# Mining Object-Oriented Design Models for Detecting Identical Design Structures

Umut Tekin, Ural Erdemir
*Center of Research for Advanced Technologies of Informatics & Information Security*
*Kocaeli, Turkey*
*{umuttekin,ural}@bilgem.tubitak.gov.tr*

Feza Buzluca
*Computer Engineering Department*
*Istanbul Technical University*
*Istanbul, Turkey*
*buzluca@itu.edu.tr*

***Abstract*—The object-oriented design is the most popular design methodology of the last twenty-five years. Several design patterns and principles are defined to improve the design quality of object-oriented software systems. In addition, designers can use unique design motifs which are particular for the specific application domain. Another common habit is cloning and modifying some parts of the software while creating new modules. Therefore, object-oriented programs can include many identical design structures. This work proposes a sub-graph mining based approach to detect identical design structures in object-oriented systems. By identifying and analyzing these structures, we can obtain useful information about the design, such as commonly-used design patterns, most frequent design defects, domain-specific patterns, and design clones, which may help developers to improve their knowledge about the software architecture. Furthermore, problematic parts of frequent identical design structures are the appropriate refactoring opportunities because they affect multiple areas of the architecture. Experiments with several open-source projects show that we can successfully find many identical design structures in each project. We observe that usually most of the identical structures are an implementation of common design patterns; however we also detect various anti-patterns, domain-specific patterns, and design-level clones.**

***Keywords*-software design models; identical design structures; software motifs; pattern extraction; graph mining; clones**

## I. INTRODUCTION

Many software projects contain a significant number of software clones [1] those are duplicated parts of source code or design models. One reason of design-level cloning is the frequent usage of software design principles and design patterns (e.g. GRASP [2], GoF [3]). Besides, there are also domain specific patterns [4] that are optimal for a particular application or a specific problem, so that they are used repeatedly in a project. In addition, there are also unfavorable sources of clones such as anti-patterns and common design defects. Consequently, a software system can include many identical parts at the design level. In the article 'Draw Me a Picture' [5], Booch pointed out that those hidden patterns of software are crucial to understand software architecture and assess its quality. Understanding of software architecture also plays a key role in refactoring processes, which constitute the reactive part of maintenance tasks.

Refactoring improves the internal design structure of software by preventing production of poor quality products. More than half of the development effort is spent on maintenance process [6]. Generally frequent identical design structures are the most reusable parts of the design which can be good candidates to be used in future designs. Also identifying identical design structures may provide significant advantage in reducing the cost of maintenance because some of the most commonly-used structures in software design are the best places to look for refactoring opportunities as they affect multiple parts of the design. For example, non-standard structures that are similar to design patterns might be modified to conform to standard forms, or common design defects can be quickly identified to fixing them in multiple areas at once.

In this paper, we propose a graph mining-based approach to detect identical parts in object-oriented software architecture. The proposed approach contains three main steps. In the first step, the AST of the source code of the systems is analyzed and UML-based design level of abstraction is created. In the next step, we apply graph partitioning algorithm to divide the software model graph into small pieces. Finally, in the last step, sub-graph mining algorithms are applied to discover identical design structures in the generated software model. As the scope of our study is primarily focused on the identification stage, analyzing and classifying these structures could be an interesting topic for future studies. We also present several interesting examples of our findings that may inspire future studies.

The rest of the paper is organized as follows: Background and related work are presented in the next section. Graph representation and definitions are given in Section III. Identification process is detailed in Section IV. In Section V, results obtained from example projects are presented and the last section concludes the paper.

## II. RELATED WORK

There are many graph-based design pattern detection approaches proposed in the literature [7][8], but most of them heavily depend on pre-known pattern templates. However, during the development of software the way of using the patterns may vary according to the particular programming language or target environment. In contrast to the previous pattern detection studies, our approach does not require any pre-known patterns and we can discover any

kind of identical design structures even though designers are not aware of it.

Koschke provides a survey about the studies in the clone detection area that try to find identical parts of software [9]. At design level, there is a suffix-tree comparison- based clone detection study [10] that tries to find identical parts of UML sequence diagrams and there exists another pattern matching based study [11] that works on XMI tree structure of UML's domain models. However in most cases, well-structured UML documentation of software is absent, incomplete or outdated [12].

