Finding Patterns in Software with Data Analysis

By,
Alekhya Changelpet

Lewis University, ABQ

Abstract

Improving the understandability, maintainability, and reusability of object-oriented programs is essential, and one approach is to automatically detect common design patterns within them. However, existing detection methods often rely on rigid conditions based on static analysis of class structures, making it challenging to identify patterns with similar class structures and accommodate the diverse applications of design patterns. In this paper, we propose a novel design pattern detection technique leveraging metrics and machine learning. Our method evaluates candidate roles within design patterns using machine learning and metric measurements, analyzing the relationships between these candidates to detect patterns. By mitigating false negatives and distinguishing patterns with similar class structures, our technique outperforms existing methods, as demonstrated through experimental comparisons

INTRODUCTION

Design patterns, such as those outlined in the GoF patterns, offer standardized solutions to common design problems in object-oriented software, enhancing maintainability, reusability, and comprehensibility. They facilitate effective communication among developers by defining roles and capabilities of objects.

Understanding and applying patterns in third-party or open-source software can be time-consuming. Detecting patterns manually from existing programs is inefficient and prone to oversight. Many studies have attempted to automate pattern detection using static analysis, but this approach struggles with identifying patterns with similar class structures or those with minimal features. Moreover, strict conditions in previous methods may lead to overlooking patterns or misapplication. We propose a novel pattern detection technique employing software metrics and machine learning. Unlike previous methods relying on strict conditions, our approach leverages machine learning to identify pattern elements based on metric measurements, avoiding the need for detailed structural descriptions. Software metrics offer quantitative standards for evaluating software development from various angles, such as the number of methods in a class (NOM). Machine learning allows us to uncover previously unknown pattern elements by analyzing metric combinations. To ensure adaptability across diverse pattern applications, our method utilizes a range of learning data during the machine learning phase.
We conducted experiments comparing our technique with two existing methods, demonstrating its superior accuracy.

PREVIOUS DESIGN PATTERN DETECTION TECHNIQUES AND THEIR PROBLEMS

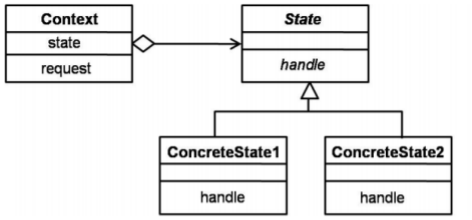

Figure 1. State pattern

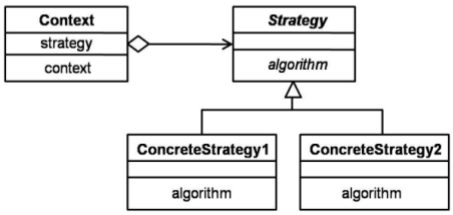

Figure 2. Strategy pattern

Many previous pattern detection techniques rely on static analysis, primarily analyzing class structures against specific conditions. However, deviations from these strict conditions or the assignment of multiple roles within a class can lead to oversight by developers.
One technique detects patterns by comparing similarity between pattern structure graphs and program graphs, but it struggles to distinguish between similar patterns like the State and Strategy patterns due to their comparable class structures. In contrast, our approach utilizes distinguishing elements identified through metrics and machine learning to discern such patterns.

Another technique identifies pattern candidates based on metric measurements but necessitates manual confirmation, relying heavily on developer expertise. Our method eliminates the need for manual filtering by incorporating metrics, machine learning, and class structure analysis.
Some techniques enhance precision by using machine learning to filter detection results based on measurements of classes and methods. However, our approach employs machine learning throughout the entire process, without relying on existing static analytical techniques.

Certain techniques analyze programs before and after pattern application, requiring access to program revision histories. In contrast, our technique analyzes patterns solely within current programs.

Other techniques classify patterns based on class structure and system behavior, but may struggle with patterns applied multiple times or with diverse applications. Our approach accommodates such scenarios effectively.

Dynamic analysis techniques rely on program execution route information, but may struggle with analyzing entire execution routes or using fragmented class sets. Moreover, results may vary based on the representativeness of execution sequences.

Multilayered approaches involve analyzing patterns across multiple layers, but may face challenges in distinguishing patterns with identical structures. Our technique, however, can detect patterns of all categories and attempt to differentiate between similar patterns like the State and Strategy patterns using metrics and machine learning.

There's also a technique utilizing formal definitions in OWL (Web Ontology Language) but may encounter false negatives due to a lack of accommodation for transformed patterns. Our method mitigates false negatives by accommodating diverse pattern applications and distinguishing patterns with similar class structures using metrics and machine learning. Finally, only certain previous techniques mentioned have been released as tools.

