Science of Computer Programming 95 (2014) 406–425

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A graph mining approach for detecting identical design structures in object-oriented design models

Umut Tekin [a,*], Feza Buzluca [b]

[a] Informatics and Information Security Research Center, Kocaeli, Turkey
[b] Computer Engineering Department of Istanbul Technical University, Istanbul, Turkey

HIGHLIGHTS

- We propose a graph mining-based approach to detect identical design structures.
- We evaluate our approach by analyzing several open-source and industrial projects.
- Identical designs structures can help in understanding the high-level architecture.
- Identical designs structures can help in discovering reusability possibilities.
- Identical designs structures can help in detecting repeated design flaws.

ARTICLE INFO

Article history:
Received 7 December 2012
Received in revised form 30 April 2013
Accepted 29 September 2013
Available online 10 October 2013

Keywords:
Software design models
Identical design structures
Software motifs
Pattern extraction
Graph mining

ABSTRACT

The object-oriented approach has been the most popular software design methodology for the past 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 that are designed for the specific application domains. Another commonly used technique 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 for detecting 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 reused design clones, which could help developers to improve their knowledge about the software architecture. Furthermore, problematic parts of frequent identical design structures are appropriate refactoring opportunities because they affect multiple areas of the architecture. Experiments with several open-source and industrial projects show that we can successfully find many identical design structures within a project (intra-project) and between different projects (inter-project). We observe that usually most of the detected identical structures are an implementation of common design patterns; however, we also detect various anti-patterns, domain-specific patterns, reused design parts and design-level clones.

1. Introduction

Many software projects contain a significant number of software clones [1], which are duplicated parts of source code or design models. One reason for design-level cloning is the frequent usage of software design principles and design patterns (e.g., GRASP [2], GoF [3]). Furthermore, there are domain-specific patterns [4] that are optimal for a specific application or

* Corresponding author.
E-mail addresses: umut.tekin@tubitak.gov.tr (U. Tekin), buzluca@itu.edu.tr (F. Buzluca).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.015

problem, which allows them to be used repeatedly in a project. In addition, there are 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 noted that the hidden patterns (domain-specific patterns) in software are crucial to understanding software architecture and to assessing its quality. Domain-specific patterns are the reused design structures in a specific project domain that are specialized to accomplish similar jobs in a common design form. Detection of these structures will give us the opportunity to improve and publish them for all developers. Theoretically, it is also possible to explore currently unnamed design patterns by examining reused design structures in specific project domains.

Studies on software maintenance indicate that more than 2/3 of the total development cost is spent on software maintenance activities [6,7], and more than half of the maintenance cost is spent on comprehension activities [8]. Nevertheless, several real-world evaluations present that the availability of documented design patterns will reduce the cost of program comprehension for object-oriented systems [9,10]. Additionally for these reasons, it is important to discover reused design patterns.

The comprehension 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 the production of poor quality products. Identifying these types of identical design structures (i.e., non-standard design patterns and common design defects) could provide a significant advantage in terms of reducing the cost of maintenance; the reason is that the most commonly-used structures in software design are the best places to look for refactoring opportunities because 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, and common design defects can be quickly identified, which allows them to be fixed in multiple areas at once. In addition, frequently repeated identical design structures are usually the most reusable parts of the design; these parts can provide good candidates for additional use in future designs.

Another source of the clones is the replicated code due to copy–paste activities. Mostly, developers modify these replicated parts separately to allow their source code to change, but the design remains same [11]. The code quality could decrease if developers apply a bug fix to one structure but fail to apply the same correction to its copies. With the help of our approach, copy–paste type design structures can be detected even if their source code is modified. These replicated structures can be combined into a single library entity, to be used efficiently in different parts of the current project or in future projects. This approach will also avoid inconsistent bug fixes.

The area of clone detection is considered to be an important part of several software engineering tasks [12]. Finding repetitive design matches could help developers and architects when evaluating reused design patterns, refactoring duplicated parts, understanding the program architectures and detecting plagiarism.

In this paper, we propose a graph mining-based approach to detecting identical parts in an object-oriented software architecture. The proposed approach contains three main steps. In the first step, the AST (abstract syntax tree) of the source code of the system is analyzed and the UML-based design level of abstraction is created. Based on this abstraction, we construct a software model graph, in which classes, interfaces and templates of software constitute the vertices, and the relations between them form the directed edges. According to the importance of the relation type, we assign weight values to the edges of the graph. In the next step, we apply a graph partitioning algorithm to divide the directed and weighted software model graph into small pieces. Finally, in the last step, a sub-graph mining algorithm is applied to discover identical design structures in the generated software model. Because the scope of our study is primarily focused on the detection stage, analyzing and automatically classifying these structures could be an interesting topic for future studies. However, we also present several evaluations with a manual classification to find useful and meaningful structures from open-source and industrial projects that could inspire future studies.

The remainder of this paper is organized as follows: Background and related work are presented in the next section. The graph representation and definitions are given in Section 3. The identification process is detailed in Section 4. In Section 5, the results obtained from exemplary projects are presented. In Section 6, we discuss critical parts and the efficiency of our approach, and the last section concludes the paper.

2. Related work

There are some graph-based design pattern detection approaches that are proposed in the literature [13,14], but most of them depend heavily on pre-known pattern templates. These approaches usually offer ways to define custom templates that could correspond to design flaws or non-standard pattern structures; however, during the development of the software, the method of using the patterns could vary according to the specific developer, programming language or target environment. In practice, it is not possible to cover all of the types of copy–paste design structures, domain-specific patterns, and non-standard design motifs for pre-known templates that are appropriate for all types of project domains. In contrast to the previous pattern detection studies, our approach does not require that any patterns are known in advance, and we can discover any type of identical design structures even though designers are not aware of them at all.

Koschke provides a survey about the studies in the clone-detection area that attempt to find identical parts of the software [15]. The suffix-tree comparison-based clone detection study in [16] works at the design level and attempts to find identical parts of UML sequence diagrams. Another design-level, pattern-matching-based study [17] operates on the XMI

tree structure of UML's domain models. However, in most cases, well-structured UML documentation of software is absent, incomplete or outdated [18].

There are also studies that use frequent sub-graph mining algorithms for clone detection in Matlab/Simulink models [19,20] and low-level program dependency graphs [22]. Mathlab/Simulink models are electrical circuit models in which only a single type of relation (the electrical link) between entities exists. These models have more sparsely-connected graph representations than the graph representations for software designs; hence, their average edge density (number of edges/number of vertices) is smaller. In our study, we address highly connected large graphs in which there are different and multiple types of relations between entities (classes and interfaces). For that reason, the design model of the software graphs requires excessively long running time for analysis. Most of the other clone detection techniques usually work on low-level attributes of programs such as source code, flow diagrams and variable dependency graphs [15,21]. We target high-level design models of software because detecting design clones can help in the comprehension and maintenance of object-oriented software, and it can provide reusability of commonly-used design structures.

In another high-level similarity detection study [23], simple source-level clones are grouped according to their appearance in source files, and then, data mining techniques are 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 add pieces or change the names or types of some of the attributes in the copied source code, the design structure will still remain the same.

Currently, the study that is the most relevant to our work is another graph mining approach with a different purpose, which focuses on the detection of unchanged/reused micro-architectures in the design model of a software system through its various versions [24]. In this study, the authors are able to find unchanged structures, including up to five entities. It is reported that the algorithm requires more than 60 hours to export different micro-architectures of the ArgoUML [25] project, for up to five vertices. The main difference from our work is that, in our graph model, we consider several high-level relations that they do not, which make our detection results more informative and more specific. In our graph model, we have four different types of class entities (class, interface, abstract, and template), whereas they represent each entity with a unique number that is related only to the real class names. Additionally, they consider only 3 generic types of relations between entities (association, aggregation and inheritance), whereas we have 11 different relation types. For example, while they consider only a single type of generic inheritance relation, we have 3 more specialized types, such as extends, implements and override. The override-relation is totally disregarded in the previous study. It is obvious that a more detailed software model graph will significantly increase the quality of the detection results by reducing the ratio of the false positive detections. Another advantage of our approach is that, with the help of partitioning, we can find identical structures within the ArgoUML project that comprise up to fifteen vertices in less than 5 hours. The other significant difference is that because their canonical text-based mining approach depends on real names (or unique IDs) of entities, they can investigate only a reused design match between different versions of the same software system, while we can also investigate repetitive identical design matches between different projects and also within a single version of a project.

We presented results of a preliminary study on the detection of design-level clones based on graph mining in [26]. We extend this study by using weighted graphs and a weighted partitioning algorithm to decrease the loss rate of important relations between entities. The results show that the weighted graphs improve the detections. In the previous study, the aim was the detection of intra-project clones. In the current study, we also investigated the detection of design-level clones between different projects (inter-project). We conduct a case study on industrial projects. By working closely with the development team of the projects, we validate our obtained results. To the best of our knowledge, there is no other published tool that combines graph partitioning and mining algorithms for the detection of identical structures in object-oriented software designs.

3. Graph representation of software design and definitions

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

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

Graph. Let $G = (V, E, Lv, Le, v, e)$ be a directed software model graph, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges, Lv is a set of labels for the vertices, Le is a set of labels for the edges, $v : V \rightarrow Lv$ is a function that assigns a label to the vertices, $e : E \rightarrow Le$ is a function that assigns a label to the edges.

Relation types in the software model graph are based on UML-like [27] 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 an "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, such as the "create" and "method call" relations between entities. Possible entity types, relation types and their labels are given in Table 1.

Table 1
Vertex and edge labels of the model graph.

