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-known design patterns in programs. Many existing detection techniques are based on static analysis and use strict conditions composed of class structure data. 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 in 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 negatives and distinguishes patterns in which the class structures are similar. We conducted experiments that showed that our technique was more accurate than two 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. Gang of Four (GoF) patterns [1] are representative patterns for object-oriented software. Patterns are composed of classes that describe the roles and abilities of objects. For example, Figure 1 shows one GoF pattern named the `State pattern`. This 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 explicit class names, comments, or attached documents in existing programs. Thus, pattern detection improves the understandability of programs. However, manually detecting patterns in existing programs is inefficient, and patterns may be overlooked.

Many studies on using automatic pattern detection to solve the above problems have used static analysis. However, static analysis has difficulty identifying patterns in 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 if the applied patterns vary from the intended conditions even a little.

We propose a pattern detection technique that uses 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 machine learning based on measurement of metrics without using strict condition descriptions (class structural data, etc.). A metric is a quantitative measure of a software property that can be used to evaluate software development. For example, one such metric, number of methods (NOM), refers to the number of methods in a class [2]. Moreover, by using machine learning, we can in some cases obtain previously unknown identifying elements from combinations of metrics. To cover a diverse range of pattern applications, our method uses a variety of learning data because the results of our technique may depend on the kind and number of learning data used during the machine learning process. Finally, we conducted experiments comparing our technique with two previous techniques and found that our approach was the most accurate of the three.

II. PREVIOUS DESIGN PATTERN DETECTION TECHNIQUES AND THEIR PROBLEMS

Most of the existing detection techniques use static analysis [3][4]. These techniques chiefly analyze information such as 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 patterns.

There is a technique that detects patterns based on the degrees of similarity between graphs of pattern structure and graphs of programs to be detected [3]. However,

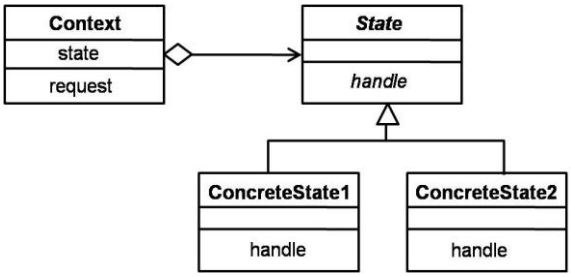

Figure 1. State pattern

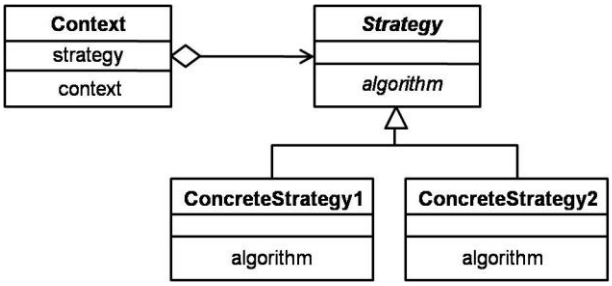

Figure 2. Strategy pattern

distinguishing the `State` pattern from the `Strategy` pattern is difficult because their class structures are similar (see Figure 1 and 2). Unlike this method, we distinguish the patterns to which the structure is similar by the identification of the roles from the quantity and the ratio of metrics by the machine learning. In addition, this technique [3] is available to the public as a web-based tool.

There is a technique that outputs pattern candidates based on features derived from 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 metrics and machine learning but also by analyzing class structure information. Moreover, this technique uses general metrics concerning an object-oriented system without using metrics for each role. Our technique uses metrics that specialize in each role.

Another existing technique improves precision by filtering detection results using machine learning. This technique inputs measurements of the classes and methods of each pattern [6]. However, it uses the existing static analytical approach, whereas our technique instead uses machine learning throughout the entire process.

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

Yet another approach detects patterns from the class structure and behavior of a system after classifying its patterns [8][9]. It is difficult to use, however, when patterns are applied more than once and when pattern application is diverse. In contrast, our technique copes well with both of these challenges.

Other detection techniques use dynamic analysis. These methods identify patterns by referring to the execution route information of programs [10][11]. However, it is difficult to analyze the entire execution route and use fragmentary class sets in an analysis. Additionally, the results of dynamic analysis depend on the representativeness of the execution sequences.