OUR TECHNIQUE

Our technique comprises a learning phase and a detection phase, each consisting of several processes detailed below. Pattern specialists and developers are involved in these processes, with the current implementation utilizing Java as the programming language.

1. **Learning Phase**

Pattern Definition:
Pattern specialists identify and define detectable patterns, specifying the structures and roles that constitute these patterns.

2. **Metric Selection:**

Pattern specialists determine relevant metrics to evaluate the roles defined in the previous step, employing techniques like the Goal Question Metric decision method.
Machine Learning Training: Pattern specialists collect measurements for each role in programs where patterns have been applied, utilizing the metrics selected earlier. These measurements are input into the machine learning system, with specialists verifying role judgments post-training. If verification results are unsatisfactory, adjustments to the selected metrics are made.

3. **Role Candidate Assessment:**

Developers gather measurements for each class in the programs targeted for pattern detection, using the metrics determined in the learning phase. These measurements are fed into the machine learning system, which assesses role candidates.

4. **Pattern Identification:**

Using the pattern structure definitions established in the learning phase and the role candidates determined in the previous step, developers proceed to detect patterns. The structure definitions correspond to the sections outlined in the methodology.

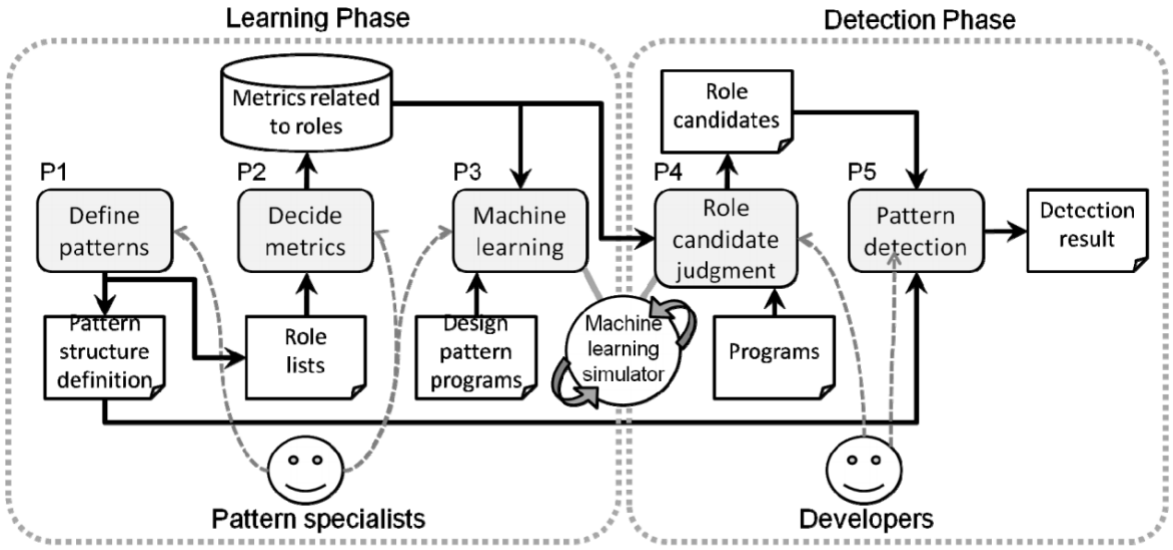

Figure 3. Entire image of our technique

5. P1. Defining Patterns

Currently, our technique addresses five GoF patterns (Singleton, TemplateMethod, Adapter, State, and Strategy) and identifies 12 roles. These GoF patterns are categorized into creational, structural, and behavioral patterns, covering a broad spectrum of pattern groups.

6. P2. Selecting Metrics

Pattern specialists utilize the Goal Question Metric decision technique (GQM) to determine relevant metrics for evaluating roles. GQM offers a structured approach to establishing relationships between goals and metrics. Initially, we experimented with using general metrics but found the results unsatisfactory due to metric instability. Therefore, we opted for GQM to ensure stability and distinguishability of metrics for role evaluation. In this approach, pattern specialists define role judgment as a goal, formulate a set of questions to assess goal achievement, and select metrics to address these questions. Questions are devised based on role attributes and operations, identified from pattern descriptions. For instance, in assessing the AbstractClass role in the TemplateMethod pattern, metrics such as the number of abstract methods (NOAM) and number of methods (NOM) are deemed useful for distinguishing the role based on the ratio of methods to abstract methods.

7. P3. Utilizing Machine Learning

Machine learning is employed to accommodate diverse pattern applications and minimize false negatives, facilitating incremental pattern detection. Our technique utilizes a neural network algorithm for its ability to handle varying input dependencies and multiple roles per class effectively. Although support vector machines could also be used for binary pattern distinction, we opted for neural networks due to their flexibility in handling complex input relationships.

