order to assess the improvements of the accuracy of our detection technique, we provided two types of the detection way; performing 1) static analysis only, and 2) both static and dynamic analyses. We measured the number of detected pattern candidates and compared the precisions of both types.

Furthermore, in order to assess the time efficiency of our detection technique, we provided two types of detection conditions; 1) following the hierarchical structure shown in Fig. 6 and 2) not following the hierarchical structure. The second type of the conditions is not asserted into the Prolog database. Note that the first type for Singleton is the same as the second one because it is not included in the hierarchical structure. We have detected the patterns by using both types of the conditions and compared the detection times. The differences between the detection times allow us to evaluate how much hierarchy of the detection conditions can contribute to the improvement of time efficiency. Additionally, we do not consider quantitative criteria of our experiments are especially important because assuming criteria depends on the target program. For example, structurally and behaviorally equal patterns, e.g. State and Strategy, are not correctly detected by our technique, and then quantitative results depend on the kinds of patterns included in the target program. We qualitatively discuss what points of our technique contribute to the experimental results.

Our experiments were done on a personal computer with Intel Pentium III 1 GHz×2, 512 MB RAM, and Microsoft Windows 2000.

6.3 Experimental Results

The detection results are shown in Table 6. In the table, the columns "C", "S", and "SD" represent Correct, the number of the patterns that were actually used, Static, the number of the patterns from static analysis only, and Static-and-Dynamic, the number of the patterns from both static and dynamic analyses, respectively. The rows "Precision" and "Recall" indicate one of the accuracy criteria and are calculated by using a set of detected results M and correct patterns C:

Table 6 Detection results.

Design patterns	Program #1			Program #2		
	C	S	SD	C	S	SD
Abstract Factory	0	0	0	0	1	0
Builder	0	0	0	0	10	0
Chain of Responsibility	0	1	1	1	2	1
Composite	1	1	1	0	0	0
Factory Method	0	0	0	1	1	1
Observer	1	8	1	0	0	0
Proxy	1	2	1	0	0	0
State	0	2	1	0	2	0
Strategy	0	2	1	0	2	0
Template Method	0	0	0	1	4	2
Visitor	1	1	1	0	2	0
Singleton	0	0	0	1	1	1
Precision		0.24	0.57		0.16	0.80
Recall		1.00	1.00		1.00	1.00

$$Precision = \frac{|M \cap C|}{|C|}, \qquad Recall = \frac{|M \cap C|}{|M|}.$$

The tool could detect all of the patterns that were actually used in both of the programs—maximum recalls. Furthermore, precisions are drastically improved by introducing dynamic analysis. As a whole program #1 and #2, we can observe the improvement of the precision from 0.19 to 0.67.

The dynamic analysis that we adopted was quite useful and effective, and it provided the information on the execution order of the method invocations and the identification of overriding methods. This information allowed the tool to filter out irrelevant candidates of patterns in early stage. In fact, more predicates of Prolog specifying the conditions on the order of method invocations and on overriding methods were used in the detection conditions compared to the predicates for static structures. They were frequently evaluated during the detection processes.

Table 7 shows the time taken to detect the patterns. The results in the case of using the hierarchical structure (our proposed approach) are included in the cells under the column of "Hierarchical conditions", while the column "Non-hierarchical conditions" is for the case of the non-hierarchical detection conditions. As for the total time for the detection, our proposed approach was 3.5 times as fast as non-hierarchical conditions in the case of the program #1, and 6.5 times faster in the program #2. This result shows that our approach of using hierarchical detection conditions is more effective.

As for the case shown in Table 6 that the value of "S" or "SD" is greater than that of "C", the tool correctly detected the parts of the programs where a pattern was really used,

Table 7 Detection time (unit: millisecond).

