International Journal of Trend in Scientific Research and Development (IJTSRD)
International Open Access Journal
ISSN No: 2456 - 6470 | www.ijtsrd.com | Volume - 2 | Issue – 1

A Pattern Mining Approach for Identifying Identical Design Structures in Object Oriented Design Model

Vijayalakshmi MM
Assistant Professor, Department of ISE,
GSSSIETW, Mysore

ABSTRACT

Object-oriented design patterns are frequently used in real-world applications. Detection of design patterns is essential for comprehension of the intent and design of a software project. This paper presents a graph-mining approach for detecting design patterns. Our approach is based on searching input design patterns in the space of model graph of the source code by isomorphic sub-graph search method. We developed a tool called DesPaD to apply our pattern detection approach in an automated-way. We successfully detected 23 GoF design patterns in the demo source code of the Applied Java Patterns book and also obtained encouraging results out of our experiments that we conducted on JUnit 3.8, JUnit 4.1 and Java AWT open source projects.

Keywords: ***Software project, design pattern, subgraphs mining, object-oriented.***

I. INTRODUCTION

Object-oriented principles and reusable design patterns are frequently used in software projects. Due to lack of documentation, it would typically take a long time for a developer to comprehend the design of the entire source code. As the developers of a software project can change during the project life cycle, getting insights of the source code for the new developers will be a repeating process. Therefore, it is crucial to have a tool for revealing the intent and design of a software project. As design patterns are used for solving a common recurring design problem in a particular context in terms of reusable object-oriented design, they are important for understanding software architecture and assessing its nature and quality. Furthermore, keeping up maintenance tasks on a software project takes more than 2/3 of the total cost, where comprehension activities constitutes considerable amount [1, 2]. Consequently, a design pattern detection tool for an object-oriented software project is essential because, by using such a tool, intent, design and general view of a software project can be extracted easily.

Detecting design patterns from a software project attracted attention after object-oriented design principles were established and design patterns like GRASP [3] and GoF [4] were described. Within this context, capturing static and dynamic aspects of the software by using reverse- engineering methods [5, 6, 3], defining patterns based on software metrics and their roles [7, 8], identification of micro-architectures similar to design patterns [9, 10] and some graph-based approaches [11, 12, 13, 14, 15] are published in the literature. Our approach is to build a high- level model graph of a given software project, to represent design patterns as graphs and to implement sub-graph mining search using open-source tool, Subdue [16, 17]. We target at a high-level understanding of a project by extracting and visualizing design patterns used in it, which will help developers or architects of the project to comprehend it conveniently.

We developed a fully automated tool, DesPaD (Design Pattern Detector) for detecting design patterns. We conducted our experiments by using the demo source code came with the Applied Java Patterns text book [23] and also on some open software projects namely JUnit 3.8, JUnit 4.1 and Java AWT projects. Our experiments showed promising results.

The rest of the paper is organized as follows. Background and related work are defined in Section 2.

DesPaD's approach for detecting design patterns is explained in Section

3. The results we obtained from our experiments are given and discussed in Section 4 Section 5 concludes the paper.

II. RELATED WORK

There is graph-based design pattern detection approaches submitted in the literature [12, 13, 14, 15]. A template matching method is implemented to detect design patterns in a given source code in [12]. They determine some features of design patterns to create templates and, then look them up in a given source code. As compared to our work, their approach does not go deep into pattern specifics as much as we do. In our work, we build a model graph of the given source code with twelve relation types, which is more specific than design features implemented in [12]. For example, while they have a single design feature to cover generalization pattern, we have three specific relations, which are extends, implements and overrides to cover the same feature. Building a detailed model graph helps us prevent from false-positives while detecting patterns. Calculating the similarity scores of each vertex in matrices representing the features of patterns is used for detecting design patterns in [13]. The drawback of this study is that the algorithm presented calculates only the similarity between vertices, instead of sub-graphs. As a result, high similarity score of two vertices can produce false-positively detected design patterns. Our matching algorithm, on the other hand, depends on isomorphic sub-graph search and we compare two graphs to find the candidate design patterns in the software project.