Some detection techniques use a multilayered (multiphase) approach [12][13]. Lucia et al. use a two-phase, static analysis approach [12]. This method has difficulty, however, in detecting creational and behavioral patterns because it analyzes pattern structures and source code level conditions. Guéhéneuc et al. use "DeMIMA," an approach that 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 in all categories and distinguish the `State` pattern from the `Strategy` pattern using metrics and machine learning.

Finally, one existing technique detects patterns using formal OWL (Web Ontology Language) definitions [14]. However, false negatives arise because this technique does not accommodate the diversity in pattern applications. The technique [14] is available to the public via the web as an Eclipse plug-in.

We suppress false negatives by using metrics and machine learning to accommodate diverse pattern applications and to distinguish patterns in which the class structures are similar. It should be noted that only techniques [3], [14] discussed above have been released as publicly accessible tools.

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 below, with pattern specialists and developers included as the parties concerned. Pattern specialists mean persons that have the knowledge about the patterns. Developers mean persons that maintain the object-oriented software. 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 input programs applied patterns into the metrics measurement system, and obtain measurements for each role. And specialists input these measurements into the machine learning simulator to learn. After machine learning they verify the judgment for each role, and if

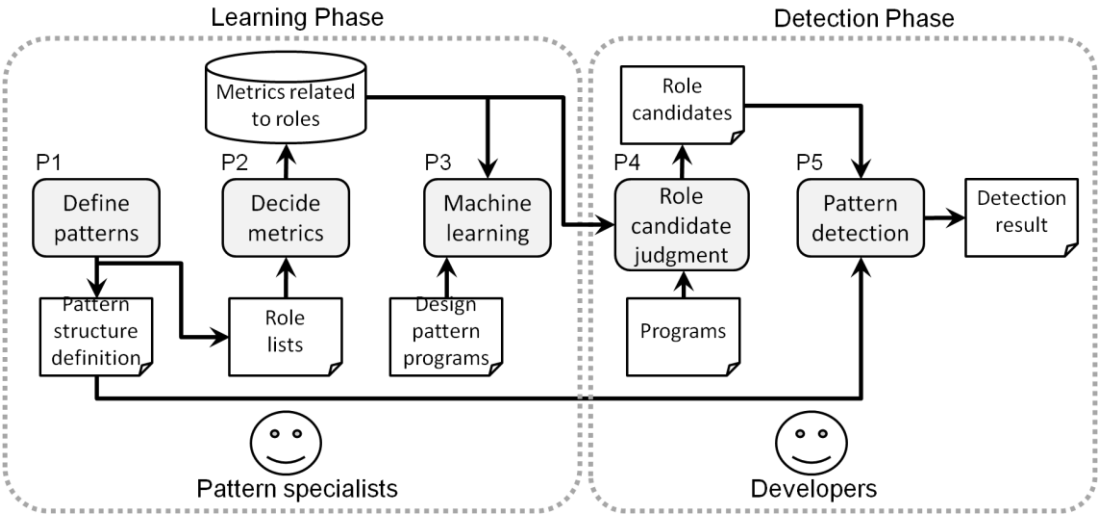

Figure 3. Process of our technique

the verification results are not good, they return to P2 and revise the metrics.

[Detection Phase]

P4. Role Candidate Judgment

Developers input programs to be detected into the metrics measurement system, and obtain measurements for each class. And developers input these measurements into the machine learning simulator. Machine learning simulator identifies role candidates.

P5. Pattern Detection

Developers input role candidates judged in P4 to the pattern detection system by using the pattern structure definitions defined in P1. This system detects patterns automatically. The structure definitions correspond to the letters P, R, and E of section III-B.

A. Learning Phase

P1. Define Patterns

Currently, our technique considers 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 all these groups.