Vertex label	Entity 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 the field type of Class B
T	Class A uses Class B in a generic type declaration
L	Class A has a method that defines a local variable with the type of Class B that is neither a filed type nor created by Class A
P	Class A has a method that has an input parameter with the type of Class B
R	Class A has a method with the return type of Class B
M	Class A has a method that calls a method of Class B
F	Class A directly accesses fields of Class B without method calls
C	Class A creates objects of Class B
O	Class A overrides methods of Class B

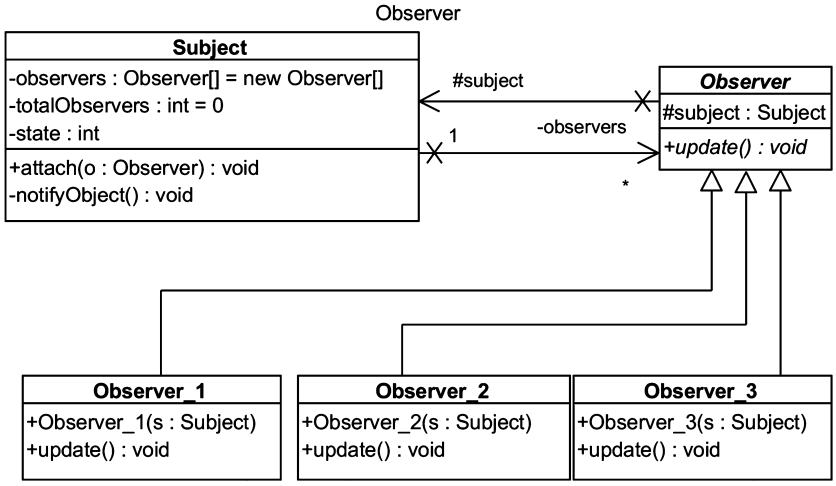

Fig. 1. The UML class diagram of a simple observer implementation.

Because of the nature of the object-oriented design, there can be more than one relation between the vertices (classes and interfaces). To create a simple and understandable graph, we collect all of the labels of parallel edges between two vertices into a single set of labels, such that '*Lij*' is a set of labels of directed edge '*eij*' that contains all relation labels from vertex '*vi*' to vertex '*vj*'. For example, if two entities have both extend {X} and method call {M} relations in the same direction, then the combined label set for this edge becomes $\{X\} \cup \{M\} = \{X, M\}$. In our approach for detecting identical design-level clones, the edges are compared using their set of the labels in such a way that, when comparing two non-empty edge label sets, '*Lu*' and '*Lv*' are considered to be equal if and only if '*Lu*' $\subseteq$ '*Lv*' and '*Lv*' $\subseteq$ '*Lu*'.

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 the UML class diagram, such as the "method call" (M), "methods parameter" (P) and "override" (O) relations.

Here, we give some definitions from graph theory that we use in the next sections:

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* ($Vs \subseteq V$ and $Es \subseteq E$).

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 all of the vertices and edges). This mapping must preserve the labels on the vertices and edges and also the direction of the 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

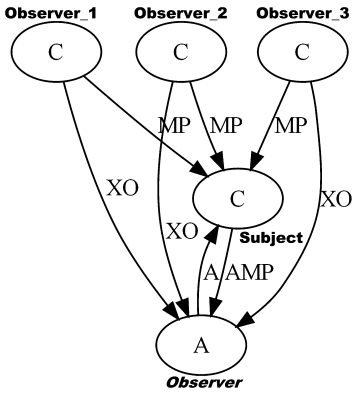

Fig. 2. Software model graph of the example observer pattern.

frequently in the graph set M. If a graph is frequent, then all of its sub-graphs are frequent. The frequency value is the number of different occurrences of an identical sub-graph in a graph dataset.

Closed frequent sub-graph. Closed frequent sub-graph G is a graph for which there is no frequent super-graph of G with the same frequency value [28].

4. Identical structure mining

In this section, we detail our mining approach about 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.

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 (AST). Then, the model builder processes the relations and generates a complete graph model of the software architecture. At the moment, we support only Java-based projects; however, it is also possible to export graph models by parsing an AST that corresponds to other types of object-oriented programming languages, such as C++ and C#. To accelerate the method and to search identical structures within a project, the software model graph is divided into small pieces by applying a graph partitioning algorithm. In the last step, we apply a closed sub-graph mining algorithm to discover identical design structures in the generated software model. The graph partitioning and the sub-graph mining algorithms, which are the main parts of our approach, are explained in detail in the following subsections.

4.1. The graph partitioning algorithm

The running time of the sub-graph mining algorithms dramatically increases depending on the number of vertices $|V|$ and edges $|E|$. Therefore, we apply the "divide and conquer" strategy, which is based on the idea of partitioning the model graph into smaller "n" sub-graphs, to find frequent identical structures, which are included for some of the "n" partitions in common. In addition to increasing the speed of our approach, partitioning is also essential to being able to investigate design-level clones within the same software project. To minimize the loss of important relations during the partitioning, we assign weight values to different relations types in our model graph.

Basically, there are three different stages in the partitioning process; these stages are illustrated on a simple exemplary graph in Fig. 4, which has a partition count of two. In the first stage, called the coarsening stage, the original large graph is transformed into a smaller graph by collapsing adjacent vertices using the maximal matching algorithm that is defined in [29]. In the second stage, the small graph is partitioned as equally as possible according to the required partition count determined by the user. Because the graph partitioning problem is an NP-complete problem, to solve it effectively, the selected algorithm uses a combination of the two fast heuristics, namely, spectral bisectioning [30] and Kernighan–Lin heuristics [31]. After the coarsening phase, the algorithm first uses spectral bisectioning to directly divide the coarsest graph into two, four or eight parts according to the required partition count. Then, it applies the Kernighan–Lin heuristic, which is an iterative optimization heuristic for the graph partitioning problem. Basically, it starts with initial random partitions of the graph, and then, in each phase, it searches subsets of vertices by swapping them between partitions to find partitions that have minimum edge cuts. At this point, several early termination conditions are defined, such that no other optimal swap operation remains. There are also some other early termination conditions according to the quality of the partitioning (i.e., the vertex balance of the partitions) [29]. In the last stage, the small graph is un-coarsened to the original form. Because each vertex in the small, coarse graph belongs to a specific partition, it is un-coarsened by simply assigning same partition membership of the embracing vertices to their collapsed vertices. Here, we explain only the basics of the partitioning process; however, there are many other features, such as the optimization of the Kernighan–Lin heuristic [32], smart strategies for the determination of initial partitioning, and fast data structures that are detailed in [29]. We use the implementation

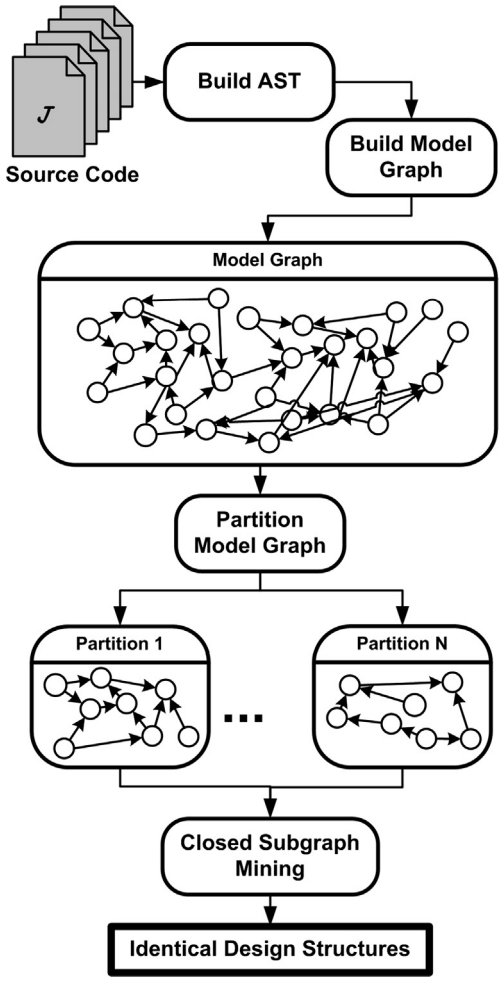

Fig. 3. Basic flow of the approach.

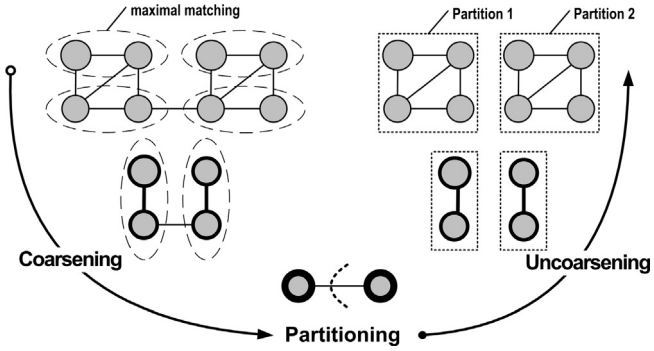

Fig. 4. Partitioning example for a simple graph with a partition count of two.

of the algorithm in the Chaco [33] graph partitioning library. In our test cases, this approach produces partitions in only a couple of seconds at most. We discuss the pros and cons of partitioning in the following paragraphs.

Partitioning is crucial for this approach; however, it could prevent identifying some interesting sub-graphs that would span between two or more partitions because we lose some weak relations during the partitioning. However, we assume that the most interesting reused software modules must be the entities that are strongly connected (highly cohesive). For example, in Java-based projects, developers usually place highly-related classes into the same packages, which usually include approximately 20 classes. Therefore, our graph partitioning algorithm attempts to partition graphs into strongly-connected

Table 2
The partitioning effect on each type of relation (average vertex degree = 5.4).

