The Layered Architecture Recovery as a Quadratic Assignment Problem

Alvine Boaye Belle[1], Ghizlane El Boussaidi[1], Christian Desrosiers[1], Sègla Kpodjedo[1], Hafedh Mili[2]

[1]Department of Software and IT engineering, École de Technologie Supérieure
[2]Department of Computer Science, Université du Québec à Montréal
Montreal, Canada

Abstract. Software architecture recovery is a bottom-up process that aims at building high-level views that support the understanding of existing software applications. Many approaches have been proposed to support architecture recovery using various techniques. However, very few approaches are driven by the architectural styles that were used to build the systems under analysis. In this paper, we address the problem of recovering layered views of existing software systems. We re-examine the layered style to extract a set of fundamental principles which encompass a set of constraints that a layered system must conform to at design time and during its evolution. These constraints are used to guide the recovery process of layered architectures. In particular, we translate the problem of recovering the layered architecture into a quadratic assignment problem (QAP) based on these constraints, and we solve the QAP using a heuristic search algorithm. In this paper, we introduce the QAP formulation of the layering recovery and we present and discuss the results of the experimentation with the approach on four open source software systems.

Keywords: Software architecture, Architecture recovery, Layered style, Architecture evolution, Quadratic assignment problem.

1 Introduction

Software architects rely on a set of patterns, commonly named architectural styles [1], to design systems. An architectural style embodies design knowledge [2] that applies to a family of software systems [1]. Common architectural styles include layered, pipes and filters, and service-oriented styles [1-3]. Each style has its own vocabulary and constraints, and it promotes some specific quality attributes. However, researchers observed that the as-built architecture of a software system does not conform to the initial style that guided its design. This is mainly due to: 1) the continuous changes undergone by the system, which increase its complexity and lead to a deviation from its initial design; and 2) violations of the style constraints due to the conceptual gap between the elements defined by the style and the constructs provided by

adfa, p. 1, 2015.
© Springer-Verlag Berlin Heidelberg 2015

programming languages [4]. Therefore, understanding and properly evolving a software system often require recovering its architecture as it is implemented.

Architecture recovery may be achieved using a bottom-up process that starts from source code to progressively construct a more abstract representation of the system [5]. In this context, various clustering-based approaches have been proposed and discussed [5 and 7]. However, these approaches generally rely on properties such as high-cohesion and low-coupling to reconstruct architectures (e.g., [6, 8]) and they do not consider the architectural style of the analyzed system. Our focus in this paper is the recovery of layered architectures as the layered style is a widely used pattern to structure large software systems. Some approaches were proposed to reconstruct layered architectures (e.g., [9-15]). However, most of these approaches propose greedy algorithms that partition elements of the analyzed system into layers using some heuristics or some particular criterion (e.g., the number of fan-in and fan-out dependencies of a module [12, 13]). This may result in partitions with very few layers (e.g., in case of a heuristic based on highly connected modules [10, 15]) or too many layers (e.g., in case of heuristics to resolve cyclic dependencies [11]) which may be too permissive with violations of the style's constraints.

In this paper, we propose an approach that aims at recovering the layered architecture of object oriented systems while relying on: 1) a set of constraints that convey the essence of the layered architectures and 2) the user's input which reveals how strictly he applied the layered principles when designing a given system. Thus, we analyze the layered style and extract a set of principles that we use to define cost factors corresponding to possible types of assignments of dependent packages to the layers of a given system. These cost factors were used to formulate the layering recovery problem as a quadratic assignment problem (QAP), a well-established combinatorial optimization formulation which has been used to model problems such as layout design or resource allocation. Experimentation with the approach on four open source projects yielded interesting results and observations.

The main contributions of this paper are: 1) the formalization of a layered architecture as a special case of a QAP; 2) an algorithm that solves the QAP to recover layered architectures; and 3) an evaluation on four open source projects. The paper is organized as follows. Section 2 discusses the layered style and the limitations of existing approaches to recover such architectures. Section 3 introduces the layering principles that we retained from our analysis of this style. In accordance with these principles, we define in section 4 a set of cost factors related to layers' assignments of dependent packages and we formulate the layering recovery problem as a special case of the QAP. Section 5 discusses the experimentation results. Related works are discussed in section 6 and we conclude and outline some future works in section 7.

2 Background and Limitations of Existing Approaches

2.1 Analysis of the Layered Style

To analyze the layered style, we studied many reference books and papers (e.g., [1-3, 10-18]). The layered style promotes a set of quality attributes which include reuse,

portability and maintainability [1-3]. It is an organized hierarchy where each layer is providing services to the layer above it and serves as a client to the layer below [1]. Different strategies can be used to partition a software system into layers. The most common layering strategies are the responsibility-based and the reuse-based layering strategies [16]. The responsibility-based strategy aims at grouping components of the system according to their responsibility and assigning each responsibility group to a layer. The reuse-based layering strategy aims at grouping components of the system according to their level of reuse and assigning the most reusable components (through applications) to the bottom layers.

In an ideal layered architecture, a layer may only use services of the next lower layer. This is referred to as strict [3] or closed [17] layering and is often violated in practice. For example, the dependence of a layer to much lower layers is a violation (named a skip-call violation in [10] and layer bridging in [2]) that is considered as a regular feature in open [17] or relaxed [3] layering. On the other hand, intra-dependencies, which are dependencies between services of the same layer, are not recommended [2, 18] but can be implemented under considerations such as portability [2]. Exceptionally, a layer may need to rely on a service offered by an upper layer. These back-calls [10] are discussed in [2] as "upward usage" and should be rare as they threaten the quality attributes promoted by the layered style.

