TABLE OF CONTENTS

Page

INTRODUCTION 1

CHAPTER 1 LITERATURE REVIEW 9
1.1 Basic concepts..9
1.1.1 What is software architecture?.. 9
1.1.2 Architectural styles 10
1.1.2.1 Definition 11
1.1.2.2 Example of architectural style: the layered style 13
1.1.3 Architectural views 16
1.1.4 Modernization of software systems 17
1.1.5 KDM: the Knowledge Discovery Metamodel 21
1.1.6 Architectural reconstruction... 24
1.2 Software architecture reconstruction approaches 26
1.2.1 Approaches targeting architecture recovery in general 28
1.2.2 Approaches targeting the layered style 31
1.3 Synthesis 36
1.3.1 Limitations of the software architecture recovery approaches 38
1.3.2 Limitations of software architecture recovery approaches targeting the layered architectures 40

CHAPTER 2 STRUCTURAL-BASED LAYERING APPROACH 43
2.1 Extracting layering rules 43
2.1.1 The abstraction rules 44
2.1.2 The responsibility rule 45
2.1.3 The transversality rule.. 46
2.1.4 The protection against variations rules 47
2.2 Overview of the proposed approach 48
2.3 Facts extraction 49
2.4 The layered architecture recovery as an optimization problem 51
2.4.1 Translating the layered architecture recovery into an optimization problem 51
2.4.1.1 Extracting layers' dependency attributes from the Incremental Layer Dependency rule 52
2.4.1.2 Extracting constraints from the layered style............................ 53
2.4.1.3 Translating the Layering recovery problem into an Optimization Problem 54
2.4.2 Solving the layering recovery optimization problem.............................. 56
2.4.2.1 Using metaheuristics to solve optimization problems 56
2.4.2.2 Using hill-climbing to solve the layered recovery problem....... 57
2.4.2.3 On the stochasticity of the layering algorithm........................... 61
2.4.2.4 Incremental computation of LaQ 61

2.5 The layered architecture recovery as a quadratic semi-assignment problem 62
2.5.1 Factors for Layers Assignment 62
2.5.2 Layers Recovery as a Quadratic Semi-Assignment Problem 64
2.5.3 Solving the layering recovery QSAP 65
2.5.3.1 A tabu search based layering algorithm 66
2.6 Chapter summary 68

CHAPTER 3 REALEITY: A TOOL FOR RECOVERING SOFTWARE LAYERS FROM OBJECT ORIENTED SYSTEMS 69
3.1 Description 69
3.2 Extraction 70
3.3 Layering 71
3.4 Visualization 73
3.5 Commands 75
3.6 Example 76
3.7 Summary 79

CHAPTER 4 EVALUATING THE STRUCTURAL-BASED RECOVERY APPROACH 81
4.1 Experimentation setup 81
4.1.1 Experimentation questions 81
4.1.2 Analyzed systems 84
4.1.3 Setting the parameters of the layering algorithms 85
4.2 EQ1: what is the effect of altering the values of the factors on the convergence of our algorithms? 87
4.2.1 Experimental results with hill climbing 88
4.2.2 Experimental results with tabu search 94
4.2.2.1 Analysis of the influence of the number of iterations 101
4.2.2.2 Analysis of the influence of the tabu list 102
4.2.3 Comparison of the two layering algorithms 104
4.3 EQ2: What are the values of factors (ap, ip, sp and bp) that best correspond to the common understanding of the layered style? 106
4.4 EQ3: How do the layering results evolve across evolutions of a software system and what does it tell about the architectural evolution of the system? 110
4.4.1 Stability of the algorithm 110
4.4.2 Similarity of the set of values of the setups factor that yield the layering that matches the known architecture of the system across its revisions . 113
4.5 EQ4: is the layering approach performant regarding the size of the system at hand? 115
4.6 EQ5: Is the approach more authoritative than other architecture recovery approaches? 116
4.7 Threats to validity 118
4.8 Chapter summary 120

CHAPTER 5 LEXICAL AND STRUCTURAL-BASED LAYERING RECOVERY APPROACH 121

5.1 Overview of the proposed approach 122
5.2 Facts extraction 124
5.2.1 LDA (Latent Dirichlet Allocation) 125
5.2.2 Extracting packages'keywords 127
5.2.3 Using LDA to generate topics from packages' keywords 128
5.3 Recovery of the system's responsibilities 133
5.3.1 Measuring linguistic cohesion and coupling of packages 133
5.3.2 A fitness function to assess the linguistic quality 136
5.3.3 Using the hill climbing to recover layers' responsibilities 137
5.4 Assigning responsibilities to layers 138
5.4.1 The layering of responsibilities as an optimization problem 138
5.4.2 Using the hill climbing to assign clusters to layers 140
5.5 Chapter summary 142

CHAPTER 6 EVALUATING THE LEXICAL AND STRUCTURAL-BASED LAYERING RECOVERY APPROACH 143
6.1 Experimental design 143
6.1.1 Implementation 143
6.1.2 Dataset 144
6.1.3 Experimentation questions 144
6.1.4 Experimental settings 149
6.2 Experimental results 150
6.2.1 Analysis of the clusters of packages (EQ1) 151
6.2.1.1 When do the lexical clusters match the package structure? 152
6.2.1.2 When do the lexical clusters not match the package structure? 154
6.2.2 Analysis of the best layering results per system and setup (EQ2) 157
6.2.3 Comparison with another layering recovery approach (EQ3) 162
6.2.4 Influence of the calibration of the LDA parameters on the layering results 164
6.3 Threats to validity 165
6.4 Chapter summary 167

CONCLUSION AND FUTURE WORKS 169
Conclusion 169
Limitations 171
Local optimum issue 171
Calibration of the algorithms used to recover the layered architectures 172
The coverage of a subset of rules 172
The use of a limited number of sources of information 173
Future works 173
Recovery of the architectures of domain-specific systems 173
Recover architectures compliant to other architectural styles 174
Use the recovery results to initiate a refactoring process 174

APPENDIX I LAYERING RESULTS OBTAINED WITH THE STRUCTURAL-BASED APPROACH 177

APPENDIX II MANUAL DECOMPOSITIONS OF THE ANALYZED SYSTEMS 199

LIST OF REFERENCES 207

LIST OF TABLES

Page

Table 3.1 Explanation on the elements in the layering recovery wizard 72

Table 4.1 Statistics of the analyzed systems 85

Table 4.2 Setups chosen for the 5 scenarios 87

Table 4.3 LaQ variations with the SAHCLayering algorithm applied on Apache 1.6.2 88

Table 4.4 LaQ variations with the SAHCLayering algorithm applied on JUnit 4.10 89

Table 4.5 LaQ variations with the SAHCLayering algorithm applied on JFreeChart 1.0.15 89

Table 4.6 LaQ variations with the SAHCLayering algorithm applied on jEdit 5.0.0 90

Table 4.7 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 60b1 90

Table 4.8 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.0.7 91

Table 4.9 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.4.1 91

Table 4.10 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.6 92

Table 4.11 Percentage of the optimal solutions per setup and per analyzed system ...93

Table 4.12 Correlation between percentage of optimal solution's quality 93

Table 4.13 LaQ variations with the TabuLayering algorithm applied on Apache 1.6.2 95

Table 4.14 LaQ variations with the TabuLayering algorithm applied on JUnit 4.10 95

Table 4.15 LaQ variations with the TabuLayering algorithm applied on JFreeChart 1.0.15 96

Table 4.16 LaQ variations with the TabuLayering algorithm applied on .... jEdit 5.0.0 96

Table 4.17 LaQ variations with the TabuLayering algorithm applied on JHotDraw 60b1 97

Table 4.18 LaQ variations with the TabuLayering algorithm applied on JHotDraw 707 97

Table 4.19 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.4.1 98

Table 4.20 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.6 98

Table 4.21 Percentage of the optimal solutions per setup and per analyzed system .. 99

Table 4.22 Correlation between percentage of optimal solution's quality 100

Table 4.23 The identicality of the optimal quality LaQ 104

Table 4.24 Best results returned by the *SAHCLayering* recovery algorithm 107

Table 4.25 Comparison of the packages per layer for versions 7.4.1 and 7.5.1 112

Table 4.26 Comparison of the packages per layer for versions 7.5.1 and 7.6 112

Table 4.27 JHotDraw Total weight dependencies and the layers' dependency attributes values obtained using setup 4' of the scenario 1 113

Table 4.28 Correlation values with the size of the system 114

Table 4.29 Execution times for all the versions of JHotDraw using Setup 4' of scenario 1 116

Table 5.1 Sample topics extracted from three packages of JHotDraw 7.0.7 135

Table 6.1 Statistics of the analyzed systems 144

Table 6.2 Distribution of packages according to the identified factors 153

Table 6.3 Average and standard deviation of the layering results obtained with and without lexical information for the 3 factors setups 158

Table 6.4 Layering results with and without lexical information for the 5 setups of factors 159

Table 6.5 Precision, recall and F-measure using the Lattix, the NLI and the LI approaches 163

clicours.com

LIST OF FIGURES

Page

Figure 0.1 Phases of the research methodology 4

Figure 1.1 Structure of a layered architecture 13

Figure 1.2 Illustration of the TCP/IP communication protocol.................................16

Figure 1.3 Lifecycle of an information system 18

Figure 1.4 ADM Horseshoe Lifecycle 20

Figure 1.5 The layers of KDM and their respective packages 22

Figure 1.6 An example of a system, its architecture and the layering obtained applying different existing approaches 41

Figure 2.1 Overview of the approach...49

Figure 2.2 Example of the calculation of the 4 types of layer dependencies 53

Figure 2.3 An example of a layered system and its related matrices 64

Figure 3.1 ReALEITY design 70

Figure 3.2 Layering recovery parameters wizard: case of JHotDraw........................72

Figure 3.3 ReALEITY interface...74

Figure 3.4 The two commands of ReALEITY 75

Figure 3.5 Progression of the layering phase for a system named JHotdraw 7.0.6 76

Figure 3.6 JHotDraw 7.0.6 extracted facts 77

Figure 3.7 JHotDraw 7.0.6’s resulting layered architecture......................................77

Figure 3.8 JHotDraw 7.0.6’s layers’ dependency attributes results 78

Figure 3.9 ReALEITY interface after the layering of JHotDraw 706.........................78

Figure 4.1 Density of dependencies by type for the best matched solution of each system 109

Figure 5.1 Overview of the proposed approach 122

Figure 5.2 Source code of the XMLGenerator class 130

Figure 5.3 Source code of the XSLTTransformer class 131

Figure 6.1 Example of clustering to illustrate the computation of C2SC 146

Figure 6.2 An excerpt of the recovered layering of Apache Ant 1.6.2 160

LIST OF ALGORITHMS

Algorithm 2.1 A high level view of the layering algorithm 58

Algorithm 2.2 Hill climbing based optimization algorithm 60

Algorithm 2.3 A high level view of the tabu search-based layering algorithm.................67

Algorithm 5.1 A high level view of the clustering algorithm 137

Algorithm 5.2 Hill climbing based optimization algorithm 141

LIST OF ABREVIATIONS

ACDC	Algorithm for Comprehension-Driven Clustering
ADM	Architecture-Driven Modernization
API	Application Programming Interface
ARC	Architecture Recovery using Concerns
ARCADE	Architecture Recovery, Change, And Decay Evaluator
ASTM	Abstract Syntax Tree Metamodel
C2SC	Clusters to Structure Conformance
CCBC	Conceptual Coupling Between Classes
CCM	Conceptual Coupling Between Methods
CCP	Conceptual Coupling Between Packages
CCF	Conceptual Cluster Factor
CO	Combinatorial Optimization
CQ	Conceptual Quality
GUI	Graphical User Interface

FAMIX	the FAMOOS Information Exchange Model
FTP	File Transfer Protocol
HC	Hill Climbing
GA	Genetic Algorithm
ILQ	Individual layering quality
ILD	Incremental Layer Dependency
IP	Internet Protocol
ISO/IEC	International Organization for Standardization/International Electrotechnical Commission
KDM	Knowlegde Discovery Metamodel
LaQ	Layering quality
LDA	Latent Dirichlet Allocation
LI	Lexical Information
LQ	Lexical Quality
LSI	Latent Semantic Indexing
MDG	Module Dependency Graph

MQ	Modularization Quality
NAHC	Next Ascent Hill Climbing
NLI	Non Lexical Information
NP	Non Polynomial
OMG	Object Management Group
OO	Object Oriented
OSI	Open Systems Interconnection
PASTA	Package Structure Analysis Tool
QAP	Quadratic Assignment Problem
QSAP	Quadratic Semi Assignment Problem
ReALEITY	REcovering softwAre Layers from objEct orIenTed sYstems
SAHC	Steepest Ascent Hill Climbing
SMM	Structured Metrics Meta-Model
TCP	Transmission Control Protocol
VSM	Vector Space Model

XMI Metadata Interchange

XML Extensible Markup Language

XSLT eXtensible Stylesheet Language Transformations

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

ap	factor adjoined to the adjacent dependencies.
bp	factor adjoined to the back-calls.
sp	factor adjoined to the skip-calls.
ip	factor adjoined to the intra-dependencies.
$\varepsilon_{i,j}$	average lexical coupling between all the ordered pairs of packages p_k, p_h respectively belonging to two distinct clusters i and j.
μ_i	average lexical cohesion between all the unordered pairs of packages p_k, p_h comprised in a cluster i.
P_i	package number i.
c_{kl}	the factor of assigning packages i and j to layers k and l.
W ($[W]_{ij} = w_{ij}$)	$m \times m$ dependency weight matrix.
C ($[C]_{kl} = c_{kl}$)	$n \times n$ matrix of layer assignment factors.
x_{ik}	binary decision variable representing the assignment of package i to layer k.
X ($[X]_{ik} = x_{ik}$)	the $m \times n$ package assignment matrix.
$M = (m_{ij})$	package-keyword matrix.
rho	Spearman correlation coefficient.
α	parameter affecting the topic distribution per document.
β	parameter affecting the word distribution per topic.
T	number of topics to be identified from the corpus.
N	number of iterations of the Gibbs Sampling method.

clicours.com

INTRODUCTION

Context

Software systems are critical assets for enterprises since they embed an important knowledge acquired over the years (Comella-Dorda et al., 2000). Due to the rapid evolution of the technologies, they progressively become the embodiment of the following expression:"*Legacy code is code written yesterday*" (Seacord et al., 2003). The technologies on which they rely on gradually tend to disappear as they are replaced by new efficient ones. Thus, the hardware used by these systems increasingly becomes rare which makes their replacement problematic. Besides, this hardware is expensive to maintain and might not be conciliable with new organizational IT purchasing policies (Sommerville, 2007). In addition, the programming languages with which these systems have been developed in the past lack experimented developers (Comella-Dorda et al., 2000), since most of them are either retired or have joined other companies. To this end, these systems need to be modernized in order to face all these challenges and to be able to keep meeting the needs for which they were designed as well as the new ones that arise. Several technological solutions have been proposed as part of the modernization process: more advanced programming languages, new running platforms, modern graphical user interfaces, databases gateways, and so on (Comella-Dorda et al., 2000). However, only a small percentage of modernization projects complete successfully while respecting the time frame allocated to them (Seacord et al., 2003). Sneed's study on reengineering projects has shown that over 50% of these projects fail (Sneed, 2005). This is notably due to the lack of formalization and standardization (Kazman et al., 1998). It is therefore necessary to find ways and means to improve the modernization process.

In particular, to carry out the modernization of an existing software system, it is mandatory to first understand its architecture. However the as-built architecture is often insufficiently documented (Stoermer et al., 2003). Moreover, this architecture has often deviated from the initial design because of the changes undergone by the system. Hence, an architecture

recovery process is required to reconstruct and document its architecture. The reconstructed architecture notably enables to understand the system and to restructure it as needed. In the context of this thesis, we are interested in recovering layered architectures of object oriented systems.

Research problem

Many works have been proposed to support the architecture recovery process (e.g., Sangal et al. 2005; Mitchell et al. 2008; Garcia et al. 2013; Maqbool and Babri 2007; Harris et al., 1995; Tzerpos and Holt, 2000; Wiggerts, 1997; Lung et al., 2004). These approaches generally rely on properties such as high-cohesion and low-coupling to cluster elements of the system under analysis into layers so as to reconstruct its architecture. However, these approaches usually target specific languages and systems and do not use a standard representation of the data of the system under analysis (El Boussaidi et al., 2012; Kazman et al., 1998). Consequently, resulting tools do not interoperate with each other (Ulrich and Newcomb, 2010). Besides, most of the architecture recovery approaches do not take into account the architectural style of the analyzed system (e.g., Mitchell et al. 2008; Saiedi et al., 2015) whereas software systems are practically built using architectural styles. Our focus in this thesis 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., (Müller et al., 1993; Laval et al., 2013; Sarkar et al., 2009; Andreopoulos et al., 2007; Scanniello et al., 2010a; Sangal et al., 2005a)). However, these approaches usually rely on algorithms that use some heuristics or some particular criterion (e.g., the number of fan-in and fan-out dependencies of a module) to partition elements of the analyzed system into layers. This might lead to partitions with either very few layers (e.g., in case of a heuristic based on highly connected modules) or too many layers (e.g., in case of heuristics to resolve cyclic dependencies). In both cases the so-obtained layered architectures might be too permissive with violations of the style's constraints. In addition, the study by Garcia et al. (Garcia et al., 2013) showed that the overall accuracy of existing architecture recovery techniques is relatively low: in particular, the average accuracy yielded by the best recovery

techniques studied was under 55% for most of the analyzed systems (Garcia et al., 2013). This calls for the proposal of more accurate recovery techniques.

Research Objectives

Our motivation is to support architects and designers in the modernization process of existing systems. To this end, our goal is to develop techniques that will enable the automatic extraction of architectural information. In the context of this thesis, our main objective is to build an approach that exploits the rules of the layered architectural style to recover a layered view of the system under analysis. For this purpose, we decompose our main objective into the following sub-objectives:

- identify the rules that characterize the layered architectures,
- derive layers' dependency attributes and constraints from these rules,
- rely on these layers' dependency attributes and constraints to propose algorithms supporting the recovery of layered architectures,
- assess these algorithms and draw lessons about the feasibility and limitations of the proposed approach.

Research Methodology

To achieve our objectives, we propose an approach that relies on the OMG's standard for software modernization (i.e., the KDM specification standard (OMG Specifications, 2015)) to reconstruct and document software architectural views of existing systems. We focus on systems built according to the layered style which is a widely used pattern to structure large software systems. From the analysis of several definitions and descriptions of the layered style, we identify a set of rules – in the literature, these rules are sometimes referred to as principles – that can guide the recovery of a layered architectural view of software systems. We rely on these rules to derive a set of layers' dependency attributes and constraints that a layered software system should satisfy. We then translate the problem of recovering layers of

software systems into an optimization problem that to be solved using search-based algorithms.

To reach our research objectives, we follow the three-phase research methodology illustrated by Figure 0.1. The different phases of this methodology are described in the following sections.

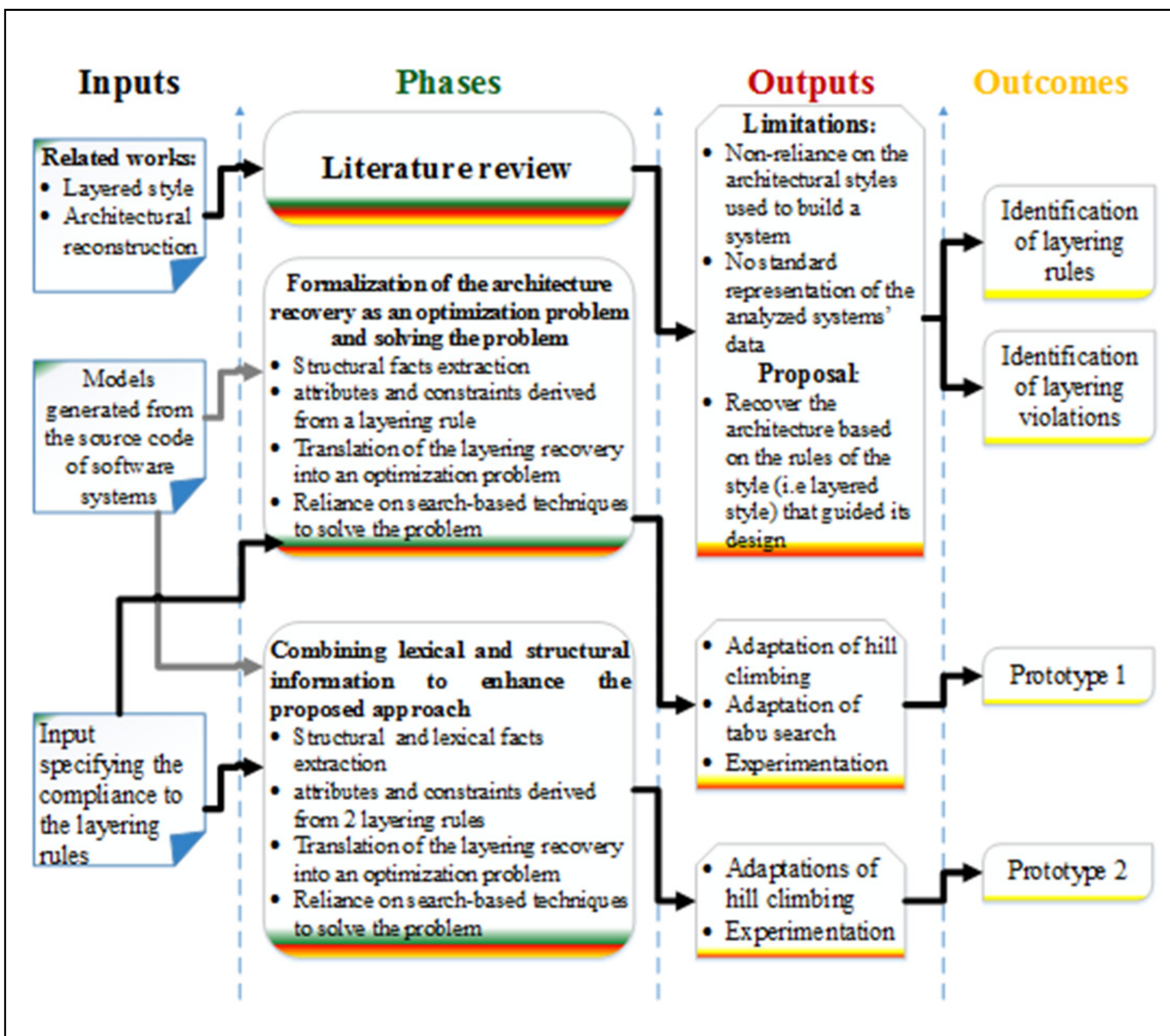


Figure 0.1 Phases of the research methodology

Phase 1: literature review

The first phase of our research methodology consists in studying the existing works related to our research theme. This phase takes as input various books and articles describing the layered style as well as the architectural recovery techniques. This phase leads to the identification of the limitations of the existing works and enables us to define our proposal. The outcomes of this first phase are the identification of the layering rules as well as the layering violations. In particular, we classify these rules into four categories: abstraction, responsibility, transversality and protection against variations rules.

