Detection of Design Pattern Using Graph Isomorphism and Normalized Cross Correlation

Prayasee Pradhan[1], Ashish Kumar Dwivedi[2], Santanu Kumar Rath[3]
Department of Computer Science and Engineering
National Institute of Technology
Rourkela, 769008, Odisha, India
Email: prayasee91@gmail.com[1], shil2007@gmail.com[2], skrath@nitrkl.ac.in[3]

***Abstract*—Present day software engineering concept gives emphasis on the use of design patterns for developing software which are recurring in nature. Detection of design pattern is one of the emerging field of Software Reverse Engineering. In this work, an attempt has been made to present an approach for design pattern detection with the help of Graph Isomorphism and Normalized Cross Correlation techniques. In this study, system and design pattern are presented in the form of graphs. The Graph Isomorphism technique finds the pattern subgraph in the system graph and Normalized Cross Correlation provides a way to formulate the percentage existence of the design pattern in the system. An Eclipse Plug-in i.e., ObjectAid is used to extract Unified Modeling Language (UML) class diagrams as well as eXtensible Markup Language (XML) files from the software system and design patterns. An algorithm is proposed to identify relevant information from the XML files. Then Graph Isomorphism and Normalized Cross Correlation techniques are used to find the pattern subgraph and its percentage existence in the system. This approach has been applied on four open source software tools for the evaluation of five design patterns, such as Composite, Facade, Flyweight, State, and Template Method.**

***Index Terms*—Design Patterns, Design Pattern Detection, Graph Isomorphism, Normalized Cross Correlation, UML.**

I. Introduction

The concept of design pattern is applied to find reusable and well documented solutions for the recurring design problems. A good number of design patterns for use in software development have been identified, documented, and analyzed in the literature [1] [2] [3] [4] [5] [6]. These design patterns facilitate the understandability and construction of systems. The design pattern tools support predictable and uninterrupted use of services as well as resources, which are offer to user. Each pattern is described by using a pattern template. These pattern templates promote to express a solution for the recurring design problem [7]. Templates are used to capture all the elements of a pattern and describe its issues, motivation, strategies, technologies, applicable scenarios, solutions, and examples. Gamma et al. [2] also known as Gang of Four (GoF) proposed standard templates for their twenty three design patterns. Design patterns are used improve software maintenance by making explicit representation of class and object interactions.

Detection of design pattern is an essential task providing support to the program comprehension, design recovery, maintenance, and reverse engineering of software systems. The process of reverse engineering encompasses a number of methods and tools to derive information and knowledge from available artifacts and provide it into software engineering process. To obtain the abstractions and views from a target system, system developers rely on reverse engineering activities to maintain, evolve and eventually re-engineer the system. Design pattern detection (DPD) is a vast area of research in the field of reverse engineering and reconstruction of software architecture [8]. Detection of design pattern also helps in the re-documentation phase of software development life cycle and enhances the maintainability of the software. Design pattern detection is useful to provide better comprehension of a software components and its architecture without having adequate knowledge of programming implementations. A good number of design pattern detection tools are available in the literature [9]. But these tools have certain limitations in terms of usability.

There are various reasons why it is difficult to detect a design pattern in a software. Finding a design pattern in large software system is a difficult task, because of larger exploration space. Secondly, a class may play more than one role under different design patterns. Thirdly, design pattern detection tools produce ambiguous results. Another problem is increasing size space of the number of software design patterns, such as architectural patterns, design patterns, security patterns, etc. To find the accuracy of design pattern detection is essential task in terms of percentage matching. Hence, in this study the Graph Isomorphism and Normalized Cross Correlation techniques are applied to find pattern subgraph and its percentage existence in the system.

Various techniques have been previously adopted to detect design pattern from source code as well as design model such as UML class diagrams [10] [11]. However, these methods are not fully automated. In this work, we have proposed a noble approach to detect design patterns from UML class diagrams automatically. The reverse engineering process of extracting UML class diagram from the source code has been done with the help of an Eclipse plug-in i.e., ObjectAid [12]. The extracted XML files corresponding to the system diagram as well as the design pattern diagram are further evaluated to find the existence of the pattern instances. We have applied our technique on various open source Java-based projects for detection of a number of design patterns.