A neural network comprises input, hidden, and output layers, with weights assigned to the connections between units in each layer. Weight adjustments, typically performed using back propagation, aim to minimize the error margin between the network's output and the correct answer. We employ a hierarchical neural network simulator, employing back propagation, with the number of layers set to three, input and hidden layer unit counts matching selected metrics, and output layer units corresponding to role judgments. The input consists of metric measurements from programs where patterns are already applied, while the output represents expected roles. Learning iterations continue until the error margin curve converges, with convergence manually determined for now.

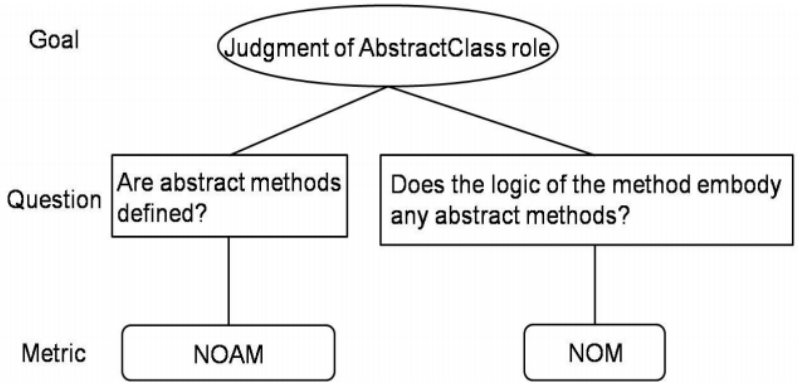

Figure 4. Example of GQM（AbstractClass role）

```
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）

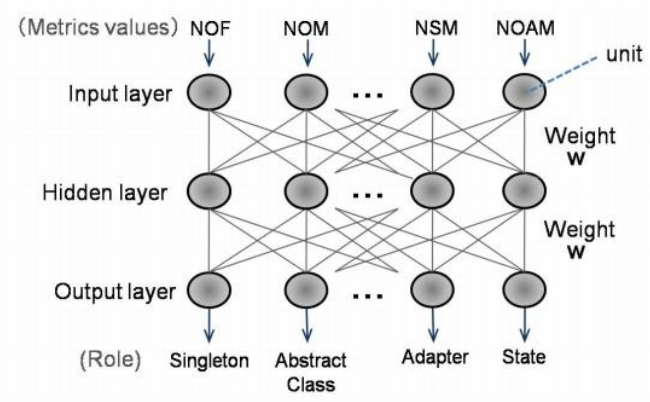

Figure 6. Neural network

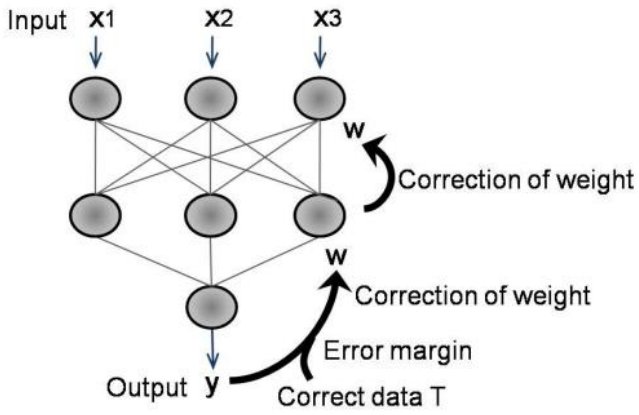

Figure 7. Back propagation

Detection Phase :

Role candidate judgment: Metric measurements are gathered for each class within the programs. These measurements serve as inputs to the machine learning simulator, which produces output values ranging between 0 and 1 for all roles to be assessed. These output values are normalized to ensure that their sum equals 1, and they are referred to as role agreement values. A higher role agreement value signifies a higher likelihood that the role candidate is correct. To determine role candidates, a threshold is established as the reciprocal of the number of roles to be detected, currently set at 1/12 = 0.0834, given that 12 roles are presently under consideration. For instance, let's consider a class with a Normalized Object Methods (NOM) of 3, Normalized Object Attributes (NOAM) of 2, and other metrics set to 0. As depicted in Figure 8, the role candidate judgment results based on these measurements show that the output value for the AbstractClass role is the highest. Therefore, after regularizing the values shown in Figure 8, the roles are determined to be AbstractClass and Target.

Pattern Detection:

Patterns are detected by examining relationships between role candidates within the pattern structure. The search progresses sequentially from role candidates with the highest agreement value to those with the lowest. All combinations of role candidates conforming to the pattern structures are explored. Detection occurs when the directions of relations between role candidates align with the pattern structure and when the role candidates match the roles at both ends of the relations. The agreement values of relations reflect the type of relation.

Currently, our method considers inheritance, interface implementation, and aggregation relations. As more patterns are incorporated in the future, additional types of relations will be accommodated. The relation agreement value is 1.0 when the relation type matches that of the pattern, and 0.5 when it does not. If the relation agreement value is 0 due to a mismatch in relation type, the pattern agreement value may become 0, resulting in the classes not being detected as part of the pattern. This situation could lead to issues similar to those encountered with previous detection techniques, where differences in relation types are not properly recognized.