There are also studies that use frequent sub-graph mining algorithms for clone detection in Matlab/Simulink models [13] [14] and low-level program dependency graphs [15]. Mathlab/Simulink models are electrical circuit models where there exists only a single type of relation (electrical link) between entities. They have slightly small and loosely connected graph representations compared to graph representations of software designs that require excessive long running time for analysis. In our study, we deal with highly connected large graphs where there are different and multiple types of relations between entities (classes and interfaces). Most of the clone detection techniques usually work on low-level attributes of programs like source code, flow diagrams and variable dependency graphs [9]. We target high-level design model of software, because detecting design clones can help in comprehension and maintenance of software and it can provide reusability of commonly used design structures.

In another high-level similarity detection study [16], simple source-level clones are grouped according to their appearance in source files and then data mining techniques applied on these groups to detect high-level identical design structures. In our study, we do not depend on the existence of source-level clones, because there can be a structural design similarity in a program even if it does not include any source-level clones. For example, if programmers change the names or types of some attributes in the copied source code, the design structure will still remain the same.

Currently, most relevant study to our work is another graph mining approach that focuses on detection of reused micro-architectures in design model of software through its different versions [17]. In this study, the authors are able to find reused identical structure sets including up to five nodes. The main difference of our work is that we assign same labels to each class and interface (i.e. "C" for classes, "I" for interfaces, "A" for abstract classes…), but they assign unique integer numbers to each class or interface in their graph model. That property of our graph model makes it possible to investigate identical structures that are inside the same software, while in [17], they investigate reused class groups between different versions of a software system. In our graph model we also consider several high level relations that they do not. Another advantage of our approach is that we are able to find identical structures that consist of up to fifteen nodes.

To the best of our knowledge, there is not any published tool that combines graph partitioning and mining algorithms for detection of identical structures in software designs.

## III. Graph Representation of Software Design and Definitions

In this section, we explain the constructed software model graph and give some definitions.

We represent a high level view of software architecture as a simple directed and labeled graph. The vertices of this graph are classes, abstract classes, template classes and interfaces. The edges of the graph represent directed relation between these entities (vertices).

Relation types in the software model graph are based on UML-like [18] relations. At this point, we especially consider class and sequence diagrams of the UML. Moreover, we handle some important relations that are visually hidden in the UML diagrams. For example if a method of a class has the same signature with a method of the parent class then there is a "override" relation between these classes that is not visually observable in UML class diagrams. We also include some important high-level relations from UML sequence diagrams like "create" and "method call" relations between entities. Possible node types, relation types and their labels are given in Table I.

TABLE I. Node and Edge Labels

| **Node Label** | **Node Type** |
|---|---|
| C | Class |
| I | Interface |
| A | Abstract Class |
| T | Template Class |
| **Edge Label** | **Relation Type (Edges are directed from A to B)** |
| X | Class A extends Class B |
| I | Class A implements Class B |
| A | Class A has field type of Class B |
| T | Class A uses Class B in generic type declaration |
| L | Class A methods has a local parameter of Class B |
| P | Class A uses Class B in its methods parameter |
| R | Class A has methods has return type of Class B |
| M | Class A has method call to Class B |
| F | Class A access fields of Class B |
| C | Class A creates Class B |
| O | Class A overrides methods of Class B |

By the nature of the object-oriented design, there can be more than one relation between entities (classes and interfaces). To create a simple and understandable graph, we merge all parallel edges into single edges by combining their labels with pre-defined deterministic order. For example, if two entities have both extend (X) and method call (M) relation, then the combined label for this relation becomes (X) + (M) = (XM), if the extend relation (X) comes first in the pre-defined order. We use the pre-defined order to ensure that the combination of the same relations gets always the same label, as in the example above, we always have the label (XM) and not (MX). In our approach to detect design-

level clones, we take the complete label match into consideration.

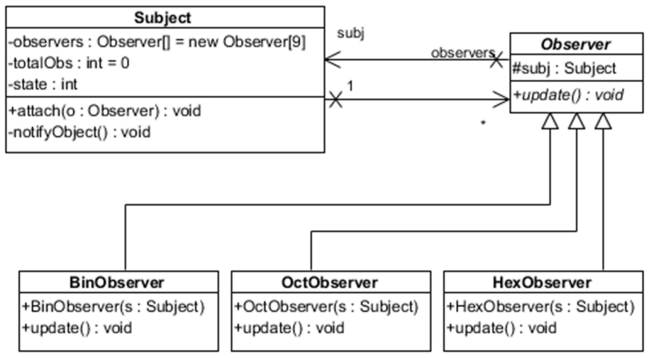


Figure 1. Class Diagram of Simple Observer Example

Fig. 1 shows the UML class diagram of an observer design pattern example, and Fig. 2 represents the related software graph model that we constructed. In Fig. 2, we can see that the software model graph includes additional information compared to UML class diagram, such as method call (M) and methods parameter (P) relations.