The rest of the paper is organized as follows. Section two

978-1-4673-7948-9/15/$31.00 ©2015 IEEE

introduces some of the related works in the field of design pattern detection. Section three covers proposed pattern detection method using Graph Isomorphism and Normalized Cross Correlation techniques. Section four presents the conclusion and future work.

II. Related Work

Gupta et al. [13] have presented an approach to detect the design patterns by the application of Normalized Cross Correlation while taking design pattern as a template to find its presence in the system design. Zanoni et al. [14] presented a design pattern detection process using machine learning techniques. They have considered five design patterns such as Singleton, Adapter, Composite, Decorator, and factory Methods for the evaluation of their study.

Ba-Brahem and Qureshi [15] have proposed an approach to detect design pattern instances in a system design which uses the graph implementation to produce both the system as well as the design pattern UML diagrams in Graph of 4-tuple elements. Tsantails et al. [16] have proposed a methodology to detect a design pattern based on similarity scoring between graph vertices which is capable of recognizing patterns that are modified from their standard representations. Instead of relying on pattern-specific heuristic, the approach reduces the search space by taking the fact into consideration that pattern resides in one or more inheritance hierarchies.

Dong et al. [17] have adopted a template matching algorithm to detect design patterns from a software system by the use of Normalized Cross Correlation. They have extracted exact matches as well as partial instances for design patterns. Dongjin et al. [18] have proposed a method to detect deign pattern instances in which they have identified all the candidate classes in the system graph satisfying pattern classes. They have selected some of candidate classes to form the sub-graphs to check their isomorphic behavior towards the pattern graph corresponding to the design pattern.

Albin-Amiot and Guéhéneuc [19] have proposed an approach to use a meta-model in order to obtain a representation of design patterns which will further allow both automatic code generation and design pattern detection. Heuzeroth et al. [20] proposed a method to detect design patterns in legacy code by combining static and dynamic analysis techniques. They have developed a tool and classified potential pattern instances according to the information provided by their tool. They have provided their analysis for various design patterns on the Java SwingSetExample.

Begenti and Poggi [21] have presented a system called IDEA (Interactive DEsign Assistant) which can automatically detect patterns in a UML class diagram and can also produce critiques about the detected patterns. They have also integrated the concept of IDEA with the CASE tool ArgoUML [22]. Gupta et al. [23] have applied a graph matching algorithm to detect design patterns in the UML class diagram of a system. The algorithm decomposes the graph matching process into K-phases, where K ranges from 1 to the minimum of the numbers of nodes in the two graphs to be matched.

Wenzel et al. [24] have proposed a method to detect design pattern instances in software systems regarding model-driven development. Their proposed approach allows developer to use UML diagram editors to specify patterns. They have used a difference algorithm called as SiDiff to compute the differences between graph-structured UML diagrams.

III. Proposed Work

Design patterns are detected with the help of Graph Isomorphism and Normalized Cross Correlation techniques. In this approach, both the system UML class diagram and design pattern class diagrams are converted into directed graphs. The nodes of the graph act as classes where as the edges connecting the nodes refers to the relationship among the corresponding classes. The block diagram of proposed approach is shown in Figure 1. For the realization of this study, some required notations have been taken into consideration.

TABLE I: Relationship Weight Table

Sl. No.	Relationship	Relationship Weight
1.	Association	2
2.	Generalization	3
3.	Realization	5
4.	Other or Disconnected	1

Notation 1: Graph G, is represented as a 3-tuple entity. G= (V, E, f(E)), where:

1) V = Set of nodes corresponding to classes of a UML class diagram.
2) E is a function from $V \rightarrow V$, corresponds to the set of edges connecting the nodes.
3) $f(E) : (E \rightarrow W_e)$, function relating the edge to a numeric weight. The value of W_e depends on the type of relationship among the classes. We have taken certain values to define various relationships which is shown in Table I.

If any two classes are connected, having more than one relationship, then the relationship weight becomes the multiplication of the individual weights corresponding to the relationship. For example, if two classes are having both Association and Generalization relationship, then the edge connecting the classes will have relationship weight = 2*3 = 6. If two classes are not connected by any of these relationship, then the relationship weight becomes 1. Hence W_e = {2,3,5,6,10,15,30} by considering all types of relationships in Table I.