Type of relation	Total number of relations in model graph	After un-weighted partitioning	Relation loss percentage	After weighted partitioning	Relation loss percentage
X	36	27	25.0%	31	13.9%
I	19	13	31.6%	16	15.8%
A	56	43	23.2%	42	25.0%
T	0	0	0%	0	0%
L	129	97	24.8%	89	31.0%
P	99	74	25.3%	74	25.3%
R	78	57	26.9%	54	30.8%
M	257	201	21.8%	187	27.2%
F	0	0	0%	0	0%
C	109	79	27.5%	71	34.9%
O	27	20	25.9%	23	14.8%

Table 3
Effect of the weight value on the detected identical structures in the E-Quality project (AVD = 6.7).

Weight value of edges that are type X and I	Detection ratio of most remarkable design matches	Total number of detections
Minimum value (1)	11 of 12	38
AVD/2	12 of 12	39
AVD	12 of 12	39
10 × AVD	11 of 12	36
100 × AVD	10 of 12	31
1000 × AVD	7 of 12	23
Maximum value (65 535)	5 of 12	20

partitions by removing weak relations as much as possible. For this purpose, we assign weight values to edges (relations) of the graph and use an algorithm that relies on the minimum cost (edge cut) graph bisection method.

In object-oriented programming, inheritance is an important type of relation because parent entity properties (i.e., variables and methods) propagate through the inheritance tree; thus, it is not possible to change relations of this type without major modifications at the architectural level and also at the source level. Therefore, we want to minimize the loss of inheritance-like relations, such as extend and implement, during the partitioning process. To determine the weight value, first we calculate the average vertex degree ($2 \times |E|/|V|$) of the whole model graph. Then, we assign this value as the weight value for the relations that are types of inheritance. The other relation types (such as method call and creates) receive the minimum weight value of 1. In Table 2, we show the effect of the weighted (average vertex degree = 5.4) and un-weighted versions of the partitioning algorithm on each type of relation for the open-source project Zest [34], which has 8 partitions. As seen from the table, the weighted version of the partitioning algorithm reduces the loss of the important design relations, such as implements (I) and extends (X); however, it also has a small negative effect on the other type of relations, such as the method call (M). After the graph partitioning process, it is inevitable that some edges are lost. Therefore, our aim is to avoid (or at least reduce) the loss of more important information.

We observe that there is a trade-off between the weight value that we assign to the inheritance relations and the detection quality of our approach. It is obvious that weight values that are too small will result in the loss of too many important edges. On the other hand, if the weight value is selected to be a number that is too high to protect the important inheritance relations, then the partitioning algorithm gathers all of the weighted edges into a few specific partitions, which makes our detection quality decrease. As an extreme case, it is possible to set the weight value to the maximum possible value to prevent nearly all of the inheritance relations from being cut. However, in this case, due to some of the very common inheritance relations that reside at the very top of the inheritance trees, most of the vertices that are connected by common inheritance relations will be collected in only a few partitions. These types of high-level inheritance relations usually represent the most common interface entities of the software system, and they must be removed to separate highly cohesive modules into different partitions. The maximum optimal weight value for the important edges can vary according to different test subjects, but we observe that the average vertex degree is well-suited for our test projects. We also present results from an experiment to demonstrate the effect of different weight values in the following paragraphs.

To investigate the impact of the weight value assigned to the inheritance relation on the detected identical structures, we performed an experiment on our software analysis tool E-Quality [35] by setting the partition count equal to 16 and the weight value as the average vertex degree (AVD). After the graph mining process, we obtained a set of 39 identical structures, and we selected 12 of them that we consider to be remarkable after analyzing the result set manually. The selected 12 structures include 2 copy–paste structures, 3 domain-specific patterns, 2 design defects and 5 regular design patterns. Other design matches, such as common structures or irrelevant design matches, are not included, and they are considered to be uninteresting detections. Then, we investigated the changes in our remarkable (valuable) detection set for different weight values. As seen from Table 3, too small (1) and too high values ($> 100 \times$ AVD) of the weight have a negative

Table 4
Partitioning effect and performance results for different partition counts (frequency value $\geqslant 2$, vertex count > 2).

Number of graph partitions	Sum of all edges (combined edges)	Percentage of lost edges	Number of identical structures	Maximum vertex count in partitions	Maximum memory consumption (in GB)	Running time (in sec.)
2	345	11.50%	NA	74	NA	NA
3	337	13.50%	NA	50	NA	NA
4	332	14.80%	NA	37	NA	NA
5	321	18.70%	NA	31	NA	NA
6	313	19.70%	26	27	27.9	4731.19
7	301	22.80%	32	23	15.3	552.60
8	286	26.60%	37	21	7.2	0.75

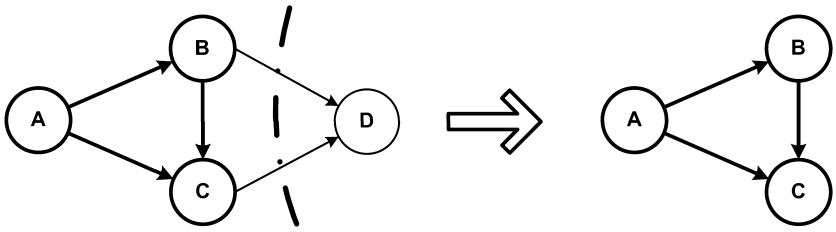

Fig. 5. Effect of an increase in the partition counts.

effect on the number of remarkable (valuable) detections and also on the total number of detected structures in the result set. We also note that, with the weight values of AVD and (AVD/2), we detect one more remarkable design match rather than the others. We also observe that, when the weight value changes around AVD, the detected structures include more edges with the inheritance relation type, which is important for the design-level analysis.

Another important parameter of the partitioning algorithm is the count of the partitions that are determined by the user. In Table 4, we present the sum of all of the edges and the percentage of lost edges for the different numbers of partitions in the Zest project. It can be seen from the table that the number of detections increases with the number of partitions. However, as expected, the detected identical structures become smaller when we increase the partition count because of more edge losses and smaller partition sizes. For example, if the Zest project is divided into 6 partitions, then the largest detected structure has 10 vertices and 21 edges, whereas there are 9 vertices and 18 edges with 8 partitions. However, both of the mentioned structures are related to the same part of the program; in other words, the smaller structure is a sub-graph of the large structure. This effect of partitioning on the detection results is illustrated in Fig. 5. In our test system, the graph mining algorithm does not terminate successfully because the memory is not sufficient if the partition count is smaller than six. From the running time and memory consumption displayed in Table 4, we can easily realize that the bottleneck of the frequent sub-graph mining algorithm is the memory consumption.

4.2. The sub-graph mining algorithm

After the partitioning process, in the last step of our approach, we apply the complete frequent sub-graph mining algorithm CloseGraph [28] to discover all of the frequent isomorphic sub-graphs that are shared by those partitions. The CloseGraph uses the depth-first search strategy to mine frequent connected sub-graphs, and it is capable of finding all of the closed frequent isomorphic graphs. The algorithm also 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, and it is also worthwhile to notice that when a large identical sub-graph is detected, the smaller sub-graphs within it are not reported by the algorithm. Because the problem of sub-graph isomorphism is NP-complete and the CloseGraph algorithm produces a complete output, this stage is the most time- and resource-consuming stage of the detection phase. We use a ParSeMiS [36] implementation of the CloseGraph algorithm, which is used successfully in many research projects.

The partitioning and detection phases are illustrated on a simple exemplary graph in Fig. 6. In this example, we partition the graph into two pieces, and the simple isomorphic sub-graph detection is highlighted as $G = \{A, B, C, D\}$. Because we are looking for architectural design-level clones, the detected structures comprise entities that are strongly related and therefore copied or reused as a whole. These structures are mostly a design module that implements a specific service. Usually, they are used in different programs or in different parts of the same program in various ways. Therefore, these entities (classes or interfaces) of these modules could have additional relationships (incoming or outgoing) with other entities of the system that do not match. However, their internal relations are the same.

Although CloseGraph is considered to be one of the best algorithms for the frequent sub-graph mining problem [37], when the number or the size of the partitions increases, the running time increases exponentially, and especially the memory consumption becomes a bottleneck. Current complete frequent sub-graph mining algorithms are especially designed to

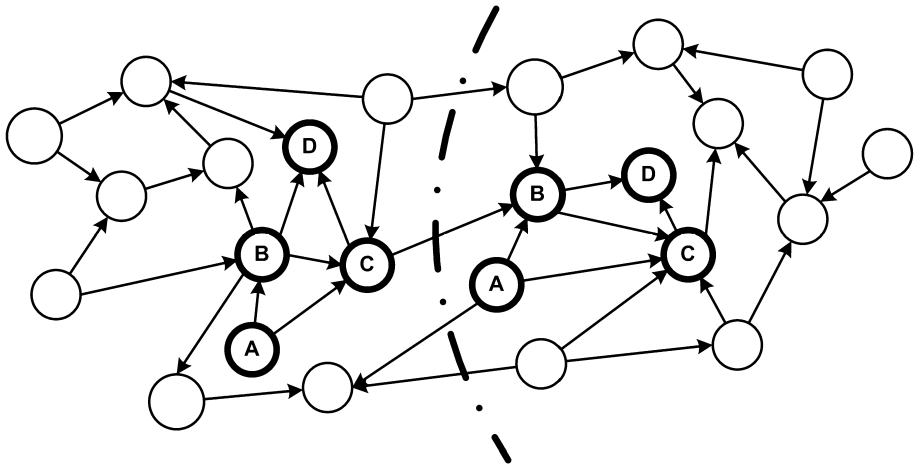

Fig. 6. Partitioning and detection approach.