Phase 2: Formalization of the architecture recovery as an optimization problem and solving the problem

The second phase of our research methodology is tasked with translating the layering recovery problem into an optimization problem that is solved. To this end, this phase starts with the choice and extraction of the facts that will populate the layered views of the system at hand during the recovery process. These facts correspond to concepts as well as relationships between concepts found in the system under study. These elements should be selected regardless of the analyzed system's specificities (programming languages, platform, etc.). These concepts and relationships should be relevant to the architectural view that we want to recover. In this phase, we focus on an abstraction rule. We therefore extract structural facts from the set of models describing the source code of the OO system under analysis provided as input. These models are compliant with the OMG's standard for software modernization (i.e., the KDM specification standard (OMG Specifications, 2015)).

The next step of this phase consists in deriving layers' dependency attributes and constraints from the considered abstraction rule. These layers' dependency attributes and constraints help translating the recovery of layered architectures into an optimization problem that we will solve using a search-based algorithm: an adaptation of the hill climbing. We then refine this optimization problem into a specific optimization problem known as the quadratic

assignment problem (QAP). We solve the latter using another search-based algorithm: an adaptation of the tabu search. Each search-based algorithm proposed in this phase relies on an input indicating the degree of compliance to the considered abstraction rule. We evaluate each of these algorithms through experimentations carried out on five open-source systems.

This phase results in the development of a tool that automates the recovery of layered architectures through the implementation of the proposed algorithms. The outcomes of this phase also include the proposed recovery approaches.

Phase 3: Combining lexical and structural information to enhance the proposed recovery technique

The third phase of our research methodology consists in combining lexical and structural information to improve the recovery techniques described in the second phase of the methodology. This third phase starts with the choice and retrieval of the facts that will populate the layered views of the analyzed system during the recovery process. As stated in phase 2, the selection of these facts should be independent of the analyzed system's specificities. This phase focuses on an abstraction rule and on a responsibility rule. Thus, we will extract structural facts as well as lexical facts from the set of models describing the source code of the input system.

In the next step of this phase, we derive measures from a responsibility rule. We use these measures to formulate the recovery of layers's responsibilities into an optimization problem that we will solve using a search-based algorithm: an adaptation of the hill climbing. We will then derive layers' dependency attributes and constraints from the considered abstraction rule. We will use these attributes and constraints to translate the problem of assigning the recovered responsibilities to layers into an optimization problem. We will solve the latter using a search-based algorithm: an adaptation of the hill climbing. This algorithm relies on an input indicating the degree of compliance to the considered abstraction rule. We assess

each of the algorithms proposed in this phase through experimentations performed on four open software systems.

This phase leads to the development of another tool automating the proposed algorithms to support the reconstruction of layered architectures. The outcomes of this phase also include the proposed recovery approach.

Thesis roadmap

This thesis is organized as follows:
Chapter 1 provides the definitions of some key concepts and present some related works. Chapter 2 presents the structural-based approach we propose to recover the layered architectures. Chapter 3 describes a tool automating our structural-based recovery approach. Chapter 4 presents a validation of our structural-based approach by applying it to a set of systems known to be layered. Chapter 5 presents the hybrid recovery approach i.e. the lexical and structural based approach we propose to recover the layered architectures. Chapter 6 validates our hybrid recovery approach by performing experiments on a set of layered systems. Finally, we conclude, outline some limitations of our work and discuss some possible future directions of our thesis.

CHAPTER 1

LITERATURE REVIEW

This chapter provides the definitions of some key concepts and discusses some related works. We also survey some architectural recovery approaches and outline their limitations regarding our contributions.

Section 1.1 describes some relevant basic concepts including the software architecture, the architectural styles and views, the modernization, KDM and the architectural reconstruction. Section 1.2 surveys existing software architectures approaches. Finally, Section 1.3 discusses the limitations of these approaches.

1.1 Basic concepts

1.1.1 What is software architecture?

Software architecture is the result of the decomposition of a system into subsystems providing a set of functionalities and collaborating in the realization of its requirements. It is defined by Bass et al. as the structure of the system made of software components, their externally visible properties, and the relationships between these components (Bass et al., 2003; Bass et al., 2012). By casting details aside, the architecture depicts the software systems at a level of abstraction that is high enough to ease the reasoning about the different steps of their life cycle (Ducasse and Pollet, 2009). These include the understanding, construction, reuse, analysis, evolution, and management of the software (Garlan, 2000). Bass et al. further discuss different advantages of an explicitly designed and documented architecture (Bass et al. 2003):

- Stakeholder communication: the architecture represents the system at a high level of abstraction. As such it can serve as a basis for the discussion between the different stakeholders involved in the system development.

- System analysis: explicitly representing the architecture at the beginning of the system development requires some analysis. Indeed, decisions related to architectural design have a significant impact on the ability of the system to respond or not to critical requirements (e.g., maintainability, availability, and performance).
- Large-scale reuse: the architecture of systems that meet similar requirements is often the same and can therefore support large-scale software reuse.

Hofsmeister et al. argue that the different steps of architectural design drive the designers to early take into account the key elements of the design process (Hofsmeister et al., 2000). They explain that the software architecture can act as a design plan that allows negotiating the system requirements and structuring the discussions between the stakeholders. In addition, they suggest that the software architecture is essential to address the complexity of a system.

As stated in (Ducasse and Pollet, 2009), the literature (e.g., (Kazman and Carriere, 1999; Medvidovic and Jakobac, 2006)) generally distinguishes between the conceptual architecture and the concrete architecture. The term concrete architecture designates the architecture that is derived from source code's artifacts. The concrete architecture is also called the as-built, as-implemented, realized, or physical architecture. The conceptual architecture designates in turn the architecture that either exists in human minds or in the documentation of software. The conceptual architecture is also referred as the idealized, as-designed, logical or intended architecture.

1.1.2 Architectural styles

The business value of a system is the fruit of its quality attributes as perceived by acquirers as well as end-users (Sury, Abran and April, 2003). The architecture of a system determines the quality attributes that this system will have at the end of its development. In this context, a quality attribute is defined as "a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders" (Bass et al., 2012). The

organization of the architecture of a system with respect to some quality attributes can be achieved using architectural styles that support these attributes. These quality attributes are also referred as non-functional requirements (Abran et al. 2004; Bourque and Fairley, 2014). Since the lack of quality attributes unsatisfies users, leads to financial loss, and might also threaten lives, quality attributes have become critical for the system (Sury, Abran and April, 2003).

1.1.2.1 Definition

Medvidovic and Jakobak (Medvidovic and Jakobak, 2006) describe the architectural style as “a key design idiom that implicitly captures a large number of design decisions, the rationale behind them, effective compositions of architectural elements, and system qualities that will likely result from the style’s use”. According to Le Métayer (Le Métayer, 1998), "the architecture style can be seen as the "type" (or form) that the architecture must have at run time, that is to say the possible connections between its individual components". It therefore represents a form of codification of the aspects that must be met by the architecture it structures (Bhattacharya and Perry, 2005). An architectural style governs the vocabulary of components and connectors used when instantiating this style, together with a set of constraints on how they should be arranged (Garlan and Shaw, 1996). As such, the definition of an architectural style involves the following concepts:

- The components: they are the "principal processing units and data stores" (Clements et al., 2003). A component is implemented as an abstract unit of software instructions and internal state that allows transforming data through its interface (Fielding, 2000).
- The connectors: they are the "pathways of interaction between the components" (Clements et al., 2003). Hence, a component has a set of connectors determining how it can interact with other components. Connectors allow transferring data from one component’s interface to another without altering these data (Fielding, 2000).
- The constraints: they are the rules that define how the components and connectors must be assembled in order to create a valid instance of the style (Clements et al., 2003). These constraints can either be topological or semantic. A system that is

> compliant to a given architectural style must conform to its constraints at design time and during its evolution (Clements et al., 2003). These constraints express the fundamental rules embodied by the architectural style.

An architectural style is an element of primary importance in the architectural design (Mikkonen et al., 2004). It allows describing a family of architectures conforming to the set of constraints that it specifies (Tamzalit and Mens, 2010). These constraints allow the architectural style to guide the evolution of architecture toward this family of architectures (Tamzalit and Mens, 2010). This being said, most systems need to combine several architectural styles in order to organize their structure, since every architectural style supports specific quality attributes. This is achieved by using an architectural style for structuring the whole system, and using different architectural styles to organize the internal structure of the various parts of the system (Sommerville, 2007). A description of many common architectural styles can be found in (Shaw and Garlan, 1996; Clements et al., 2003; Bass et al., 2003). Examples of such styles include the layered, pipes and filters, and service-oriented styles.

Architectural styles are usually classified into three categories of styles (Clements et al., 2003), namely: the module, the component-and-connectors and allocation styles. The module styles allow describing the architecture of a system as a static partition of its software's functionalities. The component-and-connectors styles allow depicting the runtime behaviour of a system. The allocation styles allow describing the mapping of a software's entities to the resources (e.g., hardware, file systems) of its development and execution environments.

Noteworthy, two distinct "schools of thought" have emerged in the literature with respect to the nature of architectural styles. The first one employs the expression "architectural pattern" (e.g., (Buschmann et al., 1996; Schmidt et al., 2000; Voelter et al., 2004)) while the other employs the expression "architectural style" (e.g., (Shaw and Garlan, 1996; Shaw and Clements, 1997; Bass et al., 2003; Clements et al., 2003). Although there are some differences in the formalization of architectural patterns and architectural styles (Avgeriou

and Zdun, 2005), most authors agree that they are essentially identical (Harrisson et al., 2007).

1.1.2.2 Example of architectural style: the layered style

The origin of the layered style dates back to 1968, year at which Dijkstra (Dijkstra, 1968) laid the foundations of this architectural style. Numerous reference books and papers have since proposed a description of this widely used style (e.g., (Bourquin and Keller, 2007; Buschmann et al., 1996; Clements et al., 2003 et al., 2003; Szyperski, 1998; Eeles, 2002; Laval et al., 2013; Sangal et al., 2005a; Sarkar et al., 2009, Scanniello et al., 2005a)). The layered style allows structuring a system by decomposing it into sets of tasks (Buschmann et al., 1996). Each set of tasks corresponds to a given level of abstraction and represents a layer of the system. Each layer uses the services of the lower layer and provides services to its immediate higher layer. The structure of the system is then achieved by arranging the layers the one above the other, in an increasing level of abstraction. Figure 1.1 illustrates the architecture of a layered system.

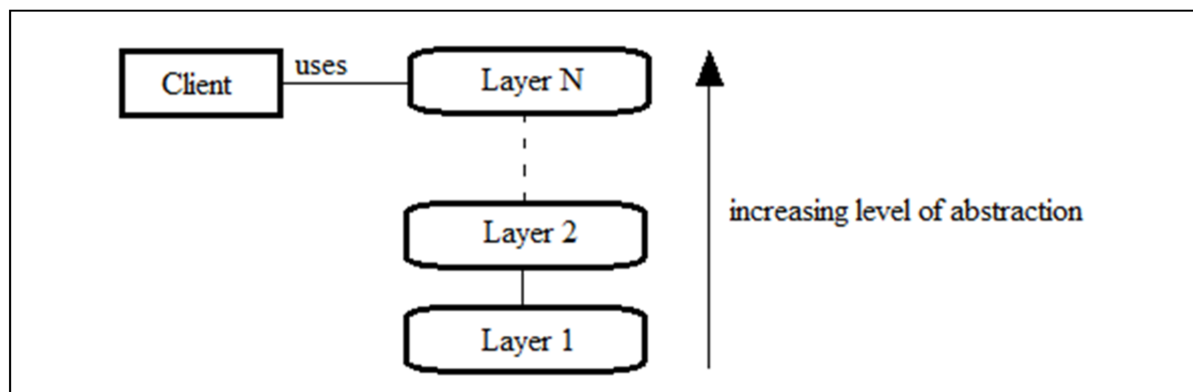


Figure 1.1 Structure of a layered architecture
Adapted from (Buschmann et al., 1996)

In a layered architecture, each layer comprises a set of modules which are cohesive with respect to their responsibilities (Clements et al., 2003). It is common ground that in a well-designed layered architecture, the layers should interact according to a strict ordering relation

(Clements et al., 2003) i.e. a layer may only use services of the next lower layer. This is referred to as strict layering in (Buschmann et al., 1996) and as closed layering in (Szyperski, 1998). 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 (Eeles, 2002). The responsibility-based layering 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 to the bottom layers. Respective examples of these two strategies include the OSI[1] model (Zimmermann, 1980) and the e-learning systems (Paris, 2004). In particular, in the OSI model, a layer uses services provided by lower layers and adds value to them to provide services needed by higher layers.

The layered style promotes many quality attributes such as reuse, portability, maintainability, understandability, and exchangeability. But this style also comes with some liabilities such as the lack of flexibility and a weak performance (Buschmann et al., 1996). To address these liabilities, a current practice is to build layers in such a way that layers are allowed to use services of any lower layer without restriction (Clements et al., 2003). However, this practice is considered as a violation named a skip-call violation in (Sarkar et al., 2009) and layer bridging in (Clements et al., 2003). Exceptionally, a layer may need to rely on a service offered by an upper layer. These dependencies are called back-calls violation in (Sarkar et al., 2009) and are discussed in (Clements et al., 2003) under the name “upward usage”. However, the quality attributes promoted by the layered style are no longer supported when layers are allowed to use services of higher layers without restriction (Clements et al., 2003). This leads to the formation of cyclic dependencies that are likely to make the system monolithic and therefore unbreakable into multiple layers (Sarkar et al., 2009). Hence, the structure of a layered architecture must be a directed acyclic graph or at least a directed graph with very few cycles connecting layers. The former statement relates to the Acyclic Dependency

[1] Open Systems Interconnection.

Principle which is an object-oriented design principle stating that: “the dependencies between packages must form no cycles” (Martin, 2000).

One of the most important concerns when designing a layered architecture is to find the right number of layers. This requires making some compromises since a high number of layers introduces an unnecessary overhead, whereas a low number of layers may lead to a monolithic structure of the system (Buschmann et al., 1996). While the OSI reference model has 7 layers (Zimmermann, 1980), most of web-based applications have three layers. In (Kruchten, 1995), Kruchten recommends defining some 4 to 6 layers of subsystems. The decomposition of the system into an appropriate number of layers and the assignment of tasks to these layers is not always easy (Buschmann et al., 1996).

In layered systems, the components are layers. The connectors are in turn defined by the protocols that determine the interaction between the layers (Shaw and Garlan, 1996). Among the layered style constraints, we can cite that the services of a given layer j are only used by the layer $j + 1$, or that the layer J should be a black box, a gray box or a white box for layer $j + 1$ (Buschmann et al., 1996).

Garlan and Shaw (Garlan and Shaw, 1996) indicate that the communication protocols in layers are the best known examples of this architectural style. Among these is the TCP/IP protocole which is used as an example by Buschmann et al. in (Buschmann et al., 1996). The TCP/IP protocole is used for transferring data on the internet and consists of 4 layers: FTP, TCP, IP and Ethernet. Figure 1.2 illustrates two systems implementing the TCP/IP protocol and interconnected by a physical medium. The communication between the layers is peer-to-peer, and is performed as if two layers of the same level of abstraction and respectively belonging to these systems were exchanging messages without intermediairy (see the dotted lines). But in reality, these messages pass through the layers that are connected by solid lines, as well as the physical medium, before arriving at destination. As such, TCP/IP defines a *virtual protocol*. The information exchanged between the layers can be modified before

being sent. TCP/IP strictly defines the behavior of each layer as well as the structure of the data exchanged between them, in order to facilitate communication between TCP/IP stacks.

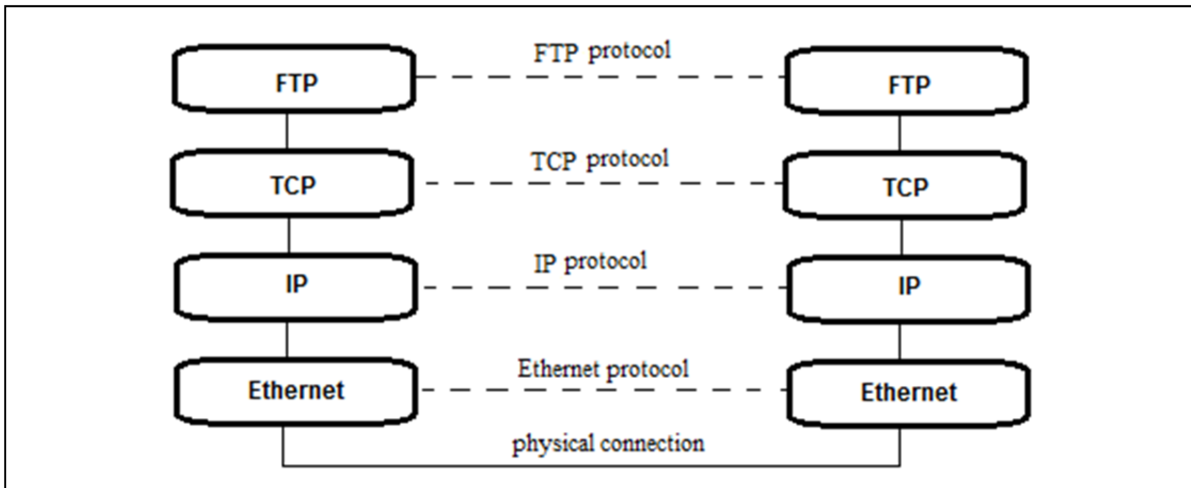


Figure 1.2 Illustration of the TCP/IP communication protocol
Adapted from (Buschmann et al., 1996)

Databases and operating systems are other examples of layered systems (Garlan and Shaw, 1996).

1.1.3 Architectural views

The concept of architectural view refers to a partial representation describing a system as a collection of parts and presenting some of the relationships between them (Clements et al., 2003). Thus, in the same way as the views that have a plumber, electrician or even a carpenter on a home vary, architectural views focus on different aspects of a software system (Seacord et al., 2003). Software architecture can therefore be described using different high-level facets provided by different views (Bourque and Fairley, 2014) that are complementary (El Boussaidi et al., 2012).

An architectural view can be static or dynamic. A static view provides a representation of the system before its execution (Seacord et al., 2003) by illustrating its implementation units and

their structural relations (El Boussaidi et al., 2012). A dynamic view describes in turn a running system (Seacord et al., 2003) by providing a representation of the control flow and the data flow between implementation units (El Boussaidi et al., 2012). For instance, the layered style yields a module decomposition that is a static/structural view of the system; a pipe and filter style yields a dynamic view based on the data flow between components; and a client-server style yields a dynamic view based on the control flow (request/reply interactions) (El Boussaidi et al., 2012).

A view is obtained by applying an architectural style on a system (Clements et al., 2003). For instance, the layered view and the decomposition view are respectively the result of applying the layered style and the decomposition style on a system. In particular, the decomposition view is a static view that depicts the structure of the source code in terms of modules and submodules and shows the repartition of the responsibilities across the system (Clements et al., 2003). A module (class, collection of classes, layer or any artefact coming form the source code) has various properties such as responsibilities, information on its visibility or its authorship (Seacord et al., 2003). A module can aggregate other modules to form a subsystem (Clements et al., 2003). For instance, in a decomposition view, the relationship between modules is a containment relation expressing that a module can be a part of only one aggregate (Clements et al., 2003).

