SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2010; **40**:897–916
Published online 21 July 2010 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.988

Architectural layer recovery for software system understanding and evolution

Giuseppe Scanniello*,†, Anna D'Amico, Carmela D'Amico and Teodora D'Amico

Dipartimento di Matematica e Informatica, University of Basilicata, Viale Dell'Ateneo, Macchia Romana, 85100 Potenza, Italy

SUMMARY

This paper presents an approach to identify software layers for the understanding and evolution of software systems implemented with any object-oriented programming language. The approach first identifies relations between the classes of a software system and then uses a link analysis algorithm (i.e. the Kleinberg algorithm) to group them into layers. Additionally to assess the approach and the underlying techniques, the paper also presents a prototype of a supporting tool and the results from a case study. Copyright © 2010 John Wiley & Sons, Ltd.

Received 28 August 2009; Revised 7 May 2010; Accepted 10 May 2010

KEY WORDS: software understanding; architecture recovery; link analysis; static analysis

1. INTRODUCTION

The architecture of a software system provides a vocabulary to share a common understanding on the design and to rationalize and understand the developers' decisions. Furthermore, the architecture of a software system should be defined before its implementation [1] and updated to reflect changes occurring during the maintenance phase. Unfortunately, very often the architecture documentation is neither updated nor documented at all [2], thus making it only available in the minds of the developers. In these cases, reverse engineering tools and approaches have to be employed to create the architecture design or to align it with the actual implemented software system [3].

Software architecture recovery represents a longstanding and relevant research topic. In fact, a number of approaches, techniques, and tools have been designed and developed to support the architecture recovery and the remodularization of legacy software systems [4–7]. The greater of these approaches is based on clustering techniques [8–10] on large software systems to group source files into clusters; hence, that files containing source code implementing similar functionality are placed in the same cluster. On the other hand, source files are placed in different clusters in case they implement different functionality.

Clustering-based approaches very often do not consider the fact that the architecture of a software system has a hierarchical structure, where subsystems are grouped into layers (e.g. classic 2 or 3 tiered software systems). The lowest layer accesses the persistent data (i.e. database or file), whereas upper layers deal with user interface issues or implement services on top of the lowest layer. Although layers should be spread on well-defined physical nodes (e.g. servers), it could

*Correspondence to: Giuseppe Scanniello, Dipartimento di Matematica e Informatica, University of Basilicata, Viale Dell'Ateneo, Macchia Romana, 85100 Potenza, Italy.
†E-mail: giuseppe.scanniello@unibas.it

happen that parts of two or more layers are on the same node, thus making them indistinguishable in practice. Similarly, this could happen also on centralized/desktop software systems where classes should be spread on physical components (e.g. packages). The indistinguishable problem may be due to maintenance operations (e.g. adaptive [11]) that deteriorate the software system original structure [12].

The recovery of a hierarchical structure of a given software system is hard and is considered useful for long-term maintenance operations [13]. For example, it could be useful to understand the coherence between design documents of large-scale systems (e.g. telecommunication systems) and the software architectures they actually implement [14]. Furthermore, the implemented software architecture may be recovered partitioning the identified software layers. Manual, automatic, or semiautomatic procedures, such as one of the approaches presented in [6, 10], could be employed. This will enable the identification of software components that have the same level of abstraction, but implement different functionality/services.

This paper presents an approach to identify software layers for the understanding and evolution of existing object-oriented software systems. The approach is semiautomatic and is based on a process that first identifies relations between classes and then decomposes the system into layers using the Kleinberg algorithm [15]. In particular, the algorithm determines for each class authority and hub values. A high authority value indicates that the class is used within the software system by many different classes. On the other hand, a high hub value shows that a class uses a huge number of other classes. Classes with high authority value and low hub value are grouped in the lowest layer and the classes with low authority value and high hub value in the highest layer. The remaining classes are grouped in the middle layer. The algorithm may be applied only on the classes of the middle layer to further decompose the software system into layers. This represents a critical point for our approach as it is required that the software engineer has some knowledge on the software system understudy.

In order to automate the approach and to facilitate its adoption, we have also developed a prototype of a supporting system. The approach and the prototype have been applied in a case study involving Java software systems and is presented in this paper, which has proved their feasibility. The approach presented here is based on the idea that an experienced software engineer is usually able to recover the layered architecture of a software system better than an automatic/semiautomatic procedure. However, this process is time consuming, tedious, and hard, thus justifying the definition first and the use then of supporting tools to make the recovery of software layers easier.

The presented paper is based on the work previously presented in [16]. In particular, compared with it, in this paper we have enhanced the approach and the supporting tool and their description has been improved. Furthermore, the case study has been extended and a deeper discussion of the results has been provided. The remainder of this paper is organized as follows. In Section 2, we provide some useful notions to better understand the approach, which is presented and discussed in Section 3. The prototype of a supporting system implementing the approach is described in Section 4. The design of the case study and the discussion of the obtained results are presented in Sections 5 and 6, respectively. Section 7 discusses a number of works related to the architectural recovery, software architecture evolution, and application of graph theory to object-oriented software systems. The paper is concluded discussing final remarks and drawing possible future directions for our work.

2. BACKGROUND

In the following subsections, we provide some useful notions to better comprehend the algorithm to decompose a software system into layers.

2.1. The Kleinberg algorithm

The Kleinberg algorithm [15] has been originally developed for extracting information from the link structures of hyperlinked environments. However, the algorithm effectiveness has been assessed

DOI: 10.1002/spe

on a variety of contexts on the web focusing on the use of links for analyzing collections of pages relevant to search topics, and to discover the most 'authoritative' pages on such topics. The key idea is that the hyperlinks encode a considerable amount of latent human judgment. The creation of a link in a web page to another page indicates that the developer in some measure conferred authority on the page. However, the quality of a page is not only related to the number of pages that point to it (i.e. hubs), but also to the quality of these hubs.

The algorithm determines two values for a page: the authority and hub. An authority value is computed as the sum of the scaled hub values that point to that page. Conversely, a hub value is the sum of the scaled authority values of the pages it points to. Authorities and hubs have a mutually reinforcing relationship: if a page points to many pages with large authority values, then it should receive a large hub value; and if this page is pointed to by many pages with large hub values, then it should receive a large authority value.

2.2. Software system representation

The static aspects of the architecture of an object-oriented system are today modeled by employing one or more Unified Modeling Language (UML) [17] diagrams. Among all diagrams, the most common representation is the class diagram, which shows the classes of a software system, their methods and attributes, and most importantly the relationships between them. Several kinds of relationships among classes can be shown. The main are:

- *Inheritance/Realization*. A class extends/implements a class/interface;
- *Aggregation/Composition*. A class is part of another class. The composition relationship is a special case of aggregation with stronger constraints;
- *Association*. A class holds a stable reference toward another class. This relationship subsumes aggregation.

The inheritance and realization relationships are easily identified at the syntactic level performing static analysis of the source code. On the other hand, aggregation and composition are almost indistinguishable. These relationships can be recovered from the code when an attribute has a way to reference an object of another class, either by means of a Java reference (i.e. a pointer) or a container (e.g. an array, a list, or a hash table). It is worth mentioning that typical limitations of reverse engineering tools could regard the target class of associations that is not always available in the source code. For example, if a variable implementing an association is declared of interface type, the referenced class can be only determined at run time.

Once a class diagram is recovered from an existing object-oriented source code it could be mapped onto a graph [18, 19], where vertices (or nodes) and edges represent the classes and a selected type of relationship (e.g. association, generalization, composition, etc.), respectively.