P2. Decide Metrics

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

We experimented with judging roles by using general metrics without GQM. However, the machine learning results were unsatisfactory because the measurements of some metrics were irregular. Consequently, we chose GQM so that the machine learning could function appropriately by stable metrics in each role. With our technique, the pattern specialists set as a goal the accurate judgment of each role. To achieve this goal they defined a set of questions to be evaluated. Next, they decided on useful metrics to help answer the questions they had established. The pattern specialists decide metrics to identify roles by the quantity and the ratio of measurements. Therefore, they decide questions by paying attention to the attributes and operations of the roles by reading the description of the pattern definition. They decide simple metrics concerning the static aspect like structure first to improve the recall. However, the lack of questions might occur because GQM is qualitative. Therefore, if the machine learning results were unsatisfactory by irregular measurements of metrics, the procedure loops back to P2 to reconsider metrics also concerning behavior. Moreover, it will be possible to apply GQM to roles with new patterns in the future.

For example, Figure 4 illustrates 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 as shown in Figure 5. The `AbstractClass` role can be distinguished by the ratio of

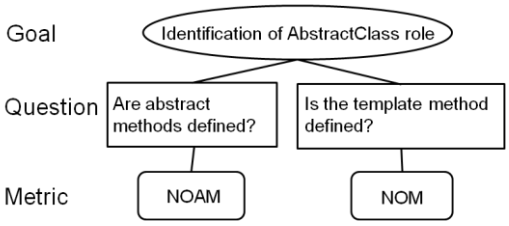

Figure 4. Example of GQM（AbstractClass role）

```
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)

TABLE I. RESULTS OF APRLING GQM

Pattern	Role	Goal	Question	Metric
Singleton	Singleton	Identification of Singleton role	Is the static field defined?	NSF
			Is the constructor called from other class?	NOPC
			Is the method that return singleton instance?	NSM
Template Method	AbstractClass	Identification of AbstractClass role	Are abstract methods defined?	NOAM
			Is the template method defined?	NOM
	ConcreteClass	Identification of ConcreteClass role	Is the override method defined?	NORM
Adaper	Target	Identification of Target role	Are abstract methods defined?	NOAM
			Is the class defined as an interface?	NOI
	Adapter	Identification of Adapter role	Are override methods defined?	NORM
			Is Adaptee field defined?	NOF
				NOOF
	Adaptee	Identification of Adaptee role	Are methods used by Adapter role defined?	NOM
			Is the class referred by other classes?	NCOF
State	Context	Identification of Context role	Are methods to set states defined?	NOM
			Is State field defined?	NOF
				NOOF
	State	Identification of State role	Are abstract methods defined?	NOAM
			Is the class defined as an interface?	NOI
			Is the class referred by other classes?	NCOF
	Concrete State	Identification of ConcreteState role	Is the override method defined?	NORM
			Is the method that describes change state defined?	NOM
				NMGI
Strategy	Context	Identification of Context role	Are methods to set states defined?	NOM
			Is Strategy field defined?	NOF
				NOOF
	Strategy	Identification of Strategy role	Are abstract methods defined?	NOAM
			Is the class defined as an interface?	NOI
			Is the class referred by other classes?	NCOF
	Concrete Strategy	Identification of ConcreteStrategy role	Is the override method defined?	NORM

the number of methods to the number of abstract methods because with this role the former exceeds the latter. Therefore, the number of abstract methods (NOAM) and number of methods (NOM) are useful metrics for judging this role. Table I shows the results of applying GQM to all roles. The details of metrics are described in Table II of paragraph IV-*A*.

The previous technique [5] uses GQM, too. In this technique, the goal is set as "Recover design patterns". And this technique uses general metrics concerning an object-oriented system without deciding metrics at each role. On the other hand, our technique uses metrics that specialize in each role.

P3. Machine Learning