1.1.4 Modernization of software systems

According to Comella-Dorda et al. (Comella-Dorda et al., 2000), the evolution of (legacy) systems is necessary for their viability. Indeed, this evolution is carried out in order to reflect progress in the business practices and allow the adaptation to new technologies. It can span from the simple addition of a field in a database to the entire re-implementation of a system. The evolution of a system can be performed through its maintenance, modernization or replacement. Figure 1.3 depicts the possible categories of evolution of a system throughout its lifecycle.

The dotted line shows the growth of the business needs while the continuous line indicates the functionnalities that the system supports. Thus, maintenance activities allow meeting the business needs for a while, but when the system gets too old, they become less and less sufficient and consideration should be given to modernization. Replacing a system becomes necessary when it can no longer be modified.

Maintenance is the set of activities needed to provide cost-effective support to software (Abran et al., 2004; Bourque and Fairley, 2014). This iterative and incremental process aims at making slight modifications to the system: bug fixes, minor functionnalities enhancements, etc. However, maintenance has some limitations including its cost and the scarcity of expertise in technologies dating back many years.

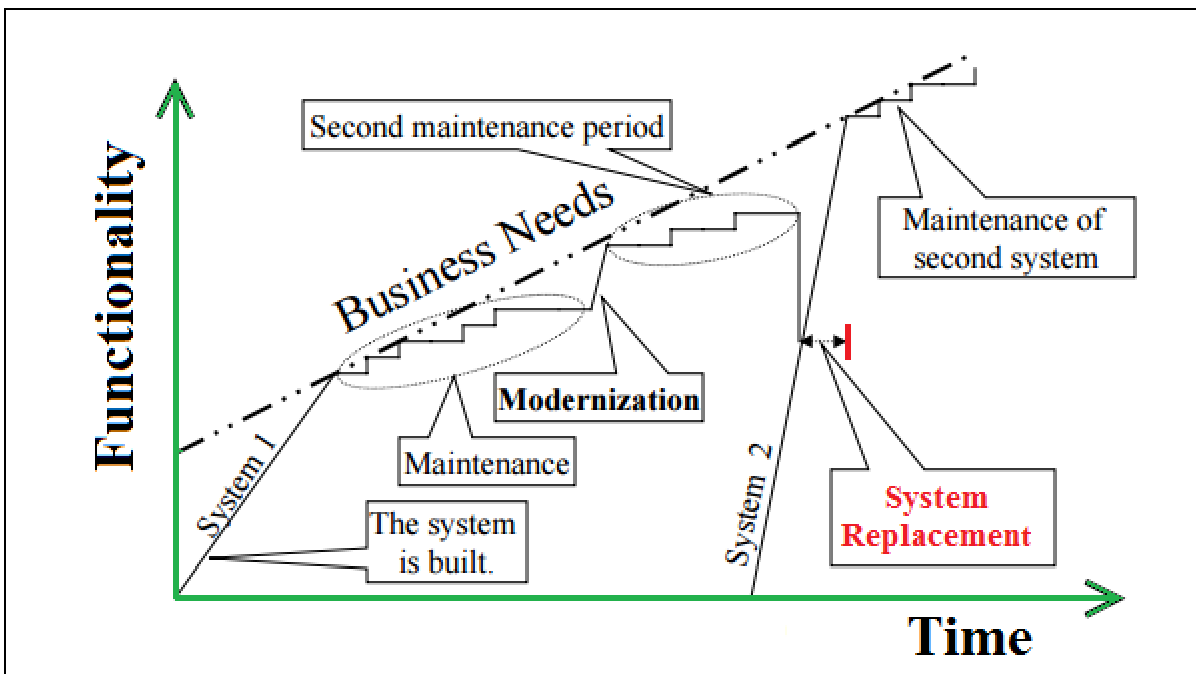


Figure 1.3 Lifecycle of an information system
Adapted from (Comella-Dorda et al., 2000)

Modernization is used when a (legacy) system that still has a significant commercial value, requires further modifications than those performed during maintenance. The modernization process preserves a significant portion of the existing system and may consist in the

restructuration of the system, major functional enhancements, or the introduction of additional software attributes. The modernization is therefore performed at the functionnal/logic level (e.g., OO wrapping[2], components wrapping), at the data level (e.g., XML integration, database replication, use of a database gateway), or even at the level of the users interfaces (Comella-Dorda et al., 2000). The modernization process may consist in (Comella-Dorda et al., 2000):

- ***a white-box modernization***: which requires understanding the code of the system (domain modeling, extraction of the information from the source code and creation of abstractions to understand the system structure) by notably reverse engineering its code. The analysis and understanding of the code might be followed by a restructuring process.
- ***a black-box modernization***: which aims at understanding the system's interfaces via the study of its inputs and outputs. It is generally based on the wrapping and can sometimes require the use of white-box modernization techniques in order to get a better understanding of the system's internals.

The modernization projects can be carried out according to several scenarios. These scenarios can be realized via transformations performed through the incremental migration of the current solution to the target elements solution (OMG, 2007). It is claimed in (OMG, 2007) that these transformations are improved thanks to ADM[3] (OMG, 2007), which proposes to carry out a modernization driven by the architecture. To this end, the ADM Horseshoe model, as displayed by Figure 1.4, is used to describe the transformations made during the modernization process.

This model therefore addresses the discovery of the knowledge embedded in the current solution as well as the description of that solution through three levels of abstraction. These

---

[2] Operation aiming at surrounding a system with a software layer that hides its complexity behind a modern interface (Comella-Dorda et al., 2000).
[3] Architecture-Driven Modernization.

ClicOurs.com

levels respectively correspond to the code, the fonctionnalities and the architecture of the current solution. The Horseshoe model also describes the successive steps leading to the target solution, namely: the reconstruction, the transformation and the refinement. According to ADM the more these steps are performed at a high level of abstraction, the greater the impact of the so-performed modernization. This justifies the choice of performing the modernization at the architecture level.

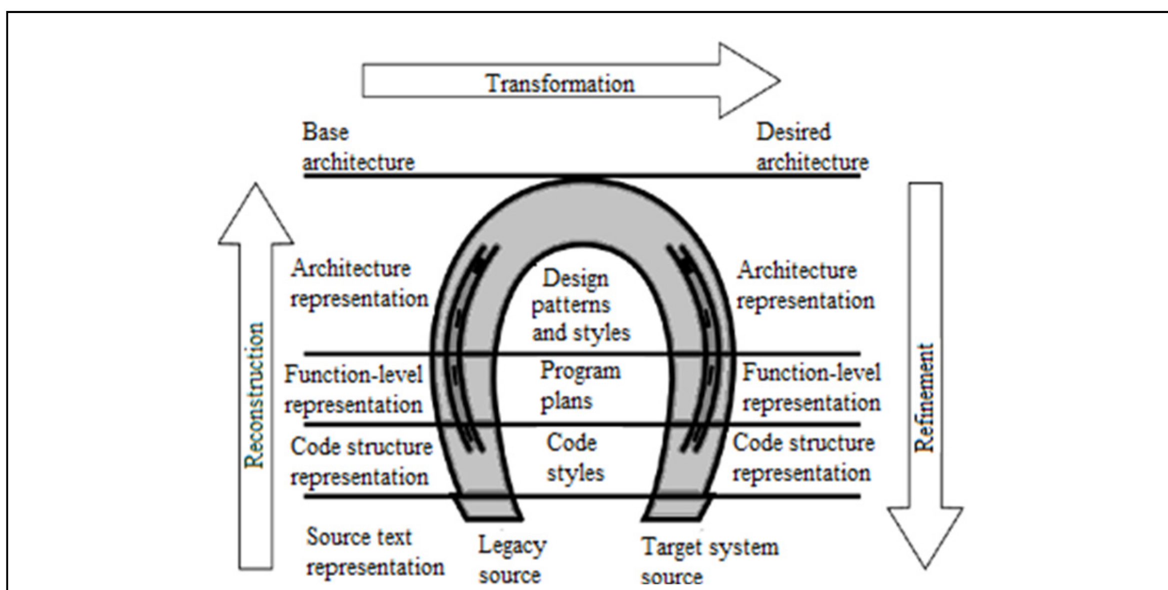


Figure 1.4 ADM Horseshoe Lifecycle
Adapted from (Seacord et al., 2003)

Regarding the replacement of a system, Comella-Dorda et al. (Comella-Dorda et al., 2000) point out that it is suitable for systems that fail to keep pace with the business needs, and for which modernization is either impossible or not profitable. Systems that are undocumented, obsolete or not extensible are often replaced. However, there are some risks relating to the replacement. These include the intensive resources required for its implementation, the need to perform extensive tests on the new system. Besides, there is no guarantee that the new system will be robust or as functional as the former one.

The modernization is therefore an acceptable mid-term solution to evolve a system that has undergone too many maintenance activities and for which a replacement would be too risky.

It often requires understanding first how the system was designed. ADM proposes many standards supporting the modernization of software systems. Among them are ASTM[4], SMM[5] and KDM. We are particularly interested in the latter which is described in (Pérez-Castillo et al., 2011) as a meta-model for describing the software (e.g., entities, relationships, attributes, runtime platform and environments) involved in a modernization process. The next section describes KDM.

1.1.5 KDM: the Knowledge Discovery Metamodel

Most of the methods proposed to extract the knowledge embedded in systems using high-level abstractions are limited in their representations and often suffer from the standardization problem (Pérez-Castillo et al., 2011). Hence, they are not applicable in the same way to all types of systems, domains or even technologies. The KDM standard was created to address this issue. To this end, it allows the modeling and management of the knowledge embedded in a system throughout the modernization process. This reduces the impact of the degradation of the system. Also known under the name ISO/IEC 19506, KDM defines a meta-model to represent existing softwares, their elements, associations and operational environments (OMG Specifications, 2015).

As shown in Figure 1.5, KDM comprises four layers representing the physical and logical artifacts of a legacy information system at different levels of abstraction. Each layer of KDM comprises a set of packages and each package usually defines a specific KDM model.

The KDM infrastructure layer corresponds to the lowest level of abstraction and defines the concepts used in the KDM specification. It contains the following packages:

---

[4] Abstract Syntax Tree Metamodel.

[5] Structured Metrics Meta-Model.

- The ***Core*** package: it defines the basic constructs of KDM. These constructs allow describing the metamodel elements in the different packages of KDM.
- The ***KDM*** package: it defines the infrastructure for other packages in the metamodel. Each KDM representation will thus consist of one or several segments each having several KDM models. It also defines a package extension mechanism that supports extending the semantic of KDM representations through the introduction of new types of extended metamodel elements called *stereotypes*.
- The ***Source*** package: it defines the *InventoryModel* that lists the system's physical artifacts (e.g., source files, images, configuration files and resource files) as KDM entities. It also defines a traceability mechanism for linking these entities to their representation in the code.

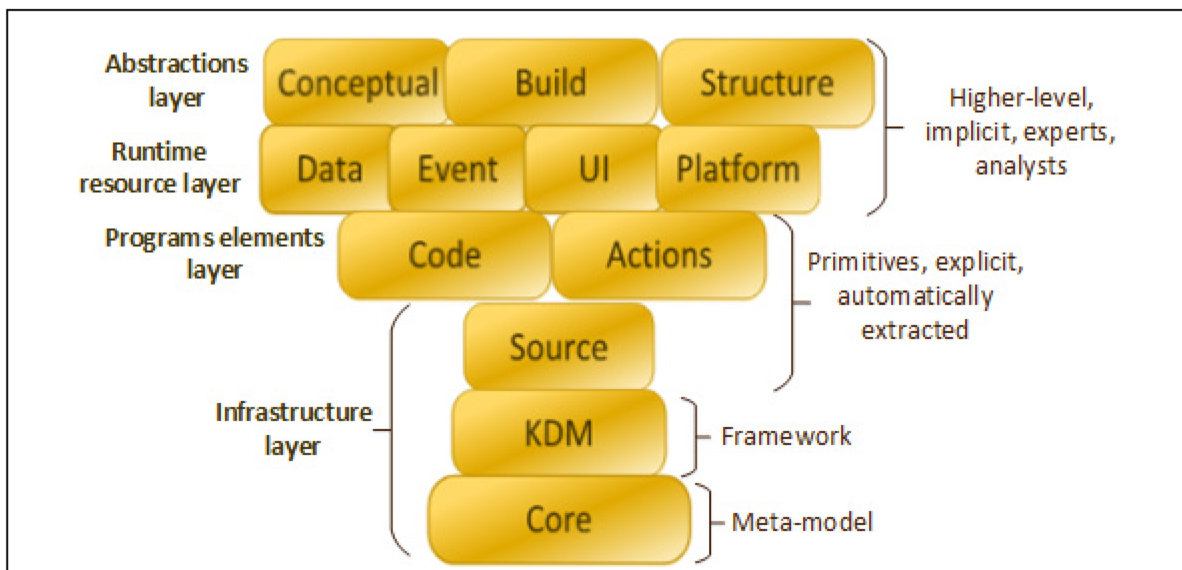


Figure 1.5 The layers of KDM and their respective packages
Adapted from (Pérez-Castillo et al., 2011)

The KDM program elements layer defines a broad set of elements of the meta-model whose goal is to provide an intermediate representation for the various constructions governed by the common programming languages (OMG Specifications, 2015). This representation is independent of these languages. This layer represents the level of implementation of the

program elements and their associations, and therefore provides a logical view of the legacy system. It contains two packages (Pérez-Castillo et al., 2011):

- The ***Code*** package: it defines through the *CodeModel* a set of elements supported by different programming languages (data types, classes, procedures, methods, templates and interfaces).
- The ***Action*** package: it extends the *CodeModel* through more elements representing the behavioral descriptions and relationships of control and data flows between the code elements.

The runtime resource layer of KDM allows representing the knowledge of the legacy system as well as its execution environment. It therefore represents the resources managed by the runtime platform and comprises four packages (Pérez-Castillo et al., 2011) respectively named Data, Event, UI and Platform. We do not dwell on their description since we do not use them in the remaining of this thesis.

The KDM abstraction layer defines a set of elements of the metamodel in order to represent the domain specific knowledge and provides a business overwiew of the legacy information systems. It consists of three packages (Pérez-Castillo et al., 2011):

- The ***Structure*** package: it provides a representation of the architectural components of the legacy system (e.g., subsystems, layers and packages). It defines the *StructureModel* which allows representing the organization of the system's source code. This model allows describing layers, components, architectural views and the relationships between them.
- The ***Conceptual*** package: it represents domain-specific information of a legacy system within the *ConceptualModel*. The latter can be a behavior graph with paths of the application logic and associated conditions.
- The ***Build*** package: through the *BuildModel*, it defines the artifacts related to the engineering view of the legacy system (roles, deliverables and development activities).

KDM is not limited to a particular language, to a given transformation or even to a specific platform. As stated earlier, it is usable in the modernization process defined by ADM. KDM facilitates the modernization of software systems by ensuring the interoperability and the data exchange between tools supplied by different vendors (OMG Specifications, 2015). This interoperability between reverse engineering tools and modernization tools is guaranteed thanks to the representation of its models as XMI files having a well-defined format and complying with the KDM standard.

1.1.6 Architectural reconstruction

As mentioned above, the modernization process defined by ADM includes a first phase called recontruction. This phase consists in generating a high-level representation of the system in order to ease its understanding. It enables the extraction of the system's information corresponding to a given level of abstraction, so as to support the analysis of this system. According to ADM, the impact of modernization increases with the level of abstraction. To this end, we will focus on the reconstruction of the architecture since it corresponds to the highest level of abstraction. According to Riva (Riva, 2002), the architectural reconstruction is a reverse engineering activity aiming at recovering the missing decisions on a system. These decisions are missing because they are either unknown (e.g., lack of documentation, departure of original developers) or recent (e.g., due to changes in the system). Retrieving these decisions leads to the reconstruction of the system's architectural views (Ducasse and Pollet, 2009).

The literature refers to the software architecture reconstruction using different terms: reverse architecting, or architecture extraction, mining, recovery, or discovery. In this context, the discovery designates a top-down process while the recovery refers to a bottom-up process (Ducasse and Pollet, 2009). In particular, the architecture recovery starts from source code to progressively construct a more abstract representation of the system (Ducasse and Pollet, 2009). In this thesis, we will use a bottom-up process to recover the architecture and will therefore use the expression architecture recovery to refer to architecture reconstruction.

Software systems are usually built by combining and composing architectural styles. However, many researchers observed that the continuous growth and evolution of these systems leads to a non-conformance of their architecture with the initial style that guided their design (e.g., (Stoermer et al., 2003; Ducasse and Pollet, 2009). This deviation is mainly due to (El Boussaidi et al., 2012): 1) the conceptual gap between the abstract elements that define a style and the concrete source code constructs that implements the system (Harris et al., 1995); 2) violations of the style constraints due to their misinterpretation; and 3) the continuous and cumulative changes undergone by the system, which increases its complexity and leads to a deviation from its initial design (Seacord et al., 2003). Moreover, the as-built architecture is usually either inexistent, insufficiently and inaccurately documented (Stoermer et al., 2003; Kazman and Carriere, 1999) or even inappropriate for the task at hand (Harris et al., 1995). Hence, a software architecture reconstruction process is required to reconstruct and document its architecture.

Recovering the software architecture supports many architectural related activities spanning from the re-documentation and the understanding of existing systems to their restructuring and migration (El Boussaidi, 2012). According to (Ducasse and Pollet, 2009), recovering the architecture of the system at hand allows restoring high-level abstractions that capture the current software's implementation. The resulting recovered architectural view(s) serves as the software's documentation and eases the understanding or the system at hand. By enabling the identification of components from existing systems, the recovered architecture not only eases their components' reuse but also fosters the migration of these systems toward other platforms supporting new technologies.

The recovered architecture also serves as a medium to conduct diverse analyses such as style conformance, dependence analysis, or quality attribute analysis. These analyses provide assistance by giving useful information during the decision-making process. Another important concern addressed by the recovered architecture is the conformance checking one (Wiggerts, 1997; Ducasse and Pollet, 2009; Stoermer et al., 2003). To this end, the recovered

architecture is used to check whether the system's implementation conforms to its intended architecture.

Recovering the architecture also enables tracing the architectural artifacts back to the source code. This is notably useful to ensure the co-evolution i.e. the synchronization between the respective evolutions of the architecture and the source code so as to avoid the erosion of the architecture (Ducasse and Pollet, 2009) and to constrain its future evolution. Besides, the recovered architecture also supports the restructuration of the system at hand (Wiggerts, 1997).

The reconstruction process is carried out by analyzing the system artifacts (Riva, 2002). As explained by Koschke (Koschke, 2009) and El Boussaidi et al. (El Boussaidi et al., 2012), the architectural reconstruction is done in three steps, namely:

1. The extraction of the data[6] from a system (e.g., structural and lexical information).
2. The construction of higher-level models using the right abstraction technique.
3. The presentation/visualization of the resulting models.

1.2 Software architecture reconstruction approaches

There is a large body of work dedicated to the software architecture reconstruction (e.g., (Scanniello et al., 2010b; Laval et al., 2013; Sarkar et al., 2009; Maqbool et al., 2007; Riva, 2002; Bittencourt and Guerrero, 2009; El Boussaidi et al., 2012; Anquetil and Lethbridge, 1999; Garcia et al., 2011; Saedi et al., 2015)). Architectural reconstruction approaches can be classified using various criteria. The ones proposed by the taxonomy of Ducasse and Pollet (Ducasse and Pollet, 2009) cover a wide range of architectural reconstruction approaches and are described below:

[6] These data are notably retrieved from the documentation of the system, its source code, its execution, the opinion of experts, the domain knowledge and by inferring the architectural information in order to make them more obvious (Riva, 2002; Koschke, 2009).

- **The goals**: they constitute the motivations that guide an architectural reconstruction process. They can for instance be the re-documentation and understanding of a system or the study of the conformity between the concrete and conceptual architecture of a system. The development and maintenance and the analysis of the architecture of a system are other examples of architectural reconstruction goals.
- **The inputs**: they are the elements that are taken as parameters by the architectural reconstruction process. Inputs may include non-architectural data, architectural data such as architectural styles or a combination of both types of data. The non-architectural data include structural, textual and dynamic information as well as human expertise. In particular, the structural information corresponds to the syntactic structure of the source code together with the control and dataflow that it represents (Maletic et al., 2001). The textual information corresponds in turn to the source code's identifiers (e.g., package name, class name method name, method's parameters names and variables names) and comments. Regarding the architectural styles serving as inputs, we are particularly interested by two criteria:
    - ***The rules of the architectural style taken into account during the reconstruction***: the reconstruction process is sometimes about recovering an architecture that complies with a specific style. This architecture should be characterized by a set of rules or best practices conveyed by this style. In the literature, these rules are sometimes referred to as principles.
    - ***The architectural violations considered during reconstruction***: these are the architectural constructions which do not respect the rules of the targeted architectural style, thus violating its integrity. As stated before, for concerns such as performance, a number of violations can be tolerated during the architectural development process.
- **The processes**: An architectural reconstruction activity process may follow a bottom-up process taking as input low level data in order to generate a high-level representation of the system that will ease its understanding. Conversely, if an architectural reconstruction activity proceeds in the opposite direction, it is classified

as a top-down process. A reconstruction approach that combines the two processes is referred as a hybrid process.