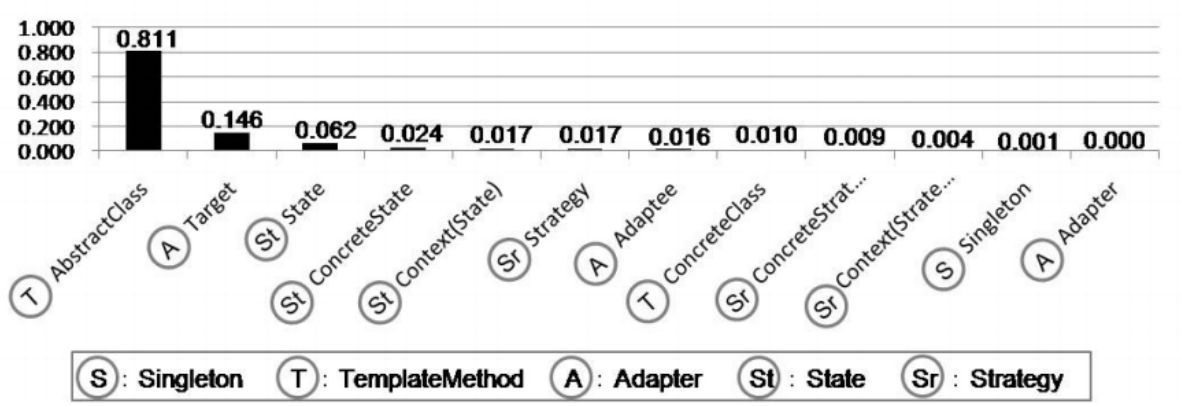

Figure 8. Example of machine learning output

The pattern agreement value (Pat) is determined based on the role agreement values (Role) and relation agreement values (Rel).

- The pattern to be detected is denoted as P, with the set of roles composing the pattern denoted as R, and the set of relations denoted as E.
- The program targeted for detection is defined as P', with the set of classes judged as role candidates denoted as R', and the set of relations between elements of R' denoted as E'.
- Role agreement value is represented as Role, while 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.
- Pat is calculated as the average of Com.
- When calculating Com, the average of two Roles is computed, and the average value of Com is adjusted to ensure both Pat and Role fall within the range of 0 to 1.
- If the directions of the relations do not align, Rel is assumed to be 0.

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

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

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

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

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

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

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

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

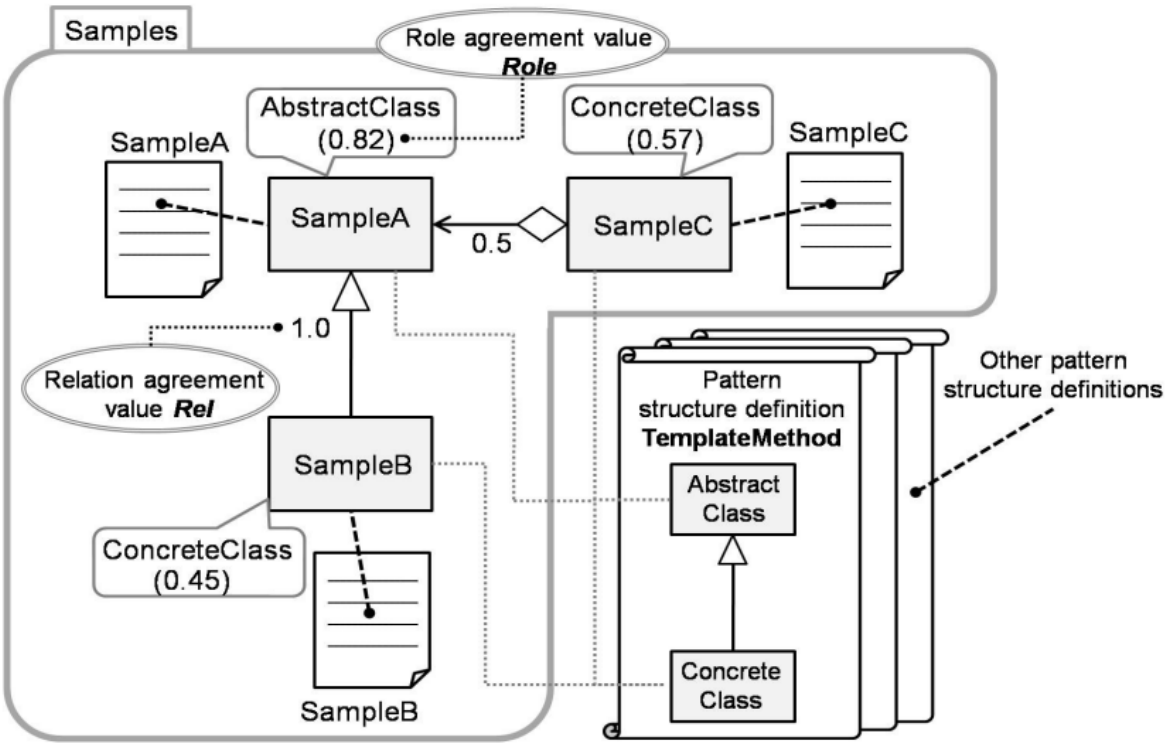

Figure 9. Example of pattern detection (TemplateMethod pattern)

Figure 9 illustrates an example of detecting the TemplateMethod pattern. In this scenario, it is presumed that class SampleA exhibits the highest role agreement value for an AbstractClass. The calculation of the pattern agreement value between the program Samples and the TemplateMethod pattern is conducted using 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 depicted in Figure 9, the pattern agreement value of the TemplateMethod pattern was calculated to be 0.492. Pattern agreement values, like role agreement values, are normalized within the range of 0 to 1.

For the detection process, classes with pattern agreement values exceeding a predefined threshold are output as the detection result. The threshold is determined as the reciprocal of the number of roles for detection, which in this case is 0.0834. Pattern agreement values higher than this threshold are considered sufficient for detection.

In Figure 9, SampleA, SampleB, and SampleC were detected as TemplateMethod patterns. Additionally, SampleA and SampleB, as well as SampleA and SampleC, can also be considered to represent the TemplateMethod pattern.

In this case, the relation "SampleA < - SampleB" is deemed to be more similar to a TemplateMethod pattern compared to the relation "SampleA < - SampleC". This determination is based on their respective agreement values, with the agreement value for "SampleA < - SampleB" being 0.635, while the agreement value for "SampleA < - SampleC" is only 0.348.

EVALUATION AND DISCUSSION

Verification of Role Candidate Judgment: To verify the judgment of role candidates, we employed cross-validation. This method involves dividing the data into n groups, with one group used for testing and the remaining n-1 groups for learning. We conducted the test five times, dividing the data into five groups.