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. Because the software graphs are highly connected [38] to run algorithms efficiently, the sizes of the partitions must be selected properly. In our test cases, we realized that we were able to find identical structures in partitions that had approximately 20 vertices in a reasonable running time with reasonable memory consumption. Therefore, we determined the partition counts for our test subjects by taking the total number of vertices in the graphs of the whole models into consideration, to allow each partition to have approximately 20 vertices. For larger partitions, the memory consumption becomes a bottleneck, and a large search space dramatically increases the running time because of the graph explosion problem (an n-edge frequent graph can have 2^n (2^n) sub-graphs). Although the algorithm has several properties that attempt to prune the search space, the graph explosion problem exponentially increases the candidate search space (memory consumption) and also the number of the required graph comparisons (running time). There are studies in the literature, such as [37], that address the performance issue of sub-graph mining. We do not completely cover sub-graph mining performance issues in the present study; however, we present several performance values from test projects that could give an idea of the performance of the algorithm.

5. Experiments and results

5.1. Experiments on open-source projects

This section presents the results obtained from our experiments on open-source projects. In this section, we investigate identical design structures within five open-source projects with different sizes (YARI [39], Zest [34], JUnit [40], JFreeChart [41], ArgoUML [25]) and our software analysis tool E-Quality [35]. YARI is a small tool suite for inspecting 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 a UML modeling tool. E-Quality is a software visualization and analysis tool for object-oriented systems that was developed by our research group. We run our experiments on a 2-GHz 4-CPU 64-bit Linux system with 32 GB of RAM. The partitioning algorithm is implemented as a single thread. However, the implementation of the frequent sub-graph mining algorithm is configured to run in parallel by creating four tasks that are attached to different processors. The given running-time values in our case studies are all elapsed real times.

In Table 5, we present for each project lines of code (LOC) that contain characters other than white spaces and comments, properties of the produced software model graph (SMG), such as the number of partitions, the maximum vertex count in each partition, the running-time in seconds, and the number of discovered identical structures that appear in at least two different partitions (frequency $\geqslant$2). The number of vertices (classes and interfaces) in these identical structures varies from three to fifteen. The structures with one and two vertices are filtered because they are too small to address. As seen from Table 5, the running time dramatically increases with the number of partitions. For example, the running time increases to at most 253 min, and the memory usage increases to at most 25 GB, during the analysis of the ArgoUML project.

In Table 6, we also present the number of detected identical structures with a certain frequency (number of occurrences) of values for the analyzed projects. The number of partitions in the graph dataset determines the maximum frequency value: for example, in the YARI project, the maximum possible frequency value is three. The possibility of finding identical structures that reside in many partitions is smaller. Therefore, with an increase in the frequency value, the number of exact matches decreases. We also present the performance results depending on the frequency values for the ArgoUML project, which is the largest project in our test base. Because the CloseGraph algorithm is especially designed to detect high-frequency sub-graphs in protein or chemical networks, its running time dramatically increases with lower frequency values because the number of comparisons and the search space increase.

Table 5
Sub-graph mining results (frequency value $\geqslant 2$, vertex count >2).

	Yari	Zest	JUnit	E-Quality	JFreeChart	ArgoUML
Number of vertices in SMG	69	144	271	281	614	1571
Number of edges in SMG	148	390	1046	939	2409	6509
Line of Code (LOC)	4727	14,432	15,247	21,662	56,925	117,903
Number of graph partitions	3	8	16	16	32	96
Maximum vertex count in partitions	23	21	18	18	20	17
Number of identical structure	15	37	61	39	307	745
Running time in seconds	0.8	0.75	76.56	17.13	1102.78	15,126.92

Table 6
Number of identical structures for different frequency values (vertex count >2).

Frequency value	Yari	Zest	JUnit	E-Quality	JFreeChart	ArgoUML	Running time in Sec. (for ArgoUML)
2	14	29	38	27	207	374	14,161.78
3	1	7	14	9	58	174	3502.44
4	NA	1	5	3	26	92	2190.22
5	NA	0	3	0	6	41	1401.56
6	NA	0	1	0	4	27	761.67
7	NA	0	0	0	3	15	157.13
8	NA	0	0	0	2	10	84.71
9	NA	NA	0	0	1	7	44.10
10	NA	NA	0	0	0	4	8.51
11	NA	NA	0	0	0	1	3.23

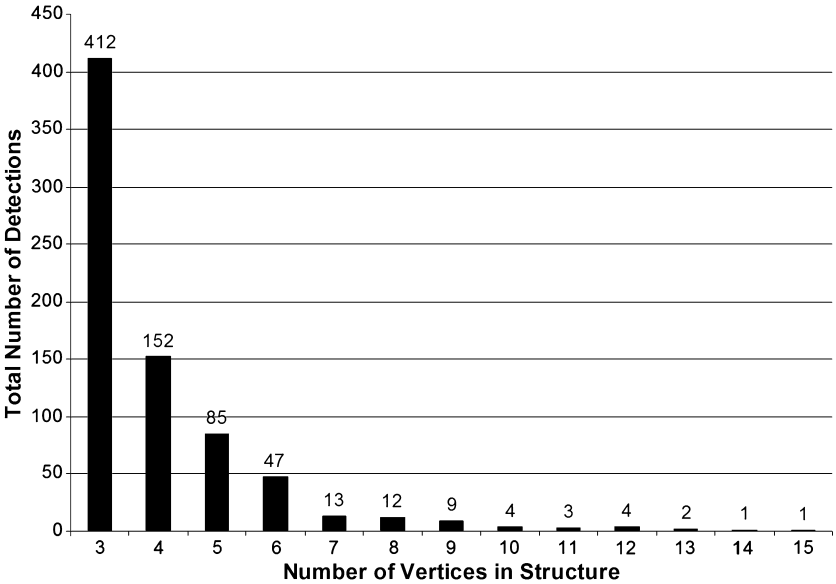

Fig. 7. Distribution of the number of vertices in the detected identical structures.

Fig. 7 displays distributions of the number of vertices in the detected identical structures for the ArgoUML project. The total number of detections strongly reduces with the increase in the size of the identical structures. Moreover, we observe that for the ArgoUML project, structures with large graphs ($|V| > 5$) are mostly much more noticeable and interesting than the smaller structures. Most of the small structures ($|V| < 5$) are common structures of the object-oriented programming, and usually, they are the uninteresting detections. As seen in Fig. 7, only 96 of the 745 structures ($\sim$13%) have more than five vertices, which should be analyzed more carefully to interpret the valuable detections. These observations for ArgoUML might not be the same for every project; for example, for the project E-Quality, nearly 40 percent of the detections has more than five vertices.

In the following pages, we briefly explain some of the identical design structures that were discovered in the different projects. Fig. 8 shows two of the most common design structures that we identified in our test projects. These types of design structures have a basic graph representation with few relations, for example, few vertexes with a single “extend” or “method call”. They appear in almost every project with high frequency values. Many times, these structures are ignorable and uninteresting because they are common structures of object-oriented programming. The percentage of these structures

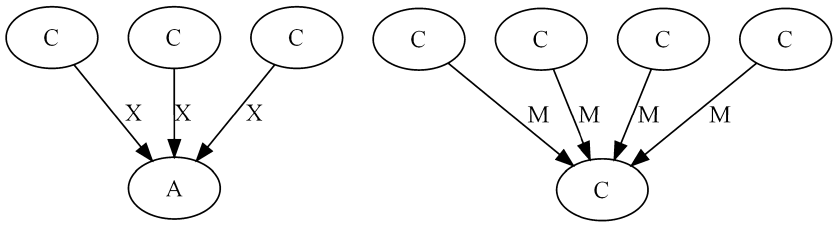

Fig. 8. Common design structures.

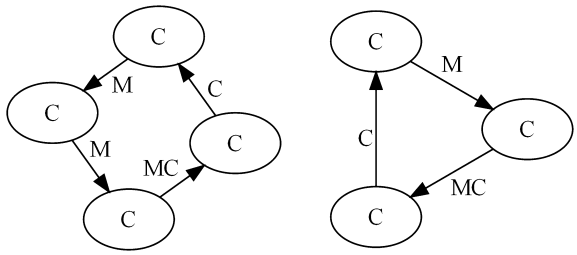

Fig. 9. Circular dependency structures in JUnit.

over the total number of discovered structures is usually high; for example, for the project YARI, it is ∼40% (6 out of 15), and for the project E-Quality, it is ∼53% (21 out of 39). We realize that this percentage grows with the size of the projects because they are the most common design characteristics of the object-oriented programming. To spend more time on useful identical structures, these types of common design structures must be eliminated. However, it is very easy for a designer to eliminate them if the designer is somewhat familiar with the project. For example, after sorting the detection set by the vertex and edge numbers, we could manually eliminate each of them in a couple of minutes for the project E-Quality. It is also possible in future work to create a set of templates for these types of structures, to enable them to be eliminated automatically.

Fig. 9 shows two examples of identical structures that we found in the JUnit project. The design structure on the right side has a frequency value of four, and the design structure on the left side has the value of two. After we had analyzed these structures, we observed that they are examples of a circular dependency anti-pattern that is considered to be bad or harmful [42]. We also detect several other identical structures of up to six vertices that contain circular dependencies in the ArgoUML project. Based on the observation that circular dependencies are sources of many identical structures, we deduce that they are repeated in many parts of the system and that they can be identified as a common design defect for the JUnit and ArgoUML projects. Because these structures could reduce the reusability, quality and manageability of the software, it is better to refactor them. Typically, they reduce the reusability because the separate reuse of a single entity is not possible. The quality is reduced because some garbage collection systems cannot remove these entities because they are usually referencing each other cyclically, which could cause several memory leaks. The maintenance becomes harder and the cost of change becomes higher because the changes in one entity can easily affect other entities [42]. Because they affect multiple areas in the design, they must be at least identified and reviewed by developers to avoid future problems. In fact, some circular dependencies could be harmless or beneficial, but it is always good to know where they are.