- **The techniques**: they refer to the degree of automation of the architectural reconstruction process. Thus, a technique may be quasi-manual (e.g, construction and exploration), semi-automatic (e.g., abstraction through the use of queries, investigation carried out using graph pattern matching or state engine) or quasi-automatic (e.g., clustering and concepts analysis).
- **The outputs**: they refer to the result generated by an architectural reconstruction activity. It may for instance be the architecture of the analyzed system or the analyses carried out on the system at hand.

In the following, we give an overview of the existing architectural recovery approaches, classifying them into two categories, namely: the architectural recovery approaches in general and the architectural recovery approaches targeting the layered style.

1.2.1 Approaches targeting architecture recovery in general

Many approaches were proposed to support architecture recovery using various techniques and producing different tools that support them (Stoermer et al., 2003). Among these techniques is the clustering which is a common practice to support architecture reconstruction (e.g., (Mitchell et al., 2008; Lung et al., 2004; Andritsos and Tzerpos, 2005; Anquetil and Lethbridge, 1999; Corazza et al., 2013; Zhang et al., 2010)). Generally speaking, the clustering is defined as the unsupervised classification of patterns (observations, data elements, feature vectors) into groups (clusters) (Xu and Wunsch II, 2005). Given N objects and a dissimilarity matrix C = (cij), the clustering problem consists in finding a partition of these objects into *m* clusters (i.e. classes) for which the sum of dissimilarities of objects belonging to the same cluster is minimal (Pardalos et al., 1994). In software engineering, many approaches leverage rules such as information hiding, strong cohesion and loose coupling to cluster a software system (Andritsos and Tzerpos, 2005). These approaches usually aim at finding a clustering of the system that maximizes the

cohesion of each component (i.e. cluster) while minimizing the coupling between resulting components (e.g., (Lung et al., 2004; El Boussaidi et al., 2012)).

To identify clusters, these approaches usually take as input the dependencies between software artifacts (functions, source files, etc.) (Andritsos and Tzerpos, 2005). To this end, the clustering process can exploit various sources of information such as naming information (file names, words extracted from the comments of the source code, etc.) or even links associating a developer to the part of the system that she implemented. Finding a solution to a clustering problem allows creating a logical decomposition of a system into subsystems whose management and understanding are much easier. Clustering-based approaches vary mostly depending on the similarity layers' dependency attributes used to determine the resemblance of the components of the analyzed systems and the algorithms used to perform the clustering (Wiggerts, 1997; Andritsos and Tzerpos, 2005).

For instance, the modularization tool called Bunch by Mancoridis et al. (Mancoridis et al., 1998; Mancoridis et al., 1999) uses a family of search-based algorithms such as hill climbing (HC) and genetic algorithms (GA) to produce a high level view of an analyzed system. The approach supported by Bunch depicts the analyzed system as a Module Dependency Graph (MDG) whose nodes and edges respectively represent the system's modules and the dependencies between these modules. This approach defines the clustering as a problem consisting in finding the right partition of the graph into cohesive clusters that are loosely coupled. The objective of this optimization problem is to maximize the value of the fitness function MQ (Modularization Quality) computed from the dependencies between modules. To solve this problem, Bunch relies on an input algorithm (e.g., hill climbing, genetic algorithm (Mitchell et al., 2001)). In addition, Bunch determines the omnipresent[7] modules that it isolates from the partitioning in order not to obscure its results.

[7] An omnipresent module is a module that uses or is used by a high number of other modules in the system (Mancoridis et al., 1999).

Subsequent works, such as (Mitchell and Mancoridis, 2002a; Mitchell et al., 2006; Mitchell et al., 2008) have improved Bunch by making it more flexible in the choice of the clustering algorithm, in the computation of the partitions and by refining the formulation of MQ. Given a graph comprising *k* clusters, the most recent expression of MQ is described by the equation (1.1) below (Mitchell et al., 2008):

$$MQ = \sum_{i=1}^{k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \dfrac{2\mu_i}{2\mu_i + \sum_{\substack{j=1 \\ j \neq i}}^{k} (\varepsilon_{i,j} + \varepsilon_{j,i})} & \text{otherwise} \end{cases} \tag{1.1}$$

In this equation CF_i is the Cluster Factor for the cluster *i* and μ_i is the number of internal edges within cluster i. ε_{ij} and ε_{ji} are respectively the number of edges between 2 distinct clusters *i* and *j*.

Anquetil and Lethbridge (Anquetil and Lethbridge, 1999) used file name similarity as a clustering criterion. To this end, they introduced various techniques to retrieve a list of concepts from each file name. Using a lattice, they then cluster together files whose names contain similar concepts.

More recently, Kuhn et al. (Kuhn et al., 2008) introduced a semantic clustering approach that relies on LSI (Latent Semantic Indexing) (Deerwester et al., 1990) to cluster software artifacts based on their vocabulary similarity. They interpreted the resulting clusters as linguistic topics. They classified these topics into five categories (e.g., well-encapsulated topics or cross-cutting topics). Also, their evaluation showed that the semantic clusters do not always match clusters derived using structural information.

Saiedi et al. (Saeidi et al., 2015) cluster modules using two objective functions. The first one is the structural-based function MQ introduced in (Mitchell et al., 2008) whereas the second one is a lexical-based function CQ (Conceptual Quality). Derived from MQ and computed using the lexical information provided by LSI (Deerwester et al., 1990), CQ aims at grouping together conceptually similar modules.

Garcia et al. (Garcia et al., 2011) introduce an approach that relies on concerns to reconstruct the software architecture. To this end, they first input source code's comments and identifiers to LDA[8] (Blei et al., 2003) in order to recover software concerns (e.g. system's role, responsibility) considered as topics. They then use a hierarchical clustering technique that recovers software components and connectors using a similarity measure computed from both the recovered concerns and the structural information. This approach is further referred to as ARC in the literature (e.g. (Garcia et al., 2013))

A recent study by Le et al. (Le et al., 2015) proposes ARCADE, a workbench that implements a set of existing architecture recovery techniques as well as a set of layers' dependency attributes that assess the architectural changes in software systems at both the system and component levels. They concluded that, even though some techniques (i.e. ARC (Garcia et al., 2011) and ACDC (Tzerpos and Holt, 2000)) outperform others, the overall accuracy of software architecture recovery techniques is relatively low.

1.2.2 Approaches targeting the layered style

As pointed out by Ducasse and Pollet (Ducasse and Pollet, 2009), is it a challenge to identify architectural patterns or styles since they span many architectural elements and their implementation can be done in diverse ways. Some approaches have attempted to address this challenge (e.g., (Ding and Medvidovic, 2001; Medvidovic and Jakobac, 2006; Yan et al., 2004; Tzerpos and Holt, 2001; Kazman and Carriere, 1999; Sartipi, 2003; Cai et al., 2013; Laval et al., 2013; Scanniello et al., 2010a; Hassan and Holt, 2002; Sora et al., 2012; Andreoupoulos et al., 2005; Sangal et al., 2005a; Constantinou et al., 2011; Sarkar et al., 2009)).

[8] Latent Dirichlet Allocation.

For instance, ACDC (Tzerpos and Holt, 2000) relies on familiar patterns observed in large systems to find clusters while keeping the size of the clusters at a manageable level. Proposed subsystem patterns include the source file pattern, support library pattern and the subgraph dominator pattern. For example, the source file pattern is a basic pattern that creates a cluster grouping procedures and variables contained in the same source file.

In this thesis, we are particularly interested in approaches proposed to recover or analyze layered architectures and that are the most relevant to our work. Most of these approaches rely on some heuristics in the process of recovering layers and do not explicitly consider the rules and constraints defined by the layered style.

In particular, Scanniello et al. (Scanniello et al., 2010b) describe an approach to partition hierarchical OO systems. First, they exploit their previous work (Scanniello et al., 2010a) to recover the layered architecture of a given system using a link analysis algorithm. The uppermost layer comprises the classes that rely on many other classes. The lowermost layer is made of the classes that are used by many other classes, while the middle layer comprises the remaining classes. They then use the K-means algorithm to cluster each layer's entities into modules. As a clustering criterion, they rely on the lexical similarity computed between each layer's entities using VSM (Vector Space Model) (Manning et al., 2008).

Schmidt et al. (Schmidt et al., 2011) introduce a framework that supports the reconstruction of software architectures. For this purpose, they first rely on a greedy[9] algorithm to generate a start solution comprising clusters based on low coupling and high cohesion criteria. To assign these clusters to layers, they use a hill-climbing algorithm which exploits the clusters' ratio of fan-in and fan-out. This hill-climbing algorithm is similar to the HC algorithm presented in (Mitchell et al., 2008).

---

[9] A greedy algorithm is a technique which consists of building incrementally the solution to a problem by selecting at each step a local optimum in order to attempt to find the global optimum.

Müller et al. (Müller et al., 1993; Kienle and Müller, 2010) propose a tool named Rigi which implements a set of heuristics allowing the user to analyze, explore, summarize and document software systems. The artifacts exploited during the analysis of these systems are stored in a repository and abstracted using a query mechanism. To support the documentation of layered subsystem hierarchies, the maintainer can interact with the Rigi tool to identify aggregates of similar low-level entities based on clustering criteria. These aggregates are progressively refined till the analyzed system is represented at the desired abstraction levels. These groupings can also be generated using scripting languages.

Laguë et al (Laguë et al. 1998) propose a framework for analyzing layered style compliant-systems' architectures so as to assess the coherence between the description of the architecture given in design documents and the structure of the source code. The framework relies on a set of questions for evaluating the properties of a layered system and a set of layers' dependency attributes that help answering these questions. This empirical study has shown that strict layering is not enforced in layered systems as skip-calls are made extensively whereas there are no back-calls. However, this framework does not support the recovery of layered architectures.

One of the most common features in large software systems is the presence of cyclic dependencies (Falleri et al., 2011; Ducasse and Pollet, 2009). For instance, in ArgoUML, the largest cycle comprises almost half of system's packages. In jEdit, the largest cycle contains almost two-thirds of the packages (Falleri et al., 2011). Since the presence of cyclic dependencies might hinder the recovery process – particularly for software architectures compliant to the layered style–, some architecture recovery approaches rely on heuristics to tackle the cyclic dependencies problem. This notably leads to the detection of strongly connected components to resolve cyclic dependencies (e.g. (Sarkar et al., 2009; Sangal et al., 2005a)) prior to the layering.

In particular, Sarkar et al. (Sarkar et al., 2009) recover layered architectures from a direct acyclic graph representing the system at hand. First, they assign ranks to the system's

modules based on their respective levels in the directed acyclic graph obtained through the detection of strongly connected components in the analyzed system's artifacts. They then compute the domain concepts frequency distribution in the modules. The so-obtained ranks and frequencies are fed to the K-means clustering algorithm so as to assign modules to layers. Sarkar et al. also present three layering rules, namely: the skip-call, back-call and cyclic dependency rules. These rules are then used to formulate violations bearing the same name. From the defined rules, the authors derive a set of measures for evaluating the compliance with the layered style. Noteworthy, in this approach, the determination of structural information rely on the existence of cyclic dependencies in the system being analyzed. In addition, the approach proposed by the authors is limited to systems developed in C or C ++.

Sangal et al. (Sangal et al., 2005a) developed a tool called Lattix which identifies software layers and cycles. First, the tool parses the source code of the analyzed system to generate a Dependency Structure Matrix (DSM) showing the depencencies between modules in a tabular form. It then uses optimization algorithms to rearrange the elements of this matrix in an order that reflects a given architecture. Thus, if most of the dependencies of the matrix elements end up in the lower diagonal, this indicates that the analyzed system is layered. In this approach, a cyclic dependency can be eliminated by the use of partitioning algorithms which create subsystems from the modules connected through a cyclic dependency. To create layered architectures, Lattix puts each cycle in a different layer. Packages which are not connected through cyclic dependencies are respectively put in lower or higher layers depending on whether they use or are used by the other packages. Lattix also supports the specification of design rules reflecting the architectural constructions allowed in the analyzed system. The application of these rules enables the detection of architectural violations and ensures that during the evolution of a system, the conceptual architecture remains consistent with its concrete architecture. Lattix supports many languages such as Java, Ada and C. However, it involves resolving the cyclic dependencies.

To resolve the cyclic dependency problem, other approaches focus on the removal of undesired dependencies causing cycles (Hautus, 2005; Laval et al., 2013). For instance, Hautus (Hautus, 2002) proposes a tool named PASTA which recovers software layers using heuristics that enable building an acyclic graph from the analyzed software by removing undesired dependencies. The resulting layering is then evaluated to estimate the percentage of the software that should be refactored in order to eliminate the cycles in the package structure.

Along the same line, is the work of Laval et al. (Laval et al., 2013) that introduces an approach, called oZone, which handles undesired cyclic dependencies and decomposes a system into layers. In their work, they make a distinction between two layering rules respectively introduced by Szyperski (Szyperski, 1998) and Bachmann (Bachmann et al., 2000): the *closed layering* rule and the *open layering* rule. The *closed layering* rule states that a layer should only use the layer directly below. This rule is also referred as *strict layering* (see Section 1.1.2.2). The *open layering* rule stipulates in turn that a layer can use any of its lower layers. In their work, the authors attempt to recover a layered architecture that complies with the *open layering* rule by breaking undesired cyclic dependencies. To this end, they rely on two algorithms. The first one takes as input a graph representing a system's package dependencies and it uses two heuristics to remove graph's cyclic dependencies hampering the layering recovery. The first heuristic eliminates direct cycles (cycles between two components) by removing the lightest dependency in the cycle. This dependency might be a design defect or a program error. The second heuristic eliminates the shared dependencies within indirect cycles (cycles involving at least three packages). The second algorithm generates a layered architecture from the acyclic graph produced by the first algorithm. To this extent, the second algorithm traverses the acyclic graph from the bottom to the top to assign its nodes to a set of layers so that each layer only uses the layers below. The user can then give her feedback on the so-obtained architecture through a browser where she can validate or disapprove the corresponding dependencies. The implementation of Ozone is based on the FAMIX language independent meta-model. This approach therefore enables

reconstructing the layered architecture of a system independently of the language used to develop it.

1.3 Synthesis

We have adopted the taxonomy of Ducasse and Pollet (Ducasse and Pollet, 2009) to summarize the related works. In this context, the goals generally targeted by an architectural reconstruction approach are: the re-documentation and understanding (Scanniello et al. 2010a, Scanniello et al., 2010b; Sarkar et al., 2009) as well as the analysis (e.g., (Laguë et al., 1998; Scanniello et al., 2010a; Scanniello et al., 2010b; Laval et al., 2013; Sarkar et al., 2009; Sangal et al., 2005a; Hautus, 2002)). Added to these are the evolution and maintenance (e.g., (Scanniello et al., 2010a, Scanniello et al., 2010b; Laval et al., 2013; Sarkar et al., 2009; Sangal et al., 2005a; Schmidt et al., 2011), and the conformance checking between the reconstructed architecture and the conceptual one (e.g., (Laguë et al., 1998; Sarkar et al., 2009; Sangal et al., 2005a)).

Besides, an architectural reconstruction approach can take as input various forms of non-architectural data. The latter can be structural information (e.g., (Sarkar et al., 2009; Scanniello et al., 2010a; Scanniello et al., 2010b; Sangal et al., 2005a)) or textual information (e.g., (Sarkar et al., 2009; Scanniello et al., 2010b; Saeidi et al., 2015; Anquetil and Lethbridge, 1999; Kuhn et al., 2008)) extracted from the analyzed system. Noteworthy, the textual information can be further processed into lexical information by applying information retrieval techniques (e.g., LDA (Blei et al., 2003), LSI (Deerwester et al., 1990), Vector Space Model (Manning et al., 2008)) on the identifiers and comments comprised in the source code. An architectural reconstruction approach can also take as input information such as human expertise (e.g., (Sarkar et al., 2009; Scanniello et al., 2010a; Scanniello et al., 2010b)), or even the physical organization of the data (e.g., (Laval et al., 2013; Hautus, 2002)). In addition, architectural input such as models (e.g., (Laval et al., 2013)) can also serve as input to an architectural reconstruction approach.

The existing architecture reconstruction approaches also implicitly or explicitly take into account various rules (also referred as principles) and violations during the reconstruction activity. The rules of the layered style considered by these approaches include: the closed layering rule (e.g., (Sarkar et al., 2009; Sangal et al., 2005a)), the open layering rule (e.g., (Laval et al., 2013; Scanniello et al., 2010a; Scanniello et al., 2010b)), the skip-call rule (e.g., (Sarkar et al., 2009)), the back-call rule (Sarkar et al., 2009), and the cyclic dependency rule (e.g., (Sarkar et al., 2009; Laval et al., 2013)). The latter is referred as the Acyclic Dependency Principle in (Hautus, 2002). The violations of the layered style considered by the studied approaches include: the skip-call (Sarkar et al., 2009; Sangal et al., 2005a), the back-call (e.g., (Sarkar et al., 2009; Laval et al., 2013; Sangal et al., 2005a; Schmidt et al., 2012)), and the cyclic dependency violations (e.g., (Sarkar et al., 2009; Laval et al., 2013; Sangal et al., 2005a)).

In addition, the studied architectural reconstruction approaches are performed according to different types of processes: bottom-up (e.g., (Scanniello et al., 2010a; Scanniello et al., 2010b; Laval et al., 2013; Hautus, 2002)), top-down (e.g., (Lägue et al., 1998)) and hybrid processes (e.g., (Sarkar et al., 2009)).

The architecture recovery and analysis approaches discussed in this chapter are either semi-automatic (e.g., (Laval et al., 2013; Sarkar et al., 2009; Scanniello et al., 2010a; Scanniello et al., 2010b; Sangal et al., 2005a)) or quasi-automatic (e.g., (Lägue et al., 98)). The semi-automatic approaches are either based on abstraction techniques such as queries (e.g, (Müller et al., 1993)) or on graph-theory techniques (e.g., (Sarkar et al., 2009; Scanniello et al., 2010a; Scanniello et al., 2010b)). Quasi-automatic approaches usually rely on clustering (e.g., (Mitchell et al., 2008; Schmidt et al., 2011)).

Furthermore, the outputs from the studied architectural reconstruction approaches are various. They include the analyzed system's architecture (e.g., (Scanniello et al., 2010a, Scanniello et al., 2010b; Sarkar et al., 2009; Laval et al., 2013; Sangal et al., 2005a)), its visualization (e.g., (Hautus, 2002; Kienle and Muller, 2010)) as well as layering measures

used to assess the quality of the recovered architecture (e.g., (Sarkar et al., 2009; Lägue et al., 1998)).

1.3.1 Limitations of the software architecture recovery approaches

Some limitations emerge from the architectural reconstruction approaches presented in this chapter. First, most of these approaches (e.g., (Mancoridis et al., 1998; Mancoridis et al., 1999; Hautus, 2002; Müller et al., 1993; Scanniello et al., 2010a; Sangal et al., 2005a)) are language and platform dependent and do not use a standard representation of the data of the system under analysis (El Boussaidi et al., 2012). Therefore, the resulting tools are not able to interoperate with each other (Ulrich and Newcomb, 2010). Besides, most of these architectural reconstruction approaches are carried out in an *ad hoc* manner and therefore lack formalization. This lack of formalization hinders the repeatability of the techniques used by these approaches (Pérez-Castillo et al., 2011).

As pointed out by Ducasse and Pollet, (Ducasse and Pollet, 2009) various sources of information can be used to initiate a software architecture reconstruction activity. However, most of the software architectures reconstruction approaches solely rely on the structural information derived from the source code, disregarding other sources of information such as the lexical information embedded within the source code (e.g., (Laval et al., 2013; Hautus, 2005; Mancoridis et al., 1998)). Yet, the lexical information is a valuable source of information that enriches the software analysis (Kuhn et al., 2007, Poshyvanyk et al., 2009; Bavota et al., 2013). When developing, a software engineer usually embeds her domain knowledge of the system through the lexical information disseminated all over the source code (Kuhn et al., 2007; Maskeri et al. 2008; Poshyvanyk et al., 2009; Bavota et al., 2013). Furthermore, the lack of correlation between structural and lexical measures allows capturing different aspects of coupling (Poshyvanyk et al., 2009). Structural and lexical information are therefore complementary. Hence, also relying on lexical information can increase the accuracy of the information used when recovering software architectures.

Software systems are practically built by combining and composing architectural styles. However some software architecture reconstruction approaches are not (explicitly) guided by the rules of the architectural styles that have helped build the architectures of the systems they analyze (e.g., (Saeidi et al., 2015) and (Kuhn et al., 2008)). This might lead to the recovery of architectures whose components and connectors do neither embody the architectural meaning nor the constraints related to the architectural style(s) initially used to build the system at hand (Cai et al., 2013). The so-recovered architectures might therefore be too permissive with architectural violations.