In this study, we classified programs into two categories: small-scale and large-scale. Small-scale programs, such as those referenced in [18], were used for testing purposes, while large-scale programs, including those from the Java library 1.6.0_13, JUnit 4.5, and Spring Framework 2.5 RC2, served as experimental data.

Real-world code segments where patterns were applied were manually collected and incorporated into the experimental data.

Table I shows the metrics selected for both small-scale and large-scale codes. Distinct metrics were chosen based on code size. For small-scale codes, the metric "number of methods generating instance" (NMGI) was preferred. This choice stemmed from the observation that in small-scale codes, methods for transit states in the ConcreteState role within the State pattern often generate other ConcreteState roles. However, NMGI was not utilized for large-scale codes due to the presence of unnecessary attributes and operations within pattern compositions, with limited implementation specialization in the State pattern as mentioned earlier. NMGI was specifically employed when the program under scrutiny closely resembled small-scale pattern codes. Conversely, it was avoided for programs of large scale, especially those developed by third parties.

Emphasis was placed on recall in our approach, as it aims to detect diverse pattern applications comprehensively. Recall indicates the extent to which the detection result is devoid of omission, or "leakage". Precision, on the other hand, signifies the absence of disagreement in the detection result. Table II serves as a reference for calculating recall, with variables such as wr, xr, yr, and zr representing numbers of roles, and wp, xp, yp, and zp representing numbers of patterns. Recall is computed using expressions derived from the data presented in Table II.

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

Table III provides an overview of the average recall associated with each role. In the evaluation

process, accurate judgment of role candidates is essential, particularly due to the structural similarities between the State pattern and Strategy pattern. Consequently, roles other than those within the State and Strategy patterns are considered accurately judged when the role agreement value surpasses a specified threshold. However, for roles within the State and Strategy patterns, accurate judgment is only achieved when the role agreement value exceeds the threshold and when both patterns are distinguished.

The recall values for large-scale codes, as depicted in Table III, are notably lower compared to those for small-scale codes. This discrepancy can be attributed to the heightened difficulty in making accurate judgments for large-scale codes, primarily due to the presence of unnecessary attributes and operations within their pattern compositions. Consequently, a substantial amount of learning data collection is deemed necessary to adequately cover the diverse landscape of large-scale codes.