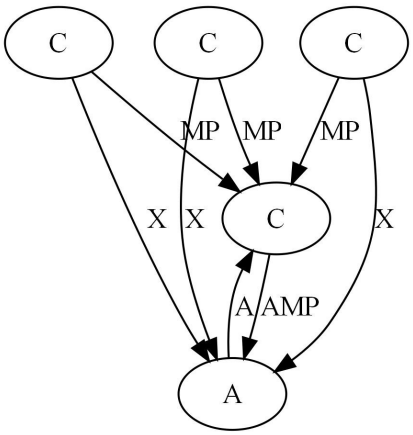


Figure 2. Software Model Graph of the Observer Pattern

Here we give some definitions from graph theory which we use in next sections.

**Graph**: Let G = (V, E, Lv, Le, v, e) be a directed software model graph where;
V is a set of vertices,
E $\subseteq$ VxV is a set of edges,
Lv is a set of labels for the vertices,
Le is a set of labels for the edges,
v: V→Lv is a function which assigns a label to the vertices,
e: E→Le is a function which assigns a label to the edges.
**Sub-graph**: A graph Gs is defined as a sub-graph of G, denoted as Gs $\subseteq$ G, if the vertices and edges of Gs form a subset of the vertices and edges of G.
**Isomorphic sub-graph:** Two different sub-graphs G1 (V1, E1) and G2 (V2, E2) are isomorphic if they are topologically identical to each other (one-to-one mapping of every vertices and edges). This mapping must also preserve the labels on the vertices and edges and also the direction of edges.
**Frequent isomorphic sub-graph**: A graph dataset is a non-empty finite set of labeled connected directed graphs. Given a graph dataset M, a frequent isomorphic sub-graph is a sub-graph whose matching isomorphic sub-graphs together are frequent in graph set of M. If a graph is frequent; also all of its sub-graphs are frequent. The frequency value is determined by the number of global graphs that the sub-graph occurs in.
**Closed frequent sub-graph:** Closed frequent sub-graph G is a graph that there is no frequent super-graph of G with same frequency value [21].

## IV. IDENTICAL STRUCTURE MINING

In this section, we detail our mining approach with the primary purpose of finding identical design structures that are globally distributed in the model graph of a large software system. Fig. 3 shows the main steps of our scheme.

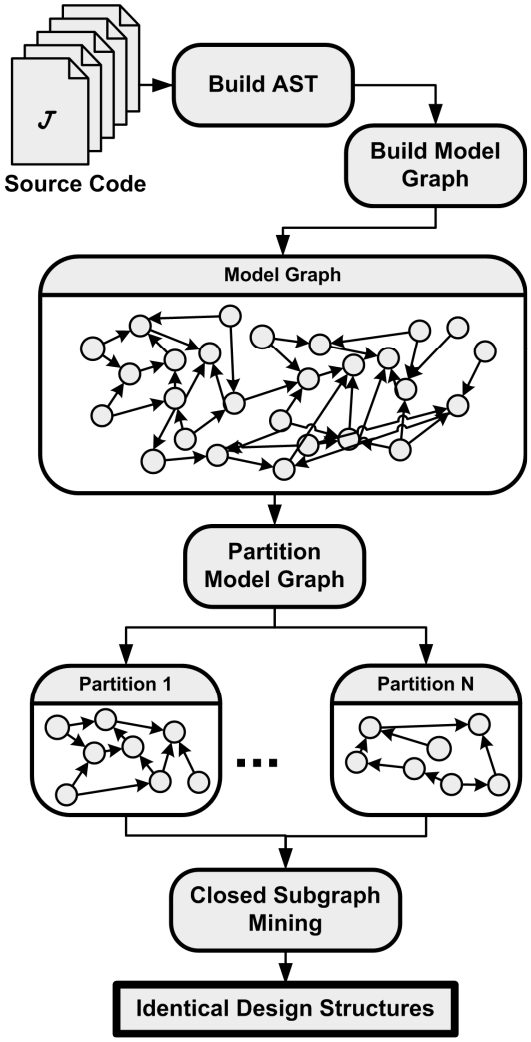


Figure 3. Basic flow of the approach

TABLE II. SUB-GRAPH MINING RESULTS (FREQUENCY VALUE >= 2 VERTEXCOUNT > 2)