		Hierarchical conditions		Non-hierarchical conditions	
		Program #1	Program #2	Program #1	Program #2
Common conditions	for meta patterns	234	22078		
	for connection	218	12532		
Meta patterns	1:1 Connection	141	2015		
	1:N Connection	94	2719		
	1:1 Recursive Connection	156	1906		
	1:N Recursive Connection	62	2000		
	Unification	47	3678		
	1:1 Recursive Unification	62	1828		
	1:N Recursive Unification	94	5187		
Design patterns	Abstract Factory	62	2203	1031	57219
	Builder	203	2828	984	56844
	Chain of Responsibility	63	2234	204	27672
	Composite	156	1906	1250	55891
	Factory Method	47	2203	297	24359
	Observer	204	3297	1078	59578
	Proxy	156	3843	1109	60828
	State	187	2156	1016	56266
	Strategy	187	2125	1032	55859
	Template Method	62	2203	265	24141
	Visitor	156	2047	1203	56031
	Singleton	109	1969	110	1875
Total		2700	82957	9579	536563

but it took the wrong patterns. In the example of the program #1, the tool detected Chain of Responsibility, but its part was structured with Composite. The tool detected the occurrences of State and Strategy at the same part of the program #1, but Observer was actually used there. These errors resulted from the incorrect decision of the role of the methods as constituents of the patterns. The reason why State and Strategy patterns were detected at the same part is their similarity in static structure. In the case of the program #2, an occurrence of Template Method was detected but the pattern was not actually used. The detected part was included in the occurrence of Factory Method, which was correctly detected. This error arises from the fact that Factory Method can be a special form of Template Method.

These errors are not related to the structural and behavioral but the semantic differences. To detect these patterns precisely, we are considering to introduce some semantic analyses, such as program identification analysis [18], or heuristic-based approach, such as the technique using a machine learning [6]. Some of the failing reasons of our detection technique are considered as follows: 1) not followed variations of patterns, 2) irrelevant execution sequences, and 3) pattern that has no structural characteristics. To detect non-structural patterns, Prolog representation of a program behavior other than executions is required.

7. Related Work

There are several researches to automatically detect occurrences of design patterns, and some of them use metrics that measure the possibility of the existence of patterns [2]–[5]. These metrics techniques can be combined with a statistical approach and learning algorithms to improve the accuracy of detection [6].

In the works by Correa et al. and Krämer et al., the conditions that should be satisfied on the static structure of a source code are defined with FOPL. The pattern occurrences are detected by checking the true/false value of the conditions [7], [8]. To provide an easy way to describe the conditions on so-called pattern specifications, Balanyi et al. proposed an XML based language named DPML [9]. However, these approaches only use the static structure of a code such as inheritance, aggregation, association and method invocation relationships. Therefore it leads to inaccuracy when detecting patterns. In particular, in DPML, although Factory Method could be completely detected, the other patterns, Builder, Proxy, and Visitor, could be detected at correctness ratios of 14%, 50%, and 0%, respectively. The types of patterns that could be detected were limited.

Some approaches use Prolog or FOPL for reasoning about structural properties of object-oriented programs, including static method invocation properties [7], [8], [13]–[15]. In particular, the tool of Kniesel et al. is based on the static source code analysis and detects patterns rapidly [17]. However, most of them do not consider maintainability of pattern descriptions, i.e. programs in Prolog or FOPL expressions themselves.

Some researches use the dynamic data such as execution traces [10]. The combination of static and dynamic analyses seems to be one of the ways to achieve more accurate pattern detection [28]. However, the computation cost may be a problem of automated detection.

Since formal concept analysis helps us to extract the abstract static structures that commonly appear, it is applied to pattern mining as well as pattern detection [11], [12]. Pattern mining approach strongly requires time efficiency because they are applied to many projects and larger-scale software.

Semantic analysis using natural language processing, e.g. an analysis of identifications or class/method names on a source code, is also effective to detect design patterns. The work by Dong et al. indicates that the collaboration of the approaches brings good results [18], [19]. However, considerations of extensibility and maintainability of pattern descriptions are not mentioned.