Notation 2: Let S_n and P_n be the number of classes in system graph (SG) and number of classes in pattern graph (PG) respectively , then

1) SGM corresponds to System Graph Matrix (V, E, f(E)). $SGM[i,j] \in W_e$, where 'i' and 'j' are nodes corresponding to classes of system graph.
2) DPM corresponds to Design Pattern Graph Matrix (V, E, f(E)), $DPM[i,j] \in W_e$, where 'i' and 'j' are nodes corresponding to classes of design pattern graph.
3) CGM is the Connectivity Graph Matrix (V, E, p); where

$$
\begin{aligned}
p &= 1 \; if \; f(E) = 1 \\
&= 0 \; otherwise
\end{aligned} \tag{1}
$$

CGM_s is the Connectivity Graph Matrix for System graph (SG) and CGM_d is the connectivity Graph Matrix for Design Pattern Graph (PG).

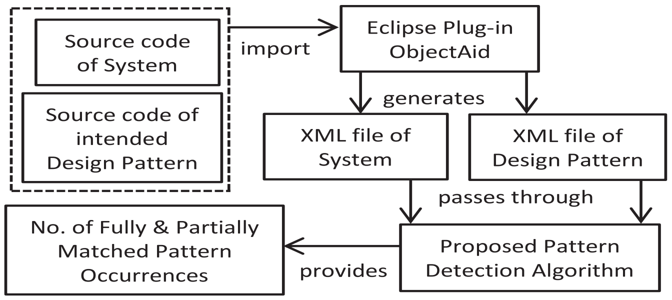

Source code of System
Source code of intended Design Pattern
import
Eclipse Plug-in ObjectAid
generates
XML file of System
XML file of Design Pattern
passes through
No. of Fully & Partially Matched Pattern Occurrences
provides
Proposed Pattern Detection Algorithm

Fig. 1: Block Diagram of Proposed Approach

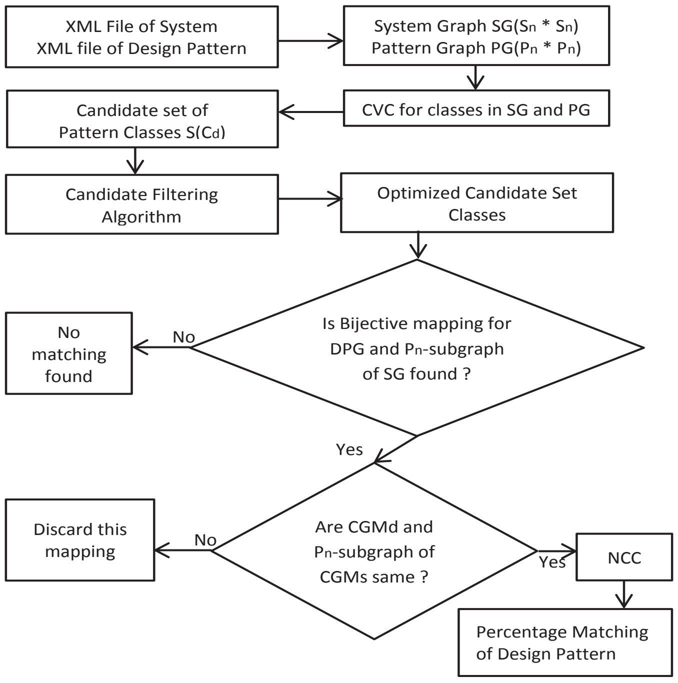

XML File of System
XML file of Design Pattern
System Graph SG(Sn * Sn)
Pattern Graph PG(Pn * Pn)
CVC for classes in SG and PG
Candidate set of Pattern Classes S(Cd)
Candidate Filtering Algorithm
Optimized Candidate Set Classes
Is Bijective mapping for DPG and Pn-subgraph of SG found ?
No
No matching found
Yes
Are CGMd and Pn-subgraph of CGMs same ?
No
Discard this mapping
Yes
NCC
Percentage Matching of Design Pattern

Fig. 2: Flow-chart of Proposed Pattern Detection Approach

Notation 3: Let CVC be the *Contribution Value of Class*. It is the multiplication value of the weights of edges connecting to all of the classes in a class diagram. If 's' and 'd' are the classes in system graph and design pattern graph respectively, then their contribution values of class are :

1) $CVC_s = \prod_{j=1}^{S_n} SGM[s,j] \; where \; s,j \in SG(V)$, the set of nodes in System Graph (SG).
2) $CVC_d = \prod_{j=1}^{P_n} DPM[d,j] \; where \; d,j \in PG(V)$, the set of nodes in Design Pattern Graph (PG).

