Design Pattern Detection using Software Metrics and Machine Learning

Satoru Uchiyama
Hironori Washizaki
Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
Tokyo, Japan
s.uchiyama1104@toki.waseda.jp
washizaki@waseda.jp
fukazawa@waseda.jp

Atsuto Kubo
Aoyama Media Laboratory
Tokyo, Japan
kubo@nii.ac.jp

***Abstract*—The understandability, maintainability, and reusability of object-oriented programs could be improved by automatically detecting well-know design patterns in programs. Many of the previous detection techniques are based on static analysis and use strict conditions composed of class structure information. Hence, it is difficult for them to detect design patterns in which the class structures are similar. Moreover, it is difficult for them to deal with diversity of design pattern applications. We propose a design pattern detection technique using metrics and machine learning. Our technique judges candidates for the roles that compose the design patterns by using machine learning and measurements of metrics, and it detects design patterns by analyzing the relations between candidates. It suppresses false negative and distinguishes patterns in which the class structures are similar. We conducted experiments comparing our technique with two previous techniques. These results showed that our technique was more accurate than the previous techniques.**

Keywords—component; Object-oriented software, Design pattern, Software metrics, Machine learning

I. Introduction

Design patterns (hereafter, patterns) are defined as descriptions of communicating classes that form a common solution to a common design problem. GoF patterns [1] are representative patterns for object-oriented software. Patterns are composed from classes that describe the roles and abilities of objects. For example, Figure 1 shows one GoF pattern, `State` pattern. `State` pattern is composed of roles named `Context`, `State`, and `ConcreteState`. The use of patterns enables software development with high maintainability, high reusability, and improved understandability, and it facilitates smooth communications between developers.

Programs implemented by a third party and open source software may take a lot of time to understand, and patterns may be applied without having to describe class names, comments, or attached documents in existing programs. So pattern detection improves the understandability of programs. However, manual pattern detection from the existing programs is inefficient. Moreover, developers might overlook patterns during manual detection.

Many studies on using automatic detection of patterns to solve the above problems have used static analysis. However, in static analysis, it is difficult to identify patterns to which class structures are similar and patterns with few features. In addition, there is still a possibility that software developers might overlook patterns if they use strict conditions like the class structure analysis, and applied patterns vary from the intended conditions even a little.

We propose a pattern detection technique using software metrics (hereafter, metrics) and machine learning. Although our technique can be classified as a type of static analysis, unlike previous detection techniques, it detects patterns by using identifying elements derived by the machine learning using measurements of metrics without using strict condition descriptions (class structural information, etc.). Metrics mean a set of a quantitative standard "Metric" that can be used to evaluate the software development from various aspects. For example, one such metric, number of methods (NOM), means the number of methods in a class [2]. Moreover, by using machine learning, we can, in some cases, get previously unknown identifying elements from combinations of metrics. To cover a diversity of pattern applications, our method uses a variety of learning data because our technique result may depend on the kind and number of learning data used during the machine learning stage. Finally, we conducted experiments comparing our technique with two previous techniques. These results showed that our technique was more accurate than the previous techniques.

II. Previous Design Pattern Detection Techniques and Their Problems

Many of the previous detection techniques use static analysis [3][4]. These techniques chiefly analyze information like class structures that satisfy certain conditions. If they vary from the intended strict conditions even a little, or two or more roles are assigned in a class, there is a possibility that developers might overlook something.

There is the technique for detecting patterns from the degrees of similarity between graphs of pattern structure and graphs of programs to be detected [3]. However,

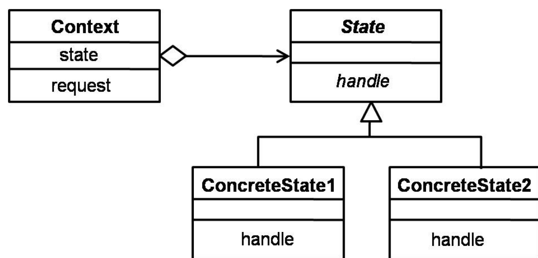

Figure 1. State pattern

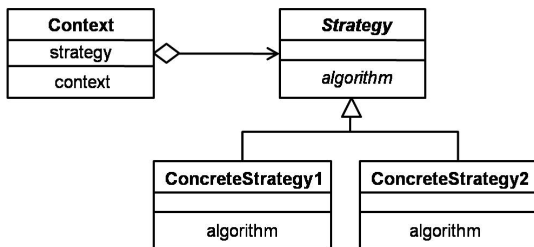

Figure 2. Strategy pattern

distinguishing `State` pattern from `Strategy` pattern is difficult because their class structures are similar (see Figures 1 and 2). Unlike this method, we derive distinguishing elements and machine learning and detect patterns that are similar in terms of these metrics. In addition, this technique [3] is open to the public to the web as the tool.

There is a technique for outputting pattern candidates from features of metric measurements [5]. However, it requires manual confirmation; this technique can roughly identify pattern candidates, but the final choice depends on the developer's skill. Our technique detects patterns without manual filtering by using not only distinguishing elements determined by metrics and machine learning but also by analyzing class structure information.

There is a technique for improving precision by filtering the detection results by using machine learning. This technique inputs measurements on the classes and methods of each pattern [6]. However, it uses the existing static analytical technique, whereas our technique uses machine learning throughout the whole process without the existing technique.

There is a technique that analyzes programs both before and after patterns are applied [7]. This technique requires a revision history of the programs used. Our technique detects patterns by analyzing only the current programs.

There is a technique for detecting patterns from the class structure and behavior of a system after classifying its patterns [8][9]. It is difficult to use it when patterns are applied more than once and when there is a diversity of patterns application. In contrast, our technique can deal with patterns that are applied more than once and it can deal with a diversity of pattern application.