Circular dependency is a specific type of design flaw that we found in the repeated identical structures that we detected in the projects. In fact, the purpose of our approach is not to find unique patterns (i.e., single circular dependency or single observer patterns) because we focus on the detection of identical repetitive design structures. However, we can detect design flaws when they are repeated in at least two parts of the system, and after an analysis, we identify them as common design defects. For future study, it is possible to extend the project graphs with custom graph templates that correspond to specific design defects or design patterns, to detect unique patterns in the projects.

Fig. 10 shows a structure whose copies we found in different parts of the ArgoUML project. This example is a typical factory-like pattern implementation. The class located on the top in the figure creates objects of related types. “Profile-GoodPractices” 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. 10. Examples of these factory structures are detected with different sizes and even with higher frequency values because they are frequently used in object-oriented projects.

A design pattern could have various implementations with different numbers of concrete subclasses. For example, a project can contain one observer pattern implementation with n concentrate subclasses and another implementation with $(n+m)$ concentrate subclasses. In such a case, our approach finds a design match between these two structures that includes n concentrate subclasses (their maximum intersection set). After the detection phase, a quick inspection on the source code can help developers to discover undetected parts (the other m subclasses) of the larger implementation. If there was a third

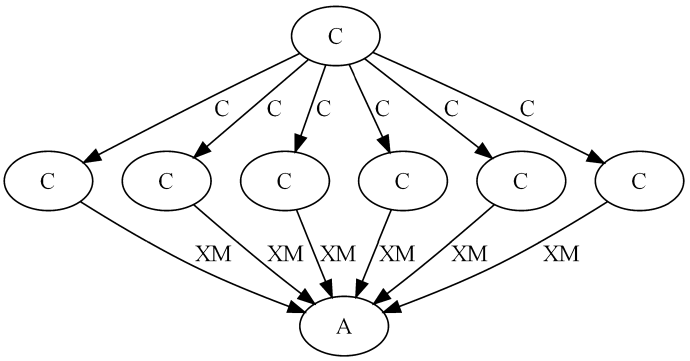

Fig. 10. Factory-like structure.

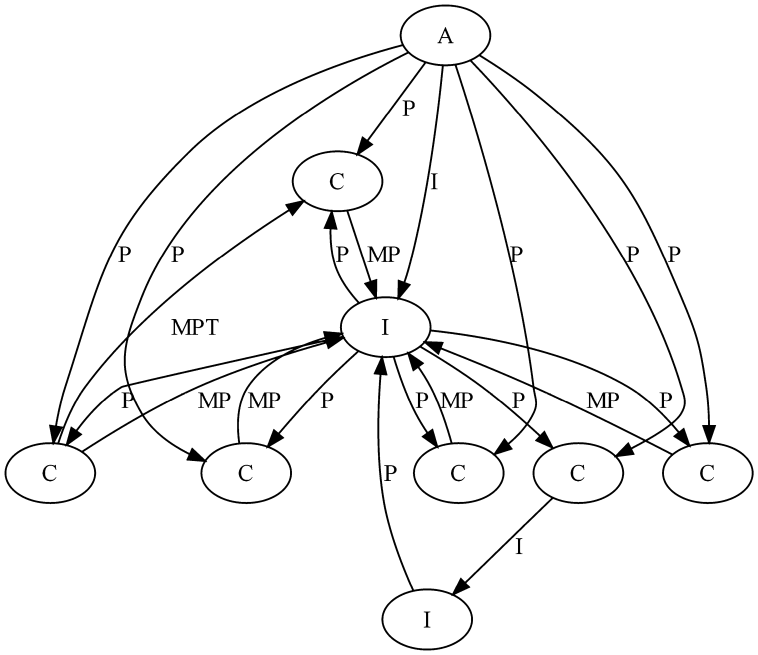

Fig. 11. The visitor pattern.

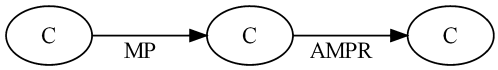

Fig. 12. A detected copy–paste structure.

implementation of the same pattern with $(n+m+t)$ concentrate subclasses, then the algorithm would report two separate detections, namely one detection including $(n+m)$ subclasses and having a frequency value of 2 and another detection with n subclasses and a frequency value of 3.

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. 11, with a frequency value of two. We use this pattern to walk through the software model graph in two places for different purposes with the same design philosophy. Additionally in E-Quality, we found an identical structure with a frequency value of three that is a source-level copy–paste clone of a database access module that contains three classes, as shown in Fig. 10. We are planning to modify the structure in Fig. 11 to a standard visitor form and to remove multiple copies of structures shown in Fig. 12 to improve the design quality of our program.

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. 13, takes place in the "org.jfree.chart.renderer.category", and the other instance takes place in the "org.jfree.chart.renderer.xy" packages. The package "renderer.category" contains 21 entities (classes or interfaces) with 4835 lines of code, and the package "renderer.xy" contains 27 entities with 5471 lines of code. We analyzed these packages manually by reading source code and automatically using CodePro's [43] similarity and clone detection

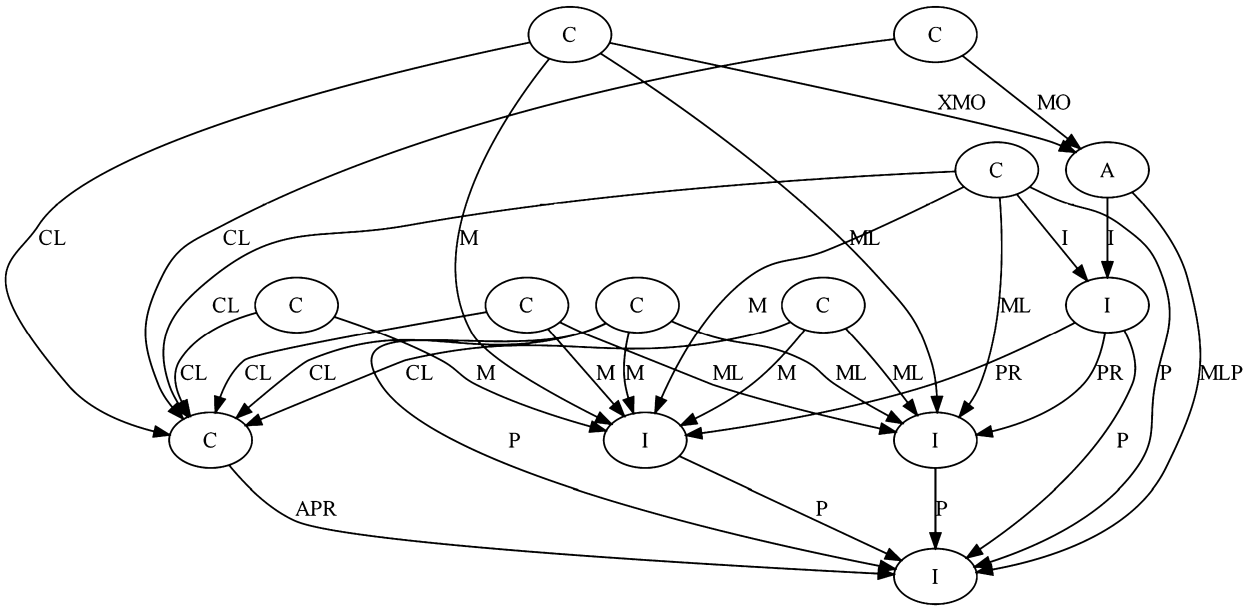

Fig. 13. A complex software clone in the JFreeChart.

interface. We found three pairs of classes among the packages that included a large amount of copy–paste code clones, which are given below:

xy.XYBoxAndWhiskerRenderer ⇔ **category**.BoxAndWhiskerRenderer
xy.AbstractXYItemRenderer ⇔ **category**.AbstractCategoryItemRenderer
xy.StackedXYAreaRenderer ⇔ **category**.StackedAreaRenderer

Although 3 of the 13 entities of the structure in Fig. 13 correspond to source-level clones, the whole structure cannot be classified as a direct copy–paste clone. According to our architectural understanding, this structure includes groups of classes that have the same behaviors for similar types of work. Most likely a designer must follow at least a similar design philosophy to add another "renderer" package to this project. Therefore, we classified this case as a domain-specific design match for the JFreeChart project rather than as a direct copy–paste detection.

We discovered also some identical design structures that cannot be classified as design- or domain-specific patterns. Fig. 14 shows an exemplary identical structure from JUnit that might be considered as only a simple or causal design match with a frequency value of three. Unless they are copy–paste clones, these types of structures are also usually uninteresting design matches, and they could appear in only a low percentage of the total detection results. For example, in the E-Quality project, we considered 6 out of 39 detected structures to be simple design matches.

By applying our method to open-source projects, several identical design structures have been identified, from simple to complex structures. We can separate these structures into two categories, useful and uninteresting (or irrelevant) structures. For example, after a manual inspection, we classified ∼31% of the detections (12 out of 39) in the E-Quality project as useful structures (2 copy–paste structures, 3 domain-specific patterns, 2 design defects and 5 design patterns), and ∼69% of the detections (27 out of 39) as uninteresting design matches (21 common structures and 6 causal matches). Further analysis, especially on the useful structures, could reveal some important insights about the quality of the projects. Developers and/or designers can suggest several useful improvements for these structures, such as fixing the design defects, removing unnecessarily repeating code or making domain-specific structures more modular, reliable and reusable.