Machine learning is a technique that analyzes sample data by computer and acquires useful rules with which to make forecasts about unknown data. We used the machine learning so as to be able to evaluate patterns with a variety of application forms. Machine learning suppresses false negatives and achieves extensive detection.

Our technique uses a neural network [16] algorithm. A 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, taking into consideration the dependency among the different metrics. Therefore, it can deal with cases in which one class plays 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. Values are given a weight when they move 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 in the input layer and the hidden layer are set to the number of decided metrics, and the number of units of the output layer is set to the number of roles being judged.

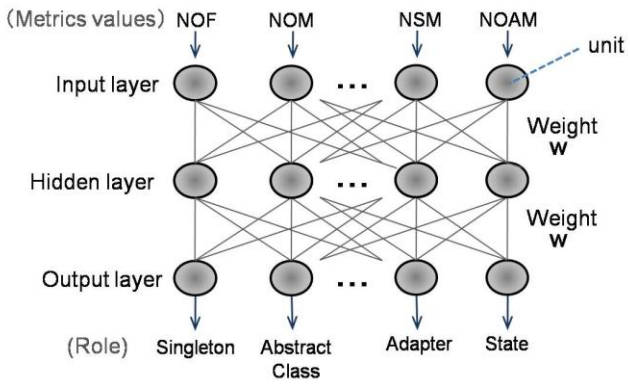

Figure 6. Neural network

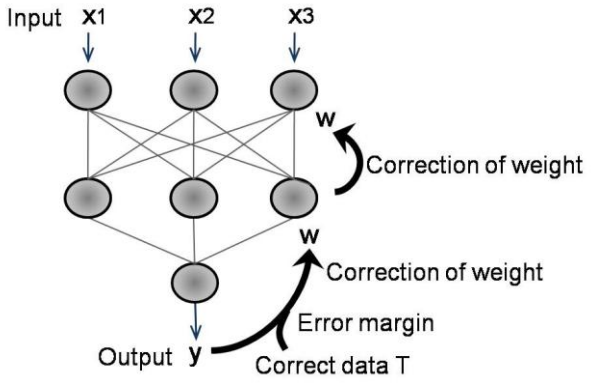

Figure 7. Back propagation

The input consists of the metric measurements of each role in a program to which patterns have already been applied, and the output is an expected role. Pattern specialists obtain measurements for each role by using metrics measurement system. And, specialists input these measurements into the machine learning simulator to learn. The learning repetitions cease when the error margin curve of the simulator converges. The specialists verify the convergence of the error margin curve manually at present. After machine learning they verify the judgment for each role, and if the verification results are not good, they return to P2 and revise the metrics.

B. Detection Phase

P4. Role Candidate Judgment

Developers input programs to be detected into the metrics measurement system, and obtain measurements for each class. And developers input these measurements to the machine learning simulator. This simulator outputs values between 0–1 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 (i.e., 0.0834) because we treat 12 roles at present. The sum of the output values is different at each input in the neural net work. Therefore, to use a common threshold for all the output, our technique normalizes the output value.

For example, Figure 8 shows the role candidate judgment results with NOM of 3 and NOAM of 2 and other metrics of 0; the output value of `AbstractClass` is the highest value. By regularizing the values of Figure 8, the roles are judged to be `AbstractClass` and `Target`.

P5. Pattern Detection

Developers input role candidates judged in P4 into the pattern detection system by using the pattern structure definitions defined in P1. And, this system detects patterns by matching the direction of the relations between role candidates and the roles of pattern in programs. The matching moves sequentially from the role candidate with the highest agreement value to that with the lowest value. The pattern detection system searches all combinations of role candidates that accord with the pattern structures. The pattern detection system detects patterns 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. The kind of relations 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 then the kind of relation does not agree, the pattern agreement value might become 0, and these classes will not be detected as patterns. In such cases, a problem similar to those of the previous detection techniques will occur because the difference in the kind of relation is not recognized.

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 comprising the role candidates is *R'*, and 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*. *Role* means the function which is input the element of *R* and the one of *R'* , and . *Rel* means the function which is input the element of *E* and the one of *E'*. 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.

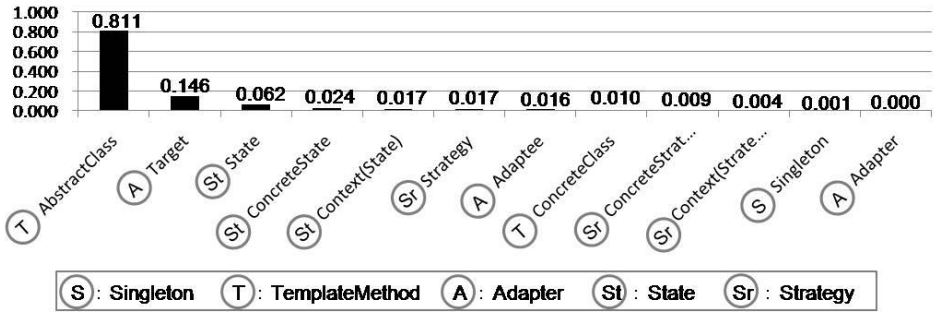

Figure 8. Example of machine learning output

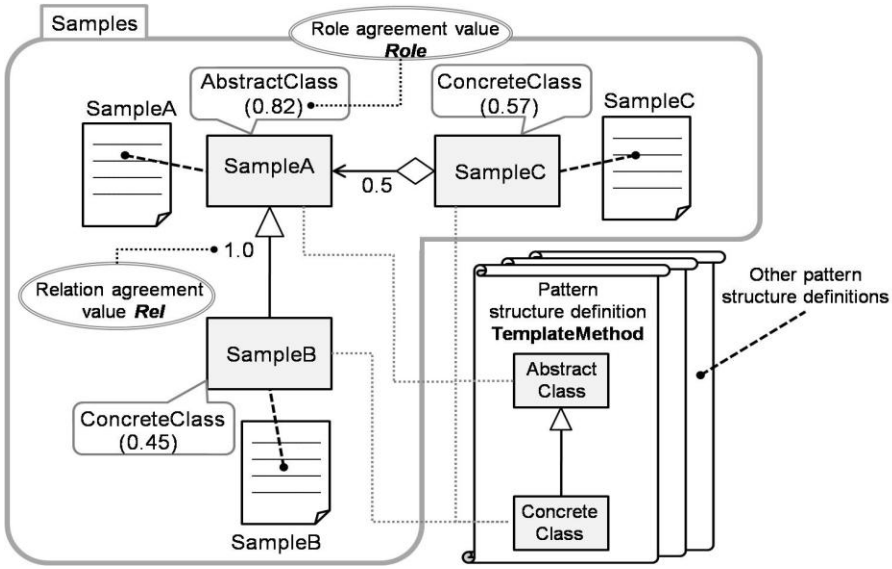

Figure 9. Example of pattern detection (TemplateMethod pattern)

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

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

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

$Role(r_m, r'_n) =$ The role agreement value $\quad r_m \in R, r'_n \in R'$