| | Yari | Zest | JUnit | EQuality | JFreeChart | ArgoUML |
|---|---|---|---|---|---|---|
| **Number of Vertices of SMG** | 69 | 144 | 271 | 281 | 614 | 1571 |
| **Number of Edges of SMG** | 148 | 390 | 1046 | 939 | 2409 | 6509 |
| **Number of Graph Partitions** | 3 | 8 | 16 | 16 | 32 | 96 |
| **Maximum Vertex Count in Partitions** | 21 | 16 | 16 | 17 | 19 | 16 |
| **Number of Identical Structure** | 15 | 37 | 64 | 38 | 299 | 726 |
| **Run Time in Seconds** | 0.8 | 0.744 | 71.678 | 16.493 | 1069.285 | 13055.736 |

TABLE III. NUMBER OF IDENTICAL STRUCTURES FOR DIFFERENT FREQUENCY VALUES (VERTEXCOUNT > 2)

| Frequency Value | Yari | Zest | JUnit | EQuality | JFreeChart | ArgoUML | Run Time in Sec. (for ArgoUML) |
|---|---|---|---|---|---|---|---|
| **= 2** | 14 | 30 | 41 | 26 | 201 | 369 | **11723.324** |
| **= 3** | 1 | 6 | 14 | 9 | 56 | 164 | **3305.544** |
| **= 4** | NA | 1 | 5 | 3 | 26 | 88 | **2050.724** |
| **= 5** | NA | 0 | 3 | 0 | 6 | 41 | **1352.869** |
| **= 6** | NA | 0 | 1 | 0 | 4 | 27 | **750.491** |
| **= 7** | NA | 0 | 0 | 0 | 3 | 15 | **154.934** |
| **= 8** | NA | 0 | 0 | 0 | 2 | 10 | **82.528** |
| **= 9** | NA | NA | 0 | 0 | 1 | 7 | **43.507** |
| **= 10** | NA | NA | 0 | 0 | 0 | 4 | **8.390** |
| **= 11** | NA | NA | 0 | 0 | 0 | 1 | **3.112** |

In the first step, we extract software entities (i.e., classes and interfaces) and relations directly from Java source code by parsing the abstract syntax trees. Then, the model builder processes the relations and generates a complete graph model of the software architecture.

Running time of sub-graph mining algorithms dramatically increases dependent on the |V|. Therefore, we apply divide and conquer strategy that is based on the idea to partition the input graph into smaller n sub-graph, and to find frequent identical structures within these "n" partitions. We apply multi-level spectral graph partitioning algorithm [19] included in Chaco [20] graph partitioning library to create several graph partitions (sub-graphs). The produced partitions consist of strongly connected entities and we lose minimum number of edges, because the algorithm relies on minimum cost (edge cut) graph bisection method. The graph partitioning problem is a NP-complete problem; therefore the selected algorithm uses fast heuristics to effectively solve the problem. In our test cases, it produces partitions just in a couple of seconds at most. We discuss pros and cons of partitioning in the discussion section.

In the last step, we apply the CloseGraph [21] complete graph mining algorithm to discover all frequent isomorphic sub-graphs within those partitions. The CloseGraph is capable of finding all closed frequent isomorphic graphs and it prunes the search space by eliminating many repetitive structures. Closed frequent isomorphic sub-graphs are important because they are the embracing isomorphic sub-graphs that we want to discover. As the problem of sub-graph isomorphism is NP-complete and the CloseGraph algorithm produces complete output, this is the most time and resource consuming stage of the detection phase. We use a ParSeMiS [22] implementation of CloseGraph algorithm.

Current complete frequent sub-graph mining algorithms are especially designed to mine patterns in small and loosely connected graphs such as chemical molecules. Their running time and resource consumption directly depend on the number of vertices and edges in the sub-graphs. As the software graphs are highly connected [23] to run algorithms efficiently, the sizes of partitions must be selected properly. In our test cases, we are able to find identical structures in partitions with the size of up to 20 vertices in a reasonable running time with reasonable memory consumptions. For larger partitions, memory consumption becomes a bottleneck and this dramatically increases the running time because of the graph explosion problem (n-edge frequent graph may have $2^n$ sub-graphs). We do not cover sub-graph mining performance issues in the present study but there are studies in the literature [24] that addresses the performance issue.