Notation 4: We call a class C_s in system graph (SG) to be the candidate of a class C_d in design pattern graph (PG), if and only if CVC for C_s is a multiple of the CVC for C_d. Candidate Set of a class C_d = S(C_d), in design pattern graph is the set of all candidate classes C_s in system graph. S(C_d) = C_s where CVC(C_s) is a multiplier of CVC(C_d).

Notation 5: Correspondence Graph (CRG): $P_n * S_n$ graph, where $CRG[i,j] = 1$, if class 'j' in system graph is one of the Candidate Set of the class 'i' in pattern graph. i.e.,
$CRG[i,j] = 1 \; if \; Cj \in S(Ci), \; where \; i \in PG(V) \; and \; j \in SG(V)$.

A. Proposed Methodology

The flow-chart of proposed pattern detection approach is shown in Figure 2 and the proposed methodology is described as follows:

1) Generate XML files using Eclipse plug-in i.e., ObjectAid from Java source code corresponding to the system class diagram as well as the pattern class diagram.
2) Extract information regarding the relationship among classes in both the graphs corresponding to class diagrams.
3) Find CVC for all the classes in both system graph (SG) and pattern graph (PG).
4) Find candidate set for all the classes of design pattern graph, i.e., S(C_d) is to be found.
5) Use filtering algorithm to find most probable candidates for all the pattern classes.
6) Find bijective relations between pattern class and the system class and extract the P_n-subgraph which may contain the pattern instance.
7) Extract P_n-subgraph of Connectivity Graph Matrix for System Graph (CGM_s) and compare with the Connectivity Graph Matrix for Design Pattern Graph (CGM_d). If both the matrices are same, then extract the system P_n-subgraph matrix from SGM, denoted as SGM_k , where $k = P_n$.
8) Perform Normalized Cross Correlation (NCC) between the extracted system P_n-subgraph and the pattern graph.
9) From the NCC value, find percentage of matching occurrences.

The proposed methodology is implemented by using programming language C++, which takes the XML files corresponding to the system UML diagram and the design pattern diagram and provides total number of fully matched occurrences and partially matched occurrences.

B. Flitering Algorithm

In this study two algorithms are presented. First algorithm takes CRG, SGM, and DPM as input and produces Optimized Candidate Class Set as an output. Second algorithm takes two numbers and produces boolean value as an output.

Filtering Candidates Algorithm
Input : Correspondence Graph (CRG), System Graph Matrix (SGM), Design Pattern Matrix (DPM)
P_n = total number of pattern classes in design pattern graph (PG).

S_n= total number of classes in system graph (SG).

```
Data: Correspondence Graph (CRG), System Graph
      Matrix (SGM), Design Pattern Matrix (DPM)
$P_n$ = total number of pattern classes in design pattern
graph (PG).
$S_n$= total number of classes in system graph (SG).
Result: Optimized Candidate Class Set
initialization;
for $i = 0$ to $P_n$ do
    for $j = 0$ to $S_n$ do
        value = DPM[i,j]
        for $p = 0$ to $S_n$ do
            if $CRG[i, p] = 1$ then
                flag = 0;
                for $k = 0$ to $S_n$ do
                    if $CRG[j, k] =$
                    $1$ and $isfactorial(SGM[p, k], value) =$
                    $true$ then
                        flag = 1
                    else

                    end
                end
                if $flag = 0$ then
                    CRG[i,p] = 0
                else

                end
            else

            end
        end
    end
end
```

Algorithm 1: Filtering Algorithm

```
Data: Values of a and b
Result: Boolean
initialization;
if $a\%b = 0$ then
    return true;
else
    return false;
end
```

Algorithm 2: IsFactorial Algorithm

C. Implementation

Figure 3 represents the system class diagram that need to be evaluated. Figure 4 represents the class diagram of template design pattern. The graph corresponding to the system class diagram is shown in Figure 5, known as System Graph (SG), and the graph corresponding to the design pattern diagram is shown in Figure 6, known as Pattern Graph(PG). According to the notations, the system graph matrix (SGM) and design pattern matrix (DPM) are shown in Table III and Table IV respectively. Similarly Connectivity Graph Matrix for System Graph and Pattern Graph are shown in Table V and Table VI respectively. Contribution value of class (CVC) for all classes in System Diagram and Design Pattern Diagram are as shown in Table II.