8. Conclusion

In this paper, we proposed the approach to detect design patterns from a source code more accurately and efficiently. To improve accuracy, we adopted an integrated method of static and dynamic analyses. Furthermore, we focused on the common properties that design patterns have and represented them as Pree's meta patterns. The meta patterns helped us to hierarchically structure the properties of design patterns, and we could reduce the time costs of detection and the size of whole detection conditions. We adopted Prolog to represent the detection condition of design patterns, so that we could easily add and modify the detection conditions and implement the efficient detection of patterns by using the backtracking mechanism and the database function, i.e. `assert` and `retract` predicates. The database function was useful to avoid re-evaluation of the same detection conditions. Our supporting tool and the experiments showed the benefits of our approach. In particular, the result of our experiments shows that all of the pattern occurrences were detected. It means that our hierarchical detection conditions are accurate enough to detect design patterns. However, it was difficult to completely recognize correct patterns, we consider that some of these types of errors can be reduced by adopting a stronger dynamic analysis. To solve the others, we should consider the *semantic aspect* of patterns, not static structure or the usage of methods and classes during program execution. One of the differences between State and Strategy, which have the same structure, is that objects in State are created as *states* while objects in Strategy are created as *strategies*. It means that we should clarify the semantic differences between State and Strategy.

We can consider the following issues as future work:

Describing detection conditions for more patterns – We have shown that our hierarchical structure by using Pree's meta patterns is effective to describe conditions of design pattern detection. Confirmations to describe conditions for other design patterns, e.g. patterns in Grand's catalog [29] or J2EE patterns [20], are effective.

Visualizing patterns – It is hard to understand constitutions and communications of patterns because our tool's outputs are just facts in Prolog. We consider a visualizing technique to enhance the understandings of the patterns. In particular, the tool to visualize classes corresponding to multiple patterns is desired.

Case study of larger scale – Our case study in this paper is pre-limited, so we should apply our technique to larger scale one.

Acknowledgments

The authors would like to thank Mr. Rodion Moiseev, Mr. Hironobu Kanai, and anonymous reviewers of IEICE for their valuable comments to the earlier version of this paper.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[2] H. Kim and C. Boldyreff, "A method to recover design patterns using software product metrics," Proc. 6th International Conference on Software Reuse, pp.318–335, 2000.
[3] T. Muraki and M. Saeki, "Metrics for applying GoF design patterns in refactoring processes," Proc. 4th International Workshop on Principles of Software Evolution, pp.27–36, 2001.
[4] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactoring via change metrics," Proc. 15th Conference on Object-Oriented Programming, Systems, Languages and Applications, pp.166–177, 2000.
[5] Y.G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," Proc. 11th Working Conference on Reverse Engineering, pp.172–181, 2004.
[6] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," Proc. 21st International Conference on Software Maintenance, pp.295–304, 2005.
[7] A.L. Correa, C.M.L. Werner, and G. Zaverucha, "Object oriented design expertise reuse: An approach based on heuristics, design patterns and anti-patterns," Proc. 6th International Conference on Software Reuse, pp.336–352, 2000.
[8] C. Krämer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," Proc. 3rd Working Conference on Reverse Engineering, pp.208–215, 1996.
[9] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," Proc. 19th International Conference on Software Maintenance, pp.305–314, 2003.
[10] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe, "Automatic design pattern detection," Proc. 11th International Workshop on Program Comprehension, pp.94–103, 2003.
[11] P. Tonella and G. Antoniol, "Object oriented design pattern inference," Proc. 15th International Conference on Software Maintenance, pp.230–239, 1999.
[12] G. Arevalo, F. Buchli, and O. Nierstrasz, "Detecting implicit collaboration patterns," Proc. 11th Working Conference on Reverse Engineering, pp.122–131, 2004.
[13] R. Wuyts, "Declarative reasoning about the structure object-oriented systems," Proc. TOOLS USA '98 Conference, pp.112–124, 1998.
[14] J.M. Smith and D. Stotts, "SPQR: Flexible automated design pattern extraction from source code," Proc. 18th International Conference on Automated Software Engineering, pp.215–224, 2003.
[15] D. Heuzeroth, S. Mandel, and W. Loẅe, "Generating design pattern detectors from pattern specifications," Proc. 18th International Conference on Automated Software Engineering, pp.245–248, 2003.
[16] N. Shi and R.A. Olsson, "Reverse engineering of design patterns from Java source code," Proc. 21st International Conference on Automated Software Engineering, pp.123–134, 2006.
[17] G. Kniesel, J. Hannemann, and T. Rho, "A comparison of logic-based infrastructures for concern detection and extraction," Proc. 3rd Workshop on Linking Aspect Technology and Evolution, 2007.
[18] J. Dong, D.S. Lad, and Y. Zhao, "DP-Miner: Design pattern discovery using matrix," Proc. 14th International Conference and Work-