There are also detection techniques using dynamic analysis. These methods identify patterns by referring to the execution route information, etc., of a program [10][11]. However, it is difficult to analyze the whole execution route and use fragmentary class sets in an analysis. Additionally, the result of dynamic analysis depends on the representativeness of the execution sequences.

There are detection techniques by using multilayered (multiphase) approach [12][13]. [12] is two phases approach by static analysis. However it is difficult to detect creational patterns and behavioral patterns because this technique analyzes pattern structures and source code level constraints. [13] is three-layered approach "DeMIMA". This consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify patterns in the abstract model. However distinguishing the `State` pattern from the `Strategy` pattern is difficult because their structures are identical. Our technique can detect patterns of all categories and try distinction of `State` pattern and `Strategy` pattern by using metrics and machine learning.

There is a technique for detecting patterns with formal definitions in OWL (Web Ontology Language) [14]. However, false negative occurs because this technique doesn't accommodate to the transformed pattern. In addition, this technique [14] is open to the public to the web as the Eclipse plug-in.

We suppress false negative by accommodating the diversity of pattern applications and distinguish patterns in which the class structures are similar with metrics and machine learning. Finally, only the previous techniques [3] [14] in the above-mentioned have been released as the tool.

III. OUR TECHNIQUE

Our technique is composed of a learning phase and a detection phase. The learning phase is composed of three processes, and the detection phase is composed of two processes, as shown in Figure 3. Each process is described, with pattern specialists and developers included as parties concerned, as follows. Our technique currently uses Java as the program language.

[Learning Phase]

P1. Define patterns

Pattern specialists determine the detectable patterns and define the structures and roles composing these patterns.

P2. Decide metrics

Pattern specialists determine useful metrics to judge the roles defined in P1 by using the Goal Question Metric decision technique.

P3. Machine learning

Pattern specialists get measurements on each role in programs to which patterns have already been applied by using the metrics defined in P2 and input them to the machine learning system. They verify the role judgments after learning, and if the verification result is not good, they return to P2 and revise the metrics.

[Detection Phase]

P4. Role candidate judgment

Developers get measurements for each class in the programs to be detected by using the metrics defined in P2. These are input to the machine learning system, which uses them to judge role candidates.

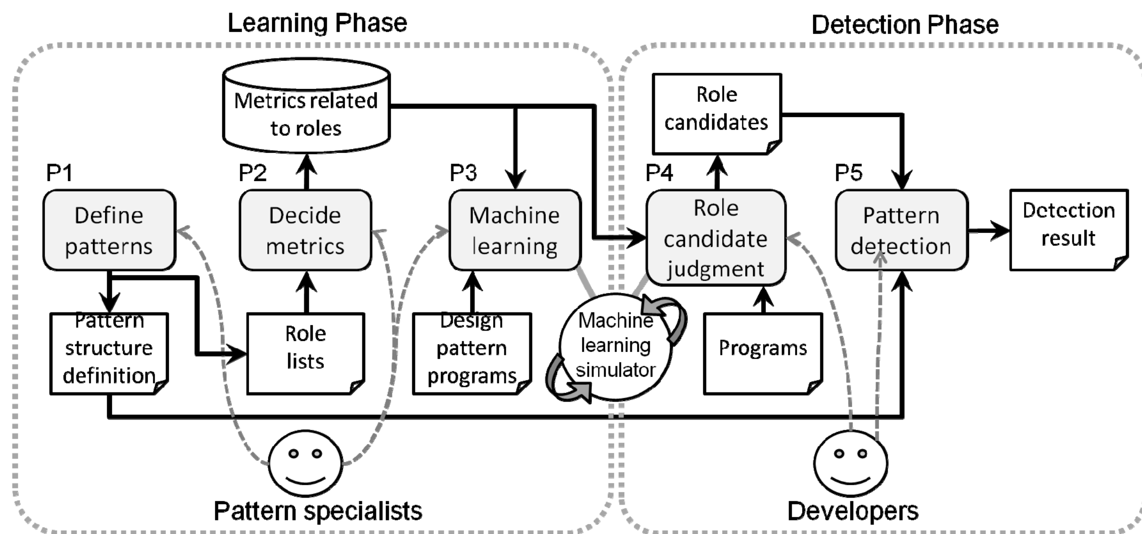

Figure 3. Entire image of our technique

P5. Pattern detection

Developers detect patterns by using the pattern structure definitions defined in P1 and the role candidates determined in P4. The structure definitions correspond to the letters P, R, and E of section III-ii.

III-i Learning Phase

P1. Define patterns

Currently, our technique treats five GoF patterns (`Singleton`, `TemplateMethod`, `Adapter`, `State`, and `Strategy`) and 12 roles. The GoF patterns are grouped into creational patterns, structural patterns, and behavioral patterns. Our technique uses these patterns to cover these groups.

P2. Decide metrics

Pattern specialists decide useful metrics to judge roles by using the Goal Question Metric decision technique [14] (hereafter, GQM). GQM is a top-down approach to clarifying relations between goals and metrics.

We experimented on judging roles by using general metrics without GQM. However, the machine learning result was not appropriate, since some metrics tended unstable. Consequently, we choose GQM so that the machine learning can function appropriately by distinguishing and stable metrics in each role. In our technique, the pattern specialists define the judgment of a role as a goal. Next, they define a set of questions that should be evaluated for achievement of goal. Finally, they decide useful metrics to help answer the questions. Moreover, we decide questions by paying attention to the attributes and operations of the roles. The pattern specialist figures out the attributes and the operations at each role necessary for the composition of the pattern by reading the description of the pattern definition. Next, they define whether those attributes and operations are defined as a question.