3. THE APPROACH

In this paper, we propose an approach based on a clustering algorithm, which uses the Kleinberg algorithm, to identify layers within software systems implemented using any object-oriented programming language. The algorithm takes as input a software system and produces for each class two values: authority and hub. These values are then used to decompose the software system in three ordered layers, which are composed of classes providing related services, possibly realized using classes from another layer. Each layer can depend only on lower-level layers and has no knowledge of the layers above it. The layer that does not depend on any other layer is called the bottom layer, whereas the layer that is not used by any other layer is called the top layer. Finally, disconnected classes get authority and hub values both equal to 0. Table I summarizes how classes are grouped into software layers considering their authority and hub values. The motivation behind the mapping between the layers and the authority and hub values are summarized as well.

In case the implemented architecture of a software system has more than three layers, the middle layer may be further decomposed into software layers. To this end, the approach uses again

DOI: 10.1002/spe

Table I. Rules for layering a software system.

Layer	Authority and hub values	Meaning
Top	(0, #)	Classes with authority value equal to 0 and hub value larger than 0 are placed in the top layer as they are mainly never used by other classes.
Middle	(#, #)	Classes with authority and hub values different from 0 are placed in the middle layer as they have a mutually reinforcing relationship with authorities and hubs classes.
Bottom	(#, 0)	Classes with authority value larger than 0 and hub value equal to 0 are placed in the bottom layer as they are mainly used by the classes of the other layers.
Disc	(0, 0)	In case the values of authority and hub are equal to 0, the classes have no relationships with other classes.

```
Layering (SoftwareSystem S, int k)
1.  G = getDirectedGraph(S);
2.  L       // this is the data structure used to contain the identified software layers
3.  count = 1;
4.  while (true) {
5.    auth_0 = {1, 1, …, 1}       // the dimensions of auth_0 and hub_0 are equal to |V|
6.    hub_0 = {1, 1, …, 1}
7.    for (i=1..k) {
8.      calculateAuthorityValues(auth_i, hub_i-1);
9.      calculateHubValues(hub_i, auth_i-1);
10.     normalize(auth_i, hub_i);
11.   }
12.   T = getTopLayer(G, auth_k, hub_k);
13.   M = getMiddleLayer(G, auth_k, hub_k);
14.   B = getBottomLayer(G, auth_k, hub_k);
15.   L.add(T, count);
16.   L.add(B, count);
17.   if (M has to be refined){
18.     if (|M| = |L.getM(count – 1)|) and (count > 1)){
19.         L.add(M, count);
20.         return L;
21.     }else{
22.         G=G.getSubGraph(M);
23.         count = count + 1;
24.     }
25.   }else
26.     L.add(M, count);
27.     return L;
28.   }
29. }
30. return L;
```

Figure 1. Pseudo code of the proposed algorithm.

the Kleinberg algorithm to recalculate the authority and hub values on the sub graph obtained considering only the classes of the middle layer. This means that Kleinberg is applied only to the graph obtained removing the disconnected classes and the classes of the top and bottom layers. Edges from and to the classes of the top and bottom layers are removed as well.

Figure 1 shows the pseudo code of the algorithm underlying the proposed approach. This algorithm starts (instruction 1) recovering the relationships among the software entities (i.e. packages, classes, interfaces, fields, and methods) of a software system to build a view of the corresponding UML class diagram. In particular, we only consider the classes as software entities and relationships among them (i.e. association, inheritance, realization, aggregation, and composition). Successively,

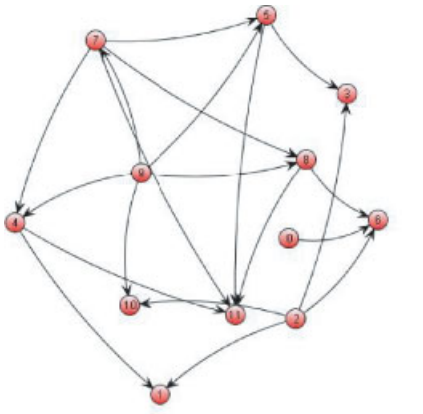
DOI: 10.1002/spe

the diagram is mapped onto a directed graph $G=(V, E)$ of the object-oriented system. V is the set of vertices and E is the set of edges, where an edge is an ordered pair of vertices in V. The set of vertices corresponds to the identified classes, whereas E represents relationships between the classes. In particular, the following are the considered relationships and how they are mapped onto G:

(i) *Inheritance/Realization*—in case a class C extends/implements a class/interface D, a direct edge (C, D) is added to E.
(ii) *Aggregation/Composition*—if a class L is a composition or an aggregation of a class M, a direct edge (L, M) is added to E.
(iii) *Association*—in case there is an association between the classes A and B with a direction from A to B, an edge (A, B) is added to E.

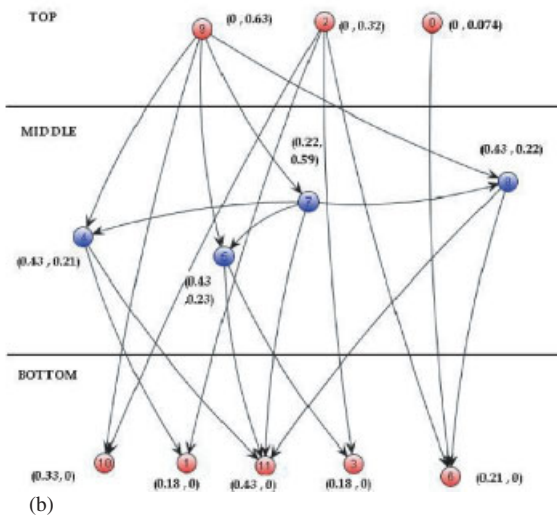
These relationships are semi-automatically identified using source code static analysis. This presents some limitations as in object-oriented software systems some relationships are obvious enough (e.g. inheritance/realization), whereas others may present some concerns since they can only be determined at run time (i.e. association). This is the motivation for which the recovery of a directed graph from an existing object-oriented software system is not automatic in the approach presented here. However, future work will be devoted to extend the approach to get relationships that the static analysis is not able to recover. Note also that the recovery of the graph obtained from a given software system is not explicitly considered in the pseudo code of the algorithm for readability reasons. In fact, we suppose that it is externally obtained by executing the function *getDirectedGraph* (see instruction 1).

To show the behavior of the proposed algorithm, an example of its application is shown in Figure 2. In particular, Figure 2(a) shows a graph excerpt of a software system analyzed in



Id	Class Name
0	*it.unibas.questionari.vista.TreeModelQuestionario.java*
1	*it.unibas.questionari.modello.Quesito*
2	*it.unibas.questionari.vista.PannelloPrincipale*
3	*it.unibas.questionari.modello.Risposte*
4	*it.unibas.questionari.controllo.AzioneCaricaQuesito*
5	*it.unibas.questionari.controllo.AzioneCaricaRisposte*
6	*it.unibas.questionari.modello.Questionario*
7	*it.unibas.questionari.controllo.Controllo*
8	*it.unibas.questionari.controllo.AzioneCaricaQuestionario*
9	*it.unibas.questionari.vista.Vista*
10	*it.unibas.questionari.modello.Modello*
11	*it.unibas.questionari.Costanti*

(a)



(b)

	0	1	2	3	4	5	6	7	8	9	10	11
0	-						*I/R*					
1		-										
2		*A/C*	-	*A/C*			*A/C*				*A*	
3				-								
4		*A*			-							*A*
5				*A*		-						*A*
6							-					
7					*A/C*	*A/C*		-	*A/C*			*A*
8							*A*		-			*A*
9					*A/C*	*A/C*		*A*	*A/C*	-	*A*	
10											-	
11												-

(c)

Figure 2. Original directed graph and the corresponding layered graph.

DOI: 10.1002/spe

the case study (i.e. Generate Questionnaire). The kinds of relationships (A/C—Aggregation or Composition; I/R—Inheritance/Realization; A–Association) between all the pairs of nodes of the graph of Figure 2(a) are shown in Figure 2(c). The direction of each relationship is specified by the edge orientation (see Figure 2(a)).