## V. EXPERIMENTS AND RESULTS

This section presents the results obtained from our experiments. Five open-source projects with different sizes (Yari [25], ZestCore [26], JUnit[27], JFreeChart[28], ArgoUML[29]) and our software analysis tool (E-Quality [30]) are studied to evaluate our approach. YARI is a small tool suite to inspect Eclipse based application GUIs; Zest is an open source project that includes visualization components for Eclipse; JFreeChart is a chart drawing library, JUnit is a testing framework and ArgoUML is an UML modeling tool. EQuality is a software visualization and analysis tool for object-orient system that is developed by our research group. We run our experiments on a 4-CPU Linux system with 32 GB of RAM.

In Table 2, we present for each project, properties of produced software model graph (SMG), partition size, maximum vertex count in each partition, running-time, and number of discovered identical structures which appear at least in two different partitions (frequency >= 2). The number of vertices in these identical structures (classes and interface) varies from three to fifteen. The structures with

one and two vertices are filtered because they are too small to address. As it can be seen from the Table 2, the running time dramatically increases with number of partitions and during the analysis of ArgoUML project memory usage increases up to 25 GB.

In Table 3, we also present number of detected identical structures with certain frequency values for different projects. The number of partitions in the graph dataset determines the maximum frequency value: for example in Yari project maximum possible frequency value is three. As frequency value increases the number of exact matches' decreases. We also present performance results dependent on frequency values for ArgoUML project that is the largest example we analyze. The performance of closeGraph algorithm dramatically increases with higher frequency values because these types of algorithms are especially designed to detect high frequent sub-graphs in protein or chemical networks.

We briefly explain some of the identical design structures discovered in different projects. Figure 4 shows two of the most common design structures that we identified in our test projects. These types of design structures that consist of basic "extend" or "method call" relations appear in almost every project and their frequency is generally more than two. Many times these structures are ignorable and uninteresting, because they are common structures of object-oriented programming.

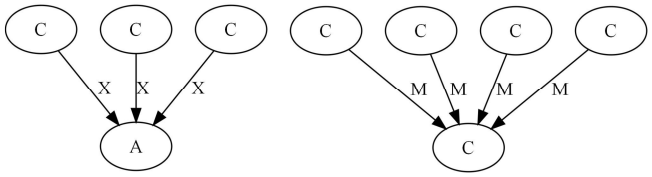


Figure 4. Common Design Structures

Figure 5 shows two examples of identical structures that we found in the JUnit project. These structures are the examples of circular dependency anti-pattern that are considered bad or harmful [31]. As these structures reduce the reusability and the manageability of software, they must be refactored. The design structure on the right side has frequency value of four and the left one has the value of two.

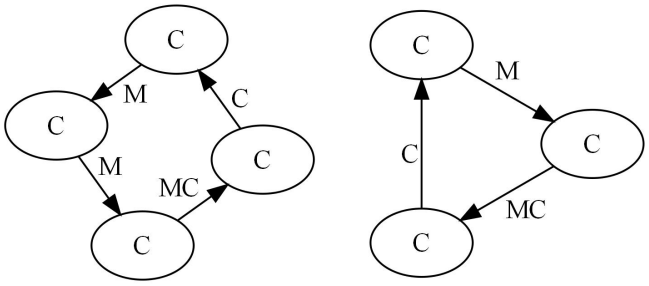


Figure 5. Circular dependency structures in Junit

Figure 6 shows a structure, whose copies we found in different parts of the ArgoUML project. It is a typical factory-like pattern implementation. The class located on the top in the figure creates objects of related types. "*ProfileGoodPractices*" and "*ToDoPane*" classes in ArgoUML are two examples of such classes. There are also several other small factory-like structures discovered in the investigated projects that are very similar to the structure in Fig. 6 with higher frequency values then two because factory like structures are frequently used in object-oriented projects.