The results presented in Table III pertain to scenarios where the State pattern and Strategy pattern could be distinguished. Notably, the Context role demonstrated high recall values, whereas the State and ConcreteState roles exhibited particularly low recalls for large-scale codes. However, when the threshold was surpassed, candidates for the State role were identified with high recall. This suggests that initiating the search from the Context role in P5 facilitates the differentiation of the State pattern, thereby improving 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

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

In Pattern Detection Results, the patterns are detected using our technique, utilizing pattern application parts as test data across both small-scale and large-scale codes. The evaluation of this detection result is presented in Table IV, which showcases the precision and recall of the detected patterns. Precision and recall metrics are derived from the data provided in Table II, utilizing the following expressions for calculation.

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

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

Both small-scale and large-scale codes exhibited a common characteristic: their recalls exceeded their precisions. This alignment with the objective indicates a tendency towards gradual detection suppression, as intended. However, a notable issue arose concerning the State patterns and Strategy patterns within the large-scale codes, where numerous disagreements were observed. To mitigate this challenge effectively, the identification of an optimal threshold becomes imperative

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

In the context of recall, while small-scale codes consistently demonstrated recall rates of 90 percent or higher, a notable decrease was observed in the large-scale codes, dropping to as low as 60 percent. This decline in recall was particularly pronounced for the Adapter pattern within the large-scale codes. The underlying cause, as indicated in Table III, can be attributed to the insufficient recall of role candidate judgments for the Adapter pattern.

A critical requirement for pattern detection is ensuring that the agreement values of all roles composing patterns surpass the specified threshold. However, in many instances, neither of the roles within the Adapter pattern was identified as a role candidate. Addressing this issue necessitates revisiting P2 and selecting new metrics to improve the detection process.

In contrast, for the State pattern, a different approach proved effective. By initiating the search from the Context role, similar to the methodology employed for State pattern detection in large-scale codes, the recall of pattern detection surpassed that of role candidate judgment. This successful strategy highlights the importance of tailored approaches for different pattern detection scenarios.

Experiment Comparing Previous Detection Technique

In our experimental evaluation, we compared our technique with two previous pattern detection techniques referenced as TSAN [3] and DIET [14]. These techniques, publicly released and targeting Java programs like ours, handle three or more patterns similar to our approach. TSAN shares four patterns with our technique: Singleton, TemplateMethod, Adapter, and State/Strategy. However, it treats the State pattern and Strategy pattern as a single entity due to an inability to distinguish