Similar to our work, an isomorphic graph matching method used to detect design patterns is given in [14]. This approach uses only class diagrams of the GoF design patterns for detecting patterns which might cause false- positive outputs. They do not consider sequence diagrams of patterns where the behavior of pattern lies. Our approach considers sequence diagrams as well as class diagrams. For example, "Class A creates an object of Class B" is a behavior type relation that we take into consideration. Shortly, our relation set is more specialized in terms of structure and behavior of patterns. Detecting design patterns by using graph matching and Constrain Satisfaction Problem (CSP) search algorithms in an Abstract Semantics Graph (ASG) of a given software project is another method applied in [15]. While they take the entire AST of a given project into the ASG of the project, we build a high-level model graph by taking only four kinds of nodes and twelve types of relations, which are considered sufficient for detecting the GoF design patterns. This helps us to find design patterns in a more simplified way.

There are also studies in the literature for detecting design patterns in a software project by means of reverse- engineering methods [5, 6]. PINOT is a tool presented in [5] which allows searching for design patterns based on their structures and then performing static program analysis, e.g. data flow analysis and control flow analysis to detect methods collaboration. As compared to our work, there are three basic differences. First, PINOT uses specific keywords to detect design patterns while we remain more generic. For instance, PINOT detects "template method" design pattern by specifically looking up final methods. Consequently, our approach decreases the rate of false-positive detected patterns. Second, while PINOT depends on the java compiler (Jikes) for searching patterns, our isomorphic sub-graph search algorithm is independent of any programming language. Third, it is not easy to add new patterns or modify existing ones in PINOT while we can simply perform such tasks without requiring any coding or compilation. An approach based on static and dynamic analysis of software project's ASG (Abstract Semantics Graph) is presented in [6]. The detection process of this approach is executed during the run-time of the software by means of log analysis. Therefore, it can only detect patterns that occur at run-time as difference to our work where we analyze the entire source code.

Properties of design patterns are correlated with some of the software metrics in other works. Creating design pattern fingerprints by specifying the roles and metrics of classes is studied in [7]. They reduce the search space by implementing a machine-learning algorithm in their repository. There also exists another one in which they implemented multi-stage reduction process by using object- oriented software metrics and structural properties to detect design patterns from a software project's source code [8]. Because they hard-coded rules for detection process and they experimented with only five GoF patterns in their implementation. Our approach, however, works on higher levels to extract design patterns and this makes our approach more flexible. Thus, we are able to experiment all of the GoF patterns.

To the best of our knowledge, the most relevant study to our work in terms of building graph model of a source code is another graph mining approach for detecting frequently used identical sub-structures in a software project by a frequent sub-graph mining method using open-source Parsemis tool [11, 18]. While we focus on detecting especially GoF design patterns in a software project and tag them automatically, they detect frequent sub-graph identical structures and then tag them manually. We also add some new relations like "Class A has the return type of Class B", "Class A related with its method of Class B" etc. in order to form design patterns' template properly.

III. DETECTING DESIGN PATTERNS

Our approach to detect design patterns consists of three basic steps. First, we analyze the source code and extract ASTs out of it. Then, we build a graph model by using these ASTs. Second, we generate templates for all the GoF design patterns. These patterns will be used basically as query items and they will be generated only once unless new design patterns are introduced in the literature. Third, we search for the pattern templates in the model graph by using Subdue's sgiso sub-graph mining algorithm. The overview of DesPaD's design pattern detection architecture is seen on Fig. 1.

We developed a fully automated, java based design pattern detection tool called DesPaD (Design Pattern Detector) to execute all the steps given above. It is fast, convenient to use and targets at finding design patterns in a high-rate of correctness. DesPaD is freely available at Github [29].

Details of these steps will be explained in the following subsections.

A. Model Graph Creation

In this step, a high-level graph representation of an object-oriented software project's source code is generated. A software project is represented as a simple labeled and directed graph (G). Formally, a graph is defined for a formation by vertices and edges connecting the vertices [19].

