Towards Lightweight Detection of Design Patterns in Source Code

Jeffy Jahfar Poozhithara Hazeline U. Asuncion Brent Lagesse
University of Washington Bothell, WA, USA
E-mail: jeffyj@uw.edu, hazeline@uw.edu, lagesse@uw.edu

Abstract

Identifying which design patterns exist in source code helps maintenance engineers better understand source code and determine if new requirements can be satisfied. Automated techniques for finding design patterns generally require much time to label training datasets or to specify rules/queries for each pattern, and is difficult to extend support to secure design patterns (SDPs) and combination patterns. To address these challenges, we introduce PatternScout, a technique for automatic generation of SPARQL queries from UML Class diagrams and Sequence diagrams. These queries are used to detect patterns in the source code. Our results indicate that PatternScout can detect object-oriented design patterns (OODP) with accuracy that is comparable or better than existing techniques. It can also generate queries for SDPs that can be represented as UML Class diagrams.

1 Introduction

Since design patterns assist with satisfying security requirements, it is important for maintenance engineers to determine which patterns already exist in the code, including SDPs. Finding design patterns can be time-consuming, due to manual work required to reverse engineer the code [1]. Automated techniques also have challenges. Mining techniques involve the time-consuming task of manual labeling the training dataset and manual checking results due to false positives [2]. On the other hand, detection using rules [3] and queries [4] also involve time-consuming specification of design patterns and suffer from false negative results, as they generally lack the flexibility to handle variations. This challenge is especially pronounced in SDPs, which have a higher level of variability than OODPs [5].

Meanwhile, Semantic Web technologies provide a means of encapsulating rich information, using a graph known as Resource Description Framework (RDF). When source code is represented as an RDF graph, we can see design- and code-level concepts (e.g., class relationships, class properties), which are not captured in other graph representations of source code (e.g., abstract syntax trees). There are several query technologies to extract information from RDF; among these SPARQL is the most popularly used [6]. Running queries on an RDF yields highly accurate results, as these only retrieve results with matching graph pattern. However, as we mention above, this technique also suffers from false negatives, as the results obtained are very specific. Furthermore, using Semantic Web query languages, such as SPARQL, is difficult because the user needs to learn not only the query language syntax, but also the vocabulary and relationships in the data [6]. This is why researchers developed automated techniques for generating SPARQL queries [7]. However, none of these techniques cater to software.

To address these challenges, we developed PatternScout which takes advantage of Semantic Web technologies while overcoming its difficulties. First, the difficulty of using SPARQL queries is handled by automatically generating queries from UML Class and Sequence diagrams. This approach works as repositories of common design patterns exist [8, 9] and they already include UML Class diagrams in their descriptions. This also applies to SDPs as many of them also include Class diagrams [10]. Second, we overcome the limitation of low recall by creating a catalogue of known design pattern variations [11] and checking for these variations in the source code.

Our main contribution is a novel technique to generate SPARQL queries that can correctly identify design patterns. Compared to other methods, our approach is more lightweight because it does not involve any manual training and does not require manually defining rules or queries. In addition to the 23 GoF patterns supported by state-of-the-art design pattern detection techniques, PatternScout can generate SPARQL queries for secure design patterns, object-oriented design pattern variants, as well as any ad-hoc pattern (e.g., combination patterns) given their UML Class diagram. Our second contribution are insights to improving design pattern detection accuracy, such as incorporating behavioral aspects of a pattern in addition to structural characteristics by incorporating stereotypes, filters, and Sequence diagrams when necessary. Our final contribution is a repository of SPARQL queries that contains object-oriented design patterns and their variants [11].

We assessed PatternScout using experiments to measure accuracy. Our results indicate that they are comparable, or outperform existing techniques (e.g., [4, 12]).

2 Detecting Design Patterns

In this section, we discuss key concepts for automatically generating queries from UML Diagrams.