2.2 Limitations of Existing Approaches to Recover Layered Architectures

Based on the above analysis of the layered style, the structure of a layered architecture must be a directed acyclic graph or at least a directed graph with very few cycles connecting different layers. The existence of cyclic dependencies between entities of the system (i.e., packages) makes it difficult to identify its layers [11]. Hence most of the approaches that were proposed to recover software layers focused their effort on proposing methods and heuristics to handle entities involved in cyclic dependencies (e.g., [10, 11, 15]). To illustrate the limitations of these approaches, we will use as an example the software system illustrated by Fig. 1(a). The latter displays a dependency graph where nodes are packages of the system and edges are dependencies between these packages. The weight of a dependency between two packages is derived from the number of dependencies between their respective classes.

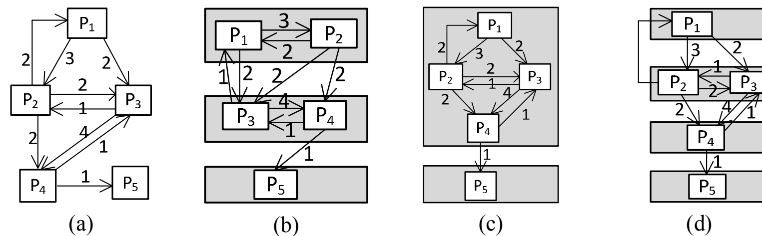


Fig. 1. An example of a system, its architecture and the layering obtained applying different existing approaches

Fig. 1(b) shows the expected layered architecture of our example system. Using a clustering algorithm that relies on modularity (e.g., [6, 8]), the recovered architecture of the system is depicted in Fig. 1(c). The clustering, in this case, puts all packages involved in a cyclic dependency in the same layer/cluster as they are tightly coupled. This is also the case for approaches relying on strongly connected components (e.g., [10, 15]). Other approaches use some heuristic to resolve cyclic dependencies and then assign packages to layers using a depth traversal of the resulting dependency graph. Using such approach as in [11], the recovered architecture of our example system is depicted in Fig. 1(d): it possesses too many layers and may be too permissive with violations such as skip-calls and back-calls.

In our example system, the architect defined three responsibilities embodied in the sets {P_1, P_2}, {P_3, P_4} and {P_5}. He then assigned each set to a layer according to its abstraction level (Fig. 1(b)). In doing so, the architect applied the responsibility-based strategy while trying to best comply with the layered style constraints. Hence, to obtain the most accurate results (i.e., a layering as in Fig. 1(b)), the layering recovery approach proposed in this paper is based on the principles of the layering style and on how strictly the architect applied them when designing his system.

3 Principles for Layers Recovery

Applying the layered style means partitioning the system into a set of layers that must be ordered according to the abstraction criterion that rules the flow of communication between components of the system. This observation encompasses two fundamental principles that should guide both design and recovery of layered architectures. These two principles are discussed in details in [27]:

- **The Layer Abstraction Uniformity**: This principle states that components of the same layer must be at the same abstraction level so that the layer has a precise meaning. The level of abstraction of a component often refers to its conceptual distance from the "physical" components of the system [3], i.e. hardware, database, files and network. Components at the highest levels are application specific; they generally contain the visible functionalities provided by the system. This principle led to many algorithms that build layered architectures based on a depth-traversal of dependency graphs built from the studied system (e.g., [10-12, 19]).
- **The Incremental Layer Dependency**: This principle is related to the "ideal layering" property that states that a component in a layer (*j*) must only rely on services of the layer below (*j-1*) [3]. This principle is the one that is mostly violated, either through back-calls, skip-calls or intra-dependencies. It is worth pointing out that there is no clear consensus among researchers on the use of intra-dependencies which are accepted by some [20] and not recommended by others [2, 18]. Our analysis of the various descriptions of the layered style and several open source projects led us to conclude that the acceptance of the intra-dependencies depend on the granularity of the components (e.g., packages) of the layer: the higher the granularity, the lower the number of intra-dependencies. The incremental dependency property should thus be stated in a way that allows the intra-dependencies and the

skip-calls and—to some extent—back-call violations. Hence, we formulate this property as "components of layer *j-1* are mainly geared towards offering services to components of layer *j*". This means that, for a given layered system, the number of skip-call and back-call dependencies must be much lower than the number of downward dependencies between adjacent layers and intra-dependencies.

In the context of this paper, we focus on object oriented systems and we work at the package level; i.e., we rely on existing decomposition of object oriented systems into packages. To comply with the first principle, the packages of the same layer should be at the same distance from the "physical" components of the system. However, the existence of back-call and skip-call dependencies introduces a discrepancy between the packages' distances, even when they belong to the same layer. Hence, compliance with the first principle derives largely from compliance with the second principle (i.e., incremental layer dependency). The latter will be used to formulate the layered architecture recovery problem as a QAP in the following section.

4 Translating the Layering Recovery into a Quadratic Semi-Assignment Problem

To formalize the incremental layer dependency principle, we define a number of cost factors related to layers assignment of two dependent packages. These cost factors are used to formulate the problem of recovering layered architectures as a special case of the QAP known as the Quadratic Semi-Assignment Problem (QSAP) [21].