The instructions from line 5 to line 10 implement the Kleinberg algorithm as presented in [15]. In particular, new authority and hub values for a given vertex/class p are calculated as follows:

$$auth(p)_i = \sum_{(p,q)\varepsilon E} hub(q)_i$$

$$hub(p)_i = \sum_{(p,q)\varepsilon E} auth(p)_i$$

Successively, the authority values are normalized; hence their squares sum to 1. Similarly, even the hub values are normalized; hence their squares sum to 1. This normalization process is performed for k times (i.e. the convergence value of the Kleinberg algorithm). Although the final hub/authority values should be determined after infinite repetitions of the algorithm, Kleinberg in [15] experimentally shows that the convergence is quite rapid ($k=20$ is the worst convergent value for the algorithm). As we applied the Kleinberg algorithm in a different context, we tried different values of k for the software systems considered in the case study. We observed that $k=20$ (the value used in our experimentation) is sufficient to let the algorithm become stable. In fact, the choice of larger values of k does not significantly impact on the authority and hub values. This confirms the results presented in [15].

The values of authority and hub for each class are stored in the vectors *auth_k* and *hub_k*, respectively. These values are successively used to group the vertices/classes in three sets T, M, and B. The first set will contain the vertices of the top layer, whereas the vertices of the middle layer are inserted in M. Finally, B will contain the vertices of the bottom layer. The layers top, middle, and bottom identified by the algorithm on the software system used as an example are shown in Figure 2(b). The values of authority and hub of the node of the recovered layered architecture are shown as well.

The classes of the top layer (e.g. classes implementing the graphical user interface (GUI)) use classes to provide services to the users and hence they will obtain high hub values and low authority values. The classes of the bottom layer may be used to manage persistent data or to access network services. In this case, the algorithm will produce high authority values and low hub values. Finally, the classes of the middle layer will obtain high values of authority and hub. In case the classes of the middle layer have different responsibilities, the software engineer may decide to reuse the algorithm to further decompose them into layers (see instruction of line 17). The rationale for applying the Kleinberg algorithm more than once relies on the fact that we could be interested in detecting classes of the data layer that have different abstraction levels (i.e. indirectly access persistent data). The same holds for classes at the presentation layer. Regarding the example presented to describe how approach works, Figure 2(b) shows the software layer obtained by applying the Kleinberg algorithm only once.

Classes with values of authority and hub both equal to 0 can be identified at each iteration of the while cycle. The first time, the classes have no relationships with other classes; hence, they can be considered as dead or unused classes. At the subsequent application of the Kleinberg algorithm, this means that the classes with values of authority and hub equal to 0 were connected with the other classes of the middle layer identified at previous application, while were disconnected at the current application of the algorithm. As these classes are not dead classes, they have to be associated with one of the identified layers accordingly with the values of authority and hub computed at the previous application. In particular, in case a class obtained an authority value larger than the hub value it is placed in the bottom layer. Conversely, it is placed in the top layer.

The proposed example shows that the approach is able to properly divide the considered classes into three layers. In fact, the classes of the top layer belonged to the *vista* package (view in English), whereas the classes of the *controllo* package (controller in English) were placed in the middle layer. On the other hand, the classes of the *modello* package (model in English) were grouped in

DOI: 10.1002/spe

the bottom layer. Note that the vertex with the label 11 (i.e. the class *Costanti*) has been placed in the bottom layer since it is used to hold constant values. Concluding, the approach was able to correctly partition the analyzed classes into the top, middle, and bottom layers.

4. SUPPORTING TOOL

The approach has been defined to divide into layers software systems implemented using any object-oriented programming language. Although the approach is general, we implemented a prototype of a supporting tool to analyze Java software systems.

This prototype has been implemented in Java. It provides a GUI component to allow selecting the software system to analyze. This component also enables the visualization of the graph corresponding to the object-oriented software system to analyze (see Figure 3(a)). To provide the most suitable representation of the graph more layouts are available, i.e. circle, spring, Fruchterman–Reingold, and Kamada–Kawai. To visualize the graph representations of a software system, the tool prototype uses the Java Universal Network/Graph (JUNG) library[‡].

The tool also enables the software engineer to move and color vertices and to get the class name that it represents. Also, the computed hub and authority values are shown within the GUI component. These values are presented in terms of a list containing for each class its name and the corresponding values of authority and hub (see on the left-hand side of Figure 3(a)).

To get directed graphs associated with the static structure of software systems, the prototype integrates Dependency Finder[§] (an open-source suite of tools for analyzing Java code). In particular, this suite is used to perform static analysis to determine relationships among classes. It extracts the dependency graph, which is represented in terms of an XML file that reports three kinds of dependency: feature (i.e. class attributes, constructors, and methods) to feature, feature to class, and class to class. Successively, the class relationships shown in Section 3 are used to map the UML class diagram of the software system understudy onto the corresponding directed graph.

The graph is stored in a GraphML file (an XML-based format for graphs). The directed graph of a software system is in turn used by the software component implementing our algorithm to recover the layered architecture of the understudy software system. Figure 3(b) shows an excerpt of the GraphML file of one of the JHotDraw version we have used in the case study. Note that directed graph representations may be modified by the software engineer if necessary (i.e. adding and/or removing edges and/or vertices). Edge could be added in case the prototype has not been able to detect relationships (e.g. associations). Future work will be devoted to extend the approach and the supporting tool with dynamic analysis technique, thus reducing as much as possible the software engineer involvement to get an accurate representation of the static structure of a given software system.

5. EXPERIMENTAL SETTING

In this section, we describe the design underlying the empirical investigation conducted to assess the feasibility of the approach and the system prototype.

5.1. Research questions

To assess the quality of the layers identified by our approach, we consider two criteria that are summarized as follows:

(i) *Authoritativeness*—It regards the resemblance between the software layers identified by the tool and an authoritative partition (i.e. the decomposition performed by a

[‡]http://jung.sourceforge.net/.
[§]http://depfind.sourceforge.net/.