Software Model Graph (G): Let G = (V, E, Le, Lv) be a labeled digraph, where V is a set of vertices or nodes, E is a set of edges or arcs, Lv is a set of labels for the vertices and Le is a set of labels for the edges [11].

Sub-graph: A graph is a sub-graph of G, defined as Gs G, if the vertices and edges of Gs embodies a subset of the vertices and edges of G (Vs V and Es E).

Isomorphic sub-graph: The two graphs G1 = (V1, E1) and G2 = (V2, E2) are isomorphic if labeling the vertices of G1 bijectively with the elements of V2 gives G2 and multiplicity of edges are maintained.

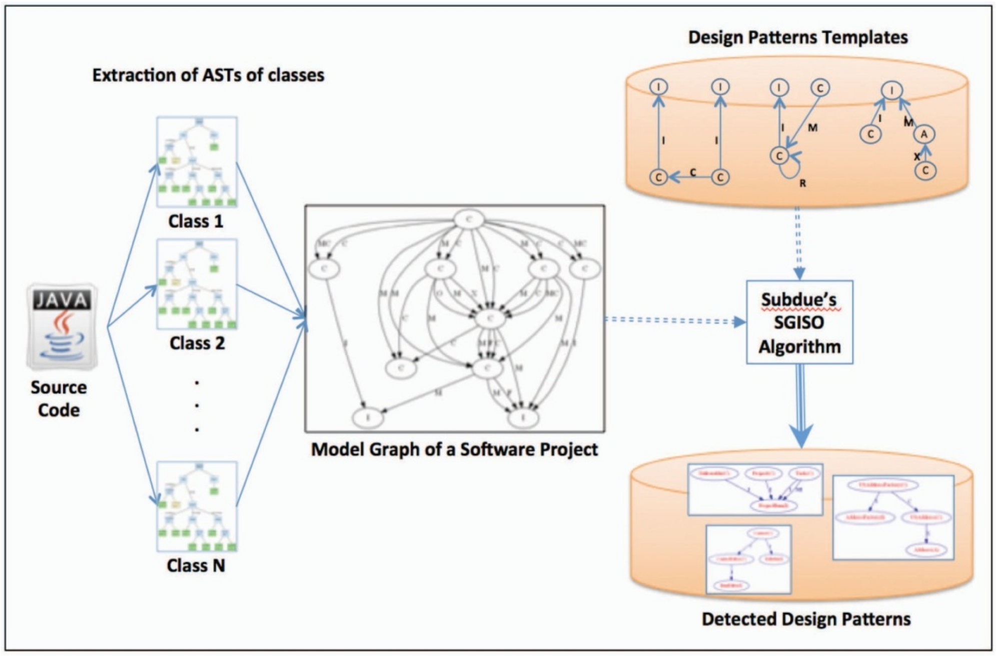
Extraction of ASTs of classes
Design Patterns Templates
Class 1
Class 2
Class N
JAVA
Source Code
Model Graph of a Software Project
Subdue's SGISO Algorithm
Detected Design Patterns

Figure 1: Overview of DesPaD's pattern detection architecture

The vertices of our model graph (G) are classes, abstract classes, template classes and interfaces. The edges of (G) include the specific relations of inheritance, aggregation, association and composition properties used commonly in object-oriented programming. The vertex and edge properties in a model graph (G) are shown in Table I and Table II, respectively.

TABLE I. VERTEX LABELS AND TYPES

Vertex Label	Entity Type
C	Class
I	Interface
A	Abstract Class
T	Template Class

TABLE II: EDGE LABELS, RELATIONS AND TYPES

Edge Label	Relation Type[a]
X	Class A extends Class B
I	Class A implements Class B
C	Class A creates object of Class B
O	Class A overrides a method of Class B
MC	Class A calls a method of Class B
F	Class A has the field type of Class B
MR	Class A has a method with the return type of Class B
ML	Class A has a method that defines a local variable with the type of Class B
MI	Class A has a method that has an input parameter with the type of Class B
M	Class A has related with its method of Class B
R	Class A has the return type of Class B
G	Class A uses Class B in a generic type declaration

As our final goal is to catch the relations of GoF design patterns in the source code, we analyzed the class diagrams and collaborations (also called sequence diagrams) within every GoF design pattern [4]. As a result of this analysis, we identified relations listed in Table II. "Class A calls method of Class B", "Class A creates an object of Class B" and "Class A has the return type of Class B" in Table II are high level behavioral relations extracted from sequence diagrams. All other relations are extracted from class diagrams.