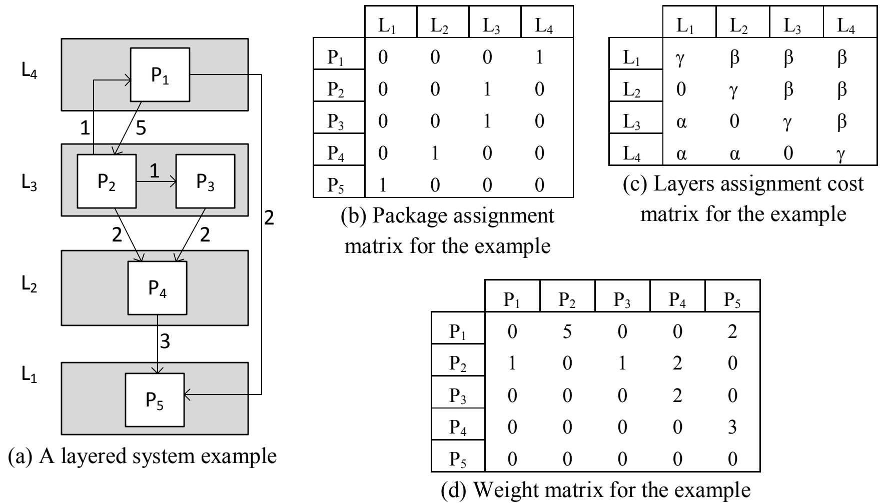
4.1 Cost Factors for Layers Assignment

Let packages i and j be two distinct packages of the system with a directed dependency from i to j. The dependency between two packages is derived from the dependencies between their respective classes and it includes class references, inheritance, method invocation and parameters. Let c_{kl} be the cost of assigning packages i and j to layers k and l, respectively. Following the incremental layer dependency principle, we distinguish four possible types of layers' assignments for packages i and j:

- Adjacent layers assignment: in this case $k = l+1$; this is the optimal and desirable assignment of two dependent packages and thus, has no cost attached to it ($c_{kl} = 0$).
- Same layer assignment: in this case $k = l$; this introduces an intra-dependency which is not recommended, unless there is a system portability concern, and has a non-zero cost $c_{kl} = \gamma$ attached to it.
- Skip layers assignment: in this case $k \geq l+2$; i.e., this introduces a skip-call dependency that can be tolerated (e.g., for performance reasons [4]) in small numbers and has a non-zero cost $c_{kl} = \alpha$ attached to it.
- Back layers assignment: in this case $k \leq l-1$; this introduces a back-call dependency that can hamper the quality attributes promoted by the layered style and is thus assigned a non-zero cost $c_{kl} = \beta$.

Consider the layered system illustrated in Fig. 2(a). The assignment of packages P_1 and P_2 to layers L_4 and L_3, respectively, has a cost value of β because of the back-call dependency relating P_2 to P_1. The assignment of packages P_1 and P_5 to layers L_4 and L_1, respectively, has a cost value of $2*\alpha$ because it introduces a skip-call dependency having a weight of 2. The assignment of packages P_2 and P_3 to the same layer L_3, has a cost value of γ because of the intra-dependency relating P_2 to P_1. The other assignments do not introduce any additional skip-calls, back-calls or intra-dependencies. Hence, the total cost of this layered system is: $(\gamma + 2*\alpha + \beta)$.

In accordance with the incremental layer dependency principle, we want to minimize the number of skip-calls and back-calls and the number of intra-dependencies. This means that, apart from the adjacent layers assignment as described above, we must minimize the number of the other assignment types. However, in practice, intra-dependencies and skip-calls are more accepted than back-calls which lead to a poorly structured system. Furthermore, according to the analysis of the open or relaxed layering ([3, 17]), skip-calls are more often used and tolerated in practice than intra-dependencies. Accordingly, we make the assumption that the values of the cost factors γ, α and β should be constrained as follows: $\alpha < \gamma < \beta$. This assumption should be validated through experimentation by analyzing a number of software systems purported to be all 1) of a layered style, and 2) of good quality.



(a) A layered system example

	L1	L2	L3	L4
P1	0	0	0	1
P2	0	0	1	0
P3	0	0	1	0
P4	0	1	0	0
P5	1	0	0	0

(b) Package assignment matrix for the example

	L1	L2	L3	L4
L1	γ	β	β	β
L2	0	γ	β	β
L3	α	0	γ	β
L4	α	α	0	γ

(c) Layers assignment cost matrix for the example

	P1	P2	P3	P4	P5
P1	0	5	0	0	2
P2	1	0	1	2	0
P3	0	0	0	2	0
P4	0	0	0	0	3
P5	0	0	0	0	0

(d) Weight matrix for the example

Fig. 2. An example of a layered system and its related matrices

4.2 Layers Recovery as a Quadratic Semi-Assignment Problem

Recovering the layered architecture of a given system consists in finding a mapping function that assigns each package to a given layer while minimizing the intra-, skip-call and back-call dependencies as discussed in the previous section. Let m be the number of packages and n the number of layers of the system under analysis. Let

w_{ij} be the weight of the directed dependency relating package i to package j. Recall that the dependency between two packages derives from the dependencies between their respective classes. Let W ($[W]_{ij} = w_{ij}$) be the $m \times m$ dependency weight matrix, and C ($[C]_{kl} = c_{kl}$) be the $n \times n$ matrix of layer assignment costs. Fig. 2(c) displays the layer assignment cost matrix and Fig. 2(d) displays the weight matrix corresponding to the system of Fig. 2(a). Let x_{ik} be the binary decision variable representing the assignment of package i to layer k (i.e., x_{ik} is set to 1 if package i is assigned to layer k, otherwise to 0), and let X ($[X]_{ik} = x_{ik}$) be the $m \times n$ package assignment matrix. Fig. 2(b) displays the package assignment matrix corresponding to the system of Fig. 2(a).