For example, Figure 4 shows the goal of making a judgment about the `AbstractClass` role in the `TemplateMethod` pattern. `AbstractClass` roles have abstract methods or methods using written logic that use abstract methods like Figure 5. `AbstractClass` role can be distinguished by the ratio of the number of methods to the number of abstract methods because the number of methods exceeds the number of abstract methods. Therefore, number of abstract methods (NOAM) and number of methods (NOM) are useful metrics for judging this role.

The lack of questions might occur because GQM is qualitative. Therefore, if the role candidate judgment is not a good one, the procedure loops back to P2 to attempt an improvement. Moreover, it will be possible to apply GQM to roles with new patterns in the future. The appendix in this paper shows the results of applying GQM to the roles of all detection targets.

P3. Machine learning

The machine learning is a technology that analyzes sample data by computer and acquires useful rules to make forecasts about unknown data. We used the machine learning so that the patterns have a variety of application forms. The machine learning suppresses false negative and achieves gradual detection.

Our technique uses a neural network [16] algorithm. On the other hand, support vector machine [16] could also be used to distinguish a pattern of two groups by using linear input elements. However, we chose a neural network because it outputs the values to all roles in consideration of the dependence of the different metrics. Therefore, it can deal with cases in which one class has two or more roles.

A neural network is composed of an input layer, hidden layers, and an output layer, as shown in Figure 6, and each layer is composed of elements called units. Weights are given when a value moves from unit to unit, and a judgment rule is acquired by changing the weights. A typical algorithm for adjusting weights is back propagation. Back propagation calculates an error margin between output result y and the correct answer T, and it sequentially adjusts weights from the

layer nearest the output to the input layer, as shown in Figure 7. These weights are adjusted until the output error margin of the network reaches a certain value.

Our technique uses a hierarchical neural network simulator [17]. This simulator uses back propagation. The hierarchy number in the neural network is set to three, the number of units of the input layer and the hidden layer are set as the number of chosen metrics, and the number of units of the output layer is set as the number of roles for the judgment. The input is the metric measurements of each role measured in a program to which patterns have already been applied, and the output is an expected role. The learning number of times is decided when the error margin curve of the simulator converges. The convergence of the error margin curve is manually determined at present.

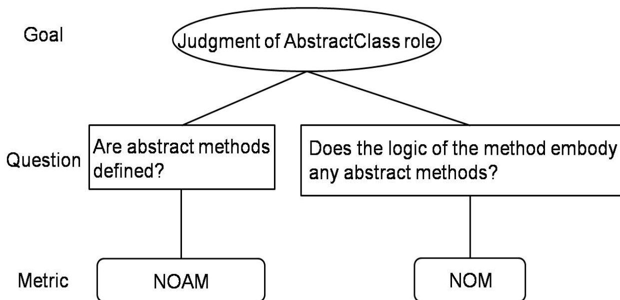

Figure 4. Example of GQM（AbstractClass role）

```
public abstract class AbstractDisplay {
    public abstract void open();
    public abstract void print();
    public abstract void close();
    public final void display() {
        open();
        for (int i = 0; i < 5; i++) {
            print();
        }
        close();
    }
}
```

Figure 5. Example of source code（AbstractClass role）

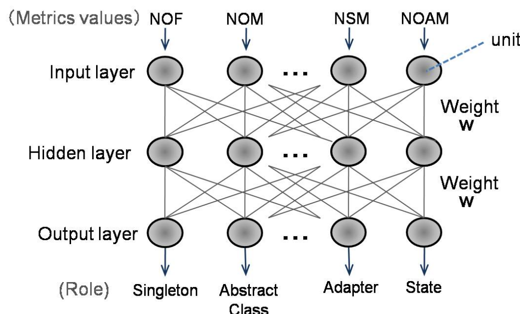

Figure 6. Neural network

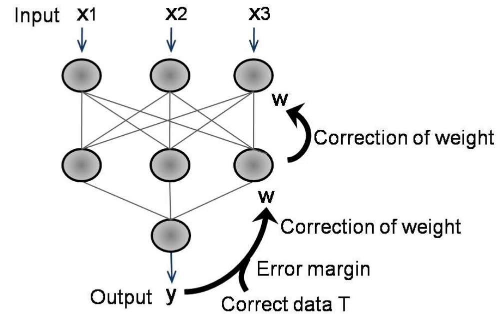

Figure 7. Back propagation

III-ii Detection Phase

P4. Role candidate judgment

Metric measurements are measured on each class in the programs. These measurements are input to the machine learning simulator, and values between 0–1 are output to all roles to be judged. The output values are normalized such that the sum of all values becomes 1. These output values are called role agreement values. A larger role agreement value means that the role candidate is more likely correct. The reciprocal of the number of roles to be detected is set as a threshold, and the role agreement values that are higher than the threshold are taken to be role candidates. The threshold is 1/12=0.0834 because we treat 12 roles at present.

Let us consider the class that assumes `AbstractClass` NOM of 3 and NOAM of 2 and other metrics of 0. Figure 8 shows the role candidate judgment results with these measurements; the output value of `AbstractClass` is the highest value. The roles are judged to be `AbstractClass` and `Target` by regularizing the values of Figure 8.

P5. Pattern Detection

Patterns are detected by searching for relations between role candidates with the pattern structure. The search goes sequentially from the role candidate with the highest agreement value to the one with the lowest value. All combinations of the role candidates that accord with the pattern structures are searched. Patterns are detected when the directions of relations between role candidates accord with the pattern structure and when the role candidates accord with roles at both ends of the relations. Moreover, the relation agreement values reflect the kind of relation.