DOI reference number: 10.18293/SEKE2022-167

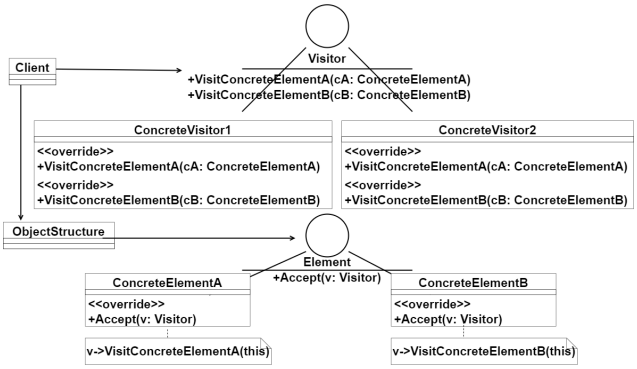

Figure 1: UML Representation of the Visitor Pattern

2.1 Identify Pattern Characteristics

Design pattern characteristics can be extracted from both UML Class and Sequence Diagrams. A Class diagram shows the objects within a design pattern and static relationships between those objects, while Sequence diagrams shows interactions between objects. We discuss how we use these diagrams in detecting patterns.

In a UML Class Diagram, PatternScout extracts relationships between classes, between classes and methods, between classes and attributes, and between methods and parameters, such as **Contain Relationships** (hasMethod, hasType, hasReturnType, hasModifiers, hasField, hasParameter, hasConstructor) and **Class Relationships** (Association, Generalization, Aggregation, Composition, Interface Realization, Dependency). PatternScout also extracts the following: **OO Entities** (Classes, Methods, Constructors, Fields, Method Parameters, Interfaces), **Visibility/Property** (Public, Private, Protected, Static, Final, Synchronized, Abstract), **Stereotypes**(Constructor, Override), and **Interactions**(Method Invocations).

We generate a SPARQL query by including relevant entities (i.e., "OO Entities") of the design pattern in the SELECT clause. We then add characteristics in the WHERE clause. For example, a project that contains a visitor pattern may contain the following RDF triples in its RDF graph:

```
PREFIX woc: <http://rdf.webofcode.org/woc/>
woc:SoldierVisitor woc:implements woc:UnitVisitor .
woc:SoldierVisitor woc:hasMethod woc:SoldierVisitor-vistSoldier() .
woc:SoldierVisitor-visitComander woc:hasParameter
woc:visitComander(com.iluwater.visitor.Commander)-parameter-0 .
woc:SoldierVisitor-visitCommander(com.iluwater.visitor.Commander)
-parameter-0 woc:hasType woc:Commander  .
```

These characteristics are captured in a UML Class diagram of a Visitor pattern as shown in Figure 1. To generate a SPARQL query, we extract all the entities in the Class diagram, such as class names and method names and add them to the SELECT clause, as shown below.

```
SELECT ?Visitor27 ?VisitConcreteElementA11 ?VisitConcreteElementB13
?VisitConcreteElementA27 ?VisitConcreteElementB29 ?Accept13
?Accept16 ?Accept20 ?VisitConcreteElementA24
?VisitConcreteElementB26 ?ConcreteVisitor15 ?Element14
?ConcreteVisitor211 ?ConcreteElementA18 ?ConcreteElementB22
```

We also add triples defining the type (role) of each entity to the WHERE clause. For example, if a Method is encountered, woc:Method type is added for that component in the WHERE clause:

```
SELECT ?ClassA ?OperationA
WHERE {
?ClassA  a woc:Class .
?OperationA  a woc:Method .
```

Next, we add the structure of the pattern, such as the aforementioned relationships between entities to the WHERE clause. A relationship is represented by an RDF triple in the format *(fromItem, relationshipType, toItem)* representing a relation from *fromItem* to *toItem*. Both *fromItem* and *toItem* are OO Entities. In the following snippet, ClassA is the *fromItem*, OperationA is the *toItem* and woc:hasMethod is the *relationshipType*.

```
?ClassA woc:hasMethod ?OperationA .
```

If characteristics related to visibility, property, data types, and return types are included in a Class diagram, PatternScout also generates the corresponding triples. Each line is a condition. All conditions in a WHERE clause must be satisfied for a design pattern match to occur. The greater the conditions, the more specific and restrictive the queries become. On the other hand, fewer conditions provide more allowance for variation in implementation. A partial list of characteristics for the above Class Diagram that would be included in a WHERE clause is below:

```
?Visitor27  a woc:Interface .
?ConcreteVisitor211  a woc:Class .
?ConcreteVisitor211 woc:implements ?Visitor27 .
?ConcreteVisitor211 woc:hasMethod ?VisitConcreteElementA11 .
?VisitConcreteElementA11 woc:hasParameter ?cA10 .
?cA10 woc:hasType ?ConcreteElementA18 .
```