The Correspondence Graph is shown in Table VII, in which value CGM[i,j] = 1, indicates the candidateship of system graph class 'j' for pattern class 'i'. After applying filtering algorithm, the updated Correspondence Graph is shown in Table VIII.

From the updated Correspondence Graph, bijective matching are found for further evaluation of the pattern existence in the 4-subgraph extracted from the Connectivity Graph Matrix of System Diagram. Let the bijective matching be $(PGC1 \rightarrow SGC1, PGC2 \rightarrow SGC8, PGC3 \rightarrow SGC3, PGC4 \rightarrow SGC5)$. The 4-subgraph of Connectivity Graph Matrix of System Diagram containing SGC1, SGC8, SGC3, SGC5 becomes the same as the Connectivity Graph Matrix of Design Pattern Diagram as shown in Table IX. Hence, the 4-subgraph of

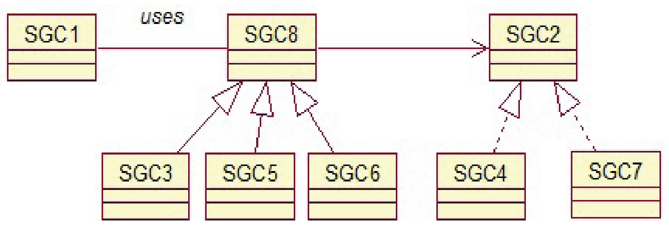


Fig. 3: UML class diagram for System Diagram under study

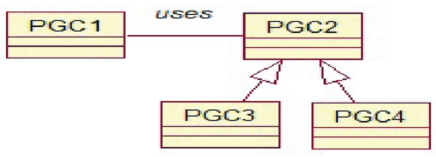


Fig. 4: UML class diagram for Template design pattern

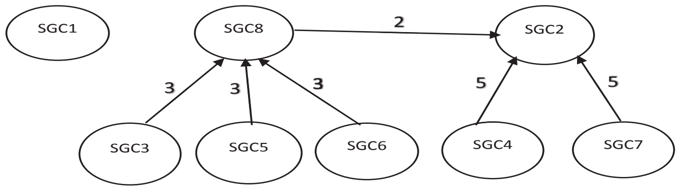


Fig. 5: System Graph corresponding to System Class Diagram

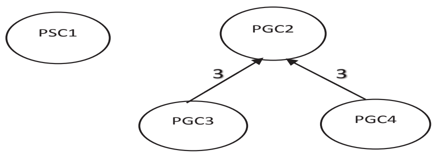


Fig. 6: Design Pattern Graph corresponding to Template Design Pattern Class Diagram

TABLE II: CVC for classes in System Diagram and Design Pattern Diagram

Class	**CVC**	**Class**	**CVC**	**Class**	**CVC**
SGC1	1	SGC2	1	SGC3	3
SGC4	5	SGC5	3	SGC6	3
SGC7	5	SGC8	2	PGC1	1
PGC2	1	PGC3	3	PGC4	3

TABLE III: System Graph Matrix (SGM)

SGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	3
4	1	5	1	1	1	1	1	1
5	1	1	1	1	1	1	1	3
6	1	1	1	1	1	1	1	3
7	1	5	1	1	1	1	1	1
8	1	2	1	1	1	1	1	1

TABLE IV: Design Pattern Matrix (DPM)

PGC \ PGC	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	3	1	1
4	1	3	1	1

TABLE V: Connectivity Graph Matrix for System Graph (CGM_s)

SGC \ SGC	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	1
7	0	1	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0

TABLE VI: Connectivity Graph Matrix for Pattern Graph(CGM_d)

PGC \ PGC	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	1	0	0
4	0	1	0	0

TABLE VII: Correspondence Graph(CRG)

PGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	0	0	1	0	1	1	0	0
4	0	0	1	0	1	1	0	0