The building process of our model graph starts with generating the abstract syntax tree of each class of the given software. ANTLR (Another Tool For Language Recognition) [20] which is an open source Java library that contains a top-down parser for a subset of context-free languages is used for generating ASTs. ANTLR library is able to generate lexers, parsers and tree parsers and, provide the ability of traversing trees.

Java language grammar is already available as BNF (Backus Normal Form) diagrams [21]. DesPaD uses these BNF diagrams to detect relations listed in Table II. For instance, the inheritance relations like "extends" and "implements" in class declaration are detected by using the BNF diagram in Fig. 2.

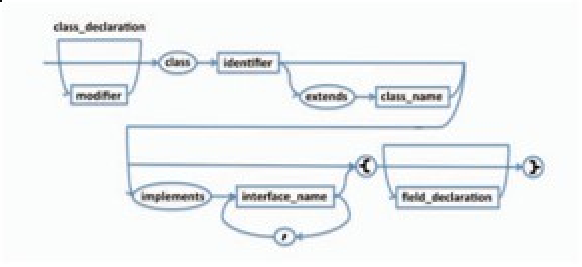

Figure 2: BNF Diagram of class declaration in Java Grammar Language

As a result, a model graph for sub-graph mining process is created similar to the one in Fig. 3. DesPaD prepares and creates the model graph in a file formatted for the open- source sub-graph mining tool, Subdue.

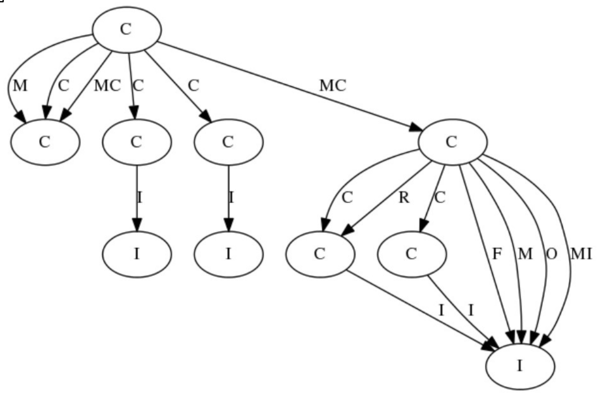

Figure 3: An example of a model graph

B. Design Pattern Template Generation

After having built the model graph as our search space, our goal is to search for sub-graphs that might represent the GoF design patterns. To achieve this, we analyzed the class and sequence diagrams of all 23 GoF design patterns and generated template graphs for each of them. An example template that was generated for the bridge design pattern is seen in Fig. 4.

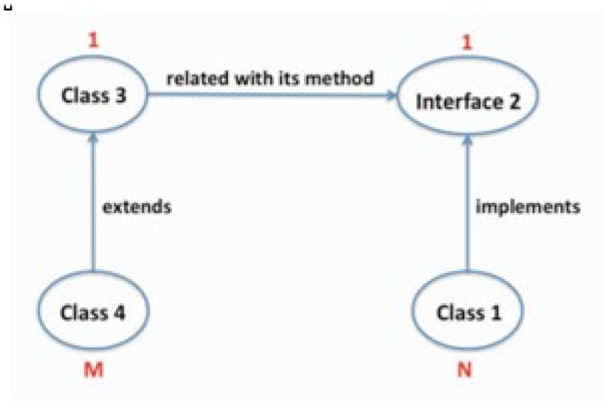

Figure 4: Bridge design pattern's template

As seen in Fig. 4, vertices were tagged with *1*, *M* and *N*. *1* means that the vertex and its edges occur only once. *M* and *N* mean that the vertex and its edges can occur more than one. DesPaD determines the maximum values for M and N by counting the numbers of times a node has a specific relation. That is, for the bridge pattern given in Fig. 4, maximum numbers of times any class in the entire source code was extended or implemented are assigned to M and N, respectively. Afterwards, all possible design patterns template graphs are generated. For example, according to the bridge pattern template graph in Fig. 4, if *M* is 11 and *N* is 5, the number of the bridge pattern template graphs that will be generated is 55. These 55 patterns are saved in input files for sub-graph mining tool, Subdue.