Some design patterns such as the Visitor pattern require behavioral information to accurately identify it. Behavioral specifications related to method invocation can be obtained from Sequence diagrams. Here is an example snippet from [13] that shows an RDF triple with a behavioral characteristic of Visitor design pattern pattern.

```
<http://rdf.webofcode.org/woc/com.iluwatar.visitor.Sergeant-
accept(com.iluwatar.visitor.UnitVisitor)>
<http://rdf.webofcode.org/woc/references>
<http://rdf.webofcode.org/woc/com.iluwatar.visitor.UnitVisitor-
visitSergeant(com.iluwatar.visitor.Sergeant)> .
```

A SPARQL query based only on the Class diagram of a Visitor pattern will include the following triple representing an association relationship:

```
?ConcreteElementA18    woc:references    ?Visitor27    .
```

This structural relationship will result in false positive results as it does not describe the defining characteristics of a Visitor pattern. Moreover, the RDF graph representation

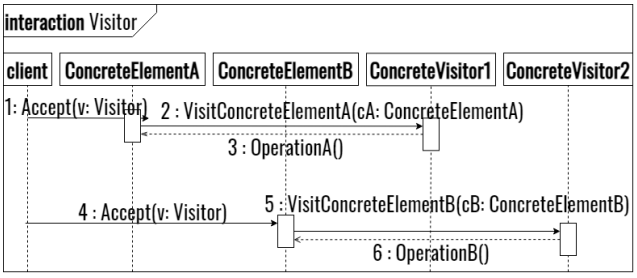

Figure 2: Sequence Diagram for Visitor Design Pattern

of the code might not include the relevant triple with references relationship if the two classes are under the same package, causing false negatives. However, by including information from the Sequence diagram shown in Figure 2, the WHERE statement would include the invocation of VisitConcreteElement method in the Accept method. This not only reduces false positives with a defining characteristic of the pattern, but is also immune to false negatives as the triple will be part of the RDF graph irrespective of the code base structure. The RDF triple is as follows:

```
?Accept16    woc:references  ?VisitConcreteElementA24  .
```

By including Sequence diagrams and consequently method invocation characteristics, PatternScout can distinguish between otherwise structurally identical design patterns (e.g., State - Strategy, Adapter - Command).

2.2 Use Stereotypes

Another way to improve accuracy of design pattern identification is to use stereotypes. Although not part of the standard UML specification, stereotypes have been used to differentiate or represent features like Constructors, Getters, Setters and Overriding of methods. Using stereotypes, Constructors can be differentiated from other Methods using *woc:Constructor* instead of the *woc:Method* type and *woc:hasConstructor* instead of the *woc:hasMethod* relationship. Similarly, methods of a child class that overrides methods of a parent class are differentiated with *woc:hasAnnotation overrides* relationship. We observed this to significantly reduce false positives, especially in detecting patterns like Proxy, Builder and Singleton.

2.3 Accommodate Variations with Query Map

Design patterns not only have many implementation variations, but some variations are combinations of existing patterns (e.g, Visitor Combinator patterns is a variant of the Visitor pattern where the GoF specification of Visitor is combined with Composite pattern for object oriented tree traversal). PatternScout can handle these variations as long as they can be represented as a Class diagram.

We use a query map to efficiently connect variants with each pattern. Instead of running one SPARQL query at a time (as shown in Figure 3), we run multiple queries at once. Each design pattern has its own section. Below each design pattern is a list of variants, with the variant name and filename as shown below:

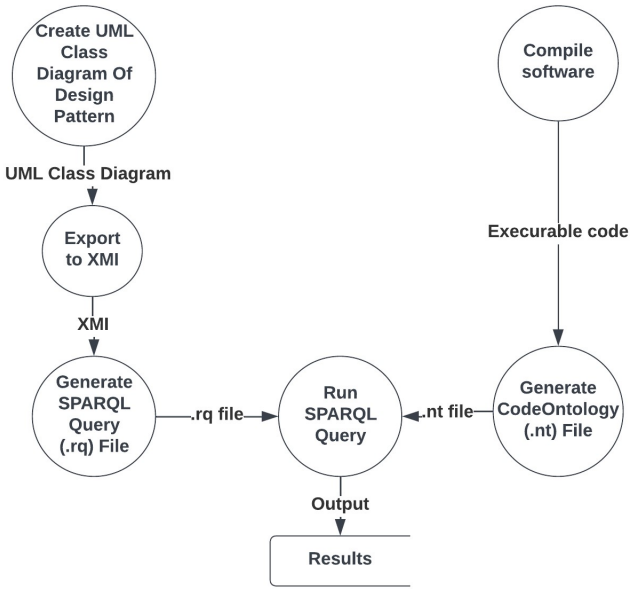

Figure 3: Approach Overview

```
"Visitor": {
    "Visitor GoF": "visitor.rq",
    "Visitor Combinators": "visitor_combinators.rq"
},
"Singleton" : {
    "Singleton GoF" : "singleton.rq",
    "Eager Instantiation": "singleton_Eager_Instantiation.rq",
```

3 Validation

We conducted two analyses to evaluate if the SPARQL queries generated by PatternScout are accurate and sufficient for detecting design patterns. The approach is summarized in Figure 3. The pre-processing required is as follows: generating an RDF graph for each project, creating a Class diagram in an XMI format and generating SPARQL queries from the Class diagram. Creating Class diagrams and generating SPARQL queries is a one time task for each pattern variant and can be reused for any project. The repository of SPARQL queries for known variants of object oriented design patterns are packaged with PatternScout. The only project-specific pre-processing needed is generating an RDF graph, using CodeOntology [14]. The preparation time taken for each project is shown in Table 1.

3.1 Detecting Presence/Absence of Patterns

Experiment: We ran an experiment to determine if PatternScout can correctly detect the presence/absence of patterns. Since these pattern instances in the selected projects are widely studied in literature (e.g., [15, 16, 12]), we used their results as our gold standard for this analysis. If a pat-

Open Source Project	LOC	Java Classes	RDF triples	Preparation Time (ms)
JHotDraw v5.1 (JHD)	8907	155	52824	2040
JRefactory v2.6.24 (JRF)	56187	569	70178	3023
JUnit v3.7 (JUN)	1347	33	9497	2001
QuickUML 2001 (QUM)	9250	156	59480	1756
MapperXML 1.9.7 (MPX)	14928	217	17147	6002
Dom4J v1.6.1 (DOM)	26350	328	29874	2059
Lizzy v1.1.1 (LZ)	12915	197	11617	1083

Table 1: Projects used for evaluation

	JHD	JRF	QUM	JUN	DOM	MPX	LZ	P	R
FM*	✓ | ✓	✓ | ✓	× | ✓	× | ✓	✓ | ✓	✓ | ×	✓ | ✓	67.67	80
PRTT	✓ | ✓	× | ×	✓ | ×	× | ×	✓ | ×	× | ×	× | ×	100	33.33
SGLT	✓ | ✓	✓ | ✓	✓ | ✓	× | ×	✓ | ✓	✓ | ✓	✓ | ✓	100	100
TPLT	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ✓	100	100
STT	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ×	× | ✓	✓ | ✓	83.33	83.33
CMD	✓ | ✓	× | ×	✓ | ×	× | ×	× | ×	× | ×	× | ×	100	50
OBSV	✓ | ✓	✓ | ✓	✓ | ✓	✓ | ×	✓ | ×	✓ | ✓	× | ×	100	66.67

Table 2: P&R based on presence(✓) or absence(×) of Patterns. Tuple represent (Actual Label | Predicted Label)

*FM: Factory Method, PRTT: Prototype, SGLT: Singleton, TPLT: Template Method, STT: State, CMD: Command, OBSV: Observer

tern is reported in a project (regardless of variation, as previous studies did not specify the variant used), we used a check mark (or True) for the actual label. If it is reported as absent, we used an X-mark (or False) (See Table 2).

Result: We summarize the precision and recall in detecting each pattern in the P and R columns of Table 2. Based on these 7 projects, we get an average precision of 92.86% and average recall of 73.33%. During manual inspection, we observed that the low recall was often due to SPARQL not interpreting the transitive nature of the inheritance relationship when parsing triples in the RDF graph.

3.2 Comparing PatternScout with Existing Tools

Experiment: We also ran experiments to compare the accuracy of PatternScout with existing tools (Table 3). In order to calculate precision and recall of each tool, we identified the true instances of each pattern in the projects used for benchmarking. We established the ground truth through manual inspection and validated with other results [15, 17]. We compared unique instances of patterns retrieved by PatternScout with Finder [18] and DPD [12]. While SparT [17] was also evaluated, we excluded it from the analysis as the off-the-shelf implementation did not include specifications for creational and structural patterns. We excluded Tools that are unavailable for download (e.g., [19, 20]) or those for which a core dependency is deprecated (e.g., [21, 15]) from the comparison.