In our software analysis tool E-Quality, we found copies of a structure that contains a non-standard implementation of the visitor pattern as shown in Fig. 7 with frequency value two. We use this pattern to walk through the software model graph in two places for different purposes with same design philosophy. Also in E-Quality, we found identical structure with frequency value of there that is a copy-paste clone of a database access module that contains three classes as show in Fig. 8. We are planning to modify the structure in Fig. 7 to a standard visitor form and to remove multiple copies of structures shown in Fig. 8 to improve the design quality of our program.

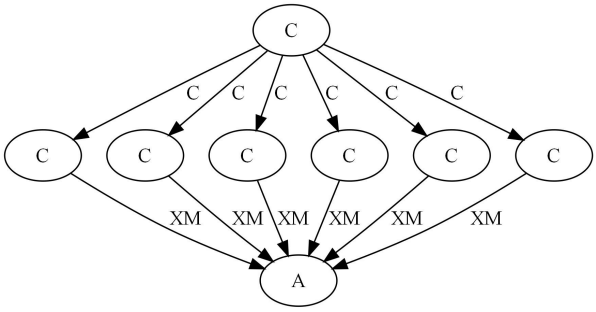


Figure 6. Factory-like Structure

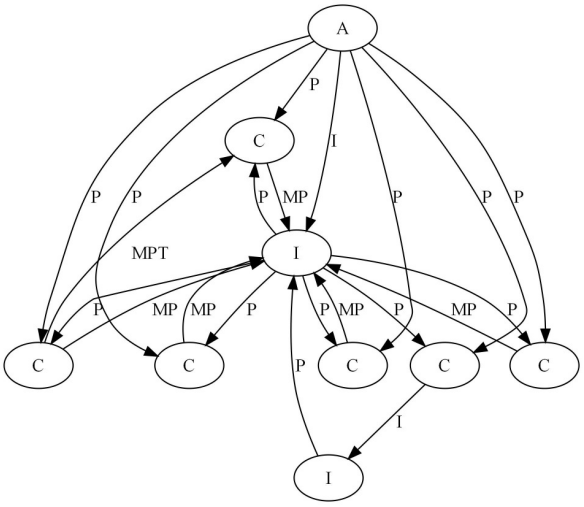


Figure 7. Visitor Pattern

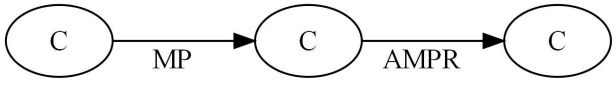


Figure 8. Copy-Paste Structure

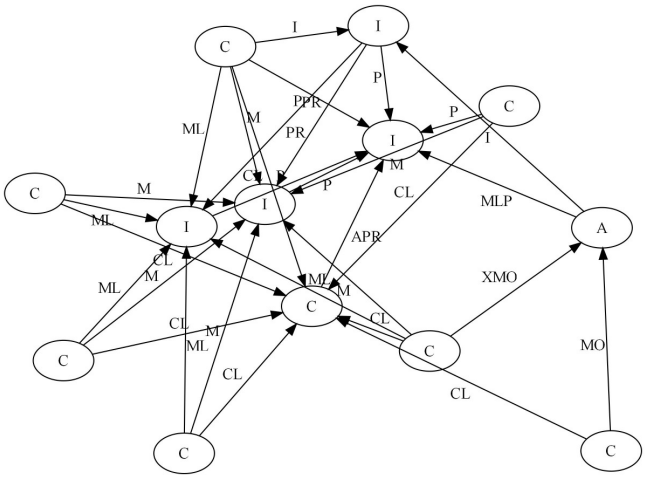


Figure 9. The Complex Software Clone in the JFreeChart

One of the most complex structural matches that contains thirteen classes is discovered in the JFreeChart project. One instance of the design structure shown in Fig. 9 takes place in the "*org.jfree.chart.renderer.category*" and the other one in the "*org.jfree.chart.renderer.xy*" packages. These structures include groups of classes with same behaviors for similar type of work. However it is not a copy-paste clone: it could be classified as a domain specific match.

We discovered also some identical design structures that cannot be classified as design or domain specific pattern. Figure 10 shows an example identical structure from JUnit that might be considered just a simple design clone with frequency value of three.