DOI: 10.1002/spe

(a)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
  <graph xmlns="" edgedefault="directed">
    <node id="n0" name="CH.ifa.draw.applet.DrawApplet" />
    <node id="n1" name="CH.ifa.draw.applet.SleeperThread"
.....
    <edge id="C2C_0" source="n0" target="n67" />
    <edge id="C2C_1" source="n0" target="n187" />
.....
    <edge id="C2C_3" source="n2" target="n67" />
    <edge id="C2C_4" source="n2" target="n187" />
.....
    <edge id="C2F_4466" source="n205" target="n202" />
    <edge id="C2F_4457" source="n205" target="n199" />
.....
    <edge id="C2F_4453" source="n205" target="n67" />
  </graph>
</graphml>
```

(b)

Figure 3. Tool snapshot and directed graph representation.

developer/architect). The layers identified by applying the approach should resemble as much as possible the groups of classes within the authoritative partition.

(ii) *Stability*—It concerns the persistence of the layer structure of two consecutive versions of an evolving software system [20, 21]. Similar layers should be produced in case of small and incremental changes between successive versions.

According to these criteria, two research questions have been formulated and will be verified in the following:

Q1: Does the software layers identified by the tool resemble an authoritative partition?
Q2: Is the layer structure identified by the tool stable?

It is worth mentioning that the research question Q2 is only useful to claim that the proposed approach is stable (i.e. it gives consistent mappings for different consecutive versions of a system). On the other hand, Q2 provides an indication on the correctness and completeness of the identified layers.

5.2. Application selection

To address the defined research questions, we selected seven Java applications. Four of these were designed and developed by four students of the Bachelor Program in Computer Science as laboratory activities of the programming language courses: Standard Object Oriented and/or Advanced Object Oriented. We selected these software systems to use the knowledge of the original developers/maintainers in order to evaluate the resemblance between the software layers identified by the tool and authoritative partitions (i.e. to address the research question Q1). These software systems have been also selected as they are based on the Model View Controller (MVC) architectural model. It is also worth mentioning that the four students who used to get the authoritative partitions (i.e. the original developers/maintainers) are last year Master students in Computer Science at the University of Basilicata, and took the Bachelor degree from the same University two/three years before conducting the experimentation presented in this paper. These students were not a part of the development team of the supporting tool presented here. Furthermore, they knew neither the objective of the experiment nor its research questions.

The name of the selected software systems and some descriptive statistics are summarized in Table II. In particular, the system names are reported in the first column, whereas the number of analyzed source files is shown in the second column. The numbers of classes and their relationships (i.e. inheritance/realization, aggregation/composition, and association) are reported in the third and fourth columns, respectively. Finally, the number of line of code is shown in the last column.

DOI: 10.1002/spe

Table II. Software systems used to investigate Q1.

System	Files	Classes (|V|)	Relat. (|E|)	LOC
Minesweeper	45	45	177	2408
Poker	27	27	109	1193
Generate questionnaire	30	30	118	1485
Car Management System	29	29	308	2484

Table III. Analyzed software systems.

System	Vers.	Files	Classes (|V|)	Relat. (|E|)	LOC
JHotDraw	9	160–298	171–300	1728–2782	9378–20375
JEdit	24	250–311	263–323	2285–4901	24319–29326
JFreeChart	33	80–430	81–434	994–5607	6765–14635

In the following, we briefly describe the analyzed software systems:

- *Minesweeper* is a game where a player has to locate all mines (bombs) in a mine field as quickly as possible by uncovering squares that do not contain a mine.
- *Poker* is a multiplayer game that implements the traditional Poker card game.
- *Generate questionnaire* enables the generation of assessment or self-assessment questionnaires from a repository of open and closed questions. This system also enables the management of the repository and the stored questions.
- *Car Management System* enables the management of cars and owners of a given car shop. It also sends e-mails to the owners in case of planned maintenance operations or promotions.

On the other hand, the research question Q2 has been investigated studying three open-source software systems. These systems have been selected according to their availability on the web and their size and to their architecture style (i.e. layered architecture). Furthermore, they are also well known and well studied in the reverse engineering field [20, 22], thus making them interesting case studies. Some statistics on these systems are shown in Table III. In particular, the first column of this table shows the names of the analyzed software systems, whereas the number of studied versions is reported in the second column. The third column shows the maximum and minimum numbers of analyzed source files among the distributions of the analyzed software. Similarly, for each system, the maximum and minimum numbers of classes and their relationships are reported in the fourth and fifth columns, respectively. Finally, the minimum and maximum number of line of code (among all the studied distributions of the considered software system) is shown in the last column. In the following, we briefly describe the analyzed software systems:

- *JHotDraw*[¶] is a Java GUI framework for technical and structured graphics.
- *JEdit*[‖] is a text editor for programming with an extensible plug-in architecture.
- *JFreeChart*[**] supports the visualization of bar charts, pie charts, line charts, scatter plots, histograms, simple Gantt charts, bubble plots, and more.

5.3. Measures

To assess the authoritativeness of the approach, we used the harmonic mean (i.e. F-measure) of precision and recall, which is defined as follows:

$$F\text{-}measure = 2 * \frac{Precision * Recall}{Precision + Recall}$$

[¶] http://www.jhotdraw.org/.
[‖] http://www.jedit.org/.
[**] www.jfree.org/.

DOI: 10.1002/spe

The rationale for using this mean relies on the necessity of evaluating the results with respect to a trade-off between correctness and completeness of the identified layers. In fact, precision is intended as a measure of correctness, whereas recall is a measure of completeness:

$$Recall = \frac{\sum_{i=1}^{\#layers} |L_{i,gold} \cap L_{i,tool}|}{\sum_{i=1}^{\#layers} |L_{i,gold}|}$$

$$Precision = \frac{\sum_{i=1}^{\#layers} |L_{i,gold} \cap L_{i,tool}|}{\sum_{i=1}^{\#layers} |L_{i,tool}|}$$

$L_{i,tool}$ represents the set of classes of the layer i identified by the tool. $L_{i,gold}$ contains the set of classes of the layer i of the authority partition identified by developers/maintainers (even software engineers with a deep knowledge of the system may be employed). The developers/maintainers were not aware of the research questions of our experiment when they were asked to indicate the authority partitions of the analyzed software systems. They were however informed on the study and its research questions when they provided the authority partitions. Let us note that precision, recall, and F-measure assume values ranging between 0 and 1. The larger are the values that these measures assume, the more authoritativeness the identified layers have.

To assess the approach stability, we have used the implementation of the optimal algorithm of the MoJo distance [23]. Indeed, we have employed the mono-directional version of this measure on all the pairs of consecutive versions of the four open-source software systems. Let A_i and A_{i+1} be the layers automatically identified by the tool on the versions i and $i+1$ of a given software system, the MoJo distance between A_i and A_{i+1} is computed as the minimum number of join and move operations to turn A_i into A_{i+1}. The less the MoJo distance, the more the stability of the layer structure of the two analyzed consecutive versions. Note that the MoJo distance is computed at class level as A_i and A_{i+1} contain sets of classes (i.e. the recovered software layers). Let us also note that this measure may threaten the assessment of the approach stability. In fact, it assumes that a node appears in both the consecutive versions of a given system. Hence, in case a class is added to a given version it does not contribute to compute the stability value. Hence, the less the number of new added classes, the better the used measure works.

6. RESULTS

In the presented empirical investigation, we made two assumptions regarding the directed graph enhancement of the static structure of an understudy software system and the number of layers to recovery. In particular, to reduce the possibility that the human factors may condition the approach results, the directed graph of the static structure of each analyzed software system has not been manually enhanced. Regarding the number of layers, we have considered only the ones recovered at first application of the link analysis algorithm. The rationale for making these assumptions relies on the fact that we would like to answer our research questions using the approach in automatic way, thus avoiding as much as possible that human factors may condition the results.