$Rel(e_p, e'_q) =$ The relation agreement value $\quad e_p \in E, e'_q \in E'$

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

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

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

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.

$P = TemplateMethod = (R, E)$

$R = \{AbstractClass, ConcreteClass\}$

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

$Samples = (R', E')$

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

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

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

$Role(AbstractClass, SampleA) = 0.82$ $\quad Role(ConcreteClass, SampleB) = 0.45$

$Role(ConcreteClass, SampleC) = 0.57$

$Rel(AbstractClass \lhd - ConcreteClass, SampleA \lhd - SampleB) = 1.0$

$Rel(AbstractClass \lhd - ConcreteClass, SampleA \leftarrow\!\diamond SampleC) = 0.5$

$$Com(AbstractClass \lhd - ConcreteClass, SampleA \lhd - SampleB) = \frac{0.82 + 0.45}{2} \times 1.0 = 0.635$$

$$Com(AbstractClass \lhd - ConcreteClass, SampleA \leftarrow\!\diamond SampleC) = \frac{0.82 + 0.57}{2} \times 0.5 = 0.348$$

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

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 role agreement values. Our technique uses the same threshold among of pattern agreement value as role agreement value because a lot of classes are detected as the pattern that composed of the only class like the singleton pattern. Classes with a pattern agreement value that exceeds the threshold are output as the detection result. The reciprocal of the number of roles for detection is taken to be

the threshold, 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 match the `TemplateMethod` pattern. In this case, the relation of "`SampleA` ◁– `SampleB`" is more similar to a `TemplateMethod` pattern than the relation of "`SampleA` ←◇ `SampleC`" because its agreement value of the former pair is 0.635 while that of the latter pair 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 approach and to confirm whether it could match its detected patterns with similar structures and diverse patterns.

A. 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 role 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 such as programs in the reference [18], etc., are called small scale, whereas programs in practical use are called large scale. We used the set of programs where patterns are applied in small-scale programs (60 in total)[1] [18][19] and large-scale programs (158 in total from the Java library 1.6.0_13 [20], JUnit 4.5 [21], and Spring Framework 2.5 RC2 [22]) as experimental data. We judged manually and qualitatively whether the patterns were appropriately applied in this set of programs.

Table II shows the metrics that were chosen for the small-scale and large-scale programs. We used different metrics depending on the magnitude of the programs. For instance, we chose the metric called number of methods generating instance (NMGI) for small-scale programs because the method for transit states in the `ConcreteState` role in the `State` pattern generates other `ConcreteState` roles in small-scale programs. We guessed that the difference appeared in ratios of metrics about State and Strategy, so we used the same metrics for the large-scale programs without NMGI. Because State pattern treats the states in State role and treats the actions of the states in the Context role. On the other hand Strategy pattern encapsulates the processing of each algorithm into a Strategy role, and Context processing becomes simpler compared with that of State pattern.

[1] All small-scale code:

http://www.washi.cs.waseda.ac.jp/ja/paper/uchiyama/dp.html

We focused our attention on recall because the purpose of our technique was detection covering diverse pattern applications. Recall indicates the degree to which detection results are free of leakage, whereas precision shows how free of disagreement these result are. The data in Table III was used to calculate 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 was calculated from the data in Table III by the following expressions.