TABLE VIII: Updated Correspondence Graph(CRG) after Applying Candidate Filtering Algorithm

PGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	0
2	0	0	0	0	0	0	0	1
3	0	0	1	0	1	1	0	0
4	0	0	1	0	1	1	0	0

System Graph Matrix containing SGC1, SGC8, SGC3, SGC5 as nodes, is extracted from System Graph Matrix, which is shown in Table IX. The Normalized Cross Correlation is applied on the matrix shown in Table X with the design pattern graph matrix (DPM), shown in Table IV.

$$NCC = \frac{\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} SGM_4[i,j] * DPM[i,j] - P_n^2 * \mu_s * \mu_d}{\sqrt{(}\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} SGM_4[i,j]^2 - P_n^2 * \mu_s^2) * (\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} DPM[i,j]^2 - P_n^2 * \mu_d^2)} \tag{2}$$

where,

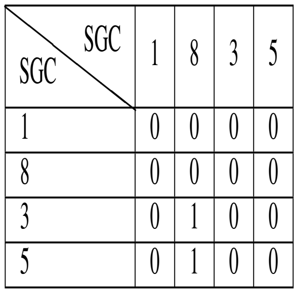
$$\mu_s = \frac{1}{P_n^2}\sum_{i=1}^{P_n} SGM_4[i,j] \tag{3}$$

$$\mu_d = \frac{1}{P_n^2}\sum_{i=1}^{P_n} DPM4[i,j] \tag{4}$$

If NCC value becomes 1 (equation 2), it assures 100% or full occurrence of template design pattern in the system graph.

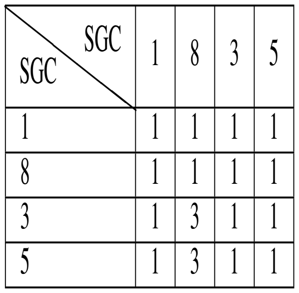
Taking another bijective matching ($PGC1 \rightarrow SGC3, PGC2 \rightarrow SGC8, PGC3 \rightarrow SGC5, PGC4 \rightarrow SGC6$), the 4-subgraph of Connectivity Graph Matrix of System Diagram containing SGC3, SGC8, SGC5, SGC6 as nodes have been extracted as shown in Table XI and it does not become the same as the Connectivity Graph Matrix of Design Pattern Diagram as shown in Table VI . Hence, this set of classes is discarded for further evaluation.

TABLE IX: 4-subgraph of CRG (CRG_4) - Testcase1



SGC \ SGC	1	8	3	5
1	0	0	0	0
8	0	0	0	0
3	0	1	0	0
5	0	1	0	0

TABLE X: 4-subgraph of SGM (SGM_4) - Testcase1



SGC \ SGC	1	8	3	5
1	1	1	1	1
8	1	1	1	1
3	1	3	1	1
5	1	3	1	1

TABLE XI: 4-subgraph of CRG (CRG_4) - Testcase2

SGC \ SGC	3	8	5	6
3	0	1	0	0
8	0	0	0	0
5	0	1	0	0
6	0	1	0	0

TABLE XIII: Implementation Results

Sl. No.	Design Pattern	Jrat 100%	$Jrat \geq 90\%$	Junit 100%	$Junit \geq 90\%$	Lexi-Alpha 100%	$Lexi - Alpha \geq 90\%$	Informa 100%	$Informa \geq 90\%$
1.	Composite	71	0	0	0	182	0	107	0
2.	Facade	124	705	564	1608	0	0	0	0
3.	Flyweight	600	8409	707	5439	581	1595	201	4759
4.	State	124	14	564	492	0	0	0	0
5.	Template Method	286	4	6098	977	522	6	4422	866

TABLE XII: Size of Open Source Java Tools

Sl. No.	Java Tools	No. of Classes	No. of Interfaces	Total classes and interfaces
1.	Jrat	60	16	76
2.	Junit	84	10	94
3.	Lexi-Alpha	95	1	96
4.	Informa	63	46	109

D. Evaluation

The proposed approach is applied on four widely adopted open source softwares having classes ranging from 76 to 109. Various tools such as Jrat, Junit, Lexi-alpha and Informa have been applied for evaluation of the existence of 5 design patterns, such as Composite, Facade, Flyweight, State, and Template Method design patterns. Table XII shows the total number of classes and interfaces in the open source Java tools. Results for 100% existence and partial existence above 90% are shown in Table XIII.