The layering recovery problem can be expressed as the following QSAP:

$$\min f(X) = \sum_{i,j=1}^{m} \sum_{k,l=1}^{n} w_{ij}\, c_{kl}\, x_{ik}\, x_{jl} \qquad (1)$$

$$x_{ik} \in \{0,1\} \quad i = 1,\ldots,m,\ k = 1,\ldots,n \qquad (2)$$

$$\sum_{k=1}^{n} x_{ik} = 1 \quad i = 1,\ldots,m \qquad (3)$$

The quadratic cost function of Eq. 1, called the layering cost in our context, defines a penalty for each possible set of assignments of packages to layers. Thus, the penalty of assigning package i to layer k, if a package j is assigned to layer l corresponds to $w_{ij} * c_{kl}$. Eq. 3 constrains the possible solutions to Eq. 1 by stating that a package may be assigned only to one layer.

4.3 Solving the Layers Recovery Problem

Because the NP-hard clustering problem is a particular case of the QSAP, finding a globally optimal solution for this problem is also a difficult task. However, since it plays a central role in many applications, much effort has been spent to solve this problem efficiently. Exact methods proposed for the QSAP, which guarantee the global optimum, include the cutting-plane and branch-and-bound algorithms. However, these methods are generally unable to solve large problems (i.e., $m \geq 20$). For large problem instances, heuristic algorithms like tabu search, local improvement methods, simulated annealing and genetic algorithms have been proposed [21]. Among these, the tabu search method [22] and the local improvement method are known to be the most accurate heuristic methods to solve the QAP. Hence, to solve the layering recovery problem, we adapted the tabu search method using our layering cost (Eq.1) as a fitness function. Briefly, the tabu search [23] starts with a feasible solution as the current solution. At each iteration, neighbors of the current solution are explored through some moves and the best neighbor is accepted as the current solution provided the related move does not belong to a tabu list. The latter records moves which are marked as tabu (i.e. forbidden) to prevent cycling and to escape from local optima. The search process stops when a termination criterion is met.

Fig. 3 gives a simplified view of our adaptation of the tabu search technique to the layering problem. The algorithm, takes as input: 1) an initial layered partition of the

system under analysis; 2) a set of values for the cost factors; and 3) the maximum number of iterations (max_it) after which the algorithm terminates. The initial partition is a 3-layer solution where packages are randomly assigned to these layers. The initial partition is then considered as the current and the best solution of the algorithm (lines 1 to 2). In the following iterations (lines 5 to 18), all the neighboring solutions of the current solution are explored to find a better layering. A neighbor is computed by moving a single package from a layer A to a layer B, provided these two layers are different (line 7). This neighbor is considered as a candidate solution if it is produced using a package move that does not belong to the tabu list (lines 8 to 10). Note that a package move may introduce an additional layer (i.e., the final layering may have more than 3 layers). The candidate solution having the lowest layering cost value is the best candidate solution and it is accepted as the current solution (line 12) to be used for the next iteration. In this case, the tabu list is updated to include the package move that led to this solution (line 13). It is also accepted as the best solution if its cost is lower than the current best-known solution (lines 14 to 16).

```
Input: initialLayeredPartition, max_it,  γ, α, β
Output: LayeredSolution
1. currentSolution ← initialLayeredPartition
2. bestSolution ← currentSolution
3. tabuList ← null
4. K ← 0
5. while (K < max_it){
6.     candidates ← null
7.     for each neighborSolution of currentSolution {
8.          if (neighborSolution is produced using a move not belonging to tabuList){
9.             candidates ← candidates + neighborSolution
10.         }
11.    } //end for
12.    currentSolution ← locateBestSolution(candidates)
13.    tabuList ← updateTabuList(currentSolution.move)
14.    if (LC(currentSolution) < LC(bestSolution)) {
15.         bestSolution ← currentSolution
16.    }
17.    K ← K +1
18. } //end while
19. return bestSolution
```

Fig. 3. A high level view of the layering algorithm

5 Experimentation with the Approach

To experiment our QSAP formulation of the layering recovery problem, we implemented a tool within the Eclipse[TM] IDE. This tool is made of two modules. The first module was built on top of the MoDisco open source tool [24] which enables to analyze source code files of the system under study and to generate platform independent representations that are compliant with the Knowledge Discovery Metamodel (KDM). The KDM was introduced by the OMG as a standard and platform-

independent meta-model for representing legacy systems [25]. In our context, the KDM representation is used by our module to extract the system's facts, i.e. packages and their dependencies. These facts are used to build the initial partition that is the input to the second module. This module implements our layering algorithm for which we set the maximum number of iterations to 1000 and the tabu list length to 10 (i.e., the tabu list records the last ten best packages' moves). Results were computed on a 2.8 GHz Intel octo-core CPU with 16Gb of RAM and took less than a second for any of the systems. For each system and setup, we ran 50 times the algorithm and retained the best (lowest layering cost) result.