In the following subsections, we show and discuss the achieved results according to the defined research questions. This section is concluded presenting and discussing the threats that could affect the obtained results. Open issues and possible future directions for our research will be presented as well.

6.1. Research Question Q1: Authoritativeness

Table IV summarizes the results obtained by applying once the Kleinberg algorithm on the software systems selected to assess the authoritativeness of the layers recovered by the tool prototype. In particular, the first column shows the name of the understudy system, whereas the precision and recall values are reported in the second and third columns, respectively. The fourth column

DOI: 10.1002/spe

Table IV. Authoritativeness results.

System	Precision	Recall	F-measure	# nodes t	# nodes m	# nodes b	Distribution of nodes within the layers (%) t	Distribution of nodes within the layers (%) m	Distribution of nodes within the layers (%) b
Minesweeper	0.78	0.69	0.73	7	22	16	15.5	48.9	35.6
Poker	0.89	0.62	0.73	1	21	5	3.7	77.8	18.5
Generate questionnaire	0.75	0.62	0.68	4	12	14	13.3	40	46.7
Car Management System	0.72	0.63	0.67	6	14	9	20.7	48.3	31

Table V. JHotDraw results.

Version	Stability	# nodes t	# nodes m	# nodes b	Distribution of nodes within the layers (%) t	Distribution of nodes within the layers (%) m	Distribution of nodes within the layers (%) b
5.2	—	29	118	22	17.2	69.8	13
5.3	6	42	137	26	20.5	66.8	12.7
5.4b1	9	59	186	47	20.2	63.7	16.1
5.4b2	1	59	186	48	20.1	63.5	16.4
6.0b1	0	59	186	48	20.1	63.5	16.4
7.0.7	0	51	134	9	26.3	69.1	4.6
7.0.8	3	63	156	11	27.4	67.8	4.8
7.0.9	0	64	168	16	25.8	67.7	6.5
7.1	5	62	175	16	24.5	69.2	6.3

shows the F-measure value. The fifth and sixth columns report the number of nodes of each identified layer (t=top, m=middle, b=bottom) and the distribution of the nodes in each layer (i.e. the percentage of nodes within the layers with respect to the total number of analyzed nodes), respectively.

The obtained results indicate that the approach enabled to get good F-measure values, thus positively answering the research question Q1. To further corroborate this result, the obtained values of precision and recall have been analyzed as well. In particular, the precision and recall values indicate that the approach exhibits a good ability in correctly and completely identifying software layers, respectively. Generally, we can affirm that the approach is conservative since the achieved results are more correct than complete. In other words, the precision values are larger than the recall values for all the considered software systems.

Despite the encouraging results, a reader may object to the fact that the number of identified layers may strongly affect the possibility of positively drawing any conclusions. Indeed, we did not apply the link analysis algorithm more than once due to the nature of the considered software systems and their size. Future work will be, however, devoted to assess the effect of applying the Kleinberg algorithm on larger software systems.

As a further analysis, we tried to identify a pattern between the nodes distribution across the layers and the results. This analysis has not indicated any relation between the precision, recall, and F-measure values and the node distribution across the layers t, m, and b. This point will be further investigated considering a larger data set. To this end, software systems with different sizes and implementing different functionalities should be considered and analyzed.

6.2. *Research Question Q2*: *Stability*

The results obtained on JHotDraw, JEdit, and JFreeChartat are summarized in Tables V–VII, respectively. The first column of each table shows the studied versions of the considered software systems, whereas the stability values are reported in the second column. Note that the first row

DOI: 10.1002/spe

Table VI. JEdit results.

Version	Stability	# nodes			Distribution of nodes within the layers (%)		
		t	m	b	t	m	b
2.3pre2	—	30	92	129	12	36.6	51.4
2.3pre3	0	31	93	132	12.1	36.3	51.6
2.3pre4	0	32	94	135	12.3	36	51.7
2.3pre5	2	31	98	138	11.6	36.7	51.7
2.3pre5-2	0	31	98	138	11.6	36.7	51.7
2.3pre6	6	30	103	132	11.3	38.9	49.8
2.3pre7	2	28	105	130	10.6	39.9	49.5
2.3final	1	28	106	129	10.6	40.4	49
2.4final	2	31	109	138	11.2	39.2	49.6
2.4.1	0	31	109	138	11.2	39.2	49.6
2.4.2	0	31	109	138	11.2	39.2	49.6
2.5final	0	31	109	138	11.2	39.2	49.6
2.5.1	0	31	109	138	11.2	39.2	49.6
2.6final	0	31	109	138	11.2	39.2	49.6
3.0	5	46	193	14	18.2	76.3	5.5
3.0.1	0	46	193	14	18.2	76.3	5.5
3.0.2	1	45	194	14	17.8	76.7	5.5
3.1	7	47	187	25	18.1	72.2	9.7
3.2	6	49	195	28	18	71.7	10.3
3.2.1	0	49	195	28	18	71.7	10.3
3.2.2	1	48	196	28	17.6	72.1	10.3
4.0	10	54	222	33	17.5	71.8	10.7
4.0.2	0	54	222	33	17.5	71.8	10.7
4.0.3	0	54	222	33	17.5	71.8	10.7

of each table (i.e. the first distribution version of the studied software system) does not show any value. This is due to the measure used to assess the stability of the approach. Finally, the distribution of the nodes (i.e. the percentage of nodes within the layers with respect the total number of analyzed nodes) within the layers t, m, and b is reported in the last column.

The values of the MoJo distance obtained on the consecutive versions of the studied software systems allow us to positively answer the research question Q2, namely the layer structure identified by applying the approach can be considered as stable. In other words, the approach gives consistent mapping for different consecutive versions of the analyzed software systems. In fact, the obtained partitions are not influenced by small and incremental changes between consecutive versions as the MoJo values were mostly 0. The stability values are slightly greater than 0 in the other cases. To better comprehend this point, we also manually analyzed the differences between the versions of the software systems that produced stability values larger than 0. We noted that this was generally due to the execution of refactoring operations. For example, given two consecutive versions i and $i+1$ of the same software system, if an incoming edge in the version $i+1$ is added to a class (originally placed in the top layer when applying the approach on the version i), this class is then placed in the middle layer.

The stability results may be strongly influenced by applying the link analysis algorithm more than once. Accordingly, we applied twice the link analysis algorithm on each software system and then we analyzed the results to get an indication on whether the number of automatically identified layers may condition the stability results. For brevity reasons, we only report here the results obtained on JHotDraw. These results are summarized in Table VIII. In particular, the first column shows the distribution versions, whereas the stability values are reported in the second column. The third column shows the number of nodes of each identified layer (i.e. t1, t2, m, b1, and b2). The distribution of the nodes within the layers is reported in the last column. Even in this case, the data analysis suggests that the approach is able to detect major changes of the layered architecture of evolving software systems.

DOI: 10.1002/spe

Table VII. JFreeChart results.

		# nodes			Distribution of nodes within the layers (%)		
Version	Stability	t	m	b	t	m	b
0.5.6	—	32	48	1	39.5	59.3	1.2
0.6.0	5	28	49	3	35	61.2	3.8
0.7.0	0	33	59	4	34.4	61.4	4.2
0.7.1	0	40	67	4	36	60.4	3.6
0.7.2	4	39	67	2	36.1	62	1.9
0.7.3	0	39	67	3	35.8	61.4	2.8
0.7.4	2	39	69	3	35.1	62.2	2.7
0.8.0	1	39	69	3	35.1	62.2	2.7
0.8.1	1	43	79	18	30.7	56.4	12.9
0.9.0	4	35	69	14	29.7	58.4	11.9
0.9.1	1	35	69	14	29.7	58.4	11.9
0.9.2	0	35	73	14	28.7	59.8	11.5
0.9.3	5	63	107	52	28.4	48.2	23.4
0.9.4	8	67	122	51	27.9	50.8	21.3
0.9.5	8	76	151	57	26.8	53.1	20.1
0.9.6	0	76	151	59	26.6	52.8	20.6
0.9.7	5	85	160	66	27.3	51.5	21.2
0.9.8	0	83	164	67	26.4	52.3	21.3
0.9.9	5	94	154	70	29.6	48.4	22
0.9.10	1	87	153	69	28.2	49.5	22.3
0.9.11	1	89	161	71	27.7	50.2	22.1
0.9.12	6	91	164	75	27.6	49.7	22.7
0.9.13	1	93	166	76	27.8	49.5	22.7
0.9.14	2	99	173	76	28.4	49.8	21.8
0.9.15	0	99	177	78	28	50	22
0.9.16	1	100	180	82	27.6	49.7	22.7
0.9.17	4	103	186	86	27.5	49.6	22.9
0.9.18	0	105	187	88	27.6	49.2	23.2
0.9.19	3	108	191	95	27.4	48.5	24.1
0.9.20	0	108	191	96	27.3	48.4	24.3
0.9.21	2	110	193	97	27.5	48.2	24.3
1.0.0-pre1	5	115	204	105	27.1	48.1	24.8
1.0.0-pre2	7	115	211	108	26.5	48.6	24.9

Table VIII. Results obtained by applying Kleinberg twice on JHotDraw.

		# nodes					Distribution of nodes within the layers (%)				
Version	Stability	t1	t2	m	b1	b2	t1	t2	m	b1	b2
5.2	—	29	18	81	22	19	17.2	10.7	47.9	13	11.2
5.3	7	42	23	90	26	24	20.5	11.2	43.9	12.7	11.7
5.4b1	10	59	26	144	47	16	20.2	8.9	49.3	16.1	5.5
5.4b2	1	59	27	120	48	39	20.1	9.2	41	16.4	13.3
6.0b1	0	59	27	120	48	39	20.1	9.2	41	16.4	13.3
7.0.7	0	51	41	83	9	10	26.3	21.1	42.8	4.6	5.2
7.0.8	3	63	43	92	11	21	27.4	18.7	40	4.8	9.1
7.0.9	1	64	45	102	16	21	25.8	18.1	41.1	6.5	8.5
7.1	9	62	47	106	16	22	24.5	18.6	41.9	6.3	8.7

6.3. Further analysis

We also manually analyzed the layers obtained by applying the approach on the four systems on which the authoritativeness of the approach has been assessed. This was due to investigation of the reasons behind the identification of false positives and false negatives. For example, false