Currently, our method deals with inheritance, interface implementation, and aggregation relations. These kind will increase as more patterns get added in the future. The relation agreement value is 1.0 when the kind agrees with the relation of the pattern, and it is 0.5 when the kind does not agree. If the relation agreement value is 0 when the kind of relation does not agree, the pattern agreement value might become 0, and these classes will not be detected as the pattern. In that case, a problem similar to those of the previous detection techniques will occur because the difference of the kind of the relation is not recognized.

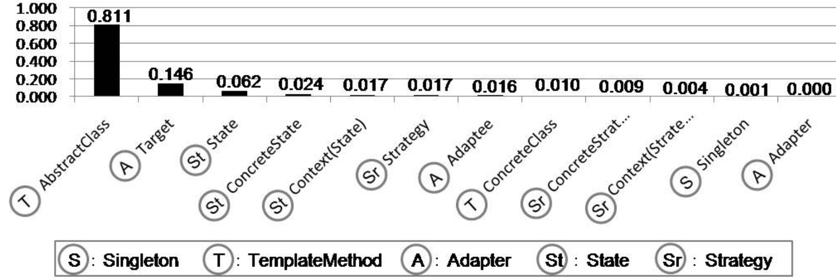

Figure 8. Example of machine learning output

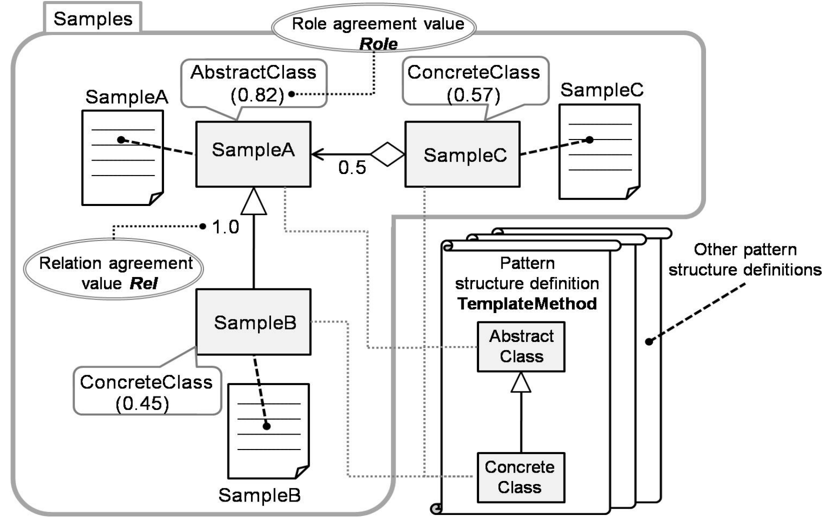

Figure 9. Example of pattern detection (TemplateMethod pattern)

The pattern agreement value is calculated from the role agreement values and the relation agreement values. The pattern to be detected is denoted as P, the role set that composes the pattern is denoted as R, and the relation set is denoted as E. Moreover, the program that is the target of detection is defined as P', the set of classes judged role candidates is R', the set of relations between elements of R' is denoted as E'. The role agreement value is denoted as *Role*, and the relation agreement is denoted as *Rel*. The product of the average of two roles at both ends of the relation and *Rel* is denoted as *Com*, and the average of *Com* is denoted as *Pat*. Moreover, the average of two *Role*s is calculated when *Com* is calculated, and the average value of *Com* is calculated to adjust *Pat* and *Role* to values from 0 to 1 when *Pat* is calculated. If the directions of the relations do not agree, *Rel* is assumed to be 0.

$$P = (R, E) \qquad P' = (R', E')$$

$$R = \{r_1, r_2, \ldots, r_i\} \qquad R' = \{r'_1, r'_2, \ldots, r'_k\}$$

$$E = \{e_1, e_2, \ldots, e_j\} \subseteq R \times R \qquad E' = \{e'_1, e'_2, \ldots, e'_l\} \subseteq R' \times R'$$

$$Role(r_m, r'_n) = \text{The output of machine learning} \qquad r_m \in R, r'_n \in R'$$

$$Rel(e_p, e'_q) = \text{The relation agreement value} \qquad e_p \in E, e'_q \in E'$$

$$Com(e_p, e'_q) = \frac{Role(r_a, r'_b) + Role(r_c, r'_d)}{2} \times Rel(e_p, e'_q)$$

$$r_a, r_c \in R, r'_b, r'_d \in R', e_p = (r_a, r_c), e'_p = (r'_b, r'_d)$$

$$Pat(P, P') = \frac{1}{\left|\left\{(e_p, e'_q) \in E \times E' \middle| Rel(e_p, e'_q) > 0\right\}\right|} \sum_{e_p \in E, e'_q \in E'} Com(e_p, e'_q)$$

Figure 9 shows an example of detecting the `TemplateMethod` pattern. In this example, it is assumed that class `SampleA` has the highest role agreement value for an `AbstractClass`. The pattern agreement value between the program Samples and the `TemplateMethod` pattern is calculated with the following algorithm.

$$TemplateMethod = (R, E)$$

$$R = \{AbstractClass, ConcreteClass\}$$

$$E = \{AbstractClass \triangleleft - ConcreteClass\}$$

$$Samples = (R', E')$$

$$R' = \{SampleA, SampleB, SampleC\}$$

$$E' = \{SampleA \triangleleft - SampleB, SampleA \leftarrow\!\diamond SampleC\}$$

($\triangleleft -$: inheritance, $\leftarrow\!\diamond$: aggregation)

$$Role(r_1, r'_1) = 0.82 \quad Role(r_2, r'_2) = 0.45 \quad Role(r_2, r'_3) = 0.57$$

$$Rel(e_1, e'_1) = 1.0 \quad Rel(e_1, e'_2) = 0.5$$

$$Com(e_1, e'_1) = \frac{0.82 + 0.45}{2} \times 1.0 = 0.635$$

$$Com(e_1, e'_2) = \frac{0.82 + 0.57}{2} \times 0.5 = 0.348$$

$$Pat(P, P') = \frac{1}{2} \times (0.635 + 0.348) = 0.492$$

In the program shown in Figure 9, the pattern agreement value of the `TemplateMethod` pattern was calculated to be 0.492. Pattern agreement values are normalized from 0 to 1, just like the role agreement values. The classes having pattern agreement value that exceeds threshold are output as the detection result. The reciprocal of the number of roles for detection is taken to be the threshold (0.0834), similar to the case of role candidate judgment, and pattern agreement values that are higher than the threshold are output as the detection result.