Once the recovery is performed, a human expertise, provided by designers and developers having an in-depth knowledge of the system under analysis is often necessary to validate and refine the recovered architecture. However, some tools supporting the architectural reconstruction requires a human assistance to validate not only the architecture they recover but also all the artifacts produced at the different steps leading to the generation of this architecture. For instance, in (Kienle and Müller, 2010), the creation of the layered hierarchy is an iterative, recursive and semi-automatic process which is interactively assisted by the reverse engineer. Such an intensive involvement of the human during the recovery process delays the exploitation of the recovered architecture.

According to Garcia et al. (Garcia et al., 2013), another important concern related to the architecture reconstruction approaches is their reliance on techniques leading to results that relatively lack precision and completeness. This is notably because some of the existing recovery approaches did not focus on the analysis of the quality of the recovered architectures and hence, did not evaluate these criteria (e.g., Sangal et al. 2005a; Kienle et al. 2010). This concern is further compounded by the lack of ground-truth architectures – or of experts able to produce these architectures – needed to evaluate these criteria (Garcia et al., 2013; Saeidi et al., 2015). There is therefore a need for techniques that generate more precise and complete architectures.

1.3.2 Limitations of software architecture recovery approaches targeting the layered architectures

Some software architecture recovery approaches were dedicated to the reconstruction of layered architectures of software systems developed according to the object oriented paradigm. Some of these approaches (e.g., (Laval et al., 2013; Scanniello et al., 2010a)) assume 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, when a module represents a common subtask exclusive to components of a middle-level layer, this module 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.

Furthermore, existing architecture recovery approaches usually recover the layered architecture using some heuristics to handle cyclic dependencies (e.g., (Sarkar et al., 2009; Laval et al., 2013)) or to layer modules according to the number of their fan-in and fan-out dependencies (e.g., (Laval et al., 2013; Scanniello et al., 2010a)). Although some of these heuristics are derived from the layering rules, they may result in architectures that are too permissive with layering violations. To give a better insight to this limitation, we will use as an example the software system depicted by Figure 1.6(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.

Figure 1.6(b) shows the expected layered architecture of our example system. Using a clustering algorithm that relies on modularity (e.g., (Schmidt et al., 2011)), the recovered architecture of the system might be the one depicted in Figure 1.6(c). The clustering (e.g., (Schmidt et al., 2011)), 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., (Sarkar et al., 2009; Sangal et al., 2005a)). In this case (i.e. Figure 1.6(c)), the number of adjacent dependencies is maximized but the number of

skip-calls can also be quite high. These approaches might therefore result in layered architectures with very few layers. 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 (Laval et al., 2013), the recovered architecture of our example system is depicted in Figure 1.6(d): it possesses too many layers and may be too permissive with violations such as skip-calls and back-calls.

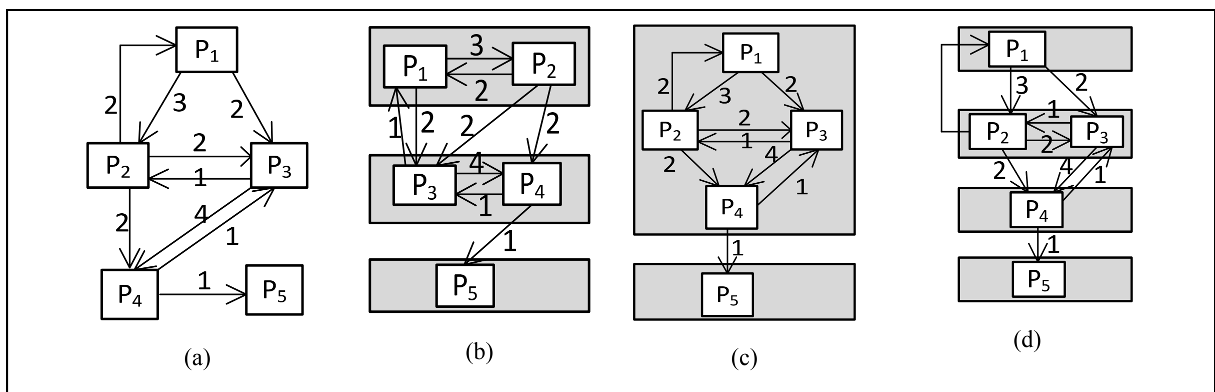


Figure 1.6 An example of a system, its architecture and the layering obtained applying different existing approaches

By generating either very few layers or too many layers, these approaches may negatively impact the quality attributes promoted by the layered style (Buschmann et al., 1996). Furthermore, an architecture recovery approach targeting the layered style should be able to build an architecture that reflects the compromises made by the designer; a compromise between the number of layers, the number of adjacent dependencies and the number of other types of dependencies (i.e., the violations permitted for certain reasons such as performance or portability).

In the example system of Figure 1.6, 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 (Figure 1.6(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 Figure 1.6(b)), the layering recovery approach

should be based on the rules of the layering style and on how strictly the architect applied them when designing her system.

CHAPTER 2

STRUCTURAL-BASED LAYERING APPROACH

In this chapter, we propose a search-based approach that will use structural information to recover the layered architecture, at the package level, of an OO system (phase 2 of the research methodology). This layering recovery approach is based on the rules of the layering style and on how strictly the architect applied them when designing her system. For this purpose, the proposed approach will rely on: 1) a set of constraints that embody the essence of the layered architectures and 2) an input which reveals how strictly the layered rules were enforced when designing a given system. Besides, to make our approach language and platform independent, we represent the structural data of the systems under study using the KDM (Knowledge Discovery Metamodel) standard.

The organization of this chapter is the following: Section 2.1 explains how we analyzed the layered style to extract a set of rules embodying the layered style. Section 2.2 provides an overview of our layering recovery approach based on the extracted rules. Section 2.3 explains how we extract the structural information used as input by our approach. Section 2.4 describes the translation of the layers recovery problem into an optimization problem that we solve using a search-based algorithm. Section 2.5 refines this translation as a quadratic assignment problem (QAP) that we will solve using another search-based algorithm. Finally, Section 2.6 presents a summary of this chapter.

2.1 Extracting layering rules

Our analysis of various definitions and descriptions of the layered style led us to the identification on four main dimensions of reasoning when it comes to applying this style: the **abstraction**, the **responsibility**, the **transversality** and the **protection against variations**. This analysis also enabled to derive the following layering rules that are respectively encompassed by these four dimensions:

2.1.1 The abstraction rules

Applying the layered style consists in 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 abstraction dimension encompasses two fundamental rules that should guide the design as well as the recovery of layered architectures:

- **The Layer Abstraction Uniformity rule**: *it 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 (Buschmann et al., 1996), 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 rule led to many algorithms that build layered architectures based on a depth-traversal of dependency graphs built from the studied system (e.g., (Sarkar et al., 2009; Laval et al., 2013; Scanniello et al., 2010a; El Boussaidi, 2012)).
- **The Incremental Layer Dependency rule**: *it 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)* (Buschmann et al., 1996). This rule 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 (Avgeriou and Zdun, 2005) and not recommended by others (Clements et al., 2003; Bourquin and Keller, 2007). Our analysis of the various descriptions of the layered style and several open source systems 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 Layer Dependency rule should thus be stated in a way that allows the intra-dependencies and the skip-calls and—to some extent—back-call violations. We therefore rephrase this rule as follows: "*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. We derived this rule from our analysis of various descriptions of the layered style (e.g., (Buschmann et al., 1996; Clements et al., 2003 et al., 2003; Szyperski, 1998; Eeles, 2002)) and several open source systems. Note that this rule is also consistent with the empirical study carried out in (Laguë et al., 1998) which has shown that strict layering is not enforced in layered systems.

The compliance with the first rule therefore implies that the packages of the same layer should be at the same distance from the "physical" components of the system. However, the presence of back-call and skip-call dependencies causes a discrepancy between the packages' distances, even when they belong to the same layer. Thus, compliance with the first rule derives largely from compliance with the second rule (i.e., Incremental Layer Dependency).

2.1.2 The responsibility rule

Applying the layered style also implies decomposing the system into a set of cohesive components (i.e., responsibilities) and properly assigning these components to a set of abstraction levels. To this extent, the **Responsibility rule** *states that each layer of the system must be assigned a given responsibility (Eeles, 2002) so that the topmost layer corresponds to the overall function of the system as perceived by the final user and the responsibilities of the lower layers contribute to those of the higher layers* (Buschmann et al., 1996). The concept of responsibility is defined in (Bass et al., 2003) as "the functionality, data, or information that a software element provides". Thus, the logic of a software system is divided into several responsibilities. Each responsibility is implemented by a set of interacting components that need to be cohesive and specific to a given domain (Clements et al., 2003). In this context, each component of the system should be designed to implement a specific service and must belong to a single layer. Therefore, each component of the system contributes to the realization of the rule of responsibility.

This rule is related to the notion of modularity that has already been subject to many studies (e.g, Lung et al., 2004; Zhang et al., 2010; Mitchell et al., 2008; El Boussaidi et al., 2012). Most of work on architecture recovery using clustering techniques focused on decomposing systems into components while minimizing the coupling between resulting components and maximizing the cohesion of each component (e.g., (Zhang et al., 2010; Lung et al., 2004; Mitchell et al., 2008; El Boussaidi et al., 2012)).

2.1.3 The transversality rule

By convention, the software layers are generally placed horizontally to the physical medium. With such an alignment of layers, components that are used by many other components are generally libraries which are assigned to lower layers. Such an assignment goes against the abstraction dimension and particularly against the **Incremental Layer Dependency rule** since it can induce a huge amount of skip-calls. Therefore, putting the most used components in a vertical layer resolves that issue by turning the skip-calls generated by these omnipresent components into adjacent dependencies directed from the horizontal layers' components to the vertical one. Accordingly, the **Transversality rule** *stipulates that components that are intensively used by other components should be put in a transversal layer instead of an horizontal one*. The layer resulting from the enforcement of this rule should therefore be aligned perpendicularly against the other layers similarly to the sidecar (Clements et al., 2003).

Of particular note, exploiting the transversality rule when recovering the architecture eases the layers reconstruction by creating a subsystem (i.e layer) which is excluded from the recovery process and which contains components that are generally referred to as omnipresent components (Müller et al., 1993). This exclusion of omnipresent components from the reconstruction process is a practice advocated by many approaches such as (Müller et al., 1993; Mancoridis et al., 1998). The main argument against the omnipresent components is that they tend to obscure the recovered architecture (Müller et al., 1993; Mancoridis et al., 1998).

2.1.4 The protection against variations rules

The protection against variations dimension relates to the following observation: each layer must be designed so that the variations affecting its inner entities have no adverse impact on other layers. This dimension is strongly inspired by the notion of protection against variations which is further discussed in (Larman, 2005) and whose essence is equivalent to the open-closed rule (Meyer, 1988). The protection against variations dimension encompasses the two following rules:

- **The interfacing rule**: *which states that a layer should have an input interface to communicate with its upper layer*. Similarly to (Larman, 2005), we consider that the term interface does not refer to a given API but rather to the medium that allows accessing a given layer. Thus, we consider that the input interface of a given layer *i* is made of the entities – classes or packages depending on the granularity on which we reason – through which its services are called by the upper layer. An interface could therefore be seen as a fence whose gate allows centralizing the accesses to the inner implementation of a layer. According to the interfacing rule, a layer should hide as much as possible its internals through the use of an interface. The latter will allow the limitation of the access points to a given layer's services (Bachmann et al., 2007). Therefore, the more this rule is enforced, the more the coupling between the layers is reduced. Thus, the impact of modifying a given layer is reduced in other layers and as a result, the maintainability and the portability of the system are increased and the analysis of the system is simplified (Büchi and Weck, 1999). The use of interfaces to hide layers's implementation details is a practice advocated by several authors (e.g., (Zimmermann, 1980; Buschmann et al., 1996; Bachmann et al., 2007; Clements et al., 2003)). Some design patterns can be used to implement the interfaces of layers. Among them is the Façade pattern (Gamma et al., 1994).
- **The stability rule**: *it stipulates that the interfaces of each layer should remain stable from a version of a system to another*. This means that if the implementation of any layer was to change, that layer should continue to provide the same services to its

clients (Buschmann and Henney, 2003; Bachmann et al., 2007; Avgeriou and Zdun, 2005)). According to this rule, if the implementation of a given layer is changed, a client of this layer should be able to keep using it (Garlan and Shaw, 1996) without having to adjust its own interfaces to cope with the new layer's implementation. The more this rule is enforced, the less the modifications of a given layer affect its users. Therefore, this rule not only allows increasing the reuse of a system (Garlan and Shaw, 1996) but also reduces the maintenance cost and effort (Alshayeb et al., 2011). The stability of a layer can be supported through the establishment of an input interface. This stability may be further enhanced by the use of an output interface that would prevent the changes in this layer to ripple on its lower layer. The use of standard interfaces is highly recommended in layered systems (Zimmermann, 1980; Clements et al., 2003; Garlan et al. 1996) and can also reinforce the stability of layers' interfaces.

The protection against variations rules are closely related to the responsibility rule. These rules highlight the way the services offered by each layer should be hidden to the user while remaining stable in case of internal changes.

2.2 Overview of the proposed approach

In the following, we present an approach to recover architectural layers using the layering rules described above. To recover the layered architectures of software systems, our approach uses constraints derived from the layering rules. This approach follows a three-step process as depicted by Figure 2.1:

1. Step 1: create a representation of the analyzed system using the KDM standard and retrieve the necessary facts from that representation. Because, we focus first on the abstraction rules and specifically on the Incremental Layer Dependency rule, we only use structural facts. Recall that the enforcement of the Layer Abstraction Uniformity rule derives from the enforcement of the Incremental Layer Dependency rule (see Section 2.1.1).

2. Step 2: derive different layers' dependency attributes and constraints from the layering rules and use these contraints to formalize the layering recovery as an optimization problem.
3. Step 3: solve the layering optimization problem to recover the architectural layers according to an input reflecting how strictly the layering rules were enforced when designing/evolving the system.

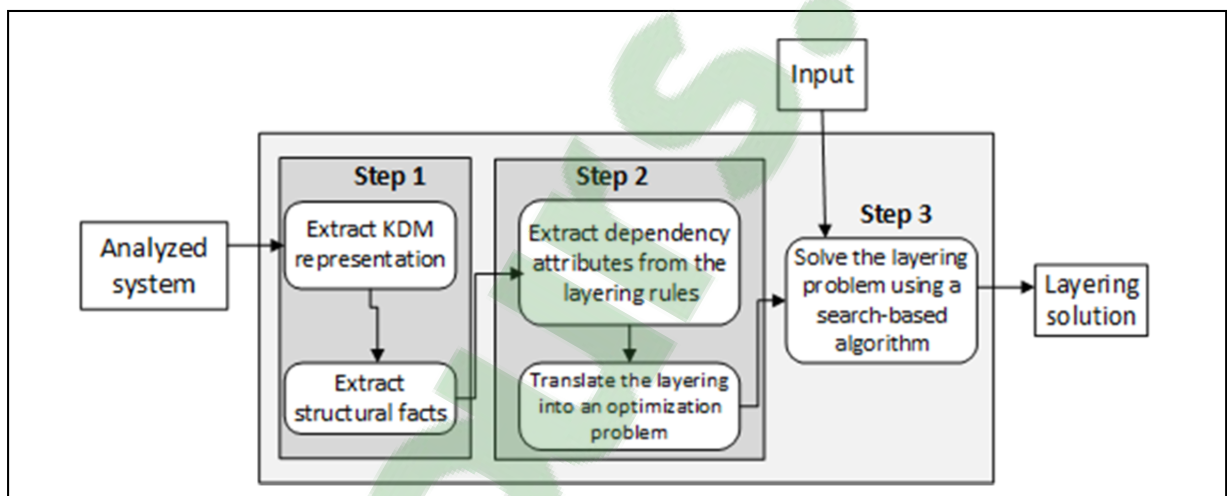


Figure 2.1 Overview of the approach

In the next sections, we further describe the three steps of our approach. In particular, we describe our first attempt to translate the layering recovery into an optimization problem in section 2.4, and a refinement of this problem into a quadratic assignment problem in Section 2.5. Note that, unlike some existing approaches, we do not rely on any heuristics to resolve cyclic dependencies problems.

2.3 Facts extraction

In the first step of our approach, we extract a system's structural concepts and relationships that comply with the KDM standard. These structural facts may vary from a system to another. In our context, we focus on object-oriented systems. Thus, we will retrieve KDM concepts representing entities such as classes, interfaces and packages; and KDM

relationships representing for instance the inheritance, implementation or methods invocations. Moreover, depending on the targeted view, we have to choose a coherent subset of concepts and relationships to be included in the view. In our context, we want to recover the layered view which is a static view of the analyzed system. Therefore, we extract the structural facts required to build the layered view of an object-oriented system. To do so, we first analyze the source code of the analyzed system and generate platform independent models that are compliant with the Knowledge Discovery Metamodel (KDM). This makes our approach independent from the nature of the languages and platforms used to develop the systems under study. We then parse the KDM models that we generated from the source code, to retrieve entities and their relationships. The retrieved entities include packages, classes, interfaces and enumerated types. These entities are respectively represented by the KDM concepts Package, ClassUnit, InterfaceUnit and EnumeratedType. The retrieved relationships (i.e., dependencies between two entities) include:

- Inheritance: this is the case when a class A (respectively an interface A) inherits from a class B (respectively an interface B). An inheritance relationship corresponds to an Extends relationship in a KDM representation.
- Implementation: this is the case when a class A implements the methods of an interface B. An implementation relationship corresponds to an Implements relationship in a KDM representation.
- Import: this is the case when a class A imports the code of another class in order to access its methods and/or attributes. An import relationship corresponds to an Imports relationship in a KDM representation.
- Method invocation: this is the case when a class/interface A invokes a method of a class/interface B. Depending on the purpose of an invocation (e.g., class instance creation, method invocation, super method invocation, super constructor invocation), the latter can for instance correspond to a Calls or to a Creates relationship in a KDM representation.
- Class reference: this is the case when a class A refers to a class B by defining an attribute of type B, specifying B as a parameter of its methods, or defining inside one

of its methods a variable of type B. These references correspond to the UsesType relationship in a KDM representation.

Besides, we choose to work at the package level. We rely on existing decomposition of object oriented systems into packages. Thus, we assume that these packages have been designed according to the responsibility rule. Accordingly, we extract the packages and their dependencies. The dependency from a package P_1 to a package P_2 is derived from the relationships, i.e., dependencies between their entities. The weight of that package dependency is obtained by summing the weights of the dependencies directed from the entities of package P_1 to those of package P_2. We consider that there is a dependency of weight 1 directed from entity A to entity B if there is a relationship from an entity A to an entity B.

Once we have extracted the structural facts, we represent them as a graph where the nodes and edges respectively denote the extracted packages and the relationships between them. This graph that is called an MDG (Module Dependency Graph) allows representing a system's facts as a generic and language-independent structure (Mitchell, 2002b). Given two packages P_1 and P_2, the weight of the MDG's edge linking the two nodes representing P_1 and P_2 is the number of dependencies between the packages P_1 and P_2.

2.4 The layered architecture recovery as an optimization problem

In the following, we translate the layered architecture recovery problem into an optimization problem that we solve by adapting existing heuristic search methods.

2.4.1 Translating the layered architecture recovery into an optimization problem

In this section, we define a set of layers' dependency attributes and constraints based on the Incremental Layer Dependency rule presented above. We then use these layers' dependency attributes and constraints to translate the layering rules as an optimization problem.

2.4.1.1 Extracting layers' dependency attributes from the Incremental Layer Dependency rule

Given two layers of a layered architecture respectively referred as layer *i* and layer *j*, we compute the dependency going from layer *i* to layer *j* as the sum of the weights of the dependencies directed from each package of layer *i* to each package of layer *j*. To formalize the Incremental Layer Dependency rule using constraints, we introduce four layers' dependency attributes related to the dependencies between layers. Assuming that the layers are numbered in a decreasing order from the highest to the lowest layers, the proposed layers' dependency attributes[10] are the following:

- AdjacencyUse(i,j) when j = i-1. AdjacencyUse(i,j) denotes the number of dependencies directed from layer *i* to its adjacent lower layer *j*.
- SkipUse(i,j) when j < i-1. SkipUse(i,j) is the number of skip-call dependencies directed from layer *i* to layer *j*.
- BackUse(i,j) when i < j. BackUse(i,j) is the number of back-call dependencies directed from layer *i* to layer *j*.
- IntraUse(i) when i = j. IntraUse(i) is the number of the dependencies inside layer *i*.

Figure 2.2 illustrates the calculation of the layer dependencies for a system comprising three layers where all dependencies have the same weight (i.e., a weight of 1).

---

[10] The measurement unit of these layers' dependency attributes is the dependency.