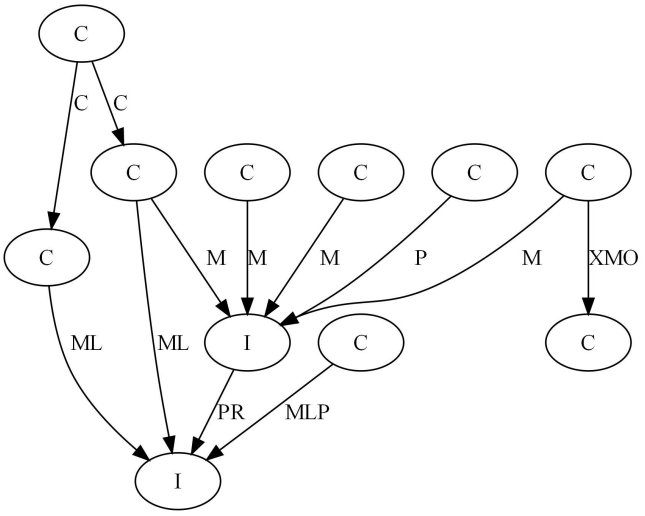


Figure 10. Design Clone in JUnit

Several identical design structures have been identified from simple to complex ones. Further analysis on these structures can reveal some useful insights about their quality and designers of them can suggest several useful improvements about these structures to make them more modular, reliable and reusable.

## VI. DISCUSSION

Graph partitioning plays a critical role in increasing the speed of our approach. Also without partition the whole graph it is not possible to investigate design-level clones in same software project. However, partitioning could prevent identifying some interesting sub-graphs that would be spanning between two or more partitions because we lose some weak relations during the partitioning. But we assume that the entities which are strongly connected (highly-cohesive) must be the most interesting reused software modules. For example in Java-based projects developers usually put highly related classes in same packages. The graph partitioning algorithms also try to partition graphs into strongly connected partitions by removing weak relations as much as possible. There are also several clustering and partition approach that try to find strongly connected software entities as a modules or services [32]. There are some heuristic algorithms such as [33] to mine large graphs without partitioning but currently they do not operate on directed graphs and also they do not produce complete outputs. In addition, we observe that mining big graphs or partitions with small frequency values is a computationally very expensive process.

Existing clone detection tools can also help in finding some types of similarities in a software system such as type-1 and type-2 clones [9]. But in our work we focus on design (model) similarities and define exact design match as a condition. In our tests some of the copy-paste structures such as Fig 8. were detected by typical source code based clone detection techniques and tools like CCFinder[34] and CodePro Analytix [35], however some of the design matches such as Fig 6, Fig 7 and Fig 9. could not be detected by them in our tests.

Benefits of finding design matches could help architects in evaluating reused design patterns. For example in our software analysis tool we detect a custom implementation of visitor pattern that is not in a standard form. We plan to modify this pattern to the standard form to increase our design quality. We also found some repetitive design defects that must be fixed such as circular dependencies. Each structure that we found occurs at least in two different places of whole architecture so quality improvement activities on them will affect multiple areas of architecture at once. We believe that most of the interesting matches are the domain specific matches that can only be make sense to its designer. Theoretically it is also possible to explore currently unnamed design patterns by applying our approach to very large set of object-oriented software projects.

## VII. CONCLUSION

In this paper we proposed a systematic sub-graph mining based approach to detect identical design structures in object-oriented software. By analyzing several open-source projects, we evaluated our approach and discussed the results. The results show that we are able to detect many identical structures that can be manually classified as software clones, common design patterns, domain specific patterns, or repeated design disharmonies. Identifying these design structures can help designers in understanding the

high-level architecture of the object-oriented software, discovering reusability possibilities and detecting repeated design flaws that need refactoring. We plan to extend our approach by analyzing partitioning effect and adding the capability of automatic classification of the detected identical design structures.

## REFERENCES


[1] B. S. Baker, "On finding duplication and near duplication in large software systems," in WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering. Washington, DC, USA: IEEE Computer Society, 1995, pp. 86+.

[2] Larman C.: Applying UML and Patterns. Prentice-Hall International, 2001.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, 1st ed. Addison-Wesley, 1994.

[4] Port, D. Derivation of Domain Specific Design Patterns. USC Center for software engineering, 1998.

[5] Grady Booch, "Draw Me a Picture," In IEEE Software 28(1): 6-7 (2011)

[6] Hanna, M., "Maintenance Burden Begging for a Remedy," Software Magazine, April 1993.