In Figure 9, `SampleA`, `SampleB`, and `SampleC` were detected as `TemplateMethod` patterns. Moreover, `SampleA` and `SampleB`, `SampleA` and `SampleC` can also be considered to be the `TemplateMethod`. In this case, the relation of "`SampleA` ◁– `SampleB`" is more similar to a `TemplateMethod` pattern than the relation of "`SampleA`←◇ `SampleC`" is because its agreement value is 0.635, whereas that of the relation of "`SampleA` ←◇ `SampleC`" is only 0.348.

IV. Evaluation and Discussion

We determined whether the machine learning simulator derived identifying elements of the roles after learning. Moreover, we compared our technique with two previous techniques to verify the precision and recall of our technique and to confirm whether it could match its detected patterns with similar structures and diverse patterns.

IV-i Verification of Role Candidate Judgement

We used cross validation to verify the role candidate judgment. In cross validation, data are divided into n groups, and a test to verify a candidate judgment is executed such that the testing data are one data group and the learning data are n-1 data groups. We executed the test five times by dividing the data into five groups. In this paper, programs like codes in the reference [18], etc., are called small-scale, whereas programs in practical use are called large-scale. The parts where patterns are applied in small-scale codes (50 places in total) [1] [18][19] and large-scale codes (158 places in total from the Java library 1.6.0_13 [20], JUnit 4.5 [21], and Spring Framework 2.5 RC2 [22]) were used as experimental data. The parts where patterns were applied in real codes were manually collected.

Table I shows the metrics that were chosen for the small-scale and large-scale codes. We used different metrics by the sizes of the codes. For instance, we chose the metric called number of methods generating instance (NMGI) for the small-scale codes because the method for transit states in the `ConcreteState` role in the `State` pattern generates other `ConcreteState` roles in the small-scale codes. We didn't use NMGI as a metrics of the large-scale codes because there were unnecessary attributes and operations in the composition of patterns and there are little implementation specialized in `State` pattern such as the above-mentioned. We used NMGI when the program to be detected is obviously similar to small-scale codes of patterns, and it should not be used when the program is large-scale codes used by a third party.

We focused our attention on the recall because the purpose of our technique is detection covering diverse pattern applications. Recall shows how free of leakage the detection result is, whereas precision shows how free of disagreement the detection result is. Table II is used to calculate the recall. w_r, x_r, y_r, and z_r are numbers of roles, and w_p, x_p, y_p, and z_p are numbers of patterns. Recall is calculated from the data in Table II by the following expressions.

Recall of role candidate judgment : $Re_r = \frac{w_r}{w_r + x_r}$

Table III shows the average recall for each role. Role candidates must be judged accurately, because the `State` pattern and `Strategy` pattern have the same class structure. Therefore, roles other than the `State` and `Strategy` patterns are assumed to have been judged accurately when the role agreement value is more than the threshold, whereas the roles of the `State` pattern and `Strategy` patterns are assumed to be have been judged accurately when the role agreement value is more than the threshold and both are distinguished.

The recalls for the large-scale codes are lower than those for the small-scale codes in Table III. The accurate judgments on the large-scale codes were more difficult because those codes possessed unnecessary attributes and operations for the composition of the patterns. Therefore, it will be necessary to collect a lot of learning data to cover a variety of large-scale codes.

Table III's results are for when the `State` pattern and `Strategy` pattern could be distinguished. The `Context` role had high recall, but `State` and `ConcreteState` roles had especially low recalls for large-scale codes. However, the candidates for the `State` role were output with high recall when the threshold was exceeded. Therefore, the `State` pattern can be distinguished by starting searching from the `Context` role in P5, and this improves recall.

TABLE I. CHOSEN METRICS

Abbreviation	Content
NOF	Number of fields
NSF	Number of static fields
NOM	Number of methods
NSM	Number of static methods
NOI	Number of interfaces
NOAM	Number of abstract methods
NORM	Number of overridden methods
NOPC	Number of private constructors
NOTC	Number of constructors with argument of object type
NOOF	Number of object fields
NCOF	Number of other classes with field of own type
NMGI	Number of methods to generate instances

[1] All small-scale codes :
http://www.washi.cs.waseda.ac.jp/ja/paper/uchiyama/dp.html

TABLE II. INTERSECTION PROCESSION

	Number detected	Number not detected
Number of agreement	w_r, w_p	x_r, x_p
Number of non-agreement	y_r, y_p	z_r, z_p

TABLE III. RECALL OF ROLE CANDIDATE JUDGMENT (AVERAGE)

		Average of recall (%)	
Pattern	Role	Small-scale codes	Large-scale codes
Singleton	Singleton	100.0	84.7
Template Method	AbstractClass	100.0	88.6
	ConcreteClass	100.0	58.5
Adapter	Target	90.0	75.2
	Adapter	100.0	66.7
	Adaptee	90.0	60.9
State	Context	60.0	70.0
	State	60.0	46.7
	ConcreteState	82.0	46.6
Strategy	Context	80.0	55.3
	Strategy	100.0	76.7
	ConcreteStrategy	100.0	72.4

IV-ii Pattern Detection Results

Patterns are detected by our technique with pattern application parts as test data in both the small-scale codes and large-scale codes, and this result is evaluated. Table IV shows precision and recall of the detected patterns. Precision and recall are calculated from the data in Table II by the following expressions.