5.2. Identical design structures between industrial projects

In this section, we used our approach to investigate identical design structures that are shared by two different industrial projects. The projects were developed and designed under the supervision of the same software architect at a software development company. The first project is Radio Link Management (RL-Man) software, which was completed two years ago. The other project is also management software (NW-Man) for new generation high-speed network security equipment, and the development team is still continuing to develop it. We have interviewed four developers and the architect of the NW-Man project. One of these developers and the software architect have also worked on the RL-Man.

Although it is possible to analyze completely different projects, in this case study, we especially target these management projects to validate our detection results with the help of the development team. Because both of the projects have been developed by the same team, it is expected that they have several similarities that will provide us more results (evidence)

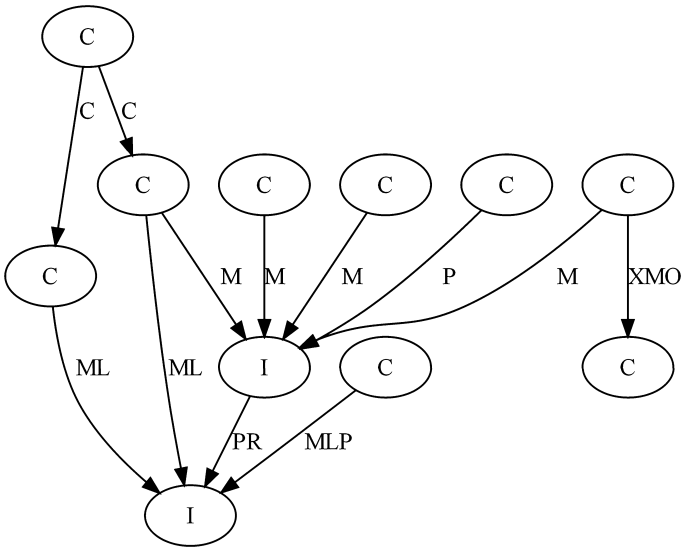

Fig. 14. A design clone in JUnit.

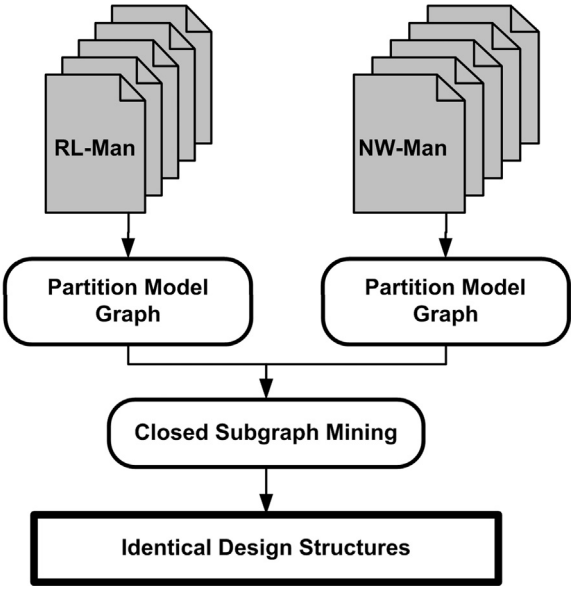

Fig. 15. Mining two projects to detect identical structures.

for different categories of design matches. Despite their similarities, two projects also have many differences due to different requirements and different technologies. For example, the new project is a web-based application server, while the older project is a classic desktop application.

To find identical design structures between two projects, we combine all of the partitions of two projects into a single set (the "Combined set" column of Table 7); then, we apply a sub-graph mining approach to this set, as shown in Fig. 15. In Table 7, we present properties of the generated software model graph (SMG), the number of partitions, the maximum vertex count in each partition, the number of detections and the running time for each project and for the combined set of projects. As seen from Table 7, we detect 107 identical structures in the combined set. When we investigated the results, we discovered that 26 out of the 107 structures are between different projects, and the others are within the same project. A total of 23 out of the 26 structures have a frequency value of two, and the other three have a frequency value of three.

After obtaining the results, first we introduced our software graph model and mining approach to the development team. That introduction phase was completed within approximately one hour. Then, we started to analyze the detected identical structures (26 out of 107) that are between the projects with the developers of the projects. The team identified 8 of the 26 structures as reused parts from the RL-Man project. This finding means that they copied and modified the source code of some of the parts from the old RL-Man project, but their design remained the same in the new project. Fig. 16 shows

Table 7
Mining results for the combined set (frequency value $\geqslant 2$, vertex count >2).

	RL-Man	NW-Man	Combined set
Number of vertices in SMG	332	547	879
Number of edges in SMG	1446	2503	3949
Number of graph partitions	16	32	48
Maximum vertex count in the partitions	23	20	23
Number of identical structures	34	55	107
Running time in seconds	34.23	1187.10	1402.34

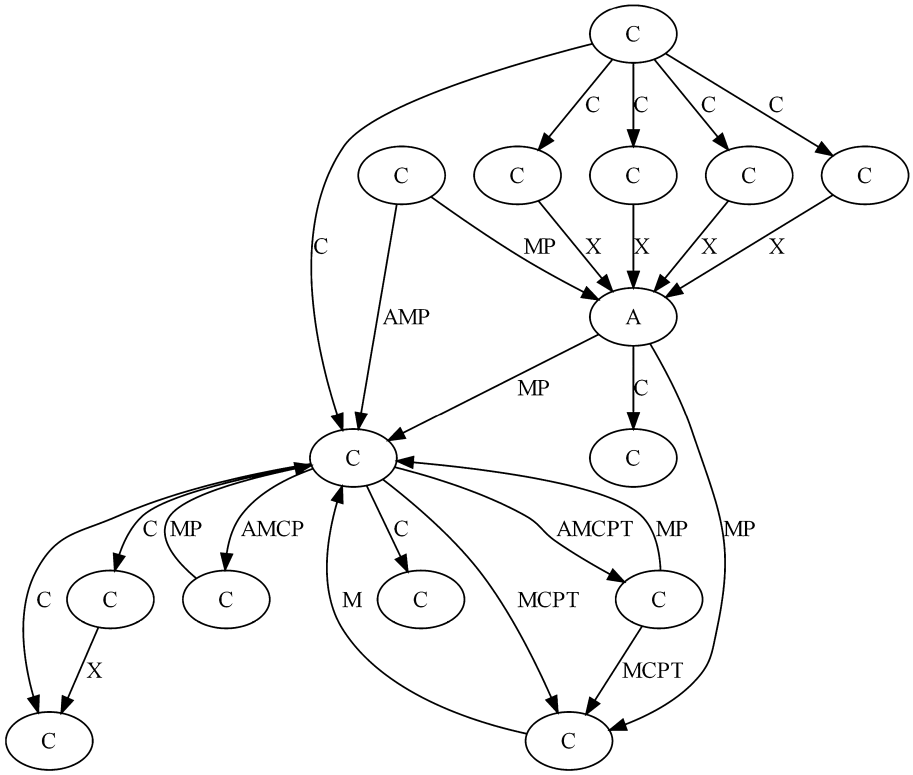

Fig. 16. A reused design structure between projects.

an exemplary identical structure among the projects that correspond to the "alarm & log management" module of the systems, and Fig. 17 represents a part of the identical design structure from a user management module of both projects. The identified 4 of the 8 reused structures correspond to the utility modules (e.g., address conversion, parsing xml inputs), and the other 2 structures correspond to part of the drawing module.

The other 7 of the 26 detections between projects were categorized as design patterns, which include factory (4), observer (2) and a visitor pattern across the projects. The remaining findings (11 out of 26) are categorized as common (9) and simple (2) design matches of object-oriented programming, which are not found to be interesting by the developers. This categorization was accomplished in less than an hour.

Then, we started to analyze our 8 valuable findings (reused structures) that were between projects. The first observation that the team made was that they were not previously aware of the 3 reused structures, including the 2 structures that are related to the drawing module and the structure that belongs to the "alarm & log management" module in Fig. 16. Further analysis on these structures at the source level revealed that there were several differences in their source codes. Some parts of the structures were heavily modified at the code level of one project for several reasons, such as performance improvement, bug fixes and new system requirements; however, the developers neglected these updates in the other project. For example, by comparing the source codes, the team realized that the "alarm and log management" module has more than 10 bug fixes and 2 performance improvements in the old project, while most of these changes do not exist in the new project. The old project, RL-Man, was completed two years ago, and it is still in use. Therefore, developers receive some feedback from the field that motivates updates in the program. Apparently, however, the team missed the fact that some of these updates should apply to the new NW-Man project that is still under development. According to the team, this finding is an important result because the copied parts of a software system change separately and independently, which makes them unmanaged pieces of duplicated software. It is especially noteworthy that missed bug fixes can be dangerous and

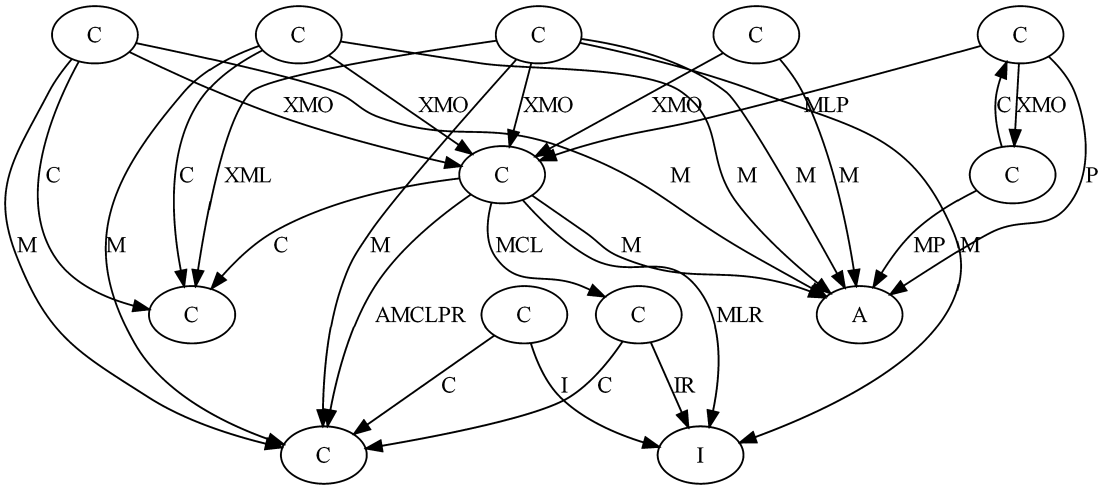

Fig. 17. Duplicated user management module between projects.