IV. Conclusion and Future Work

This study provides an approach to detect design patterns in the source code of a software. In the field of Software Reverse Engineering, this approach helps to detect design pattern instances in a software, as it automatically detects design patterns. The use of Normalized Cross Correlation method in the process of design pattern detection provides a way not only to detect full occurrence of the pattern but it also provides a measure to find the percentage matching of the pattern. The Graph Isomorphism technique is useful to detect design patterns in order to enhance usability and reliability of the proposed technique. This method is useful for software engineers to get knowledge about the pattern existence in the system.

As a future work, this approach can be applicable for other design patterns and similar evaluation can be done for various open source softwares.

References

[1] A. K. Dwivedi and S. K. Rath, "Formalization of web security patterns," *INFOCOMP Journal of Computer Science*, vol. 14, no. 1, pp. 14–25, 2015.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] A. K. Dwivedi and S. K. Rath, "Selecting and formalizing an architectural style: A comparative study," in *Contemporary Computing (IC3), 2014 Seventh International Conference on*, Aug 2014, pp. 364–369.

[4] H. Zhu and I. Bayley, "An algebra of design patterns," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 23:1–23:35, Jul. 2013.

[5] A. K. Dwivedi and S. K. Rath, "Analysis of a complex architectural style C2 using modeling language Alloy," *Computer Science and Information Technology Journal*, vol. 2, no. 3, pp. 152–164, 2014.

[6] G. Scanniello, C. Gravino, M. Risi, G. Tortora, and G. Dodero, "Documenting design-pattern instances: A family of experiments on source-code comprehensibility," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 3, p. 14, 2015.

[7] A. K. Dwivedi and S. K. Rath, "Incorporating security features in service-oriented architecture using security patterns," *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–6, 2015.

[8] C. Gerardo, M. D. Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, 2011.

[9] F. A. Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Information Sciences*, vol. 181, p. 13061324, 2011.

[10] D. Yu, Y. Zhang, and Z. Chen, "A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures," *The Journal of Systems and Software*, vol. 103, p. 116, 2015.

[11] H. Huang, S. Zhang, J. Cao, and Y. Duan, "A practical pattern recovery approach based on both structural and behavioral analysis," *Journal of Systems and Software*, vol. 75, no. 1, pp. 69–87, 2005.

[12] "Eclipse Plug-in ObjectAid," http://www.objectaid.com/.

[13] M. Gupta, A. Pande, R. Singh Rao, and A. Tripathi, "Design pattern detection by normalized cross correlation," in *Methods and Models in Computer Science (ICM2CS), 2010 International Conference on*. IEEE, 2010, pp. 81–84.

[14] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[15] A. S. Ba-Brahem and M. Qureshi, "The proposal of improved inexact isomorphic graph algorithm to detect design patterns," *arXiv preprint arXiv:1408.6147*, 2014.

[16] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *Software Engineering, IEEE Transactions on*, vol. 32, no. 11, pp. 896–909, 2006.

[17] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 765–769.

[18] Y. Dongjin, J. Ge, and W. Wu, "Detection of design pattern instances based on graph isomorphism," in *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*. IEEE, 2013, pp. 874–877.

[19] H. Albin-Amiot and Y.-G. Guéhéneuc, "Meta-modeling design patterns: Application to pattern detection and code synthesis," in *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[20] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 2003, pp. 94–103.

[21] F. Bergenti and A. Poggi, "Improving UML designs using automatic design pattern detection," in *12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Citeseer, 2000, pp. 336–343.

[22] "AgroUML," http://argouml.tigris.org/.

[23] M. Gupta, R. Singh Rao, and A. K. Tripathi, "Design pattern detection using inexact graph matching," in *Communication and Computational Intelligence (INCOCCI), 2010 International Conference on*. IEEE, 2010, pp. 211–217.

[24] S. Wenzel and U. Kelter, "Model-driven design pattern detection using difference calculation," in *1st Int. Workshop on Pattern Detection for Reverse Engineering, Co-located with 13th Working Conf. on Reverse Engineering*, 2006.