DOI: 10.1002/spe

negatives were introduced because some classes that should be placed in the top layer have incoming edges although they should not have and are then placed in the middle layer. Similar considerations can be made about the bottom layer. On the other hand, false positive were detected in case classes implementing the application logic of a given system do not have outgoings edges.

Although the open-source software systems have been selected to assess the approach stability, we also conducted a preliminary analysis on JFreeChart to get an indication on the quality of the layers recovered by the tool. The motivation behind the selection of this software system is that on the early eight analyzed versions we obtained bottom layers, whose size was smaller than the bottom layers recovered by analyzing the other versions of this system (see Table VII). As this may indicate a drawback for the approach, we manually analyzed the versions of JFreeChart from 0.5.6 to 0.8.0. This analysis revealed that all these distributions included only a few numbers of classes in charge of managing the data persistence. Regarding JFreeChart 0.8.1, the number of classes placed within the bottom layer is larger since refactoring operations on some of the bottom layer classes of JFreeChart 0.8.0 were performed. In particular, these classes were split to separate their responsibilities among a larger number of classes and then used by the same classes of the middle layer.

6.4. Threats to validity

This section presents and discusses the threats to validity that might limit the generalization of the obtained results. In our case, the reliability of the measures used to assess the achieved results (i.e. authoritativeness and stability) may condition the observed results. Regarding the measures used to assess the approach authoritativeness (i.e. precision, recall, and F-measure), the involvement of Master students to get the authoritative partition may represent a threat for the validity of the obtained results. However, this threat has been mitigated as the involved students can be considered not far from junior developers. Additionally, they knew neither the objectives nor the research questions of the experiment (they were informed later). To further investigate the effectiveness and the correctness of the proposed approach, we have planned to adopt different criteria and measures [24, 25]. In particular, we plan to adopt the criteria proposed in [26], where three models are suggested to generate synthetic static dependency networks of classes in object-oriented software systems to validate clustering algorithms.

Another threat for the validity of the results is the software systems used to assess the authoritativeness of the recovered layers. Possible issues may be related to the size of these systems and to the fact that they have been designed and developed by students as a laboratory activity of academic programming courses. A reader may object to the fact that we could study the open-source software systems used to assess the approach stability. Unfortunately, to compute precision and recall on these systems experts were needed. Software engineers with a suitable experience on an open-source software system are difficult to find and then to involve in studies like the one proposed here. The use of open-source software systems may represent however an issue for the assessment of the effectiveness of the approach, namely authoritativeness and stability. To address the issues presented above, we have planned to conduct research collaborations with one or more industries of our contact network to assess both the approach and the tool prototype on commercial software systems. In this scenario, the professional software engineers will play the expert role of software systems to be studied. This part of our research is still in progress and is actually the most challenging. The fact of using commercial software systems may also reduce the threats to validity regarding the size of the investigated software systems. In fact, limited size of the studied software systems should affect the generalization of the presented results both in terms of authoritativeness and stability.

Also, the fact of having assessed the prototype and the underlying algorithm on the layers identified by applying the link analysis algorithm once or twice might threaten the validity of the presented results. Therefore, in the future we plan to assess whether the identification of more layers affects the stability of the approach and the overall quality of the identified software layers.

DOI: 10.1002/spe

6.5. Open issues

The study presented here also opens a number of interesting concerns, which need further work and/or discussions to improve our knowledge on the possibility of using link analysis algorithms (e.g. the Kleinberg algorithm) in the reverse engineering, in general, and in the architectural layer recovery, in particular:

(i) Can the use of dynamic relationships improve the quality of the layers identified by the approach algorithm? To answer this question, we plan to extend the approach with dynamic analysis techniques. Furthermore, we are also going to analyze the effect of using relationships that are not currently managed (e.g. dependency) on the layers recovered by the tool prototype.

(ii) Are the results influenced by the chosen relationships? Although a preliminary analysis has been performed (see for example Figure 2), in the future we plan to investigate the influence of the different considered relationships (i.e. aggregation/composition, inheritance/realization, and association) on the decomposition of a software system into layers.

(iii) Can the approach be used to understand how the classes within the software layer evolve? Despite the encouraging results shown in Section 6.3, future work will be devoted to analyze how classes in the software layers evolve over time. In particular, it would be interesting to conduct a number of empirical studies to investigate and to measure the evolution of the changes of the classes within each layer (e.g. number of bugs, number of commits, etc.). Clone detection approaches could be also employed to investigate whether clones follow different evolution patterns in case they are placed within different layers.

(iv) Is the approach suitable for software systems that do not exhibit a classical tiered architecture? Even if we expect that the approach is not suitable for any software system, future work is needed.

(v) Is the approach influenced by the original modular quality of a given software system? Although the conducted study revealed that the original modular quality does not influence the layering results, a special conceived investigation is needed. In case of a significant influence of the modular quality on the approach results, the approach could be used as an indicator for the software system reengineering.

(vi) Does the application of the Kleinberg algorithm on the top or bottom influence the results? In case the top and bottom layers are bigger than the middle layer, it would be interesting to apply the approach recursively on the top or on the bottom layer. Nevertheless, the conducted study showed that it makes sense only when recursively applying the approach on the middle layer.

7. RELATED WORK

In this section, we discuss works related to the architectural recovery (i.e. module decomposition and layering) and evolution and to graph theory applied to object-oriented software systems.

7.1. Architectural recovery and evolution

To retrieve the architectural documentation of a software system, several approaches have been proposed. These can be classified as automatic, manual or semiautomatic. Automatic approaches do not need the software engineer support, whereas the architectural documentation is retrieved by software engineers in manual approaches. Semiautomatic approaches require software engineering interactions to set parameters or to enhance the results. For example, Muller *et al.* [13] propose a reverse engineering approach that requires the software engineer intervention to identify the layered subsystem structures of a subject software system implemented in C. Differently from us, their approach considers the structure of a software system to recognize related components and dependencies, to construct the layered structure, and to identify the component interfaces.

DOI: 10.1002/spe

The approach has been satisfactorily assessed on a real software system. Bowman *et al.* [27] suggest an approach to extract the architectural documentation of a software system from its implementation. The approach is based on a process that first automatically extracts relations from the code and then uses these relations to get the system architecture. As the authors are interested in the relations among subsystems, they manually created a tree/hierarchical structured decomposition of the system into subsystems, which is manually created analyzing the directory structure and the source files. This seems to be one of the most critical and challenging phase of the approach. The effectiveness of the approach is assessed on the Linux operating system. Clustering algorithms rarely consider dynamic information and often create flat decompositions. To overcome these issues Andreopoulos *et al.* [4] propose MULICsoft, a clustering algorithm that uses static and dynamic information to partition software entities into layered clusters.