5.1 Research Questions and Experimental Setup

Experimentations with our approach aimed at answering the following questions:

1) What are the (relative) values of the cost factors (γ, α, β) that best correspond to the common understanding of the layered style? For any given layered software system, assuming we already know its layered architecture, we look for the values of the cost factors that yield a set of layers that best match the known architecture of the system. However, as the system may be an imperfect application of the layered style, we need to look into a set of well-designed software systems that are known to be layered systems. The answer to this question will help assessing the extent to which the layering principles, as discussed in section 3, are applied by designers.

2) How does the layering cost evolve across revisions of a software system and what does it tell us about the architectural evolution of the system? This question is related to two aspects: 1) the stability of our layering recovery algorithm and 2) the stability of the set of values of the cost factors that yield the layering that matches the known architecture of the system across its revisions. The latter aspect can be rephrased into "*when a layered system evolves, does it maintain the same level of conformity to the layering principles?*".

To answer these questions, we carried out an experiment on four different open source projects and four different versions of one of these projects. Some characteristics of these projects are given in Table 1. All the projects (Apache Ant, JUnit, JFreeChart and JHotDraw) are purported layered systems that are actively maintained and that were analyzed in related work (e.g., [11, 12]). We performed several executions on each of these projects using different values for the layering cost factors γ, α, β. For lack of space, we present the results for 5 setups. Recall that downward adjacent dependencies are rewarded and, hence, no cost factor was associated to them. Briefly, setups 1 and 2 penalize more skip-calls than intra-dependencies while setups 3, 4 and 5 penalize more intra-dependencies than skip-calls. Thus setups 1 and 2 are appropriate for systems that favor portability over reuse. Conversely, setups 3, 4 and 5 are appropriate for systems that comply with a reuse-based layering strategy where the most (re)used packages are assigned to bottom layers. Setups 3, 4 and 5 differ in the value they assign to the back-call cost β and they are meant to analyze the extent to which the back-calls are tolerated in the analyzed systems. It should be noted that we performed tests using other setups where the cost γ of intra-dependencies was set

to zero. However, in this case, the algorithm behaves as a modularity-based clustering algorithm and it assigns all highly dependent packages to the same layer.

To find out which ones of the setups return layered solutions that best match the actual layered organizations of the analyzed systems, we compare the returned solution for each setup and system with an authoritative decomposition of the system. We rely on previous works (e.g. [15 and 26]) to specify the authoritative decomposition of the analyzed systems (e.g., Apache Ant and JUnit). For systems for which the authoritative decomposition was not available (e.g., JHotDraw and JFreeChart), we had 3 PhD students with intensive experience in software design and with a good knowledge of these systems to manually decompose them. We used the harmonic mean (F-measure) of precision and recall as introduced in [12] to evaluate each solution with respect to both correctness and completeness of its layers compared to the authoritative decomposition. Thus, we compute the precision as the number of packages correctly assigned by our tool over the total number of packages assigned by our tool. We compute the recall as the number of packages correctly assigned by our tool over the number of the packages assigned to layers in the authoritative decomposition.

Table 1. Statistics of the analyzed open source projects

Project	Number of files	LOC	Numb. of packages	Package dependencies
Apache Ant	681	171 491	67	229
JUnit 4.10	162	10 402	28	106
JFreeChart 1.0.14	596	209 711	37	207
JHotDraw 6.0.b1	498	68 509	17	72
JHotDraw 7.0.6	310	51 801	24	89
JHotDraw 7.4.1	585	111 239	62	365
JHotDraw 7.6	680	118 938	65	358

5.2 Results and discussions

Table 2 summarizes the results of executing our layering algorithm on the analyzed projects using 5 setups. The first column indicates for each setup the values of the cost factors. For each solution returned by the algorithm, Table 2 displays: 1) the layering cost (LC); 2) the number of layers (NL); 3) the total weight of all dependencies relating adjacent layers (Adj); 4) the total weight of all intra-dependencies (Intra); 5) the total weight of all skip-calls (Skip); 6) the total weight of all back-calls (Back); and 7) the F-measure. Recall that the layering cost is the value of the quadratic function in Eq. 1. Cells that are greyed in Table 2 correspond to the solutions with the highest F-measures.

As shown by Table 2, for Apache Ant and JUnit, the layering solution that best matches the actual layering of the system is returned using setup 2. In general, the most accurate results are produced by our algorithm for these two systems when using setups where the intra-dependencies cost γ is less than the skip-calls cost α (e.g., setups 1 and 2). This means that the designers of these two systems have favored intra-dependencies over skip-calls and back-calls. This is consistent with the fact that both

Apache Ant and JUnit are frameworks that target different platforms and, thus, portability is one of the concerns that drive their design. As for JFreeChart, we obtained the best match using setup 5. JFreeCHart contains several subsystems that are composed