Recall of role candidate judgment: $\mathrm{Re}_r = \dfrac{w_r}{w_r + x_r}$

Table IV 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, our technique regards the roles of the patterns other than `State` and `Strategy` patterns as role candidates when the role agreement value was above the threshold, whereas our technique regards the roles of `State` and `Strategy` patterns as role candidates when the role agreement value was above the threshold and the roles of both patterns were distinguished State pattern from Strategy pattern.

As shown in Table IV, the recalls for the large-scale programs were lower than those for the small-scale programs. Accurate judgment of large-scale programs was more difficult because these programs possessed attributes and operations that were unnecessary for pattern composition. Therefore, it will be necessary to collect a significant amount of learning data to adequately cover a variety of large-scale programs.

The results in Table IV pertain to instances where the `State` and `Strategy` patterns could be distinguished. The `Context` role had high recall, but `State` and `ConcreteState` roles had especially low recalls for large-scale programs. 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 initiating searching from the `Context` role in P5, and this improves recall.

TABLE II. 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

TABLE III. INTERSECTION PROCESSION

	detected	not detected
correct	w_r, w_p (true positive)	x_r, x_p (false negative)
incorrect	y_r, y_p (false positive)	z_r, z_p (true negative)

TABLE IV. RECALL OF CANDIDATE ROLE JUDGMENT (AVERAGE)

		Average recall (%)	
Pattern	Role	Small-scale programs	Large-scale programs
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

B. Pattern Detection Results

Our technique detects patterns using test data in both the small-scale and large-scale programs, and this result is evaluated. We used 40 sets of programs where patterns are applied in small-scale programs and 126 sets of programs where patterns are applied in large-scale programs as learning data. We judged manually and qualitatively whether the patterns were appropriately applied in the detection results. Table V shows precision and recall of the detected patterns. Precision and recall were calculated from the data in Table III by the following expressions:

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

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

Small-scale and large-scale programs shared a common point in that they both had recalls that were higher than precisions. However, there were many non-agreements about the `State` patterns and `Strategy` patterns in the large-scale programs. Recall was 90% or more with the small-scale programs, but it dropped as low as 60% with the large-scale programs.

The large-scale programs resulted in especially low recall for the `Adapter` pattern. Table IV shows the cause: The recall of the role candidate judgment for the `Adapter` pattern was not high enough. It is necessary to show that all roles that compose patterns have agreement values that 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 for the `Adapter` pattern. It will be necessary to return to P2 and choose new metrics. The `State` pattern was identified by searching from the `Context` role, for instance, in the `State` pattern detection in the large-scale programs, and the recall of the pattern detection was higher than the recall of role candidate judgment. Table V shows holistically that our technique suppresses false negatives because the recall is high.

TABLE V. PRECISION AND RECALL RATIO OF PATTERN DETECTION

	Number of test data		Precision (%)		Recall (%)	
Pattern	Small-scale programs	Large-scale programs	Small-scale programs	Large-scale programs	Small-scale programs	Large-scale programs
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

C. Experiment Comparing Previous Detection Techniques