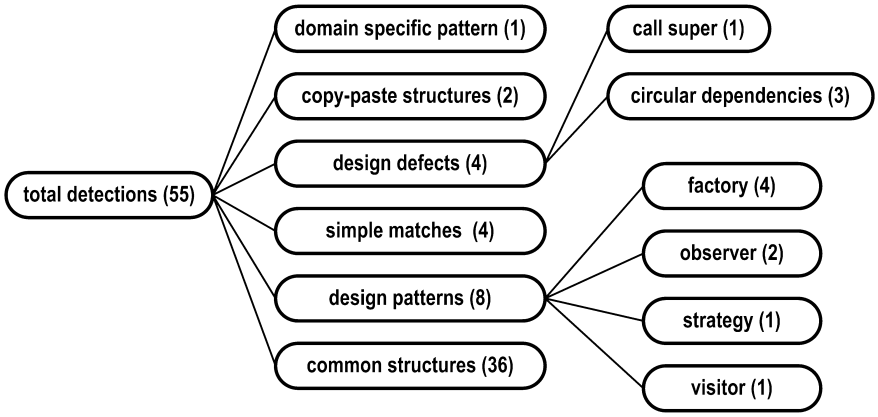

Fig. 18. Manuel classification results for the NW-Man project.

can harm the quality of the software and increase the cost of maintenance because these bugs must be detected and fixed again. According to the team, the main reason for the missing bug-fixes is that the original maintainer of these modules left the team some time ago.

After another quick source code comparison, 4 structures that correspond to the utility modules and the structure in Fig. 17 were considered to be reused copy–paste structures by the team. Even they were maintained by the same developer, who works for both projects; they included some differences at the code level. The developers noticed that those differences were caused by simple bug fixes that were applied to the new project but that were overlooked in the old project (RL-Man). The team agrees that especially those utility modules must be combined as a single library entity for quality improvements and future uses.

The team confirms that our inter-project findings are the most reusable parts of the RL-Man project because they are repeated in the NW-Man project. Developers also agree that these structures are the most suitable parts for searching refactoring opportunities because any improvement makes them more valuable for these and also for future projects. We also observe that the detected clones that are among the projects have a large number of inconsistent bug fixes. It is obvious that necessary bug fixes must be applied to these structures in both projects, to improve their quality. The analysis and discussion process on all of the inter-project detections took almost three hours. As a result, we note that the members of the team find those results to be extremely useful for improving the quality and the reusability of the projects.

We also had the chance to analyze our intra-project findings for the NW-Man project, which is currently in the development phase (it is very close to its first release). To simplify the classification process, we first sorted our detection set by the vertex and edge numbers in ascending order. Then, together with the development team, we categorized our 55 findings, as presented in Fig. 18. It took nearly two hours to complete this categorization process. At the end of the manual inspection for the MW-Man project, the team classified ∼27% of the detections (15 out of 55) as useful structures and ∼73% of the detections (40 out of 55) as uninteresting design matches.

The developers identified a small domain-specific pattern, which is given in Fig. 19; it was repeated three times in the project. According to the developers, this structure is an access right control pattern that they apply in several places of

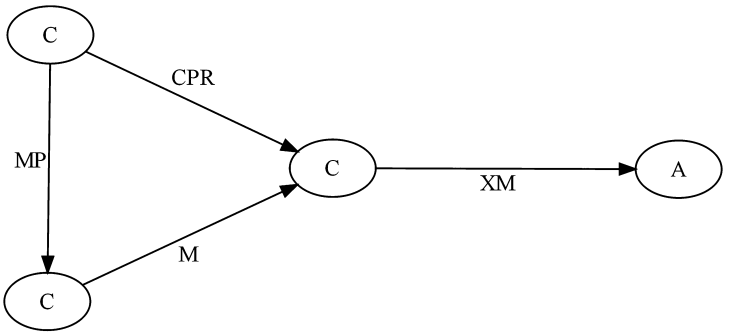

Fig. 19. Access control pattern from the NW-Man project.

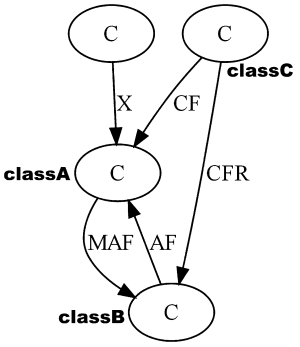

Fig. 20. Circular dependency detection for the NW-Man project.

the software system to control and monitor critical data operations, such as password access. This structure was primarily used by a single developer. After our detection, the team decided to use this specific design motif also in two or three other possible places of the software system, to improve the security level of the critical data accesses. The architect of the team also set aside this structure for future examination and standardization. Because related source code of the detected structures are completely different, these structures cannot be detected by clone-detection tools that are operating at the source code level, such as CodePro [43].

After a source code inspection, one of the two detected copy–paste clones was considered to be an unnecessary replication, and they planned to remove it. The other clone was identified as a user input simulation test module that the team was already aware of and was planning to remove.

We determined that one of the detected structures with a frequency value of 3 includes the common design defect "call super" [42] anti-pattern. This defect occurs when a subclass calls an overridden method of the abstract super class. Normally, the parent class expects its derived classes to override its methods, to extend the functionality. In this structure, there are three classes that access their parent's method that should be overridden. Three other detections, which include design defects, are related to the circular dependency pattern. One small example of these detections is given in Fig. 20. Between the classes class A and class B, there is a circular field access dependency. This section has complex source code; however, it can be simply illustrated at the code level as follows:

```
classA{                 classB{                 classC{
classB b;               classA a;               methodC(){
char data;              char.data;               new classA A;
methodA(){              methodB(){               new classB B;
  b.data = 'a';           a.data = 'b';}         A.b = B;
  b.methodB();}         }                        B.a = A;
}                                               }
```

Because the detected repeated structures include a design flaw, multiple places in the system are affected, and therefore, the team has planned a high-priority future task to investigate refactoring opportunities on those structures.

During the analysis, the architect suggested several quick improvements on the detected structures, to improve the quality and reduce the coupling of the system, such as removing unnecessary parallel calls and converting direct field access relations to public getter and setter method calls. The developers have also planned another future task, to examine these structures in detail and to make them more modular and standard to improve their reusability. We also asked developers about their opinion on the effect of these detections on software comprehension. Especially the newest two members of the

team stated that detected patterns improved their understanding of the whole design architecture of the software system. The architect also asked us to analyze their other projects.

The analysis process on detected identical structures within the NW-Man project took nearly six hours and spread over two days. We observe that most of the time was spent on the analysis of useful structures. After the experiments, the team gained more awareness of the architecture, and they discovered a domain-specific pattern and acquired many valuable refactoring opportunities, such as removing unnecessary duplicate parts, combining useful parts into a single library item, fixing common design defects, and improving design pattern structures. Experiments on real-world projects show that we can detect valuable design matches between projects and within the project NW-Man. Feedback from the development teams of the projects clearly expressed the benefits of intra- and inter-project design-level clone detection.

6. Discussion

In the frequent sub-graph mining phase of our approach, memory consumption and running time increase when the size of the partitions or the entire system increases (i.e., 25 GB memory usage and 253 min running time for the ArgoUML, on the exemplary hardware). To prevent the memory consumption problem from becoming worse, virtual memory and memory swapping techniques can be applied to utilize large disk systems. This approach will apparently increase the running time; however, we note that the whole detection approach is not required to run periodically in the short term during the development of a system because the design of the object-oriented software might not change very frequently. In a typical software development scenario, the whole design should have small amounts of changes at each iteration (which lasts a couple of weeks). In fact, the architect can decide when to apply the approach according to the growth of the project. However, it would be reasonable for the projects that are in their very beginning stages to apply our detection approach at the end of each iteration (3–4 weeks) to detect early problems or quality improvement opportunities. For the projects that are close to their final stages, because the growth of the code is slower, the detection approach could be applied after every two iterations (1.5–2 months). Additionally, advances in computer hardware support the applicability of our approach even when we are working on completed large projects. For example, we had a chance to repeat some of our experiments on a more powerful test environment (16 CPU with 64 cores); we observed that the running time reduced from 253 min to 34 min for the project ArgoUML.

The manual classification of the analysis phase appears to require significant effort, which might be true for a person who is not familiar with the target project. However, for developers and architects of the project, it is much easier to handle the classification. For example, in our case study about industrial projects, we could easily classify each detected structure (107 detections in total) in no more than a couple of minutes with the participation of the development team. After introducing our graph model and detection examples, the team was able to identify the meaning and the category of many structures just by considering the entity (i.e., classes and interfaces) names. Additionally, our analysis tool E-Quality [35] has several helper functionalities that support the analysis of detections, such as: a user can move into source code in Eclipse [44] editors by clicking an associated entity in the detection graph, and he or she can navigate through the detection result set on a graphical user interface.