shops on the Engineering of Computer-Based Systems, pp.371–380, 2007.

[19] J. Dong and Y. Zhao, "Experiments on design pattern discovery," Proc. 3rd International Workshop on Predictor Models in Software Engineering, 2007.

[20] J. Crupi, D. Malks, and D. Alur, Core J2Ee Patterns: Best Practices and Design Strategies, Prentice Hall PTR, 2001.

[21] W. Pree, Design Pattern for Object-Oriented Software Development, Addison-Wesley, 1996.

[22] Eclipse project, available at http://www.eclipse.org/

[23] Eclipse Java development tools (JDT) subproject, available at http://www.eclipse.org/jdt/

[24] E. Denti, A. Omicini, and A. Ricci, "Multi-paradigm Java-Prolog integration in tuProlog," Science of Computer Programming, vol.57, no.2, pp.217–250, 2005.

[25] tuProlog, available at http://tuprolog.alice.unibo.it/

[26] G.T. Leavens, A.L. Baker, and C. Ruby, "JML: Java modeling language," available at http://www.eecs.ucf.edu/˜leavens/JML/

[27] J. Vlissides, Pattern Hatching: Design Patterns Applied, Addison-Wesley, 1998.

[28] L. Wendehals and A. Orso, "Recognizing behavioral patterns at runtime using finite automata," Proc. 2006 International Workshop on Dynamic Systems Analysis, pp.33–40, 2006.

[29] M. Grand, Patterns in Java, John Wiley & Sons, 1998.

Shinpei Hayashi received a B. Eng. degree in information engineering from Hokkaido University in 2004, and a M. Eng. degree in computer science from Tokyo Institute of Technology in 2006. He is currently a Ph.D. student in computer science at Tokyo Institute of Technology. His research interests include software evolution and refactoring, software patterns, software development environment, and mining software repositories.

Junya Katada received B. Eng. and M. Eng. degrees in computer science from Tokyo Institute of Technology in 2003 and 2005, respectively. He is currently working at NTT COMWARE CORPORATION. His research interests include software patterns, software architecture, software design methods, and software development environment.

Ryota Sakamoto received a B. Eng. degree in computer science from Tokyo Institute of Technology in 2006. He is currently a master course student in computer science at Tokyo Institute of Technology. His research interests include software testing, software patterns, human eye modeling, and human visual processing.

Takashi Kobayashi received B. Eng., M. Eng., and Ph.D. degrees in computer science from Tokyo Institute of Technology in 1997, 1999, and 2004, respectively. He is currently a designated associate professor of Center for Embedded Computing Systems (NCES), Graduate School of Information Science, Nagoya University. His research interests include software patterns and architecture, software development method, software configuration management, Web-services compositions, workflow, multimedia information retrieval, and data mining. He is a member of IPSJ, JSSST, DBSJ, and ACM.

Motoshi Saeki received a B. Eng. degree in electrical and electronic engineering, and M. Eng. and Ph.D. degrees in computer science from Tokyo Institute of Technology, in 1978, 1980, and 1983, respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).