We executed the benchmarked tools, DPD and SparT, on a Windows 10 21H2 Virtual Machine. FINDER was executed on a Rocky Linux 8.5 Virtual Machine. Java arguments and runtime parameters for execution were as recommended by the tools. SparT did not utilize Java, but ran natively on Windows.

	PatternScout			DPD			FINDER		
	P	R	f1	P	R	f1	P	R	f1
FM	68.75	100	77.27	25	50	66.67	20	50	57.14
PRTT	100	33.33	50	100	100	100	100	33.33	50
CMD	88.89	57.14	69.57	100	57.14	72.73	55	78.57	64.71
STT	41.67	100	58.33	14.94	79.41	23.06	25	50	66.67
SGLT	100	100	100	100	100	100	66.67	55.56	90
TPLT	88.89	87.78	87.88	83.33	74.44	78.45	72.22	80	74.81

Table 3: Detection accuracy on JHD, MPX and QUM

Results: Precision, Recall and F1-score were calculated on JHD, MPX and QUM as these systems were included in the required input formats with the distributions of the selected tools (see Table 3). The average precision, recall and F1-score of SPARQL queries generated with PatternScout(81.37%, 79.71%, 73.84%) are better than DPD(70.55%, 76.83%, 73.49%) and FINDER(56.48%, 57.91%, 67.22%). DPD and FINDER rely exclusively on code structure whereas queries generated by PatternScout are able to detect both code structure and behavior.

4 Discussion: Precision & Recall Tradeoff

We now cover threats to validity, the language-agnostic potential of our tool as well its present limitations. The tradeoff in precision and recall depends upon how restrictive the SPARQL query is. For the same design pattern, the SPARQL query can be more restrictive if there are conditions specifying visibility, property, stereotypes, etc, and less restrictive if these constraints are relaxed. While a more restrictive query can reduce false positive results (increase precision), this can fail to retrieve some instances that do not conform strictly to the structure of a pattern. For patterns like Singleton where access modifiers of the constructor and instance are important, a more restrictive query is appropriate. To achieve a balance between precision and recall, we can use a query map to identify variants to detect. The tolerance for false positives and false negatives vary for different usecases (e.g., detecting SDPs to ensure a security concern is addressed needs higher precision over recall).

5 Related Work

Earlier approaches for detecting design patterns in source code ranged from sub-graph matching [22, 23] and ontology based techniques [4] to using machine learning (ML) techniques [2, 12]. A detailed meta-analysis of various design pattern mining approaches is discussed in [24]. Summary of comparison is in Table 4.

6 Conclusion

PatternScout is a lightweight tool that automatically generates SPARQL queries from Class and Sequence diagrams to find design patterns in source code. The generated query has the same granularity as input diagrams in terms of entities and relationships between those entities. Thus, it is able to identify more types of patterns than other techniques.

Technique	Limitation	PatternScout
Semantic Web / Ontology [4]	Supports variants with same number of targets; requires manual creation of queries/rules for each pattern	requires presence of Class/Sequence diagrams, many of which already exist
ML / Code Metrics [2] [25]	accommodates variations but compromises accuracy; requires manual training for each pattern	accommodates variations without compromising accuracy; no manual training
Subgraph Matching [23] [26]	can't capture design- and code-level concepts; subject to false negatives as results are very specific	captures design- and code-level concepts; minimizes false negatives using query maps

Table 4: PatternScout with Existing Techniques

While we primarily focused on OO design patterns, SDPs that can be expressed as Class or Sequence diagrams can also be detected. We evaluated PatternScout using representative patterns from three types of design patterns: creational, structural, and behavioral. Precision and recall on the open source projects indicate that our technique is comparable to, or better than related tools.

Acknowledgment

The authors thank Elif Hepateskan, Conor Barrett, Sonam Misra, Aashima Mehta, Namita Dave, Aidar Kurmanbek-Uulu, Zhijun Huang, and Logan Petersen for their assistance with performing evaluations. Matthew Hewitt assisted with evaluation and related work. Jacob McHugh assisted with providing Query Map feature. This effort is supported in part by the University of Washington Bothell Computing and Software Systems (CSS) Division Project & the CSS Division Graduate Research funds.