C. Design Pattern Detection

After having generated the model graph of the software and the design patterns' template graphs, we can execute sub-graph mining search. To do this, we used an isomorphic sub-graph mining algorithm called "sgiso" provided by the open-source graph-mining tool, Subdue [17].

The algorithm for detecting design patterns is given in Algorithm 1. Maximum numbers for *M* and *N* are given as input to the algorithm. However, it will be time and resource consuming to implement all combinations of the candidate templates to the isomorphic sub-graph search tool. For example, if you consider the bridge pattern in Fig. 4, if *M* is 11 and *N* is 5, there would be 55 combinations to run the isomorphic search for. Instead, the algorithm stops trying after some value *i*, if the sub-graph search returns nothing for *i+1*.

Algorithm 1: Detection of Design Patterns by Sub-graph Isomorphic Search.

Data: Relations' count of the design pattern template $\{M_i\}$, $\{N_j\}$ ($M \geq N$);
Generated candidate input files *input_file[M][N]*;

```
foreach x ∈ {M_i} do

  /* After running sub-graph isomorphism
     algorithm(sgiso) in Subdue, we get output
     files in outputs[]. */

     execute sgiso on input_file[x][0]; add
     output of sgiso to outputs[];

         if no output exists then
                   break;
         end
         foreach y ∈ {N_j} do
         execute sgiso on input_file[x][y]; add
         output of sgiso to outputs[];

         if no output exists then
                   break;
         end end
  end
  foreach y ∈ {N_j} do
        execute sgiso input_file[0][y];
        add output of sgiso to outputs[];

        if no output exists then
                  break;
        end
  end
```

After the algorithm is executed, there might be overlapping sub-graphs in the output list. Overlapped sub- graphs are eliminated accordingly. And finally, found design patterns can be visualized by DesPaD. To

achieve this, DesPaD uses the open source GraphViz application [22]. An example bridge pattern extracted from Java AWT
1.3. project is visualized as seen in Fig. 5.

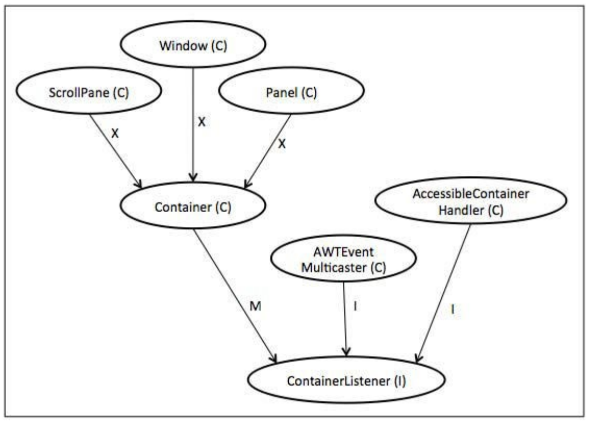

Figure 5: An example bridge pattern extracted from Java AWT 1.3.

IV. EVALUATIONS

We have conducted extensive experiments to test our approach and its performance. We also compared our tool and its results against its closest rivals, PINOT [5], HEDGEHOG [13], FUJABA [14] and DP-Miner tool [15] that we mentioned in this paper.

As test bed, we used codes from four different sources. We chose source codes that were used as benchmarks by our rivals. These are demo source codes from "Applied Java Patterns" (AJP) text book [23] and source codes of three open source projects, i.e. JUnit 3.8, JUnit 4.1 [24] and Java AWT 1.3 [25]. However, we also plan to use real-world applications on industry in the future. Projects in the test bed are all Java projects. Note that our approach is not bound to a specific project. DesPaD can be adapted for another programming language, e.g. C++ or C#.