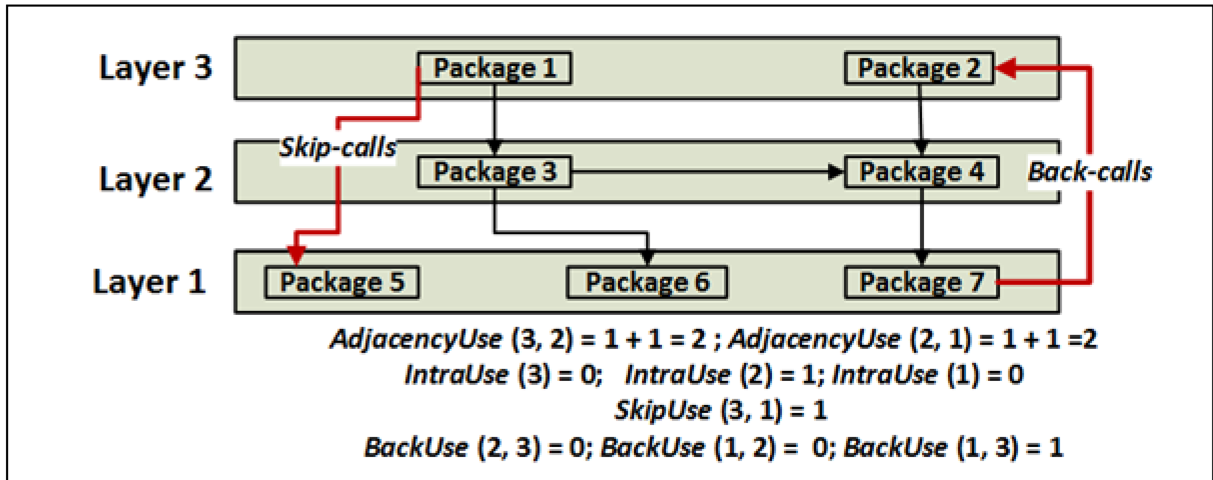


Figure 2.2 Example of the calculation of the 4 types of layer dependencies

2.4.1.2 Extracting constraints from the layered style

To comply with the Incremental Layer Dependency rule, we want to minimize the number of skip-call and back-call dependencies. This means that, apart from the upper layer adjacent to layer j, we must minimize the dependencies relating other layers to layer j. However, the skip-call dependencies are more tolerated (i.e., skip-calls are often used for performance reasons (Buschmann et al., 1996)) than the back-calls which lead to a poorly structured system. These restrictions are formalized by the following constraint:

$$\text{For each } i, j, k \mid j<i \text{ and } k<j-1, \ BackUse(j,i) \leq SkipUse(j,k) \leq AdjacencyUse(j, j-1) \quad (2.1)$$

Constraint (2.1) may be certainly satisfied when the number of the layers of the system is very small (i.e., when layers are merged). But this will introduce a large number of intra-dependencies. Noteworthy, there is no clear consensus among researchers on the use of intra-dependencies. While these dependencies are accepted by some (Avgeriou and Zdun, 2005), they are not recommended by others (Clements et al., 2003; Bourquin and Keller, 2005). In particular, Bourquin and Keller (Bourquin and Keller, 2005) argue that dependencies between packages of the same layer are not recommended unless otherwise stated while

Clements et al (Clements et al., 2003) indicate that intra dependencies should only be tolerated when some concerns as portability need to be addressed.

When analyzing various descriptions of the layered style and several open source systems, we came to the conclusion that the acceptance of the intra-dependencies depend on the granularity of the components (e.g., packages) of the layer: the lower the granularity, the higher the number of intra-dependencies. Thus, we propose another constraint that limits the number of intra-dependencies of a layer. This second constraint is formalized as follows:

$$\text{IntraUse(j)} \leq \text{AdjacencyUse(j, j-1)} \quad (2.2)$$

This constraint should be validated through experimentation by analyzing a number of layered software systems.

2.4.1.3 Translating the Layering recovery problem into an Optimization Problem

We aim at proposing an architecture reconstruction process which rewards the adjacency between layers while minimizing the other types of layers' dependencies, i.e., intra-dependencies, skip-calls and back-calls. To this end, we rely on the four layers' dependency attributes described in the previous section to guide the process of assigning the packages of a given system layers. We assign an individual layering quality (ILQ) to each layer *i* of the system. We express ILQ as follows:

$$ILQ(i) = \text{ap} * \text{AdjacencyUse(i, i} - 1) + \text{ip} * \text{IntraUse(i)} + \text{sp} * \sum_{j=i-2}^{1} \text{SkipUse(i, j)} + \text{bp} * \sum_{j=i+1}^{n} \text{BackUse(i, j)} \quad (2.3)$$

In equation (2.3), *n* is the number of layers while *ap*, *ip*, *sp* and *bp* are respectively the factors (i.e percentages) assigned to the adjacent dependencies, the intra-dependencies, the skip-call dependencies and the back-call dependencies. In Figure 2.2, since layer 3 has two adjacent dependencies and one skip-call, to compute the individual layering quality of layer 3, we

respectively assign the factors *ap* and *sp* to these two types of dependencies. Hence, ILQ(Layer 3) = 2 x *ap* + 1 x *sp*.

To comply with the Incremental Layer Dependency rule, we want to minimize the number of skip-calls and back-calls. In practice, back-calls which lead to a poorly structured system are less tolerated than intra-dependencies and skip-calls. Besides, the analysis of the open or relaxed layering (Buschmann et al., 1996; Szyperski, 1998) revealed that, in practice, skip-calls are more often tolerated and used than intra-dependencies. We therefore make the assumption that the values of the factors *ip*, *sp* and *bp* should be constrained as follows: $sp < ip < bp$. This assumption should be validated by carrying out experimentations on a number of software systems that are built according to the layered style, and which are of good quality.

The layering quality LaQ[11] corresponding to the assignment of the packages of a system to a set of *n* layers is then obtained by summing the individual layering quality for each layer *i* of the system.

$$LaQ = \sum_{i=1}^{n} \mathrm{ILQ(i)} \tag{2.4}$$

The lower LaQ is, the more the layered system conforms to the abstraction rules. Hence, attempting to reconstruct a layered architecture while minimizing its LaQ is a problem that can be solved by adapting appropriate optimization algorithms.

[11] In both (2.3) and (2.4) equations, the measurement unit is the dependency.

2.4.2 Solving the layering recovery optimization problem

2.4.2.1 Using metaheuristics to solve optimization problems

Typically, an optimization problem aims at searching an optimal solution that minimizes (or maximizes) an objective function, also called a fitness function, while satisfying a set of constraints on elements of the solution (Blum and Roli, 2003). However, for NP-hard combinatorial problems, exact methods might require an exponential computation time to find the optimal solution, which is very time-consuming and sometimes infeasible in practice. Hence, approximate algorithms are more and more used to solve these problems (Blum and Roli, 2003). These algorithms find good solutions in a significantly reduced amount of time. But they do not guarantee to find the global optimal solution, i.e., they can be stuck in local optimum. Approximate algorithms include local search methods (Blum and Roli, 2003). Local search algorithms start from an initial solution chosen in the search space and iteratively attempt to replace the current solution by a better solution belonging to the current solution's neighborhood. To allow an efficient and effective exploration of the search space so as to find (near) optimal solutions, metaheuristics[12] are frequently used. Among them are the following metaheuristics: genetic algorithms (Holland, 1975), hill climbing (Harman, 2007; Mitchell et al., 2008), tabu search (Glover, 1989) and simulated annealing (Kirkpatrick et al., 1983).

In the context of the layering recovery problem, we have selected the hill-climbing metaheuristic for the following reasons: it is a search algorithm that performs well in the context of clustering large systems. Besides, it has been successfully used in several recovery approaches (e.g., (Mancoridis et al., 1998; Saeidi et al., 2015; Schmidt et al., 2011). The hill climbing works in an iterative way. It starts by an initial partition of the system's modules into a set of clusters; usually a randomly generated partition as in (Mitchell et al., 2008). At

[12] Usually inspired from the nature, metaheuristics are general algorithmic frameworks designed to find solutions to complex optimization problems (Bianchi et al., 2009).

each iteration, elements belonging to the current solution's neighborhood are assessed to improve the current solution according to some criterion. This criterion is based on maximizing or minimizing a fitness function. The algorithm stops when a given condition is met, returning the best solution found over all the iterations. Different variants of the hill climbing exist in the literature. The most common ones are the Next Ascent Hill Climbing (NAHC), the Steepest Ascent Hill Climbing (SAHC) and the Random Restart Hill Climbing (Harman, 2007; Mitchell et al, 2001; Russell and Norvig 2002). In the Next Ascent Hill Climbing, the first current solution's neighbour that improves the fitness function is assigned to the current solution. In the Steepest Ascent Hill Climbing, all the current solution's neighbors are assessed and the one that improves the fitness function is assigned to the current solution. Finally, the Random Restart Hill Climbing iteratively performs a hill climbing, starting each time from a randomly generated initial partition.

A hill climbing algorithm can be stuck in a local optimum. However, repeatedly restarting the hill climb from different initial partitions—as in the Random Restart Hill Climbing might lead to adequate results. Even though the hill climbing suffers from the local optimum issue, it is a simple technique which is not only easy to implement but also very effective. Besides, some studies have shown that the hill climbing outperforms other search-based techniques (e.g., Simulating Annealing and Evolutionary Algorithms) (e.g., (Harman et al., 2002; Mitchell and Mancoridis, 2002a)).

2.4.2.2 Using hill-climbing to solve the layered recovery problem

To reconstruct the layering of software systems, we propose a layering algorithm that applies the SAHC (Steepest Ascent Hill-Climbing) technique (Mitchell et al., 2008). This layering algorithm, called *SAHCLayering*, uses LaQ as a fitness function. A high-level view of the *SAHCLayering* algorithm is depicted by Algorithm 2.1. This algorithm aims at producing an improved 3-layered solution from an input layered partition built from the analyzed system. However, since the analyzed system might have more than 3 layers, the algorithm recursively attempts to divide each of these 3 layers into *m* layers until the LAQ value of the so-obtained

layering can no longer be improved. The layering algorithm takes as input the system to analyze and the value of *m*. The algorithm also takes as input a table comprising the values assigned to the factors *ap*, *ip*, *sp* and *bp*.

Algorithm 2.1 A high level view of the layering algorithm

```
Algorithm SAHCLayering
Input: inputSystem, currentSolution, m, factors
Output: LayeredSolution
  1. if (currentSolution = null){
  2.     currentSolution ← initialPartition(inputSystem)
  3.     neighborhood ← firstNeighborhoodDefinition(currentSolution)
  4.     currentSolution ← HCOptimization (currentSolution, neighborhood, factors)
  5. }//end if
  6. bestSolution ← currentSolution
  7. for (layerToSplit in currentSolution.getLayerList()){
  8.       layerPartition ← initialPartition(layerToSplit, m)
  9.       fullPartition ← replaceLayer(currentSolution, layerToSplit, layerPartition)
 10.       neighborhood ← secondNeighborhoodDefinition(fullPartition)
 11.       layeredSolution ← HCOptimization (fullPartition, neighborhood, factors)
 12.       if (LC(layeredSolution, factors) < LC(bestSolution, factors))
 13.           bestSolution ← layeredSolution
 14.  }//end for
 15.  if (LC(bestSolution) < LC(currentSolution)){
 16.       currentSolution ← bestSolution
 17.       currentSolution ← SAHCLayering(inputSystem, currentSolution, m,
                                                       factors)
 18.  } //end if
 19.  else
 20.      return currentSolution
```

The *SAHCLayering* algorithm starts with an initial partition computed from the input system (line 2) i.e. the analyzed system. We compute the initial partition from the nodes and edges of the module dependency graph representing the input system. This initial partition consists of a set of 3 clusters i.e. layers. In this partition, packages are randomly assigned to each layer. This initial partition is then considered as the current solution of the algorithm (line 2). In the following step (line 4), the algorithm attempts to improve the current solution according to a first neighborhood definition (line 3). For this purpose, it relies on the *HCOptimization* algorithm described below to iteratively compute and assess all the corresponding neighboring solutions so as to find a better solution than the current one.

The layering solution found so far (at line 4) is then considered as the best solution of the layering algorithm (line 6). In the next steps (lines 7 to 18), the algorithm attempts to improve the best solution's quality by dividing each of its layers. Therefore, for each layer *layerToSplit* of the best solution, the algorithm computes a new partition (line 9) which has the same content as the best solution, at the difference that *layerToSplit* is replaced by an *m*-layered partition (line 8). The latter comprises the packages of *layerToSplit* and its layers are computed similarly as the initial partition's ones (line 2), except that the only dependencies considered when determining the *m*-layered partition are the internal dependencies of *layerToSplit*. Next, the content of the new partition is optimized (line 11) by applying the *HCOptimization* algorithm based on a second neighborhood's definition (line 10) and the resulting layering solution is accepted as the solution of the iteration (lines 12 and 13) if its quality is better than the best solution's one. The best solution is then accepted as the current solution of the algorithm if its quality is lower than the current solution's one (lines 15 to 18). In this case, the layering algorithm tries again to improve the current solution by attempting to split its layers (line 17). Otherwise, the algorithm stops (line 20).

Remark that an attempt to further divide a layer into *m* other layers (lines 7 to 18) can fail if the solution obtained when dividing the layer increases the LaQ of the layering solution, by generating extra skip-calls and back-calls.

As explained earlier, to optimize the content of its partitions, the *SAHCLayering* uses another SAHC algorithm: the *HCOptimization* algorithm. The latter which is described by Algorithm 2.2 takes as input: 1) an initial partition layered partition provided by the neighborhood; 2) the appropriate neighborhood's definition to use when computing the neighboring solutions; and 3) the values of the factors (*ap*, *ip*, *sp* and *bp*) assigned to each kind of layering dependencies. The initial partition taken as input by the *HCOptimization* is then considered as the best solution of the algorithm (line 1).

In the following iterations (lines 2 to 13), the algorithm computes all the neighboring solutions (line 4) and evaluates them based on their quality (line 6). A neighbor solution is

computed by moving a single package from a layer to a distinct one, provided that this move is in accordance with the input neighborhood's definition. The neighbor having the lowest value of LaQ is considered as the best neighbor of the iteration (lines 6 and 7). This neighbor is accepted as the new best solution if its fitness is lower than that of the best solution (lines 9 and 10). The algorithm stops if the best solution cannot be improved anymore (lines 11 and 12).

Algorithm 2.2 Hill climbing based optimization algorithm

```
Algorithm HCOptimization
Input: inputPartition, neighborhood, factors
Output: LayeredSolution
  1.  currentSolution ← inputPartition
  2.  while (TRUE){
  3.      bestNeighbor ← NULL // bestNeighbor's LC is set to +∞ by default
  4.      neighborList ← computeAllNeighbors(currentSolution, neighborhood)
  5.      for (neighbor in neighborList){
  6.          if (LC(neighbor, factors) < LC(bestNeighbor, factors))
  7.              bestNeighbor ← neighbor
  8.      }//end for
  9.      if (LC(bestNeighbor, factors) < LC(bestSolution, factors))
  10.         bestSolution ← bestNeighbor
  11.     else
  12.         END WHILE LOOP
  13. }//end while loop
  14. return currentSolution
```

The *SAHCLayering* algorithm and the *HCOptimization* use two definitions of the neighborhood. According to the first definition of the neighborhood (line 3 of the *SAHCLayering* algorithm), a neighboring solution can be computed by moving a single package from a layer A to a layer B of the current solution, provided these two layers are different. In this case, to determine the neighborhood of a partition *P* in *HCOptimization* (line 4), we compute all its possible neighbors. The second neighborhood's definition (line 10 of the *SAHCLayering* algorithm) is used to optimize the new partition derived from the best solution by replacing a layer *L* of the best solution by an *m*-layered partition. According to this second neighborhood's definition, a neighboring solution can be computed in

HCOptimization (line 4) by moving a package from a layer A to a layer B of the current solution, provided that the package and the layers A and B were initially comprised in layer *L*. This second definition of the neighborhood is meant to restrain the optimization at the *m* layers derived from *L* and as a consequence to reduce the computation cost.

Note that each neighbor does not always generate an acceptable solution (Mitchell, 2002b). In particular, a partition with a low number of layers may lead to a monolithic structure of the system (Buschmann et al., 1996). And since most applications generally have three layers or more (Stoermer et al., 2003; Kruchten, 1995), we remove from the possible neighbors list, the subset of neighbors having less than three layers.

2.4.2.3 On the stochasticity of the layering algorithm

The *SAHCLayering* algorithm (and the *HCOptimization* algorithm) is based on the hill climbing. As such, it starts from an initial random layered partition that it tries to improve by moving nodes between layers and splitting resulting layers. However, the randomness of this initial partition can drive the *SAHCLayering* to generate solutions that do not always converge toward the same optimal solution. In the literature, authors (e.g., Mitchell et al. (Mitchell et al., 2006) and Saeidi et al. (Saeidi et al., 2015)) generally overcome the stochasticity of the hill climbing, by running this algorithm many times. In this case, the solution having the best value of the fitness function (i.e. the lowest LaQ value in our context) among the so-generated solutions is usually kept as the best solution. As such, we also perform multiple runs of the *SAHCLayering* algorithm. This also has the advantage of reducing the risk of getting trapped in local optima.

2.4.2.4 Incremental computation of LaQ

Computing the value of LaQ of each partition from scratch can be time-consuming for very large systems. To overcome such scalability issues, we compute LaQ incrementally. In this case, Let NP be the neighbor computed by moving a node (i.e. package) N from a layer A to

another layer B of a partition P. When moving N toward layer B, the only dependencies affected by this move are the ones related to N. The interpretation of each of these dependencies might change depending on the position of layer B relatively to the other layers of NP. Therefore, instead of considering all the dependencies of the system to compute the quality of NP, we can incrementally compute the LaQ of NP by only taking into account the dependencies connecting N to the other nodes of the system. As such, to compute the LaQ of NP, we only need to subtract from P's LaQ, the quality part involved in the moving the node N to layer B. This quality part is the difference between the LaQ computed when the node N in layer A within P and the LaQ computed when the node N in layer B within NP.

2.5 The layered architecture recovery as a quadratic semi-assignment problem

A further review of the literature on optimization problems (e.g., (Taillard, 1991; Pardalos et al., 1994)) showed us that the assignment of packages to layers was a special case of QAP (Quadratic Assignment Problem), a well-established combinatorial optimization formulation which has been used to model problems such as layout design or resource allocation. This particular case of QAP is known as as the Quadratic Semi-Assignment Problem (QSAP) (Pardalos et al., 1994). Thus, we analyzed the Incremental Layer Dependency rule to define a number of factors corresponding to possible types of assignments of dependent packages to the layers of a given system. We then use these factors to translate the problem of recovering layered architectures as a Quadratic Semi-Assignment Problem (QSAP).

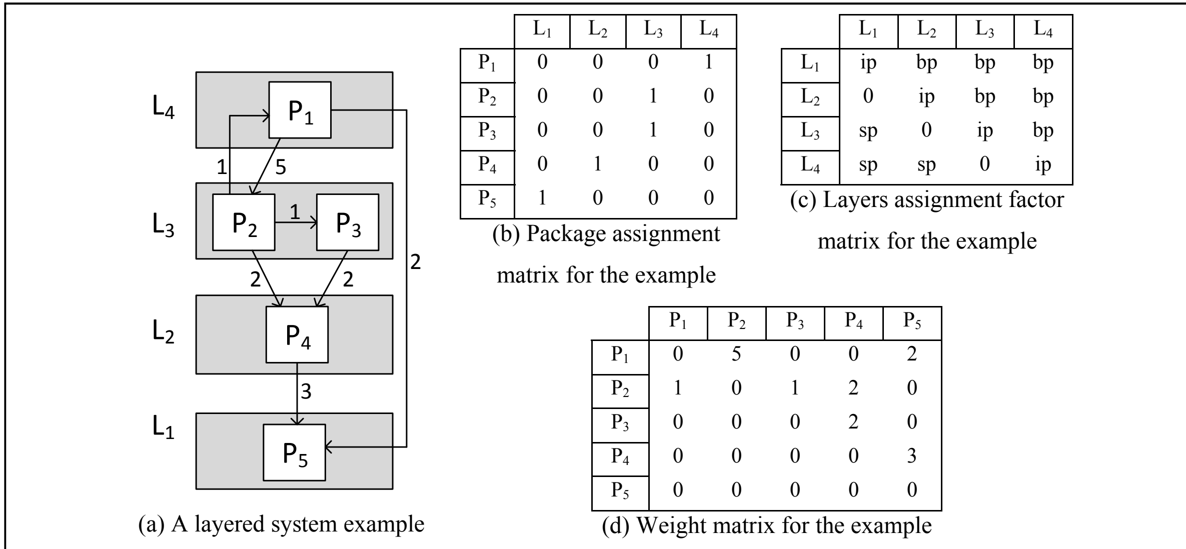
2.5.1 Factors for Layers Assignment

Let packages i and j be two distinct packages of the system with a directed dependency from i to j. Recall that the weight of a dependency from i to j is obtained by summing the weights of the dependencies directed from the entities of package P_1 to those of package P_2. Let c_{kl} be the factor (i.e percentage) adjoined to a dependency directed from package i to package j when assigning packages i and j to layers k and l, respectively. In accordance with the

Incremental Layer Dependency rule, we classify the possible assignments of two packages *i* and *j* to layers according to the four following categories:

- Adjacent layers assignment: in this case $k = l+1$; this is the optimal and desirable assignment of two dependent packages and thus, has zero factor 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 factor $c_{kl} = ip$ 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 (Harris et al. 1995)) in small numbers and has a non-zero factor $c_{kl} = sp$ 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 factor $c_{kl} = bp$.