Even on the legacy software of the future, i.e. web applications, the recovery of hierarchical architectures is relevant. For example, Hassan and Holt [28] propose a semiautomatic approach to recover the software architecture of web-based systems. In particular, this approach uses a set of parsers/extractors to analyze the source code and binaries of web applications developed for the Microsoft Windows platform. The extracted data are then manipulated to reduce the complexity of the architectural diagrams of the studied application. To assess the approach, they used several large commercial and experimental web applications. In particular, on Hopper News (a large application integrating components written in HTML, VBScript, VB, and C++) they employed the defined approach to recover its layered architecture.

Wiggerts in [7] introduces clustering algorithms commonly used in the past to group entities into software subsystems. Indeed, he provides a theoretical background for the application of cluster analysis in systems' remodularization. To this end, three concerns are mainly addressed: the entities to be analyzed, similarity measures to compare the entities, and the clustering algorithm to apply. Regarding the clustering algorithms, two categories have been suggested: supervised or unsupervised. Supervised algorithms need some *a priori* knowledge to group software entities. Such *a priori* knowledge can be the number of clusters that the algorithm should identify, for example. The greater part of the clustering algorithms has more or less supervised variants, thus requiring human decisions to identify the best partition of software entities into clusters. Anquetil and Lethbridge [8] extend the work by Wiggerts [7] presenting a comparative study of different hierarchical clustering algorithms and analyze their properties with regard to software remodularization. To get the best partition of software entities into clusters, the considered algorithms need human decisions (e.g. cutting points).

Various similarity and distance measures to be used in the software clustering in general and in the software remodularization in particular are analyzed in [29]. The main contribution of the paper is, however, the analysis of two clustering-based approaches and their experimental assessment. Both the approaches try to reduce the number of decisions to be taken during clustering process. The authors have also conducted an empirical evaluation of the clustering-based approaches on four large software systems. The evolution of these systems has not been considered.

To produce a decomposition of a system into subsystems, the Bunch clustering system is presented in [30]. In particular, this system is based on several heuristics to navigate through the search space of all possible graph partitions. To evaluate the quality of graph partitions and to find a satisfactory solution, the tool uses fitness functions and search algorithms. This represents the main difference with respect to our approach. Even, in [9] a structural approach based on genetic algorithms is proposed to group software entities into clusters. The effectiveness of the approach has been assessed on a small software system. Different from us, subsequent versions of the same system have been not considered. In [31], a different search-based approach to the automated module clustering problem is shown. The approach uses dependencies between modules to maximize cohesion within each cluster and to minimize coupling between clusters.

Generally, reverse engineering approaches, including the one proposed here, are focused on structural information to recover software architectures. Adritsos and Tzerpos in [12] present LIMBO, a hierarchical algorithm for software clustering, which considers both structural and non-structural

DOI: 10.1002/spe

attributes to reduce the complexity of the software system to be decomposed. The authors also apply LIMBO to three large software systems. Nevertheless, the domain knowledge of the developers is also embedded in the code comments. For such a reason, Kuhn *et al.* [6] describe an approach to group software artifacts based on Latent Semantic Indexing (LSI). The approach is language independent and tries to group source code containing similar terms in the comments. The authors consider different levels of abstraction to understand the semantics of the code (i.e. methods and classes). The main difference with respect to our approach relies on the fact that it does not consider relationships among classes. Maletic and Marcus in [10] propose an approach based on the combination of semantic and structural dimensions. From the semantic point of view they consider problem and development domains, whereas the structural dimension refers to the actual syntactic structure of the program along with the control and dataflow that it represents. Software entities are compared using LSI, whereas file organization is used to get structural information. There are two main differences with respect to our approach: the use of a graph theoretic algorithm to identify modules and the layered structure is not considered at all.

Regarding the analysis of the software architecture evolution, Wu *et al.* in [21] present a comparative study of a number of clustering algorithms. In particular, they consider: (i) an agglomerative clustering algorithm (based on the Jaccard coefficient and the complete linkage update rule) using 0.75 and 0.90 as cutting points; (ii) an agglomerative clustering algorithm (based on the Jaccard coefficient and the single linkage update rule) using 0.75 and 0.90 as cutting points; (iii) an algorithm based on program comprehension patterns that tries to recover subsystems that are commonly found in manually created decompositions of large software systems; and (iv) a customized configuration of an algorithm implemented in Bunch. The authors compare these algorithms on the subsequent versions of five large C/C++ open-source systems. Similarly, in [20] an empirical study is presented to evaluate four widely known clustering algorithms according to: extremity of cluster distribution, authoritativeness, and stability. The algorithms are assessed in the architecture recovery and evolution fields on 15 systems implemented in Java and C/C++. In [32], the MoJo distance [33] is employed to study the stability and the quality of a number of software clustering algorithms. Even in this case, the selected algorithms need a tuning phase to get good software partitions. The comparison among clustering algorithms is conducted generating randomly 'perturbed' versions of an example system. Successively, differences between the partition identified by the clustering algorithms and the original partition of the system are measured. Random perturbation of a fixed size system could be however considered as inadequate to simulate the behavior of a clustering algorithm on actual software systems, both commercial and open source.

7.2. Graph theory for O-O software systems

Graphs have long been used in several fields of computer science, in general, and in software engineering, in particular. For example, Chatzigeorgiou *et al.* [34] adopts graph theory to understand object-oriented software systems. Indeed, they suggest a possible graph-based representation of software systems and then use a variant of the Kleinberg algorithm to identify 'god' classes. On the other hand, Chatzigeorgiou *et al.* [18] use an approach based on graph theory for software clustering. Software systems are represented using the same graph representation as [34] and are partitioned employing a spectral graph partition technique.

An approach based on graph theory to recover design patterns has been proposed in [19]. To this end, the graph similarity algorithm presented in [35] has been properly modified. The algorithm takes as input the graph representing the system and the graph describing a given pattern to recover and compute similarity scores between the vertices of the graphs. One of the main advantages of the approach concerns the possibility of detecting variations of patterns in their basic form (i.e. the one usually found in the literature). The algorithm effectiveness and efficacy are assessed on three large open-source software systems.

Myers in [36] asserts that a software system can be represented as a complex network connecting many collaborating modules, objects, classes, methods, and subroutines. According to the recent advances in the study of complex networks, he studies the software collaboration graphs of some

DOI: 10.1002/spe

open-source software systems. The study reveals that the hierarchical nature of a software system has an impact on the corresponding network topology. In particular, generic classes and subroutines form the heavy tail of the in-degree distribution, and complex, more specialized aggregates populate the heavy tail of the out-degree distribution, with the two generally well separated from one another.

8. REMARKS AND FUTURE WORK