Table 2. Results returned by the layering recovery algorithm

		Ant	JFreeC	JUnit	JHD.60b1	JHD.706	JHD.741	JHD.76
Setup 1 γ=1 α=2 β=4	LC	569	1069	152	383	385	1176	1089
	NL	3	3	3	3	3	3	3
	Adj	1535	1018	234	864	623	1547	1522
	Intra	521	629	110	335	353	1036	909
	Skip	0	64	3	4	8	12	6
	Back	12	78	9	10	4	29	42
	F-measure	74	35.97	57	58	29	22	21
Setup 2 γ=1 α=2 β=15	LC	692	1411	247	493	429	1316	1245
	NL	3	3	3	3	3	3	3
	Adj	1502	450	168	864	623	1362	1325
	Intra	557	1332	181	335	353	1247	1141
	Skip	0	2	3	4	8	12	7
	Back	9	5	4	10	4	3	6
	F-measure	76	59	67	58	29	12	12
Setup 3 γ=2 α=1 β=4	LC	1026	1476	234	587	572	1910	1719
	NL	4	4	3	3	4	4	5
	Adj	1567	998	248	887	665	1622	1580
	Intra	417	409	78	213	222	455	526
	Skip	48	290	14	97	92	396	275
	Back	36	92	16	16	9	151	98
	F-measure	59	13	53	82	87	51	60
Setup 4 γ=2 α=1 β=15	LC	1204	2368	357	715	589	2263	2039
	NL	4	4	4	3	4	4	4
	Adj	1526	870	234	891	647	1396	1398
	Intra	494	637	109	239	234	993	874
	Skip	36	224	4	72	106	232	201
	Back	12	58	9	11	1	3	6
	F-measure	65	18	53	76	91	75	70
Setup 5 γ=2 α=1 β=20	LC	1372	2725	402	770	594	2278	2069
	NL	3	3	4	3	4	4	4
	Adj	1494	474	234	891	647	1396	1398
	Intra	533	1277	109	239	234	993	874
	Skip	36	31	4	72	106	232	201
	Back	9	7	9	11	1	3	6
	F-measure	23	65	53	76	91	75	70

of subsets of highly dependent packages; i.e., it includes a high number of cyclic dependencies. In this case, the layering result that matches best the authoritative architecture is produced using a setup where the back-calls cost β is set to a very high value compared to the intra-dependencies cost γ (e.g., setup 5). Finally, in the case of JHotDraw, we hypothesized that the best matches for the 4 analyzed versions would be produced using the same setup. As displayed by Table 2, this is the case for JHotDraw 7.0.6, 7.4.1 and 7.6 for which the best results are generated using both setups 4 and 5. But, for JHotDraw 60b1, the best match is generated using setup 3. This is due to: 1) JHotDraw 60b1 containing more layering violations compared to the 3 other versions; and 2) each of the subsequent versions 7.0.6 and 7.4 introducing substantial changes to the framework. Yet, the setups producing the best matches for all JHotDraw versions are the setups that enforce more strictly the layering principles as discussed in this paper (i.e., $\alpha < \gamma < \beta$).

Based on these observations and on the fact that JHotDraw was designed as an example for a well-designed framework, we hypothesized that the setup that produces the best matches for most of the versions of JHotDraw is the one that corresponds to the common understanding of the layered style constraints. This is the case of Setup 4 (i.e., the results of setup 4 and 5 are the same but we consider the first setup that gives most of the best results). To verify our hypothesis, we analyzed the density of violations found in each project. To do this, for each solution that best matches the system's architecture (greyed cells in Table 2), we compared the number of each type of dependency (i.e., intra-dependencies, skip-calls and back-calls) to the total number of dependencies in the system. Fig. 4 displays the dependencies by type for the best matched solution of each project. JFreeChart have the highest percentage of intra-dependencies (71%) relative to the other dependencies. JHotDraw 6.0.b1 has the lowest percentage of intra-dependencies (17%) while JHotDraw 7.0.6 has the lowest percentage of back-calls (0.1%). For the four versions of JHotDraw, the density of violations relative to the project size is smaller than the density of violations in the two of the three projects (i.e., JUnit and JFreeChart). These findings confirm our hypothesis which is consistent with the fact that JHotDraw is known to be well-designed. They also strongly suggest that setup 4 is the one that most corresponds to the common understanding of the styles constraints with respect to our first research question. This will be investigated more in future works.

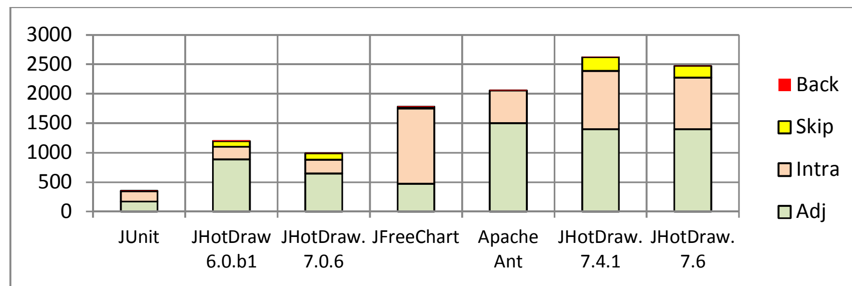


Fig. 4. Dependencies by type for the best matched solution of each project

Regarding our second question, the analysis of four versions of JHotDraw led us to some interesting observations. The best solutions for both versions 6.0.b1 and 7.0.6 have approximately the same layering cost (LC). Furthermore, the layering solutions are stable, i.e., an existing package in both versions is assigned to the same layer in the two best layering solutions. The same observations were made when we compared the results for versions 7.4.1 and 7.6. This confirms the stability of our layering recovery algorithm. Moreover, for the versions 7.0.6, 7.4.1 and 7.6., the best results are returned by the same setup. This suggests the stability of the set of cost values that yield the layering that matches best the architecture of the system across its revisions. It also suggests that JHotDraw maintains the same level of conformity to the layering principles through its evolution. To confirm this, we analyzed eight (8) versions of JHotDraw. Fig. 5 displays for each of these versions their layering cost (LC) using setup 4 and their total weight of package dependencies. Interestingly, Fig. 5 reveals that the evolution of the layering cost of JHotDraw through these 8 versions followed a linear trend line. This strongly suggests that JHotDraw maintains the same level of conformity to the layering principles through its evolution. Future work will investigate whether this trend line applies or not to other layered systems.