[7] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., et al.: "Design Pattern Detection using Similarity Scoring," IEEE Transactions on Software Engineering 32(11), 896–909 (2006)

[8] Dong, J., Sun, Y., Zhao, Y.: "Design Pattern Detection by Template Matching," In: Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Brazil, pp. 765–769 (2008)

[9] Rainer Koschke. "Survey of Research on Software Clones," In Proceedings of DagstuhlSeminar 06301: Duplication, Redundancy, and Similarity in Software, 24pp., Dagstuhl, Germany, July 2006

[10] Liu, H., Ma, Z., Zhang, L., and Shao, W. "Detecting duplications in sequence diagrams based on suffix trees," In 13th Asia Paci_c Software Engineering Conf. (APSEC) (2006), IEEE CS

[11] Harald Störrle, "Towards Clone Detection in UML Domain Models,", Nordic Workshop on Model Driven Software Engineering (NW-MODE) – 2010

[12] Christian F.J. Lange and Michel R.V. Chaudron, "In Practice: UML Software Architecture and Design Description," Eindhoven University of Technology Johan Muskens, Philips Research

[13] Deissenboeck, F., Hummel, B., Schaetz, B., Wagner, S., Girard, J., and Teuchert, S. "Clone Detection in Automotive Model-Based Development," In Proc. IEEE 30th Intl. Conf. Software Engineering (ICSE) (2008), IEEE Computer Society, pp. 603-612.

[14] Nguyen, H., Nguyen, T., Pham, N., Al-Kofahi, J., and Nguyen, T. "Accurate and efficient structural characteristic feature extraction for clone detection," In Proc. 12th Intl. Conf. Fundamental Approaches to Software Engineering (FASE) (2009), Springer, pp. 440-455.

[15] R. Komondoor and S. Horwitz. "Using slicing to identify duplication in source code," In SAS, 2001.

[16] Hamid Abdul Basit and Stan Jarzabek "Detecting Higher-level Similarity Patterns in Programs," ESEC-FSE'05, European Software Engineering Conference, Sept. 2005, Lisbon

[17] Ahmed Belderrar, Segla Kpodjedo, Yann-Gael Gueheneuc, Giuliano Antoniol, Philippe Galinier, "Sub-graph Mining: Identifying Micro-architectures in Evolving Object-oriented Software," In CSMR, 2011

[18] UML: Unified modeling language. (http://www.uml.org)

[19] B. Hendrickson, R. Leland, "A multi-level algorithm for partitioning graphs," in: Proceedings Supercomputing'95, ACM Press, 1995, p. 28

[20] B. Hendrickson and R. Leland, The Chaco User's Guide, Version 1.0, Tech. rep. SAND93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.

[21] X. Yan and J. Han. "Closegraph: Mining closed frequent graph patterns.," In KDD'03, Washington, D.C, 2003. ACM Press.

[22] Philippsen, M., et al.: ParSeMiS: The Parallel and Sequential Mining Suite, available at http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/

[23] Sergi Valverde and Ricard V. Solé, "Hierarchical Small Worlds in Software Architecture," Submitted to IEEE Transactions on Software Engineering, 2007

[24] Marc Worlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. "A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston," In Proc. Conference on Knowledge Discovery in Database (PKDD'05), Lecture Notes in Computer Science, pages 392–403, Porto, Portugal, October 2005. Springer-Verlag.

[25] YARI: http://sourcefourge.net/projects/yari

[26] Zest: http://www.eclipse.org/gef/zest/

[27] JUnit: http://www.junit.org/

[28] JFreeChart: http://www.jfree.org/jfreechart

[29] ArgoUML: http://argouml.tigris.org/

[30] U. Erdemir, U. Tekin and F. Buzluca, "E-Quality: A Graph Based Object Oriented Software Quality Visualization Tool". In Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), pp..6–13, Virginia, USA Sep. 2011.

[31] William J. Brown, Raphael C. Malveau, and Hays W. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, 1998.

[32] R. A. Bittencourt and D. D. S. Guerrero "Comparison of Graph Clustering Algorithms for Recovering Software Architecture Module Views", In Proc of European Conference on Software Maintenance and Reengineering, IEEE CS Press, 2009, pp. 251-254.

[33] Michihiro Kuramochi, George Karypis: "Finding Frequent Patterns in a Large Sparse Graph". Data Min. Knowl. Discov. 11(3): 243-271 (2005).

[34] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," IEEE Trans. Software Eng., vol. 28, no. 7, pp. 645-670, July 2002

[35] CodePro Analytix: http://code.google.com/javadevtools/codepro