References

[1] M. VanHilst and E. B. Fernandez, "Reverse engr to detect security patterns in code," in *Proc. Int'l Workshop on Software Patterns & Quality. Info Processing*, 2007.

[2] S. Uchiyama, A. Kubo, H. Washizaki, Y. Fukazawa, and others, "Detecting design patterns in object-oriented program source code by using metrics & machine learning," *Journal of Software Engr & Applications*, vol. 7, no. 12, 2014.

[3] J. Niere, M. Meyer, and L. Wendehals, "User-driven adaption in rule-based pattern recognition," University of Paderborn, Germany, Tech. Rep. tr-ri-04-249, 2004.

[4] S. Paydar and M. Kahani, "A semantic web based approach for design pattern detection from source code," in *Proc Int'l eConf on Computer & Knowledge Engr*, 2012.

[5] M. Bunke, "Security-Pattern Recognition & Validation," PhD Thesis, Universität Bremen, 2019.

[6] J. Potoniec, "Learning SPARQL Queries from Expected Results," *Computing & Informatics*, vol. 38, no. 3, 2019.

[7] F. Haag, S. Lohmann, and T. Ertl, "SparqlFilterFlow: SPARQL Query Composition for Everyone," in *The Semantic Web: ESWC Satellite Events*, 2014.

[8] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on GoF design patterns: A mapping study," *Journal of Systems & Software (JSS)*, vol. 86, no. 7, 2013.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[10] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, "Secure design patterns," CMU Softw Engr Inst, Tech. Rep., 2009.

[11] G. Rasool and H. Akhtar, "Towards A Catalog of Design Patterns Variants," in *Int'l Conf on Frontiers of Info Tech*, 2019.

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *Trans on Software Engr (TSE)*, vol. 32, no. 11, 2006.

[13] "Design patterns implemented in java," https://github.com/iluwatar/java-design-patterns/l, (accessed: 05.27.2020).

[14] M. Atzeni and M. Atzori, "CodeOntology: RDF-ization of source code," in *Int'l Semantic Web Conf*. Springer, 2017.

[15] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Detecting the behavior of design patterns through model checking & dynamic analysis," *Trans on Softw Engr & Methodology*, vol. 26, no. 4, 2018.

[16] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns & method signatures," *Journal of Systems & Software*, vol. 103, 2015.

[17] R. Xiong, D. Lo, and B. Li, "Distinguishing Similar Design Pattern Instances through Temporal Behavior Analysis," in *Proc Int'l Conf on Softw Analysis, Evolution & Reengineering*, 2020.

[18] H. Dabain, A. Manzer, and V. Tzerpos, "Design pattern detection using FINDER," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1586–1593.

[19] M. L. Bernardi, M. Cimitile, and G. Di Lucca, "Design pattern detection using a DSL-driven graph matching approach," *Journal of Software: Evolution & Process*, vol. 26, no. 12, 2014.

[20] G. Rasool and P. Mäder, "A customizable approach to design patterns recognition based on feature types," *Arabian Journal for Science & Engr*, vol. 39, no. 12, 2014.

[21] A. Binun and G. Kniesel, "Joining forces for higher precision and recall of design pattern detection," *CS Department III, Uni. Bonn, Germany, Technical report IAI-TR-2012-01*, 2012.

[22] D. Yu, Y. Zhang, J. Ge, and W. Wu, "From sub-patterns to patterns: an approach to the detection of structural design pattern instances by subgraph mining & merging," in *Proc Comp Softw & App Conf*, 2013.

[23] M. Gupta and A. Pande, "Design patterns mining using subgraph isomorphism: Relational view," *Int'l Journal of Softw Engr and Its App*, vol. 270.

[24] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *Int'l Journal of Software Engr & Knowledge Engr*, vol. 19, no. 06, 2009, publisher: World Scientific.

[25] F. Tie, J. Le, Z. Jiachen, and W. Hongyuan, "Design pattern detection method based on stacking generalization," *Journal of Software*, vol. 31, no. 6, 2020.

[26] W. Liu, C. Zhang, F. Wang, and Y. Yang, "Combining Network Analysis with Structural Matching for Design Pattern Detection," in *Proc Evaluation & Assessment in Softw Engr*, 2020.