Experiments were done on a Linux running quad-core CPU commodity computer with 8 GB of RAM. Evaluation results are analyzed in terms of precision and recall. Precision is the rate of true pattern instances found out of the total number of instances extracted by the tool. Recall is the rate of the true pattern instances found by the tool in the actual existing pattern instances. Actual true instances are based on the documentation of the open-source projects [26, 27, 28].
First, we compared DesPaD against similar tools in terms of capabilities. Table III shows which patterns in the AJP example can be detected by each tool. The AJP example is chosen since it contains all GoF design patterns. Patterns are grouped as *creational, structural* and *behavioral* in the table. *OK* means that pattern can be detected by the tool. *X* means that the tool has failed to detect that pattern. If the tool does not cover the pattern at all, it is showed with the "- " symbol. According to Table III, DesPaD is the only tool which can detect all 23 GoF patterns (100 %). The closest rival, PINOT can only detect 17 out of 23 patterns (74%).

TABLE III: COMPARISON ABOUT VERIFICATION OF DESIGN PATTERNS

	Tools			
	PINOT	***HEDGEHOG***	***FUJABA***	***DesPaD***
Creational				
Abstract Factory	*OK*	*OK*	*X*	*OK*
Builder	-	-	-	*OK*
Factory Method	*OK*	*OK*	*X*	*OK*
Prototype	-	*X*	-	*OK*
Singleton	*OK*	*OK*	*OK*	*OK*
Structural				
Adapter	*OK*	*OK*	*X*	*OK*
Bridge	*OK*	*OK*	*OK*	*OK*
Composite	*OK*	*OK*	*X*	*OK*
Decorator	*OK*	*OK*	*X*	*OK*
Facade	*OK*	-	*OK*	*OK*
Flyweight	*OK*	*OK*	*X*	*OK*
Proxy	*OK*	*OK*	-	*OK*
Behavioral				
CoR	*OK*	-	*X*	*OK*
Command	-	-	-	*OK*
Interpreter	-	-	-	*OK*
Iterator	-	*OK*	*X*	*OK*
Mediator	*OK*	-	*X*	*OK*
Memento	-	-	*X*	*OK*
Observer	*OK*	*OK*	*X*	*OK*
State	*OK*	*X*	-	*OK*
Strategy	*OK*	*OK*	*OK*	*OK*
Template Method	*OK*	*OK*	*OK*	*OK*
Visitor	*OK*	*OK*	-	*OK*

Second, we tested DesPaD against the open source projects that we have in our test bed. Numbers of classes and lines of code regarding these projects are given in Table IV.

TABLE IV: SIZE OF SELECTED PROJECTS

Project	Number of Classes	Thousands of lines of code
JUnit 3.8	54	4.7
JUnit 4.1	157	4
AWT 1.3	407	102

Test results of DesPaD against JUnit 3.8, JUnit 4.1 and Java AWT 1.3 projects are seen in Table V, Table VI and Table VII, respectively. *Actual instances* are the number of times a pattern really occurs in the source code. *Found instances is* the number of patterns that was returned by DesPaD and claimed as found in the source code? True instances are the number of correctly found patterns by DesPaD.

According to test results, we detect design patterns with 80% precision and 88% recall values in average. DesPaD works almost perfect for the smallest project in our test bed, i.e. JUnit 3.8. As the number of classes and lines of codes in projects increase, precision and recall values may suffer. However, 78% of the actual patterns are still correctly detected and, for precision values below average, only the 21% of the cases generates false positives.