between them. On the other hand, DIET, with three common patterns (Singleton, TemplateMethod, Adapter), detects patterns using formal definitions in OWL (Web Ontology Language) rather than graph similarity.

We conducted pattern detection and evaluation using both small-scale and large-scale test data, ensuring differentiation between test and learning data sets. Figures 10 and 11 illustrate the recall-precision graphs comparing our technique with TSAN and DIET, respectively. In these graphs, higher plots indicate better performance. Particularly noteworthy are the results when using small-scale codes across all techniques, as reflected in the figures. This success can be attributed to the absence of unnecessary attributes and operations in the composition of patterns within small-scale codes.

Importantly, our technique successfully distinguishes between the State pattern and Strategy pattern, unlike TSAN. Table V provides a glimpse into the metrics measurements for the Context role in the State pattern and Strategy pattern, which were distinguished through experiments conducted on large-scale codes. Specifically, the State pattern manages states within the State pattern, while the Strategy pattern handles strategy encapsulation.

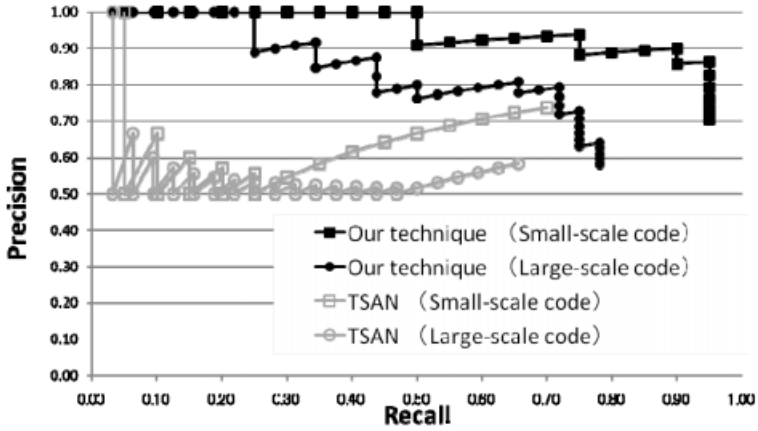

Figure 10. Recall-precision graph of the detection result (vs. TSAN)

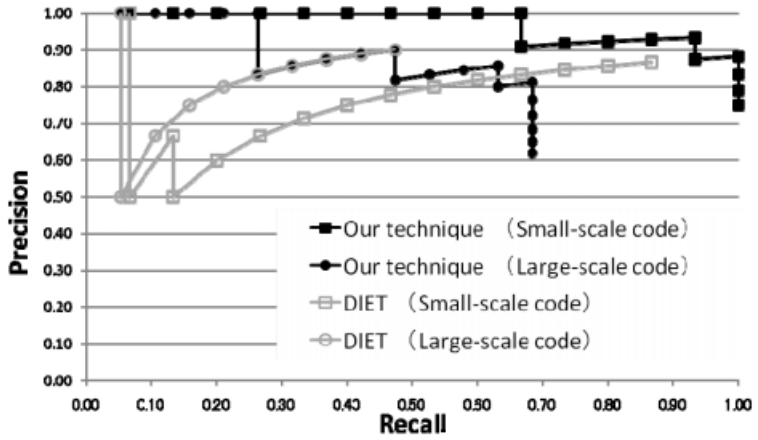

Figure 11. Recall-precision graph of the detection result (vs. DIET)

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

The Context role within the State pattern manages the states, while the Strategy pattern encapsulates each algorithm's processing into a Strategy role. As a result, the processing complexity of the Context role is simplified in comparison to that of the State pattern. Table V provides insights into the characteristics of the Context role, revealing that the State pattern's Context role comprises 45 fields and 204 methods (with 18 fields and 31 methods in the Strategy pattern's Context role). This disparity in the number of fields and methods serves as a distinguishing factor between the two patterns.

Figure 10 underscores the effectiveness of our technique, particularly in distinguishing between the State pattern and Strategy pattern, a feat not achieved by TSAN. Conversely, Figure 11 illustrates DIET's lower recall in large-scale code scenarios, attributed to its inability to accommodate the diverse applications of patterns. Large-scale codes not only feature numerous attributes and operations within pattern compositions but also encompass various subspecies of patterns.

Given these observations, our technique outperforms previous approaches, as evidenced by the superior positioning of its curve in Figures 10 and 11. Furthermore, Table VI and VII present the average F-measure for each plot depicted in the respective figures. The F-measure, calculated using the recall and precision derived from the aforementioned expressions, offers additional insight into the performance of each technique.

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