In the layered system illustrated by Figure 2.3(a), the corresponding assignment of packages P_1 and P_2 to layers L_4 and L_3, respectively, is *bp* since there is a back-call dependency relating P_2 to P_1. The corresponding assignment of the packages P_1 and P_5 to layers L_4 and L_1, respectively, is $2*sp$ because it introduces a skip-call dependency having a weight of 2. The corresponding assignment of the packages P_2 and P_3 to the same layer L_3, is *ip* since there is an 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 factor of this layered system is: ($ip + 2*sp + bp$).



	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	ip	bp	bp	bp
L2	0	ip	bp	bp
L3	sp	0	ip	bp
L4	sp	sp	0	ip

(c) Layers assignment factor 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

Figure 2.3 An example of a layered system and its related matrices

2.5.2 Layers Recovery as a Quadratic Semi-Assignment Problem

The recovery of a system's layered architecture consists in finding a mapping function that assigns each package to a given layer while minimizing the intra, skip-call and back-call dependencies. 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. As mentioned earlier, the dependency between two packages derives from the dependencies between their respective entities. 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 factors. Figure 2.3(c) displays the layer assignment matrix while Figure 2.3(d) displays the weight matrix corresponding to the system of Figure 2.3(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. Figure 2.3(b) displays the package assignment matrix corresponding to the system of Figure 2.3(a).

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

$$\min f(X) \;=\; \sum_{i,j=1}^{m} \sum_{k,l=1}^{n} w_{ij}\, c_{kl}\, x_{ik}\, x_{jl} \tag{2.5}$$

$$x_{ik} \in \{0,1\} \quad i = 1, \ldots, m,\ k = 1, \ldots, n \tag{2.6}$$

$$\sum_{k=1}^{n} x_{ik} \;=\; 1 \quad i = 1, \ldots, m \tag{2.7}$$

In our context, the quadratic function of Eq. (2.5) is called the global layering quality of dependencies. This function[13] defines a factor (i.e. percentage) for each possible set of assignments of packages to layers. Thus, the factor of assigning package *i* to layer *k*, if a package *j* is assigned to layer *l* corresponds to w_{ij} * c_{kl}. Eq. (2.7) constrains the possible solutions to Eq. (2.5) by stating that a package may be assigned only to one layer.

2.5.3 Solving the layering recovery QSAP

Taillard (Taillard, 1991) explains that the QAP consists in assigning a set of N units to a set of N different locations, so as to minimize the sum of the product of the distance between every pair of units by the flow between every pair of locations. Different common problems can be expressed in terms of QAP. These include: 1) the traveling salesman problem; and 2) the assignment of electronic components in a chip so as to minimize the total length of the connections. A particular case of the QAP is the Quadratic Semi Assignment Problem (QSAP). The QSAP aims at assigning M units to N locations (N<M) so as to minimize the sum of the factors of assigning each unit to a given location (Pardalos et al., 1994). The formulation of the QSAP does not necessarily take into account the distance between the locations or the flow between the units. QSAP Problems include the graph coloring problem

[13] The measurement unit of f(X) is the dependency.

and the clustering problem (Pardalos et al., 1994). The QSAP is an NP-hard computational problem and finding a global optimal solution for this problem is 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 (Pardalos et al., 1994). Among these, the tabu search method (Skorin-Kapov, 1990) 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 quadratic function f (Eq. (2.5)) as a fitness function. Briefly, the tabu search (Glover, 1997) 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. As pointed out in (Blum and Roli, 2003; Taillard, 1991), the size of the tabu list influences the search process. If the tabu list size is small, the search process concentrates on small areas of the search space and is likely to cycle. On the other hand, if the tabu list size is large, it forbids revisiting a high number of solutions. The search process is therefore obliged to explore larger regions, which requires a larger number of iterations to locate the desired solution. As demonstrated in (Taillard, 1991; Battiti and Tecchiolli, 1994), varying the size of the tabu list during the search leads to more robust algorithms.

2.5.3.1 A tabu search based layering algorithm

Our adaptation of the tabu search technique to the layering problem is called *TabuLayering*. A simplified view of *TabuLayering* is provided by Algorithm 2.3. This algorithm takes the

following inputs: 1) an initial partition built by randomly assigning the system's modules into a set of 3 layers; 2) a set of values assigned to the factors; and 3) the maximum number of iterations (*max_it*) after which the algorithm ends.

Algorithm 2.3 A high level view of the tabu search-based layering algorithm

```
Algorithm TabuLayering
Input: initialLayeredPartition, max_it,  ip, sp, bp
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 (f(currentSolution) < f(bestSolution)) {
15.      bestSolution ← currentSolution
16.    }
17.    K ← K +1
18. } //end while
19. return bestSolution
```

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 to another layer (line 7). This neighbor is considered as a candidate solution if it is generated using a package move that does not belong to the tabu list (lines 8 to 10). It is worth pointing out that a package move might introduce an additional layer (i.e., the final layering may have more than 3 layers). The candidate solution having the lowest quality i.e. the lowest value of the quadratic function f is the best candidate solution and 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 the value of its quadratic function is lower than the current best-known solution's one (lines 14 to 16).

Note that, similarly to the *SAHCLayering* algorithm, we also run the *TabuLayering* algorithm multiple times and keep as the best solution the solution having the best factor (i.e. the lowest factor in our context) among the so-generated solutions. Besides, as with the *SAHCLayering* algorithm, the *TabuLayering* algorithm can be applied in each layer of the best solution to further refine it.

2.6 Chapter summary

In this chapter, we presented an approach for the reconstruction of software systems'architectural layers. To this end, we revisited and analyzed the layered style and retained six rules. A subset of these rules led to the specification of a set of layers' dependency attributes and constraints. The latter allowed us to translate the layering recovery problem into an optimization problem that we solved using a search-based algorithm. To refine this formalization, we defined dependency factors corresponding to possible types of assignments of dependent packages to the layers of a given system. We then use these factors to express the layering recovery problem as a specific case of QAP, namely a Quadratic Semi-Assignment Problem (QSAP). Besides, to ensure the independence of our approach from the nature of the languages and platforms used to develop the systems under study, we represent these systems using platform independent models that are compliant with the Knowledge Discovery Metamodel (KDM).

CHAPTER 3

REALEITY: A TOOL FOR RECOVERING SOFTWARE LAYERS FROM OBJECT ORIENTED SYSTEMS

In this Chapter, we introduce a tool which enables to recover layered architectures (phase 2 of the research methodology). This tool provides the following main functionalities: 1) extracting the entities and their dependencies from an analyzed system using a standard representation, 2) reconstructing the architectural layers by assigning abstraction levels to the extracted entities using constraints and layers' dependency attributes extracted from the layered style, 3) visualizing the so-obtained layers and their corresponding layers' dependency attributes and, 4) refining the recovered layers. Our tool is useful to understand and document layered systems as well as to detect layering violations.

3.1 Description

To automate the layering approach presented in the previous Chapter, we implemented it as a plugin within the Eclipse™ environment. We named this tool ReALEITY (REcovering softwAre Layers from objEct orIenTed sYstems). To compile and execute ReALEITY we use a 64 bits Java Virtual Machine (JDK 1.8) (Oracle, 2015) which makes it portable from an environment to another. Figure 3.1 illustrates the ReALEITY modules as well as the Eclipse plug-ins on which it relies. We explain the role of each of these plug-ins below.

ReALEITY has three main functionalities which are described in the following subsections: 1) extracting the facts of an object-oriented system; 2) performing the layering of this system using the input (i.e. the values of the factors); and 3) visualizing the corresponding results. Notice that the tool user can interactively refine the so-obtained results.

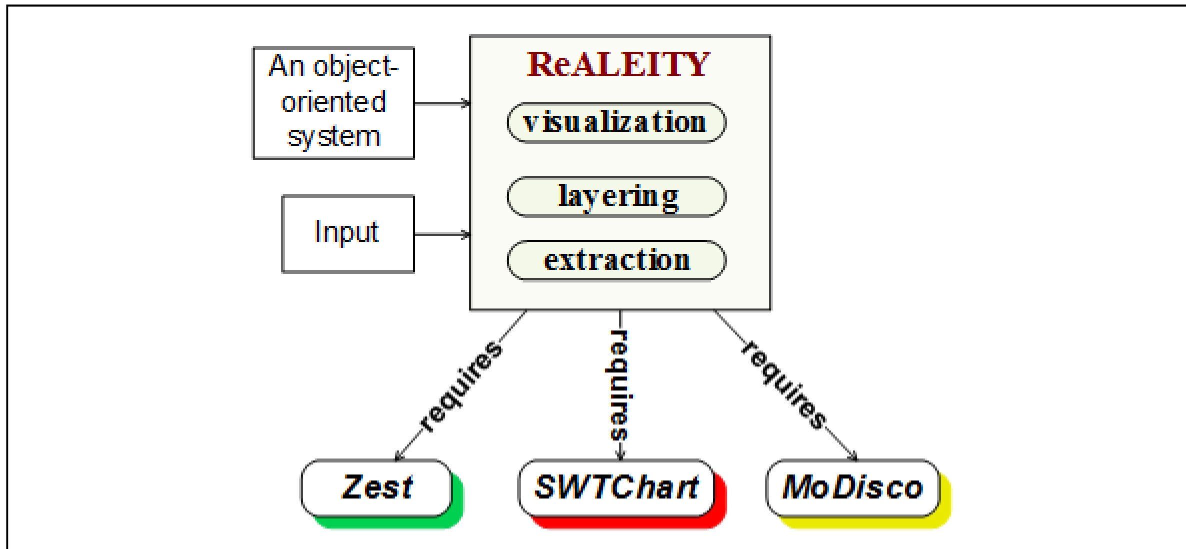


Figure 3.1 ReALEITY design

3.2 Extraction

The extraction using ReALEITY is performed in two steps: 1) generating a representation of the analyzed system; and 2) parsing this representation so as to extract the system's structural facts. The first step generates the system's representation by analyzing the source code files of the system at hand. To do so, the ReALEITY relies on MoDisco (Modisco, 2015) which is an Eclipse plug-in that allows analyzing source code files of a system and generating platform independent models that are compliant with KDM (e.g., inventory model, code model). These models are further described in Section 1.1.5.

In the second step, the extractor module parses the KDM models generated from the source code, to retrieve packages and their relationships. Dependencies between two packages are derived from the dependencies between their respective classes (i.e., class references, inheritance, method invocation and parameters). ReALEITY uses the extracted facts to build a module dependency graph. The nodes and the edges of this graph are respectively the extracted packages and the dependencies between them. The resulting module dependency graph is stored as a KDM structure model which is added to the collection of the KDM models representing the analyzed system. Therefore, the next time ReALEITY will extract

the facts from the same analyzed system, it will only read that structure model. This enables saving time by avoiding extracting once again the facts from the system's KDM models. ReALEITY also stores the system's module dependency graph in two txt files that are respectively visualizable by GraphViz (Graphviz, 2015) and Zest (Zest, 2015).

If the analyzed system has external dependencies, these dependencies are not taken into account by ReALEITY when extracting the system's facts.

3.3 Layering

The extracted facts are used as input of the layering module. ReALEITY performs the layering according to the following steps:

- **Setting the parameters of the layering recovery algorithms using a wizard**. This wizard is depicted in Figure 3.2. Table 3.1 explains in details the elements to fill in the layering recovery parameters wizard. These parameters include the factors values, the selection of a layering algorithm (i.e., *SAHCLayering* or *TabuLayering*) and the fitness function (e.g., LaQ or its QAP equivalent f(X)). The structure model that will contain the layering results is then created and named after the output name specified in the wizard. The parameters entered through the wizard are stored in that structure model as its attributes.
- **Execution of the layering algorithm**. For this purpose, ReALEITY builds an initial layered partition from the module dependency graph obtained during the extraction phase. The best partition is then obtained by improving that initial partition using the selected layering algorithm and the parameters selected in the recovery parameters wizard. At the end of the execution of the layering algorithm, the content of the best partition found is stored in the created structure model. The layers'dependency attributes corresponding to that best partition are added to the created structure model's attributes. That structure model is then added to the XMI file describing the analyzed system.

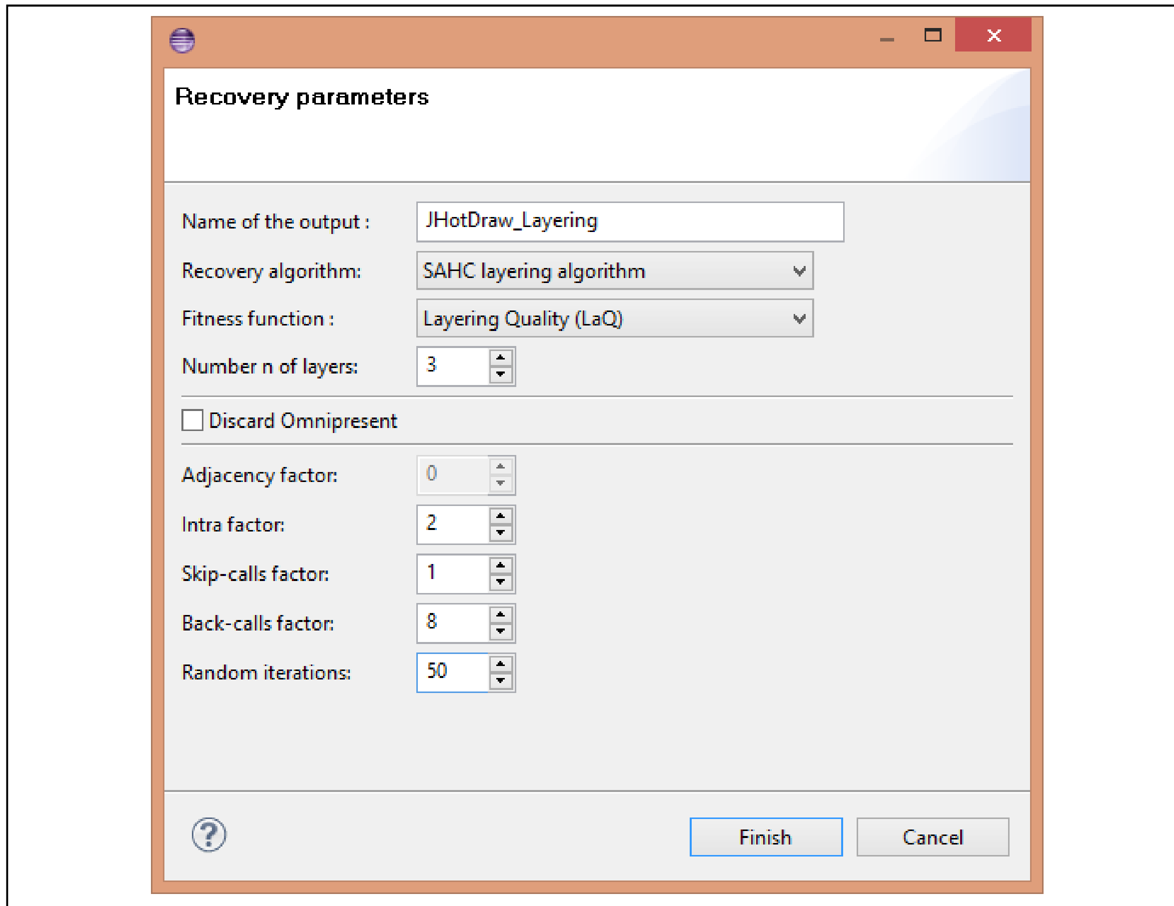


Figure 3.2 Layering recovery parameters wizard: case of JHotDraw

Table 3.1 Explanation on the elements in the layering recovery wizard

Element in the wizard	**Explanation**
Name of the output	Name of the structure holding the layering results
Recovery algorithm	layering recovery algorithm to run
Fitness function	Name of the fitness function used to evaluate the layered partitions that the recovery algorithm generates
Number n of layers	Initial number of layers to recover before recursively decomposing the layers. In our experiments, we set *n* to the default value 3.
Discard omnipresent	Indicates whether the omnipresent packages should be identified

	and discarded of the layering results
Adjacency factor (*ap*)	Factor adjoined to the adjacent dependencies. Always set to zero since our layering algorithm rewards the adjacent dependencies
Intra factor (*ip*)	Factor adjoined to the layers' internal dependencies
Skip-calls factor (*sp*)	Factor adjoined to the skip-calls dependencies
Back-calls factor (*bp*)	Factor adjoined to the back-calls dependencies. This element is disabled until the user specifies the Intra factor and the Skip-calls factor.
Random iterations	Number of multiple runs executed by the algorithm

At the end of the layering process, a folder named Layering_Results is automatically created/updated inside the analyzed system folder. Each time the user performs the layering, ReALEITY automatically add two txt files in Layering_Results. These files describe the layering results and their names both start with the output name specified in the layering recovery parameters wizard. The first file name contains the word "Zest" to indicate that Zest can display the content of that file. The second file contains the word "GraphViz" to indicate that GraphViz can display the content of that file.

3.4 Visualization

ReALEITY supports the visualization of the MDG extracted from the analyzed systems and the layering results. For this purpose, the ReALEITY interface comprises 4 resizeable views that are depicted by Figure 3.3.

These views are: ReALEITY-MDG, ReALEITY-Layering, ReALEITY-Metrics and the Package Explorer. The Package Explorer view holds the Java systems to analyze. To analyze a system, the user has to import it in Eclipse. Once the system is imported, it appears in the Package Explorer view. That view is provided by Eclipse. The ReALEITY-MDG view displays the extracted facts as a graph where the nodes and edges are respectively the extracted packages and their dependencies. The ReALEITY-Layering view displays the

resulting layered architecture of the analyzed system. That view represents the layers as rectangular boxes and the layers' packages as nodes. These nodes are related by edges that represent the packages dependencies. The ReALEITY-Layering view allows the user to refine the recovered layers by reassigning nodes to different layers or by renaming the layers. The so-modified layering can also be saved for future use.

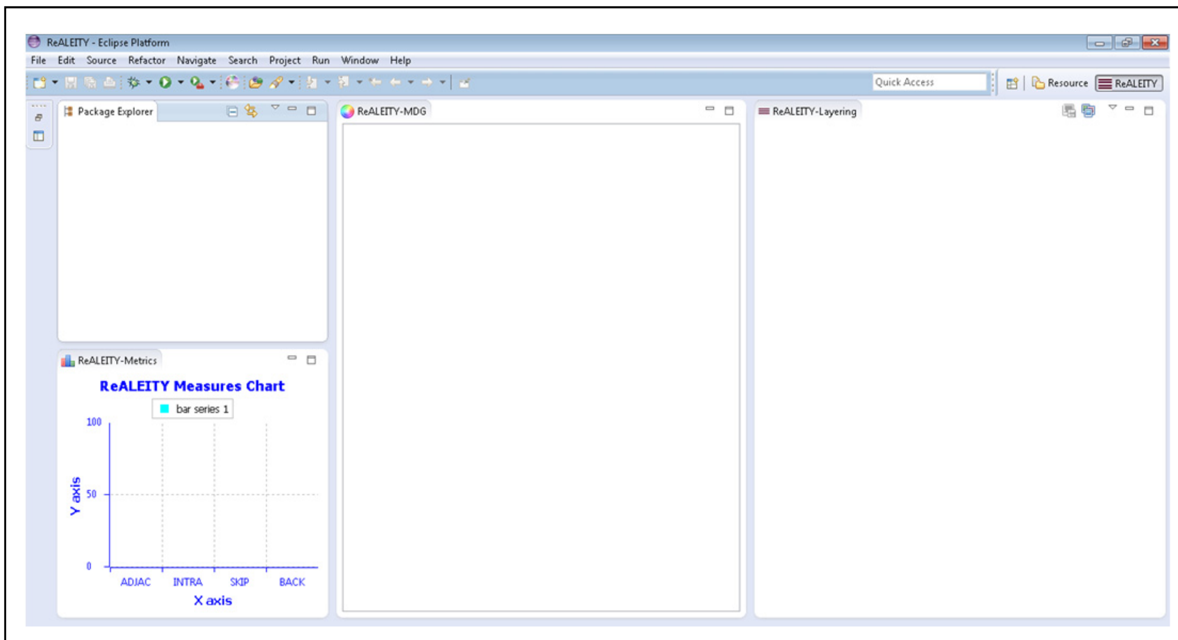


Figure 3.3 ReALEITY interface

The ReALEITY-Metrics view displays the automatically generated layers' dependency attributes values using a bar chart whose horizontal axis represents four values: ADJAC, INTRA, SKIP, and BACK. These values respectively correspond to the sum of the four layers' dependency attributes AdjacencyUse, IntraUse, SkipUse and BackUse of all the layers. If the user refines the layering, the ReALEITY-Layers' dependency attributes view is updated to display the layers' dependency attributes of this new layering.