TABLE V. JUNIT 3.8 TEST RESULTS (DESPAD)

Pattern Name	Found/ True	Actual Instances	Precision	Recall
Bridge	2/2	2	100	100
Composit	1/1	1	100	100
Decorato	1/1	1	100	100
Singleton	0/0	0	NA	NA
Template Method	12/11	11	92 %	100 %

TABLE VI: JUNIT 4.1 TEST RESULTS (DESPAD)

Pattern Name	Found/ True	Actual Instances	Precision	Recall
Bridge	4/1	1	25 %	100 %
Composite	2/2	2	100 %	100 %
Decorator	1/1	4	100 %	25 %
Singleton	4/1	1	25 %	100 %
Template Method	22/20	20	91 %	100 %

TABLE VII: JAVA AWT 1.3 TEST RESULTS (DESPAD)

Pattern Name	Found/ True	Actual Instances	Precision	Recall
Bridge	20/20	30	100 %	66 %
Composite	9/2	2	22 %	100 %
Decorator	7/7	7	100 %	100 %
Singleton	18/14	14	78 %	100 %
Template Method	55/55	128	100 %	43 %

Third, we compared our work with PINOT as we described it as the closest work in literature which is close to DesPaD in terms of capabilities the closest work available. In addition, we did not have access to HEDGEHOG [13] or FUJABA [14] test results on the chosen source codes. We were not able to produce them, either. DP-Miner [15] provides promising results similar to DesPaD. However, since it does not cover all design patterns and uses a hard- coded mechanism by using specific properties of design patterns and related programming language, we did not include it in the comparisons. We are going to consider it in our future work when we add optimizations to our algorithm similar to what DP-Miner has.

Table VIII compares precision and recall performances of DesPaD to PINOT tools when they worked against the Java AWT 1.3 source codes. Regarding precision, PINOT seems to perform 8% better then DesPaD in average.

However, recall values are 47% better for DesPaD in average. That is, DesPaD detects much more design patterns than PINOT does. Recall values for PINOT can be as low as 3% while it can be only 35% in average.

TABLE VIII: DESPAD VS. PINOT

Pattern Name	Precision		Recall	
	DesPaD	*PINOT*	*DesPaD*	*PINOT*
Bridge	100 %	75 %	66 %	10 %
Composit	22 %	67 %	100 %	100 %
Decorato	100 %	100 %	100 %	43 %
Singleton	78 %	100 %	100 %	22 %
Template Method	100 %	100 %	43 %	3 %
Averages	80	88	82	35

Finally, we evaluate the performance of DesPaD in terms of run time. Run times required to detect five different design patterns within the chosen open source projects are seen in Table IX. As isomorphic sub-graph mining search is an NP-Complete problem, it is a big challenge to reach a good performance in case of large sized software systems. JUnit 3.8 and JUnit 4.1 are small projects and DesPaD performs at the level of few seconds except for detecting the *Template Method* pattern. Java AWT 1.3, on the other hand, is a relatively large project, where the performance of DesPaD varies from few minutes to few hours. Note that we performed these evaluations on a simple commodity computer with limited CPU and memory. In case of a more powered experimental infrastructure, numbers for AWT 1.3 evaluations can be pulled down. Our future goal, however, is to optimize our algorithm to get better results.

TABLE IX: DESPAD PERFORMANCE EVALUATIONS

Pattern Name	JUnit 3.8		JUnit 4.1		AWT 1.3	
	Time	*Input file count*	*Time*	*Input file count*	*Time*	*Input file count*
Bridge	0,00	9	1	66	10560	690
Composite	0,07	9	3	36	9872	900
Decorator	0,05	18	1	132	950	1380
Singleton	22	1	2	1	5	1
Template Method	2	10	4758	42	4690	90

V. CONCLUSIONS AND FUTURE WORK

In this work, we built a high-level model graph out of source codes of a project, generated representative graphs for design patterns and tried to detect those patterns in the model graph by using an isomorphic graph-matching algorithm.

We developed an automated detection tool called DesPaD. We tested it against source codes from four different projects and compared it with the related work. To the best of our knowledge, DesPaD is the only tool which can detect all GoF design patterns. Also, it outperforms the closest work by creating 47% better recall values.