A substantial F-measure indicates higher accuracy, and the tables provided demonstrate that our technique consistently achieved a larger F-measure compared to the previous techniques. Notably, our technique exhibited the capability to detect subspecies of patterns, a capability not shared by TSAN or DIET. For instance, our technique successfully identified a Singleton pattern within source code utilizing a boolean variable, a detection missed by the previous techniques. However, it's worth noting that our technique is susceptible to false positives due to its gradual detection approach using metrics and machine learning, as opposed to stringent conditions.

False positives, particularly evident in patterns composed of only one role like the Singleton pattern, necessitate the utilization of metrics specialized for such scenarios to enhance judgment accuracy (P4). Despite this limitation, the overall evaluation affirms the superiority of our technique over previous methods, as evidenced by the graphical representations and the higher F-measure achieved.

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)

CONCLUSION

We developed a pattern detection technique utilizing metrics and machine learning. Our approach involves judging role candidates using machine learning trained on measured metrics, with patterns identified through class relations. We aimed to address challenges associated with overlooking patterns and distinguishing patterns with similar class structures.

Through experimental validation, we demonstrated the superiority of our technique over previous detection methods, particularly in distinguishing patterns with similar class structures. Furthermore, our technique successfully detected subspecies of patterns, enabling effective handling of the diverse applications of patterns. However, our method exhibited higher susceptibility to false positives compared to previous techniques due to its reliance on less stringent conditions.

In our future endeavors, we have outlined several key areas for improvement. Firstly, we intend to expand the scope of patterns detectable by our technique beyond the current five patterns. This expansion necessitates the identification of appropriate metrics for detecting additional patterns and the collection of more extensive learning data to cover the diverse landscape of pattern applications. Additionally, we plan to refine the metrics to specialize them for each role, potentially enhancing both recall and precision.

Secondly, our current method of qualitative and manual judgment for deciding whether to return to P2 and reapply the GQM process needs automation. Therefore, we aim to develop an appropriate automatic judgment method to streamline this process.

Thirdly, we seek to validate the expressions and parameters governing agreement values and thresholds to optimize pattern detection accuracy. Determining the best thresholds for role and pattern agreement values could help mitigate false positives and enhance the overall effectiveness of our technique.

Finally, we plan to automate the determination of the optimal number of learning iterations and investigate the correlation between the learning frequency and precision. This approach will enable us to fine-tune our technique and improve its performance over time.

APPENDIX

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

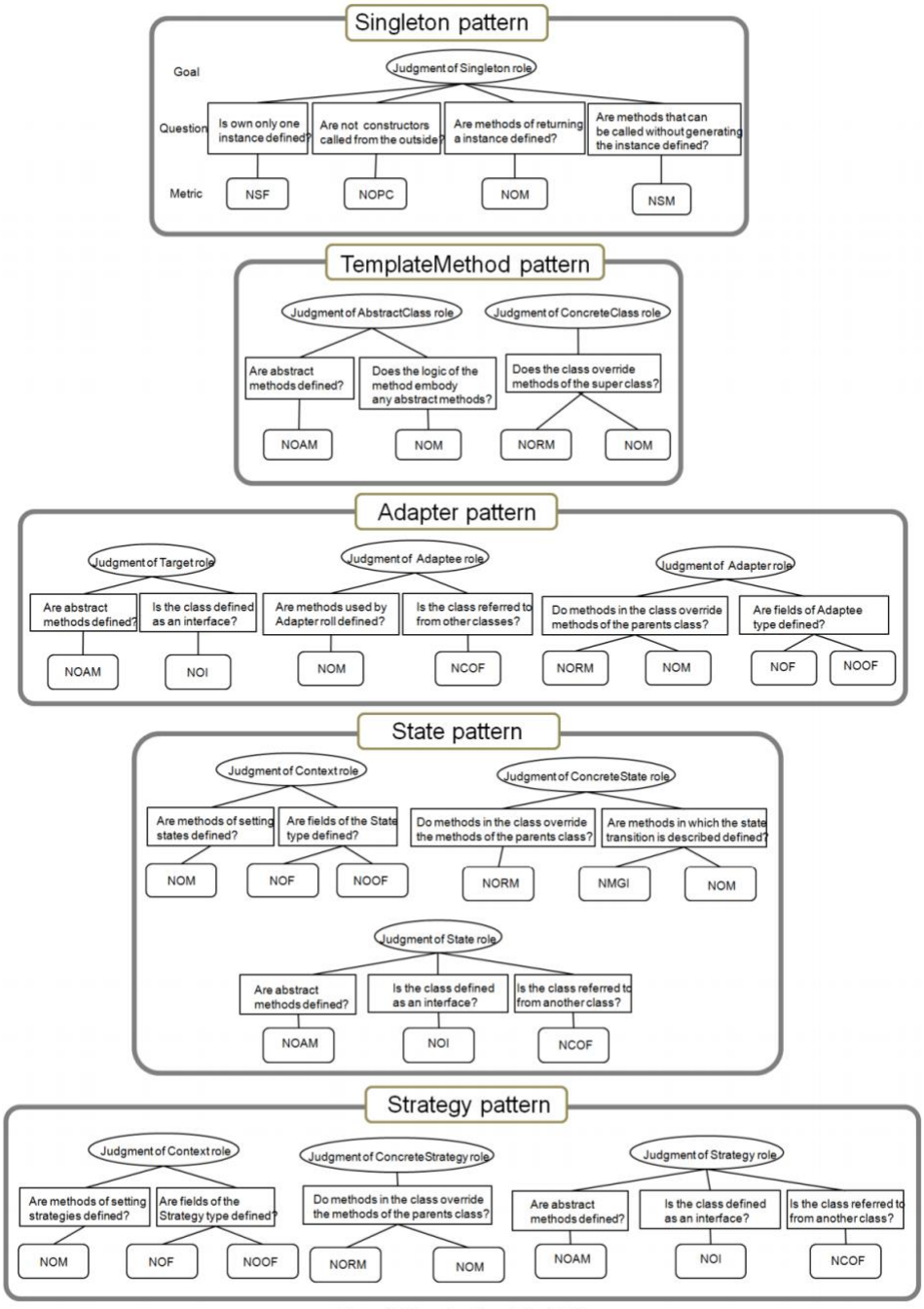

Figure 13. Results of applying GQM

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. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. H. Yuki. An introduction to design pattern to study by Java. http://www.hyuki.com/dp/