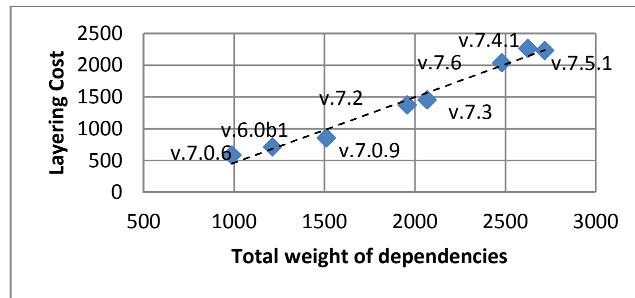


Fig. 5. Evolution of the layering cost of JHotDraw using setup 4

5.3 Threats to validity

Conclusion validity: To find out which setups return the most accurate layered solutions, we compared these solutions to authoritative architectures which come in part from the manual work of students. This issue is related to the lack of comparison baselines in the software architecture community. However, the students who participated to the experimentation were chosen based on their experience and knowledge of the analyzed systems.

Internal validity: The parameters of the tabu search were set through preliminary tests and may not have the best possible values. In any case, as a meta-heuristic, tabu search cannot guarantee a global optimal solution. We were however able to confirm that for the smallest system (JHotDraw 60b1), the algorithm was returning the global optimum. Indeed, with 3 layers and 17 packages, it was possible to examine and evaluate (with a run time of about half an hour) each of the 3^{17} possible solutions.

External validity: The experiment has been conducted on a sample of four open source Java projects. While all these projects are known to be layered systems, the observed results may not be generalizable to other projects. To minimize the threats, we have analyzed several versions of a layered system that is purported to be of good quality (i.e., JHotDraw). We plan as a future work to analyze other existing layered systems including commercial software systems.

6 Related Work

The work in this paper is related to the approaches proposed to recover layered architectures (e.g., [9-15]). Most of these approaches rely on some criterion or heuristics (e.g. [10-14]) in the process. Muller et al. [9] propose an approach aiming at supporting users in discovering, restructuring and analyzing subsystem structures using a reverse engineering tool. The proposed process involves the identification of the layered subsystem structures. The layered structure is obtained through the aggregation of system's entities into building blocks using composition operations based on principles such as low coupling and high cohesion. Schmidt et al. [28] introduced a framework that supports the reconstruction of software architectures. A greedy algorithm is used to generate clusters based on low coupling and high cohesion criteria. These clusters are then assigned to layers based on their ratio of fan-in and fan-out. Unlike [9] and [28], our approach does not include an aggregation phase since we work at the package level. However, our approach can be applied at a lower level of granularity (i.e., at class level).

Hassan and Holt [14] propose a semi-automatic approach to reconstruct the architecture of web applications. For this purpose, they analyze the source code to extract artifacts which are clustered according to some heuristics and refined using human expertise. Andreopoulos et al [13] propose a clustering algorithm which exploits both static and dynamic information to recover layered architectures. This approach assigns highly interdependent files to top layers and less interdependent files to bottom layers. Laval et al. [11] propose an approach which resolves cyclic dependencies and decomposes a system into layers. They rely on two heuristics to find undesired cyclic dependencies which are ignored when building layers of the system. In [10], the authors proposed 3 layering principles (skip-call, back-call and cyclic dependency) and a set of metrics that measure the violation of these principles. Although these principles are focused on detecting violations, they are related to the principles discussed in this paper. In [12], a semi-automatic approach is proposed to identify software layers. Classes that are used by many other classes are grouped in the lowest layer while classes that rely on many other classes are grouped in the highest layer. The remaining classes are assigned to a middle layer. The same technique is used in [29] where a lexical-based clustering is performed to further decompose each layer into modules. In [11, 12, 13 and 29], it is assumed that a module that does not have fan-out dependencies belongs to the lowest-level layer and conversely a module that does not have fan-in dependencies belongs to the highest-level layer. However, a module encapsulating a common subtask exclusive to components of a middle-level layer, will not

have any fan-out dependency but still belongs to this middle-level layer. Likewise, a module that starts some specific service of a middle-layer may not have any fan-in dependency but still belongs to this middle-level layer. Unlike all these approaches, we do not rely on any heuristic or criteria. Our approach relies on the constraints of the layering style and a set of parameters that express how rigorously the designer applied these constraints.

7 Conclusion

Recovering architectural views from existing software systems remains a challenging problem and will remain a topical issue in software engineering. In this paper, we proposed an approach to recover the layered architecture of object oriented systems using the principles of the layering style and the designer's input which describes how strictly he applied these principles when designing the analyzed system. We revisited and analyzed the layered style and retained two important principles. These principles were translated into a set of layers assignment cost factors that help formulating the layering recovery problem as a specific case of QAP, namely a Quadratic Semi-Assignment Problem (QSAP). To experiment this formulation, we implemented a tool that extracts packages' dependencies from object oriented java systems, and an algorithm to solve the layering recovery QSAP. We tested the approach on four open source Java projects using several setups with different sets of values for the layering cost factors. This experimentation yielded interesting results.