As future work, we intend to optimize our approach. Due to the complexity of the sub-graph search algorithms, DesPaD's performance might suffer in case of large-sized projects. To alleviate this problem, some partitioning or optimization algorithms will be investigated. Additionally, we plan to analyze software metrics of the given source code and use them in the pattern detection process for optimization in the future.

Accordingly, detecting design patterns which are described in some novel catalogues proposed in the literature will be the part of our future work.

REFERENCES

1. C. Gravino, M. Risi, G. Scanniello, G. Tortora, Does the documentation of design pattern instances impact on source code comprehension ? Results from two controlled experiments, Proceedings of the Working Conference on Reverse Engineering, IEEE CS, 2011, pp. 67-76.
2. L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Trans. Software Engineering 28 (6), 2002, pp. 595-606.
3. C. Larman, Applying UML and Patterns : An Introduction to Object- Oriented Analysis and Design and the Unified Process, Prentice Hall, 2001.
4. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
5. N. Shi, R. A. Olsson, Reverse Engineering of Design Patterns from Java Source Code, 21st IEEE ınternational Conference on Automated Software Engineering (ASE'06), 2006.
6. H. Lee, H. Youn, E. Lee, A Design Pattern Detection Technique that Aids Reverse Engineering, International Journal of Security and its Applications Vol. 2, No. 1, 2008.
7. Y. G. Gueheneuc, H. Sahraoui, F. Zaidi, Fingerprinting Design Patterns, Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04), 2004
8. G. Antoniol, G. Casazza, M. Di Penta, R. Fiutem, Object-oriented design patterns recovery, The Journal of Systems and Software 59, 2001, pp. 181-196.
9. Y. G. Gueheneuc, P-MARt : Pattern-like Micro Architecture Repository, Proceedings of the 1st

EuroPLoP Focus Group on Pattern Repositories (EPFPR), 2007.

10. Y. G. Gueheneuc, G. Antoniol, DeMIMA : A Multilayered Approach for Design Pattern Identification, IEEE Transactions on Software Engineering, Vol. 34, No. 5, 2008.
11. U. Tekin, F. Buzluca, A graph mining approach for detecting identical design structures in object-oriented design models, Science of Computer Programming, 2013.
12. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32, 2006, pp. 176-192.
13. Dong, J., Sun, Y., Zhao, Y., Design Pattern Detection by Template Matching, Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Brazil, 2008, pp. 765-769.
14. M. A. Soliman, I. A. M. ElMeddah and A. M. Wahba, Patterns Mining from Java Source Code, Int.J. of Software Engineering, IJSE Vol.4 No.2, 2011.
15. R. S. Rao, M. Gupta, Design Pattern Detection by a Heuristic Graph Comparison Algorithm, International Journal of Advanced Research in Computer Science and Software Engineering 3(11), 2013, pp.251- 255.
16. D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design the Pattern Detection, Proceedings of the 11 IEEE International Workshop on Program Comprehension (IWPC'03), 2003.
17. Subdue, http://ailab.wsu.edu/subdue/.
18. U. Tekin, U. Erdemir, F. Buzluca, Mining Object-Oriented Design Models for Detecting Identical Design Structures, Sixth International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, 2012, pp. 43-49.
19. K. Ruohonen, Graph Theory Lecture Notes, 2013.
20. T. Parr, The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, 2012.
21. BNF Index of Java language grammar, http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html.
22. GraphViz, www.graphviz.org.
23. S. Stelting and O. Maassen, Applied Java Patterns, Prentice Hall, Palo Alto, California, 2002.
24. JUnit, http://www.junit.org/.
25. Java AWT, http://docs.oracle.com/javase/7/docs/api/java/awt/.
26. E. Gamma, JUnit A Cook's Tour, http://junit.sourceforge.net/doc/cookstour/cookstour.htm.
27. C. Sars, P. Wessman, and M. Halme, Design Patterns and the Java AWT, http://www.niksula.hut.fi/~ged/DesignPatterns/.
28. J. Zukowski, Java AWT Reference, http://oreilly.com/catalog/javawt/book/index.html
29. DesPaD Design Pattern Detection Tool, https://github.com/muratoruc2006/DesPaD.git.