Recall of pattern detection : $$\mathrm{Re}_p = \frac{w_p}{w_p + x_p}$$

Precision of pattern detection : $$\mathrm{Pr}_p = \frac{w_p}{w_p + y_p}$$

The small-scale and large-scale codes shared a common point in that they both had recalls that were higher than precisions. This agrees with the purpose of suppressing achieves gradual detection. However, there were many non-agreements about the `State` patterns and `Strategy` patterns in the large-scale codes. To avoid this problem, we will have to find the best threshold.

TABLE IV. PRECISION AND RECALL RATIO OF PATTERN DETECTION

	Number of test data		Precision (%)		Recall (%)	
Pattern	Small-scale codes	Large-scale codes	Small-scale codes	Large-scale codes	Small-scale codes	Large-scale codes
Singleton	6	6	60.0	63.6	100.0	100.0
Template Method	6	7	85.7	71.4	100.0	83.3
Adapter	4	7	100.0	100.0	90.0	60.0
State	2	6	50.0	40.0	100.0	66.6
Strategy	2	6	66.7	30.8	100.0	80.0

Recall was 90 percent or more on the small-scale codes, but it dropped as low as 60 percent on the large-scale codes.

The large-scale codes gave especially low recall for the `Adapter` pattern. Table III shows the cause: the recall of the role candidate judgment for the `Adapter` pattern was not high enough. It is necessary to show that agreement values of all roles that compose patterns are above the threshold so that patterns will be detected. There were many cases in which neither of the roles that composed patterns was judged as a role candidate in the `Adapter` pattern. It will be necessary to return to P2 and choose new metrics. The `State` pattern was distinguished by searching from the `Context` role, as in the `State` pattern detection in the large-scale codes, and the recall of the pattern detection was higher than the recall of role candidate judgment.

IV-iii Experiment Comparing Previous Detection Technique

We experimentally compared our technique with previous detection techniques [3][14]. These previous techniques have been publicly released, and they treat three or more patterns that our technique treats. These techniques target Java program as well as our technique. The technique proposed by Tsantails [3] (hereafter, TSAN) has four patterns in common with ours (`Singleton`, `TemplateMethod`, `Adapter` and `State/Strategy`). Because this technique cannot distinguish the `State` pattern from the `Strategy` pattern, these are detected as one pattern. Dietrich's technique [14] (hereafter, DIET) has three patterns in common (`Singleton`, `TemplateMethod`, `Adapter`). TSAN detects patterns from the degree of similarity between the graphs of the pattern structure and graphs of the programs to be detected, whereas DIET detects patterns by using formal definitions in OWL (Web Ontology Language). Patterns were detected and evaluated with the small-scale and large-scale test data. Moreover, the test data and learning data were different.

Figure 10 shows the recall-precision graphs of our technique and TSAN, and Figure 11 shows the recall-precision graphs of our technique and DIET. We ranked the detection results of our technique with the pattern agreement values. Next, we calculated recall and precision according to the ranking and plotted them. Recall and precision were calculated from the data in Table II by using the expressions of paragraph IV-ii. In TSAN and DIET, we ranked alternately agreement results and non-agreement results because results of the previous detection techniques were no value to rank. In recall-precision graph, higher plots are better.

Figures 10 and 11 show particularly good results in the case used small-scale codes for all techniques. This reason is that small-scale codes don't include unnecessary attributes and operations in the composition of patterns.

Our technique distinguished `State` pattern `Strategy` pattern. Table V is an excerpt of the metrics measurements for the Context role in `State` pattern and `Strategy` pattern that were distinguished by the experiment on the large-scale codes. `State` pattern treats the states in `State`

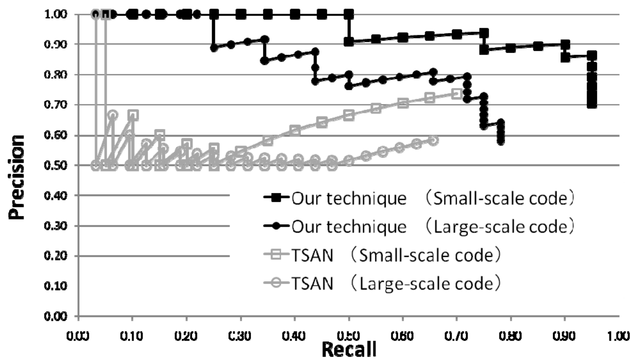

Figure 10. Recall-precision graph of the detection result (vs. TSAN)

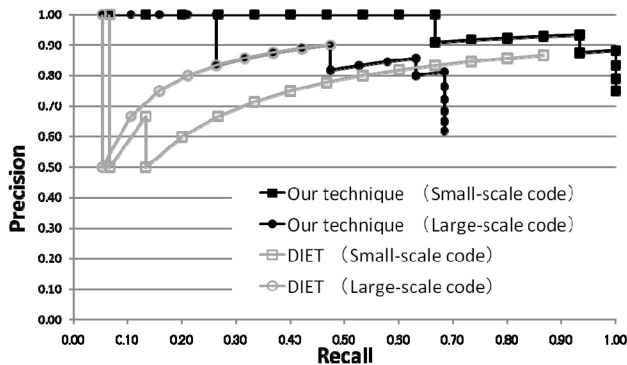

Figure 11. Recall-precision graph of the detection result (vs. DIET)