Link analysis algorithms have been successfully used in the reverse engineering field [18, 19, 36]. However, their feasibility in the architectural recovery and software architecture evolution has not been investigated in the past. In this paper, we have proposed a semiautomatic approach to decompose a classic object-oriented tiered software system into layers using the Kleinberg algorithm [15], a link analysis algorithm. Even if the approach has been mainly proposed for software understanding and evolution, it may be also used to assist a software engineer in the tedious process of identifying the hierarchical subsystem structure of given software system. In fact, the identified layers could be manually or automatically partitioned to identify subsystems providing related services. Future work will be devoted to the definition of a clustering-based approach aimed at partitioning the identified software layers into software subsystems. To this end, we plan to investigate the possibility of using semantic approach similar to the one proposed by Kuhn *et al.* in [6]. The motivation for adopting semantic approaches relies on the fact that the partition of each layer should aim at identifying software components that have the same level of abstraction, but that implement different services. The possibility of considering information concerning the development process (i.e. analysis and design documentation) represents another possible direction to extend our approach. The possibility of using the original structure of the classes within the packages will be investigated as well.

In order to automate the approach and to facilitate its adoption, we have also implemented a Java prototype. To prove the feasibility of the approach and the tool prototype, we have conducted a case study involving different Java software systems exhibiting a tiered architecture. The overall quality of the identified layers has been assessed using two criteria: authoritativeness (automatically identified layers should approximate the one produced by a software architect) and stability (the layer structure of two consecutive versions of a software system is similar). The data analysis has revealed that the identified layers exhibit a suitable authoritative level and their structure is stable. Future work will be devoted to investigate whether the approach is suitable for software systems implemented with programming languages belonging to different paradigms (e.g. procedural, declarative, functional, and logic). To this end, the approach and the tool prototype will be properly extended and then assessed on some case studies. We are also considering the feasibility of extending our approach to recover the architectural layers of web-based software systems.

ACKNOWLEDGEMENTS

The authors thank the original developers of the software systems used to validate both the approach and the tool from the point of view of the result authoritativeness.

REFERENCES

1. Zelkowitz M, Shaw A, Gannon J. *Principles of Software Engineering and Design*. Prentice-Hall: Englewood Cliffs, NJ, 1979.
2. Lehman MM. Program evolution. *Journal of Information Processing Management* 1984; **19**(1):19–36.
3. Muller HA, Jahnke JH, Smith DB, Storey MAD, Tilley SR, Wong K. Reverse engineering: A roadmap. *Proceedings of ICSE—Future of SE Track*, Limerick, Ireland, 2000; 47–60.
4. Andreopoulos B, An A, Tzerpos V, Wang X. Multiple layer clustering of large software systems. *Proceedings of Working Conference on Reverse Engineering*. IEEE CS Press: Silver Spring, MD, 2005; 79–88.

DOI: 10.1002/spe

5. Koschke R. Atomic architectural component recovery for program understanding and evolution. *PhD Thesis*, University of Stuttgart, 2000. Available at: http://www.informatik.uni-bremen.de/st/papers/koschke-diss.pdf [11 June 2010].
6. Kuhn A, Ducasse S, Girba T. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 2007; **49**(3):230–243.
7. Wiggerts TA. Using clustering algorithms in legacy systems remodularization. *Proceedings of the Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, 1997; 33–43.
8. Anquetil N, Lethbridge TC. Experiments with clustering as a software remodularization method. *Proceedings of Working Conference on Reverse Engineering*. IEEE CS Press: Silver Spring, MD, 1999; 235–255.
9. Doval D, Mancoridis S, Mitchell BS. Automatic clustering of software systems using a genetic algorithm. *Proceedings of the Software Technology and Engineering Practice*. IEEE CS Press: Silver Spring, MD, 1999; 73–82.
10. Maletic JI, Marcus A. Supporting program comprehension using semantic and structural information. *Proceedings of International Conference on Software Engineering*. IEEE CS Press: Silver Spring, MD, 2001; 103–112.
11. Lientz BP, Swanson EB, Tompkins GE. Characteristics of application software maintenance. *Communications of ACM* **21**(6):1978; 466–471. DOI: acm.org/10.1145/359511.359522.
12. Andritsos P, Tzerpos V. Information-theoretic software clustering. *IEEE Transactions on Software Engineering* 2005; **31**(2):150–165.
13. Muller HA, Orgun MA, Tilley SR, Uhl JS. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance*: *Research and Practice* 1993; **5**(4):181–204.
14. Laguë B, Leduc C, Le Bon A, Merlo E, Dagenais M. An analysis framework for understanding layered software architectures. *Proceedings of the Sixth International Workshop on Program Comprehension*. IEEE CS Press: Silver Spring, MD, 1998; 37–44.
15. Kleinberg JM. Authoritative sources in a hyperlinked environment. *Journal of the ACM* 1999; **46**(5):604–632.
16. Scanniello G, D'Amico A, D'Amico C, D'Amico T. An approach for architectural layer recovery. *Proceedings of 25th Annual International Symposium on Applied Computing*, Sierre, Switzerland, 22–26 March 2010. ACM Press: New York, 2010; 1853–1857.
17. Rumbaugh J, Jacobson I, Booch G. *Unified Modeling Language Reference Manual* (2nd edn). Addison-Wesley: Reading, MA, 2004.
18. Chatzigeorgiou A, Tsantalis N, Stephanides G. Application of graph theory to OO software engineering. *Proceedings of International Workshop on Interdisciplinary Software Engineering Research*. ACM Press: New York, 2006; 29–36.
19. Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis ST. Design pattern detection using similarity scoring. *Transactions on Software Engineering* 2006; **32**(11):896–909.
20. Bittencourt RA, Guerrero DDS. Comparison of graph clustering algorithms for recovering software architecture module views. *Proceedings of European Conference on Software Maintenance and Reengineering*. IEEE CS Press: Silver Spring, MD, 2009; 251–254.
21. Wu J, Hassan AE, Holt RC. Comparison of clustering algorithms in the context of software evolution. *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press: Silver Spring, MD, 2005; 525–535.
22. De Lucia A, Deufemia V, Gravino C, Risi M. Design pattern recovery through visual language parsing and source code analysis. *Journal of System and Software* 2009; **82**(7):1177–1193. DOI: 10.1016/j.jss.2009.02.012.
23. Wen Z, Tzerpos V. An optimal algorithm for MoJo distance. *Proceedings of the International Workshop on Program Comprehension*. IEEE CS Press: Silver Spring, MD, 2003; 227–235.
24. Champaign J, Malton A, Dong X. Stability sand volatility in the Linux kernel. *Proceedings of the Sixth International Workshop on Principles of Software Evolution*. IEEE CS Press: Helsinki, Finland, 2003; 95–102.
25. Martin R. *Agile Software Development—Principles*, *Patterns*, *and Practices*. Pearson Education: Upper Saddle River, NJ, 262–263.
26. Souza R. Modular network models for class dependencies in software. *Proceedings of 14th IEEE Conference on Software Maintenance and Reengineering*, Madrid, Spain, 15–18 March 2010. IEEE Computer Society Press: Silver Spring, MD, 2010; 245–248.
27. Bowman IT, Holt RC, Brewster NV. Linux as a case study: Its extracted software architecture. *Proceedings of the 21st International Conference on Software Engineering*. ACM Press: New York, NY, 1999; 555–563. DOI: http://doi.acm.org/10.1145/302405.302691.
28. Hassan AE, Holt RC. Architecture recovery of web applications. *Proceedings of the 24th International Conference on Software Engineering*. ACM Press: New York, NY, 2002; 349–359. DOI: http://doi.acm.org/10.1145/581339.581383.
29. Maqbool O, Babri HA. Hierarchical clustering for software architecture recovery. *Transactions on Software Engineering* 2007; **33**(11):759–780.
30. Mitchell BS, Mancoridis S. On the automatic modularization of software systems using the bunch tool. *Transactions on Software Engineering* 2006; **32**(3):193–208.
31. Mahdavi K, Harman M, Hierons RM. A multiple hill climbing approach to software module clustering. *Proceedings of the International Conference on Software Maintenance*. IEEE CS Press: Washington, DC, 2003; 315.

DOI: 10.1002/spe

32. Tzerpos V, Holt RC. On the stability of software clustering algorithms. *Proceedings of the Eighth International Workshop on Program Comprehension*, 2000; 211–218.
33. Tzerpos V, Holt RC. MoJo: A distance metric for software clusterings. *Proceedings of the Working Conference on Reverse Engineering*. IEEE CS Press: Silver Spring, MD, 1999; 187–193.
34. Chatzigeorgiou A, Xanthos S, Stephanides G. Evaluating object-oriented designs with link analysis. *Proceedings of International Conference on Software Engineering*, Edinburgh, Scotland. IEEE CS Press: Silver Spring, MD, 2004; 656–665.
35. Blondel VD, Gajardo A, Heymans M, Senellart P, Van Dooren P. A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review* 2004; **46**(4):647–666.
36. Myers CR. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Physical Review E* 2003; **68**(4):1–16.

DOI: 10.1002/spe