To profit from design-level clone detection, two additional stages are necessary, namely the classification and analysis stages. In this study, we primarily focus on the detection stage. After the detection stage, we experienced that design-level classification is easier than expected; however, we observe that most of time is spent on the analysis stage to decide on appropriate refactoring actions for quality improvements that require high-level knowledge of design patterns, anti-patterns and software design quality (i.e., coupling and cohesion). Each structure that we found occurs in at least two different places of the whole architecture, so that any quality improvement activities on them will affect multiple areas of the architecture at once. We believe that most of the interesting matches are the domain-specific matches that can make sense only to the designer. Theoretically, it is also possible to explore currently unnamed design patterns by applying our approach to a very large set of object-oriented software projects.

There are also several clustering approaches that attempt to find strongly-connected software entities such as modules or services [45]. During the analysis phase, some of the popular clustering algorithms have also been tried, but they produced very large partitions that we could not handle with our test system. For example, the FastCommunity [46] graph clustering algorithm produces clusters with $|V| > 50$ for the Zest project and even much larger clusters for larger projects. There are some heuristic algorithms, such as [47], that mine large graphs without partitioning, but they currently do not operate on directed graphs, and they do not produce complete outputs, either. In addition, we observe that mining large 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 code-level similarities in a software system, such as type-1 and type-2 clones [15]. However, in our work, we focus on design (model) similarities and define exact design matches as a condition. In our tests, some of the copy–paste structures, such as in Fig. 12, were also detected by typical source code-based clone detection techniques and tools, such as CCFinder [48] and CodePro Analytix [43]; however, other design matches, such as Figs. 10, 11, 18 and many parts of the structure in Fig. 13, could not be detected by the same techniques.

In our study, we cannot include empirical comparisons with other methods because, to the best of our knowledge, there is no other tool published for the detection of repetitive identical design structures in the graph representation of object-oriented software design models within the same project (intra-project) and among different projects (inter-project).

There exist some similarity, pattern or code-based clone detection approaches that have been mentioned in the related work section. The study in [24] can be interpreted as the closest approach to our work because it also applies a graph mining technique. However, this study is not comparable with ours because it focuses on the detection of unchanged micro-architectures through different versions, while we focus on repetitive design structures within the project or between projects. However, we validate our detection quality with the help of real-world case study interviews, manual source code inspections, and the use of source code clone detection tools. We also show that some of our detections cannot be identified by using code-based clone detection tools.

7. Conclusions

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 and industrial projects, we evaluated our approach and discussed the results. The results show that we can detect many identical structures that can be manually classified as software design clones, reused design structures between projects, common design patterns, domain-specific patterns, copy–paste-modify structures or repeated design disharmonies within and between projects. Identifying these design structures can help designers to understand the high-level architecture of the object-oriented software, discovering reusability possibilities and detecting repeated design flaws that require refactoring. We plan to extend our approach by improving the performance of the graph-mining phase and adding the capability of automatic classification of the detected identical design structures.

Acknowledgements

We would like to give special thanks to software development team members of the RL-Man and NW-Man projects who participated in our analysis.

References

[1] B. Baker, On finding duplication and near-duplication in large software systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, 1995, pp. 86–95.
[2] C. Larman, Applying UML and Patterns, Prentice Hall International, 2001.
[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison–Wesley, 1995.
[4] D. Port, Derivation of domain specific design patterns, USC Center for Software Engineering, 1998.
[5] Grady Booch, Draw Me a picture, IEEE Softw. 28 (1) (2011) 6–7.
[6] L. Erlikh, Leveraging legacy system dollars for E-business, IT Pro 2 (3) (2000) 17–23.
[7] M. Hanna, Maintenance burden begging for remedy, Softw. Mag. 53 (63) (1993).
[8] S. Peeger, J. Atlee, Software Engineering – Theory and Practice, 3rd ed., Ellis Horwood, 2006.
[9] 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, in: Proc. of the Working Conference on Reverse Engineering, IEEE CS, 2011, pp. 67–76.
[10] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, W. Tichy, Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance, IEEE Trans. Softw. Eng. 28 (6) (2002) 595–606.
[11] Z. Li, S. Lu, S. Myagmar, Y. Zhou, CP-Miner: Finding copy–paste and related bugs in large-scale software code, IEEE Trans. Softw. Eng. 32 (3) (2006) 176–192.
[12] C.K. Roy, J.R. Cordy, A survey on software clone detection research, Queen's Technical Report 541, 2007, p. 115.
[13] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Trans. Softw. Eng. 32 (2006) 896–909.
[14] J. Dong, Y. Sun, Y. Zhao, Design pattern detection by template matching, in: Proceedings of the 2008 ACM Symposium on Applied Computing, Fortaleza, Brazil, 2008, pp. 765–769.
[15] R. Koschke, Survey of research on software clones, in: Proceedings of Dagstuhl Seminar 06301: Duplication, Redundancy, and Similarity in Software, 2006, p. 24.
[16] H. Liu, Z. Ma, L. Zhang, W. Shao, Detecting duplications in sequence diagrams based on suffix trees, in: Proceedings of the 13th Asia Pacific Software Engineering Conference, APSEC 2006, 2006, pp. 269–276.
[17] H. Störrle, Towards clone detection in UML domain models, in: C. Cuesta (Ed.), Proceedings of the 4th European Conference on Software Architecture: Companion Volume, ECSA 2010, ACM, New York, NY, USA, 2010, pp. 285–293.
[18] C.F.J. Lange, M.R.V. Chaudron, J. Muskens, In practice: UML software architecture and design description, IEEE Softw. 23 (2006) 40–46.
[19] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, S. Teuchert, J. Girard, Clone detection in automotive model-based development, in: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, 2008, pp. 603–612.
[20] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, T. Nguyen, Accurate and efficient structural characteristic feature extraction for clone detection, in: Proc. 12th Intl. Conf. Fundamental Approaches to Software Engineering (FASE), Springer, 2009, pp. 440–455.
[21] C.K. Roy, J.R. Cordy, R. Koschke, Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Sci. Comput. Program. 74 (7) (2009) 470–495.
[22] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, in: Proceedings of the 8th International Symposium on Static Analysis, SAS, 2001, pp. 40–56.
[23] H.A. Basit, S. Jarzabek, Detecting higher-level similarity patterns in programs, in: Proceedings of the 10th European Software Engineering Conference, ACM, 2005, pp. 156–165.
[24] A. Belderrar, S. Kpodjedo, Y. Guéhéneuc, G. Antoniol, P. Galinier, Sub-graph mining: Identifying micro-architectures in evolving object-oriented software, in: 15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 2011, pp. 171–180.
[25] ArgoUML, http://argouml.tigris.org/.
[26] U. Tekin, U. Erdemir, F. Buzluca, Mining object-oriented design models for detecting identical design structures, in: Sixth International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012, pp. 43–49.
[27] Object Management Group, Unified modeling language specification, http://www.uml.org.

[28] X. Yan, J. Han, CloseGraph: mining closed frequent graph patterns, in: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2003, pp. 286–295.
[29] B. Hendrickson, R. Leland, A multi-level algorithm for partitioning graphs, in: Proceedings Supercomputing'95, ACM Press, 1995, p. 28.
[30] B. Hendrickson, R. Leland, An improved spectral graph partitioning algorithm for mapping parallel computations, Technical Report SAND92-1460, Sandia National Laboratories, 1992.
[31] B.W. Kernighan, S. Lin, An efficient heuristic procedure for partitioning graphs, Bell Syst. Tech. J. 49 (1) (1970) 291–307.
[32] C.M. Fiduccia, R.M. Mattheyses, A linear time heuristic for improving network partitions, in: Proceedings of the 19th IEEE Design Automation Conference, 1982, pp. 175–181.
[33] B. Hendrickson, R. Leland, The Chaco User's Guide, Version 1.0, Tech. rep. SAND 93-2339, Sandia National Laboratories, Albuquerque, NM, 1993.
[34] Zest, http://www.eclipse.org/gef/zest/.
[35] U. Erdemir, U. Tekin, 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), Virginia, USA, September 2011, pp. 6–13.
[36] ParSeMiS: The parallel and sequential mining suite, http://www2.informatik.unierlangen.de/EN/research/ParSeMiS/index.html.
[37] Marc Worlein, Thorsten Meinl, Ingrid Fischer, Michael Philippsen, A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston, in: Proc. Conference on Knowledge Discovery in Database (PKDD'05), Porto, Portugal, October 2005, in: Lect. Notes Comput. Sci., Springer-Verlag, 2005, pp. 392–403.
[38] S. Valverde, R.V. Sole, Hierarchical small worlds in software architecture, Dyn. Contin. Discrete Impuls. Syst. B Appl. Algorithms 14 (S6) (2007) 1–11.
[39] YARI, http://sourceforge.net/projects/yari.
[40] JUnit, http://www.junit.org/.
[41] JFreeChart, http://www.jfree.org/jfreechart.
[42] William J. Brown, Raphael C. Malveau, W. Hays, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, Wiley, 1998.
[43] CodePro Analytix, http://code.google.com/javadevtools/codepro.
[44] Eclipse, http://www.eclipse.org.
[45] R.A. Bittencourt, D.D.S. Guerrero, Comparison of graph clustering algorithms for recovering software architecture module views, in: Proc. European Conference on Software Maintenance and Reengineering, IEEE CS Press, 2009, pp. 251–254.
[46] M.E.J. Newman, Fast algorithm for detecting community structure in networks, Phys. Rev. E 69 (2004) 066133.
[47] Michihiro Kuramochi, George Karypis, Finding frequent patterns in a large sparse graph, Data Min. Knowl. Discov. 11 (3) (2005) 243–271.
[48] T. Kamiya, S. Kusumoto, K. Inoue, CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28 (7) (2002) 645–670.