While the results of the approach are promising, we plan to extend it in different ways. In the short-term, we want to handle issues related to library components also called omnipresent modules. Library components obscure the system's structure if considered during the decomposition of the system [9]. These components may be clustered into a vertical layer (called transversal layer in [2]) to ease the recovery of the layered architecture. In the mid- to long-term, we need to perform more experiments and analysis to properly tune the cost factors. We would also like to experiment on domain-specific systems and find out if particular setups (i.e., a set of cost factors) are related to specific domains or classes of systems. In this context, the availability of such systems and some architecture description of these systems is a challenging issue. As a future work, we plan to investigate how a given setup can be enforced so that an architect is notified when some changes made to the system introduce a "deviation" from that setup. Finally, we plan to strategy to experiment the approach on software systems implemented using other programming languages and paradigms.

8 REFERENCES

1. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996
2. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, 2003

3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons, 1996
4. Harris, D.R., Reubenstein, H.B., Yeh, A.S.: Recognizers for Extracting Architectural Features from Source Code. The 2nd WCRE, 1995, pp.252-261
5. Ducasse, S., Pollet, D.: Software Architecture Reconstruction: A Process-Oriented Taxonomy. IEEE Trans. on Soft. Eng., July/August 2009, Vol. 35, No. 4, pp. 573-591
6. Mitchell, B. S., Mancoridis, S.: On the Evaluation of the Bunch Search-Based Software Modularization Algorithm. Soft Comput., 2007, vol. 12, Issue 1, pp. 77-93
7. Maqbool, O., Babri, H.A.: Hierarchical Clustering for Software Architecture Recovery. IEEE Transactions on Software Engineering, 2007, vol.33, no.11, pp.759-780
8. Lung, C.H., Zaman, M., Nandi, A.: Applications of Clustering Techniques to Software Partitioning, Recovery and Restructuring. JSS Journal, 2004, vol. 73, pp. 227–244
9. Muller, HA., Orgun, MA., Tilley, SR., Uhl, JS.: A reverse engineering approach to subsystem structure identification. Journal of Software Maintenance, 1993, 5(4):181–204
10. Sarkar, S., Maskeri, G., Ramachandran, S.: Discovery of architectural layers and measurement of layering violations in source code. JSS Journal, 82 (11), 2009, pp. 1891-1905
11. Laval, J., Anquetil, N., Bhatti, M.U., Ducasse, S.: OZONE: Layer Identification in the presence of Cyclic Dependencies. submitted to Science of Computer Programming, 2012.
12. Scanniello, G., D'Amico, A., D'Amico, C., D'Amico, T.: Architectural layer recovery for software system understanding and evolution. SPE Journal, 2010, 40(10), pp. 897-916.
13. Andreopoulos, B., Aijun, A., Tzerpos, V., Wang, X.: Clustering large software systems at multiple layers. Information and Software Technology, 2007, 49(3), pp. 244-254
14. Hassan, AE., Holt, RC.: Architecture recovery of web applications. The 24th International Conference on Software Engineering. ACM Press: New York, NY, 2002; 349–359.
15. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In Proceedings of OOPSLA'05, pages 167–176, 2005
16. Eeles, P.: Layering Strategies. Rational Software White Paper, TP 199, 08/01, 2002.
17. Szyperski, C.A.: Component Software. Addison Wesley, 1998.
18. Bourquin, F., Keller, R.K.: High-impact Refactoring Based on Architecture Violations. In the 11th CSMR, 2007, pp. 149-158.
19. El-Boussaidi, G., Boaye-Belle, A., Vaucher, S., Mili, H.: Reconstructing Architectural Views from Legacy Systems. in the 19th WCRE, 2012.
20. Avgeriou, P., Zdun, U.: Architectural patterns revisited-a pattern language. EuroPlop, 2005
21. Pardalos, PM., Rendl, F., Wolkowicz, H.: The Quadratic assignment problem-A survey and recent developments. DIMACS, Americ. Mathemat. Society, 1994, vol.16, pp. 1–42
22. Skorin-Kapov, J.: Tabu search applied to the quadratic assignment problem. ORSA Journal on Computing, vol.2, no.1, pp.33-45, 1990.
23. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Boston, 1997
24. MoDisco site: http://www.eclipse.org/MoDisco/
25. OMG Specifications: http://www.omg.org/
26. Barros, M.d.O., Farzat, F.d.A., Travassos, G.H.: Learning from optimization: A case study with Apache Ant. Information and Software Technology, 2015, 57, 684-704.
27. Boaye, B.A., El-Boussaidi, G., Desrosiers, C., Mili, H.: The layered architecture revisited-Is it an optimization problem. In Proc. 25th Int. Conf. SEKE, 2013, pp. 344-349
28. Schmidt, F., MacDonell, S.G., Connor, A.M.: An automatic architecture reconstruction and refactoring framework. In SERA 2011, pp. 95-111
29. Scanniello, G., D'Amico, A., D'Amico, C., D'Amico, T.: Using the kleinberg algorithm and vector space model for software system clustering. In ICPC, 2010, pp. 180-189