TABLE V. EXAMPLE MEASUREMENTS OF THE CONTEXT ROLE'S METRICS

Pattern - Role	Number of fields	Number of methods
State - Context	12	58
	45	204
	11	72
Strategy - Context	18	31
	3	16
	3	5

role and treats the actions of the states in the `Context` role. `Strategy` pattern encapsulates the processing of each algorithm into a `Strategy` role, and `Context` processing becomes simpler compared with that of `State` pattern. Table V shows 45 fields and 204 methods as the largest in `Context` role in `State` pattern (18 and 31 respectively in `Context` role of `Strategy` pattern). Therefore, the complexity of `Context` role of both patterns appears in the number of fields and the number of methods, and these are distinguishing elements. Figure 10 shows that our technique is especially good because `State` pattern and `Strategy` pattern could not be distinguished with TSAN.

Figure 11 shows that the recall of DIET is low in the case of large-scale codes because this technique doesn't accommodate the diversity of pattern applications. Additionally, large-scale codes not only contain many attributes and operations in the composition of patterns but also subspecies of patterns.

Therefore, our technique is superior to previous one because the curve of our technique is above the previous in Figures 10 and 11.

Table VI and VII show the average F measure for each plot of Figure 10 and 11. The F measure is calculated with recall and precision calculated by the above-mentioned expression as follows.

$$F = \frac{1}{\frac{1}{2\,\mathrm{Pr}_p} + \frac{1}{2\,\mathrm{Re}_p}}$$

A large F measure means higher accuracy, and these tables show that our technique had a larger F measure than the previous techniques had.

Our technique detected subspecies of patterns. For example, our technique detected the source code of the `Singleton` pattern that used the boolean variable as shown in Figure 12. This `Singleton` pattern was not detected in TSAN or DIET. However, unlike the previous techniques, our technique is affected by false positives because it is a gradual detection using metrics and machine learning instead of strict conditions. False positives of the `Singleton` pattern especially stood out because `Singleton` pattern is composed of only one role. It will be necessary to use metrics that are specialized to one or a few roles to make judgments about patterns composed of one role like the `Singleton` pattern (P4).

The overall evaluation is that our technique is superior to previous one because of the above-mentioned graphs and the F measure.

TABLE VI. THE AVERAGE OF F MEASURE (VS. TSAN)

	Small-scale codes	Large-scale codes
Our technique	0.67	0.56
Previous technique (TSAN)	0.39	0.36

TABLE VII. THE AVERAGE OF F MEASURE (VS. DIET)

	Small-scale codes	Large-scale codes
Our technique	0.69	0.55
Previous technique (DIET)	0.50	0.35

```
class Connection{
  public static boolean haveOne = false;
  public Connection() throws Exception{
    if (!haveOne) {
       haveOne = true;
    }else {
      throw new Exception(
      "cannot have a second instance");
    }
  }
  public static Connection getInstance()
  throws Exception{
  ...}
}
```

Figure 12. Example of diversity of pattern application (Singleton pattern)

V. Conclusion and Future Work

We devised a pattern detection technique using metrics and machine learning. Role candidates are judged by using machine learning that uses measured metrics, and patterns are detected from the relations between classes. We worked on the problems associated with overlooking patterns and distinguishing patterns in which the class structures are similar.

We demonstrated that our technique was better than the previous detection technique by experimentally distinguishing patterns to which the class structures are similar. Moreover, subspecies of patterns were detected, so we could deal with the large diversity of patterns applications. However, our technique was more susceptible to false positives because it doesn't use strict conditions such as the previous technique. In our future work we will do the following.

First, we plan to add more patterns to be detected. Our technique deals with five patterns currently. However we predict it is possible to detect other patterns if metrics to identify others are decided. And it is necessary to collect more learning data to cover the diversity of pattern applications. Moreover, we plan to specialize the metrics to each role by returning P2 because the result might depend on data. These lead to the enhancement of recall and precision.

Second, we currently qualitatively and manually judge whether to return to P2 and to apply GQM again; hence, in the future, we should find an appropriate automatic judgment method.

Third, we plan to prove the validity of the expressions and the parameters of agreement values and thresholds, etc. We consider that it is possible to reduce the false positive by deciding best thresholds of role agreement values and pattern agreement values.

Finally, we plan to determine the learning number of times automatically and examine the correlation of the learning number of times and precision.

References

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

[2] M. Lorenz and J. Kidd Object-Oriented Software Metrics. Prentice Hall, 1994.

[3] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design Pattern Detection Using Similarity Scoring. IEEE Trans. Software Engineering, Vol.32, No.11, pp. 896-909 2006.

[4] 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, pp. 224–232, 2005.

[5] H. Kim and C. Boldyreff. A Method to Recover Design Patterns Using Software Product Metrics. In Proceedings of the 6th International Conference on Software Reuse: Advances in Software Reusability, pp. 318-335, 2000.

[6] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. Design Pattern Mining Enhanced by Machine Learning. 21st IEEE International Conference on Software Maintenance, pp. 295-304 2005.

[7] H. Washizaki, K. Fukaya, A. Kubo, and Y. Fukazawa. Detecting Design Patterns Using Source Code of Before Applying Design Patterns. 8th IEEE/ACIS International Conference on Computer and Information Science, pp. 933-938, 2009.

[8] N. Shi and R.A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 123-134, 2006.

[9] H. Lee, H. Youn, and E. Lee. Automatic Detection of Design Pattern for Reverse Engineering. 5th ACIS International Conference on Software Engineering Research, Management and Applications, pp. 577-583, 2007.