We experimentally compared our technique with previous detection techniques [3][14]. These previous techniques have been publicly released, and they consider three or more patterns addressed by our own technique. Both target Java programs, as does our own. The technique proposed by Tsantails's technique [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`) with our own. TSAN detects patterns based on 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 OWL (Web Ontology Language) definitions. 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 and precision graphs for our technique and TSAN, and Figure 11 shows the corresponding graphs for 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 III by using the expressions of paragraph IV -*B*. In the results of TSAN and DIET, we alternately plotted results because these previous detection techniques output no value to rank. In the recall and precision graphs higher values are better.

Figure 10 and 11 show particularly good results for all techniques when small-scale programs was examined. This is because small-scale programs do not include unnecessary attributes and operations in the composition of patterns.

Table VI and VII show the average F measure for each plot of Figure 10 and 11. The F measure is calculated with

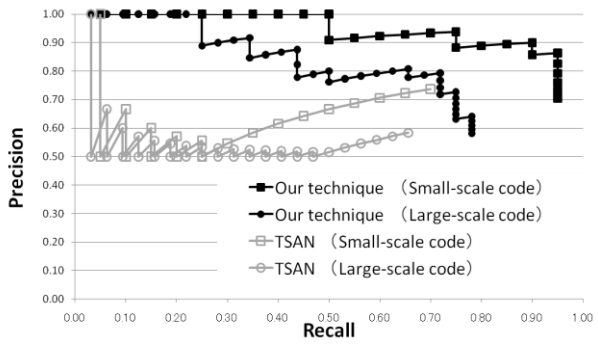

Figure 10. Recall-precision graph of detection results (vs. TSAN)

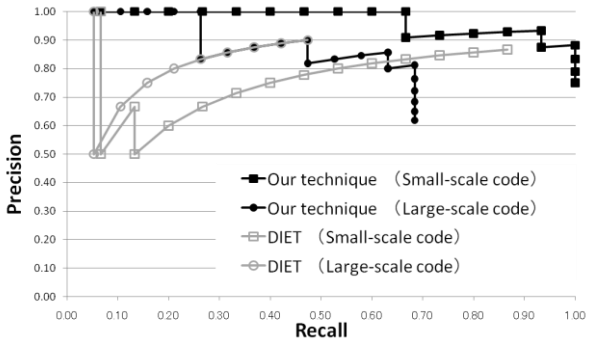

Figure 11. Recall–precision graph of detection results (vs. DIET)

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

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

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

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

recall and precision calculated by the above-mentioned expression as follows.

$$F-measure = \frac{2 \cdot \mathrm{Pr}_p \cdot \mathrm{Re}_p}{\mathrm{Pr}_p + \mathrm{Re}_p}$$

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

Distinction between State *pattern and* Strategy *pattern*

Our technique distinguished `State` pattern `Strategy` pattern. Table VIII 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 programs. `State` pattern treats the states in `State` 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 VIII 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.

Detection of Subspecies of Patterns

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

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).

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

TABLE VIII. 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

```
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 in pattern application (Singleton pattern)

V. Conclusion and Future Work

We devised a pattern detection technique using metrics and machine learning. Role candidates are judged using machine learning that relies on 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 superior to two previous detection techniques by experimentally distinguishing patterns in which the class structures are similar. Moreover, subspecies of patterns were detected, so we could deal with a very diverse set of pattern applications. However, our technique was more susceptible to false positives because it does not use strict conditions such as those used by the previous techniques.

We have several goals for our future work. First, we plan to add more patterns to be detected. Our technique can currently cope with five patterns. However, we predict it will be possible to detect other patterns if we can decide upon metrics to identify them. It is also necessary to collect more learning data to cover the diversity in pattern applications. Moreover, we plan to more narrowly adapt the metrics to each role by returning to step P2 because results might depend on the data. This process would lead to the enhancement of recall and precision.

Second, we currently qualitatively and manually judge whether to return to step 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. We consider that it is possible to reduce the false positive rate by deciding on the optimum thresholds for 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. 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] Oracle Technology Network for Java Developers. http://www.oracle.com/technetwork/java/index.html

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

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