[10] L. Wendehals and A. Orso. Recognizing Behavioral Patterns at Runtime Using Finite Automata. In 4th ICSE 2006 Workshop on Dynamic Analysis, pp. 33–40, 2006.

[11] S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, and M. Saeki. Design Pattern Detection by Using Meta Patterns. IEICE Transactions, Vol. 91-D, No. 4, pp. 933–944, 2008.

[12] A. Lucia, V. Deufemia, C. Gravino and M. Risi. Design pattern recovery through visual language parsing and source code analysis. Journal of Systems and Software, Vol.82 (7), pp. 1177-1193, 2009.

[13] Y. Guéhéneuc and G. Antoniol. DeMIMA: A Multilayered Approach for Design Pattern Identification. IEEE Trans. Software Engineering. Vol.34, No. 5, pp. 667–684, 2008.

[14] J. Dietrich, C. Elgar. Towards a Web of Patterns. In Proceedings of First International Workshop Semantic Web Enabled Software Engineering, pp. 117-132, 2005.

[15] V. R. Basili and D.M. Weiss. A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering, Vol. 10, No. 6, pp. 728–738, 1984.

[16] T. Segaran. Programming Collective Intelligence. O'Reilly. 2007.

[17] H. Hirano. Neural network implemented with C++ and Java in Japanese. Personal Media. 2008.

[18] H. Yuki. An introduction to design pattern to study by Java. http://www.hyuki.com/dp/

[19] H. Tanaka. Hello World with Java! http://www.hellohiro.com/pattern/

[20] Sun Microsystems. Sun Developer Network. http://developers.sun.com/

[21] JUnit.org. Resources for Test Driven Development. http://www.junit.org/

[22] SpringSource.org. Spring Source. http://www.springsource.org/

Appendix

Figures 13 shows the results of applying GQM to the roles of all detection targets.

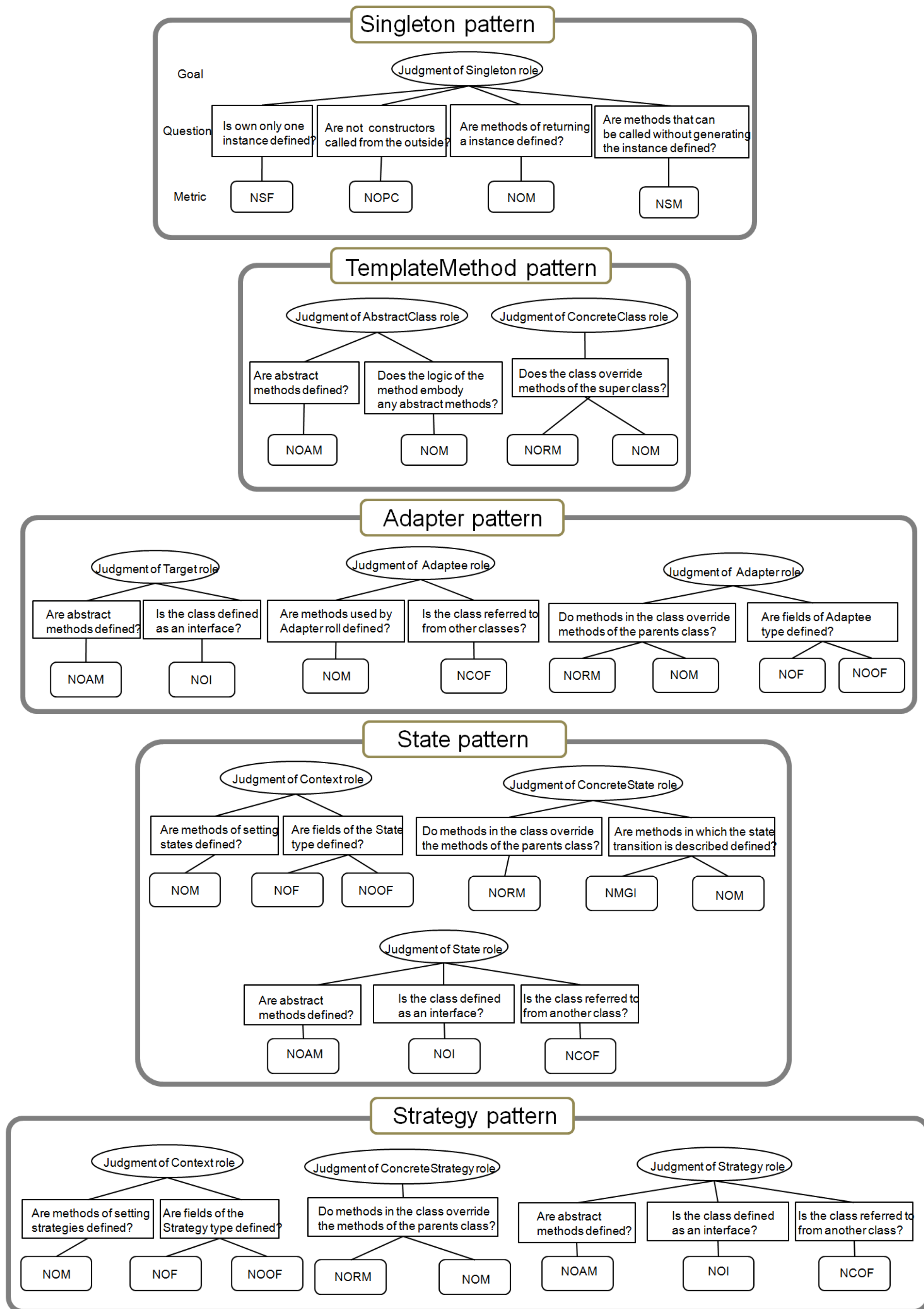

Figure 13. Results of applying GQM