The ReALEITY-MDG and the ReALEITY-Layering views rely on the Zest (Zest, 2015) functionalities to perform the display. Zest is a Visualization Toolkit that allows visualizing graphs inside the Eclipse environment and according to different layout algorithms. The

ReALEITY-Metrics view relies on SWTChart (SWTChart, 2015) to display the layers' dependency attributes. SWTChart is an Eclipse plug-in that allows visualizing charts (e.g., line charts, bar charts and stack charts).

3.5 Commands

The ReALEITY plugin comprises two commands illustrated by Figure 3.4. The first command is named **Create initial MDG**. This command allows the user to launch the extraction of the analyzed system. At the end of the extraction process, the ReALEITY-MDG view displays the module dependency graph of the analyzed system. The second command of ReALEITY is named **Create Layering**. This command allows the user to launch the layering of the analyzed system. For this purpose, the second command begins by prompting the user to specify the input needed to perform the layering through a wizard (see Figure 3.2). The user's input is mainly constituted of the factors (*ap*, *ip*, *sp*, and *bp*), the choice of the layering algorithm and the fitness function. Once the layering is over, the second command makes the ReALEITY-Layering view and the ReALEITY-Metrics view respectively display the recovered layers and the corresponding layers' dependency attributes. To generate an alternative layering, the user can click once again on the command Create Layering and specify other values for the factors.

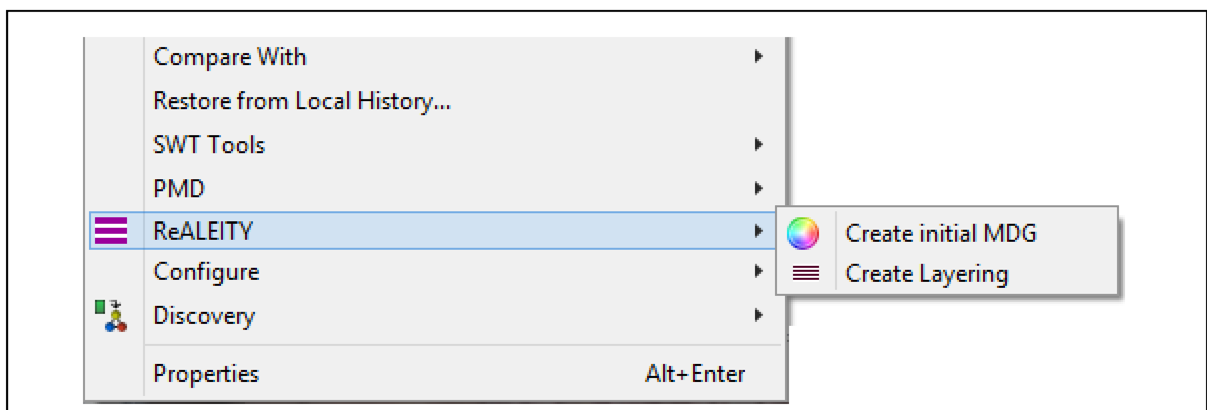


Figure 3.4 The two commands of ReALEITY

Note that the execution times of the respective tasks performed by the ReALEITY commands depend on the size of the analyzed system. These tasks are then respectively monitored using a progress monitor that indicates their progression (see Figure 3.5).

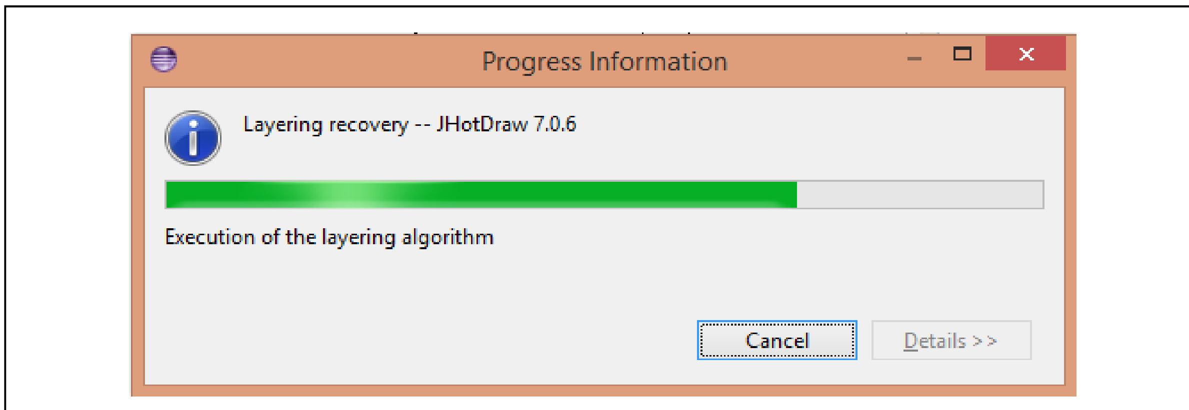


Figure 3.5 Progression of the layering phase for a system named JHotdraw 7.0.6

3.6 Example

This section presents an example of the use of ReALEITY to recover the layers of JHotDraw 7.0.6, a 51 KLOC layered system developed in Java. The extraction of JHotDraw 7.0.6 with ReALEITY shows that JHotDraw 7.0.6 contains 24 packages. It also shows that these packages comprise 310 classes and are interconnected by the mean of 89 connections. Figure 3.6 shows the JHotDraw 7.0.6's extracted facts as displayed by the ReALEITY-MDG view.

Figure 3.7 illustrates JHotDraw 7.0.6's best layering solution as displayed by the ReALEITY-Layering view. This figure is obtained when performing a single run of the *SAHCLayering* algorithm on JHotDraw 7.0.6, using a given set of factors (i.e., *ap*=0, *ip*=2, *sp*=1 and *bp*=4) and a value of *n*=3 as the number of layers of the initial partition. Note that moving the cursor's mouse over any node displayed in that view allows displaying the node's name, i.e., the complete namespace of the package represented by this node.

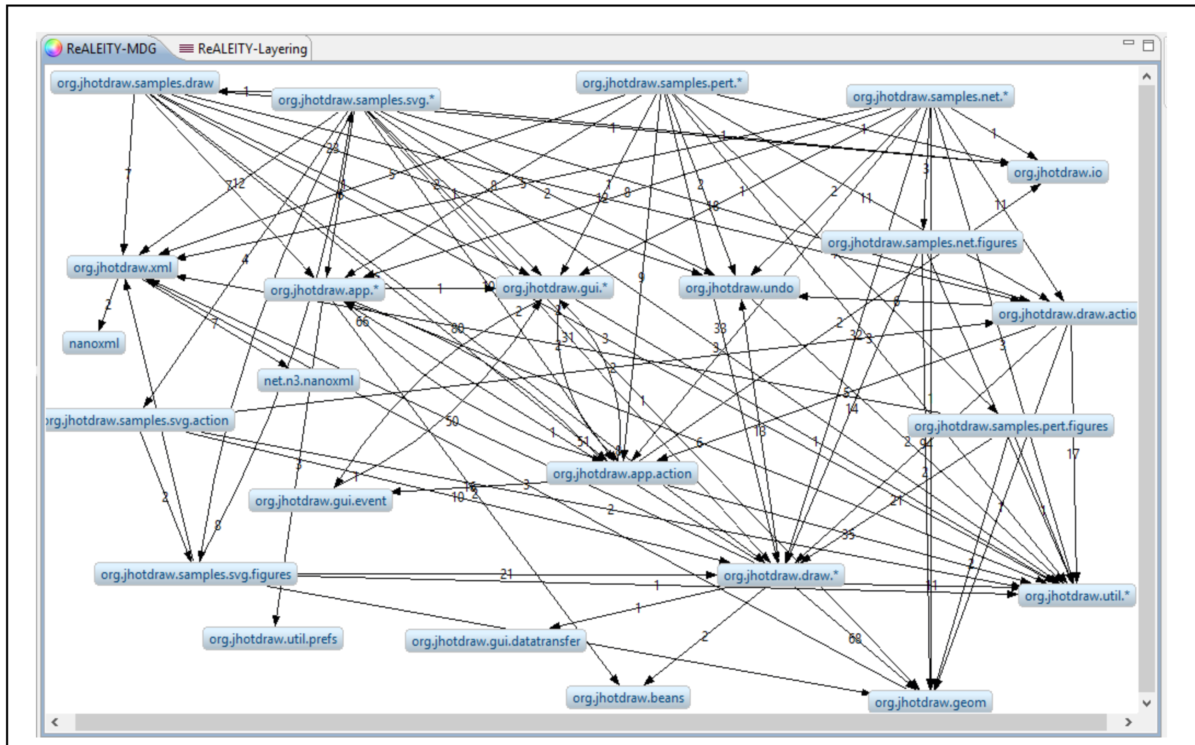


Figure 3.6 JHotDraw 7.0.6 extracted facts

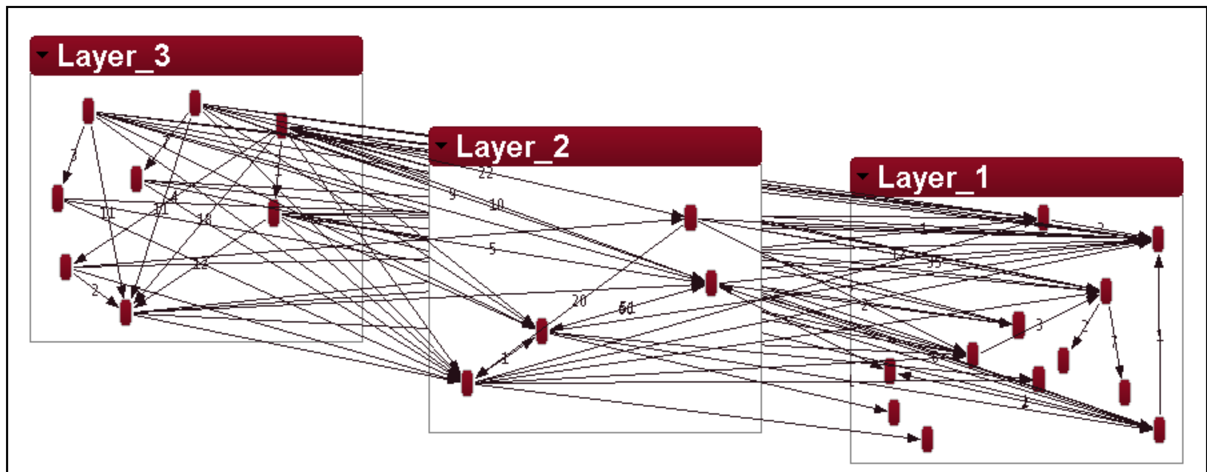


Figure 3.7 JHotDraw 7.0.6's resulting layered architecture

Figure 3.8 shows the JHotDraw 7.0.6's best layering solution layers' dependency attributes values (obtained with the specified factors) as displayed by the ReALEITY-Metrics view.

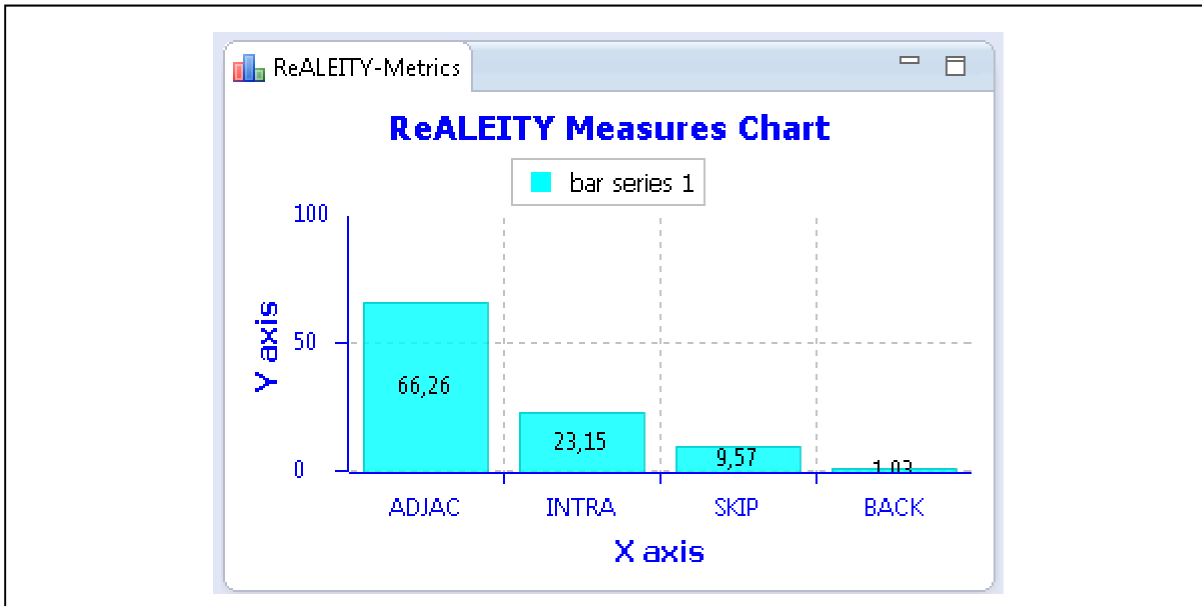


Figure 3.8 JHotDraw 7.0.6's layers' dependency attributes results

Figure 3.9 shows the interface of ReALEITY once the layering is performed on JHotDraw 7.0.6 using the parameters specified above (at the beginning of this section).

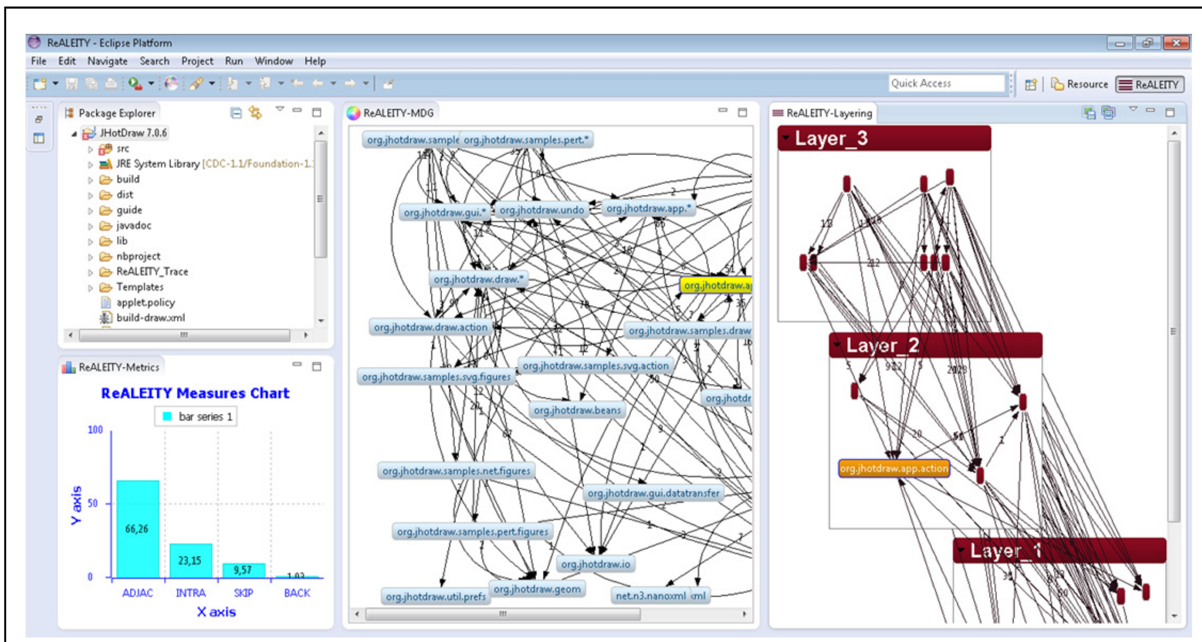


Figure 3.9 ReALEITY interface after the layering of JHotDraw 706

3.7 Summary

This Chapter has presented ReALEITY, a tool that supports: 1) extracting facts from the analyzed software system and using a standard and platform-independent representation of the system; 2) performing the layering using these facts; 3) visualizing the resulting layering architecture and related layers'dependency attributes; and 4) modifying the so-obtained layering results. ReALEITY is implemented as a plug-in within the Eclipse™ environment and uses three Eclipse plug-ins: MoDisco, Zest and SWTChart. The ReALEITY tool, its user manual and its demo are available at (ReALEITY, 2015).

clicours.com

CHAPTER 4

EVALUATING THE STRUCTURAL-BASED RECOVERY APPROACH

In this chapter, we present and discuss the results of our structural-based layering approach when applied to five analyzed systems (phase 2 of the research methodology). To this end, Section 4.1 describes our experimental setup including the set of experimentation questions that our validation aims at answering and the systems analyzed during this validation. Sections 4.2 to 4.6 respectively discuss the results computed to address each experimentation question. Section 4.7 outlines some of the threats to validity to which our approach is subject and summarizes the chapter in Section 4.8.

4.1 Experimentation setup

In the following sections, we present the set of experimentation questions that our validation aims at answering as well as the set of systems that we used as subjects during our validation. To compute the results during the validation, we rely on a machine with an Intel(R) Core i7-3778 CPU @3.40GHz and 16 GB of RAM.

4.1.1 Experimentation questions

The purpose of the evaluation of our approach is to address the following questions:

EQ1: *What is the effect of altering the values of the factors on the convergence* [14]*of the algorithms?* For a given setup of the factors (*ap*, *ip*, *sp* and *bp*), we consider that a solution is optimal if its quality LaQ is the lowest among the quality of the solutions generated over multiple runs. To answer this experimentation question, we will study the influence of the

---

[14] An algorithm converges when it is no longer able to find a partition whose quality is better than the current partition's one [Mitchell2006].

factors on the layering quality LaQ of the results respectively obtained with the hill climbing algorithm and the tabu search algorithm. This is meant to determine which of the two algorithms is more likely to lead to an optimal solution.

EQ2: *What are the values of factors (ap, ip, sp and bp) that generate software layers that best correspond to the common understanding of the layered style?* For any given layered software system, assuming that its layered architecture is already known, the point of this question is to look for the values of the 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, there is a 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 rules, as discussed in CHAPTER 2, are enforced by designers. To investigate this question, we carry out a quantitative analysis of our layering results using the harmonic mean (F-measure) (Baeza-Yates and Ribeiro-Neto, 1999) of their precision and recall. We used the formulas introduced in (Scanniello et al., 2010a) to compute the precision and recall. We therefore compute the precision as the number of packages correctly assigned by our tool over the total number of packages that our tool assigns. We compute the recall as the number of packages that our tool correctly assigns over the number of the packages assigned to layers in the authoritative decomposition[15].

We rely on previous works (e.g. (Sangal et al., 2005a and De Oliveira Barros et al., 2015)) to specify the authoritative decomposition of two of the analyzed systems: Apache Ant and JUnit. We (i.e two professors[16] and the researcher) manually decomposed the other two systems for which the authoritative decomposition was not available (e.g., JHotDraw). This choice was motivated by: 1) the lack of experts to produce the reference architectures for all the analyzed systems; 2) our familiarity with the analyzed systems (i.e., we have from two to

[15] Noteworthy, the precision and recall might differ when there is a package that is not related to others. In this case, the tool cannot assign this package to any layer while it is assigned in the authoritative decomposition.

[16] From the Department of Software and IT engineering at École de technologie supérieure.

five years of experience with the architecture and the code of these systems); and 3) our solid background in object-oriented design. At first, we individually produced a preliminary decomposition of the analyzed systems. Then, we worked all together to discuss these decompositions so as to produce the final decompositions. These decompositions are reported in APPENDIX II, p. 199.

EQ3: *How do the layering results evolve across revisions of a software system and what does it tell about the architectural evolution of the system*? This question is related to two aspects, namely: 1) the stability of our layering recovery algorithm and 2) the similarity of the set of values of the setups of factors that yield the layering that matches the known architecture of the system across its versions. Regarding the first aspect, we consider, as in (Scanniello et al., 2010b), that a stable algorithm should generate similar layerings in case of small changes between successive versions. The second aspect can be rephrased into "*when a layered system evolves, does it maintain the same level of conformity to the layering rules?*". To answer this question, we will analyze different versions of JHotDraw.

EQ4: *Is the layering approach performant regarding the size of the system at hand*? The purpose of this experimentation question is to assess the performance of our layering approach as the size (i.e total weight of package dependencies/ quantity of package dependencies) of the system at hand changes during its evolution. To answer this experimentation question, we will analyze the performance of our layering approach when recovering the layering of different versions of the same system i.e JHotDraw.

EQ5: *Is the approach more authoritative than other architecture recovery approaches supported by ARCADE*? This question aims at finding out whether our structural-based recovery approach is more likely to generate layering results close from the manual decomposition of the system at hand than other approaches. To answer EQ5, we will compare our structural-based layering approach to the recovery approaches based on clustering techniques. In this regard, our focus is on the most accurate architecture recovery

techniques implemented by ARCADE[17], namely: ACDC and ARC. These algorithms work at the class level and they partition the analyzed system into clusters with finer granularity compared to the ones of the packages used by our approach. To compare our layering approach with these two clustering approaches, we will then combine each of these two algorithms (ACDC and ARC) with our layering technique presented in Chapter 2. We will therefore derive a first layering technique from ACDC and another one from ARC. We will then use the harmonic mean (F-measure) to compare our layering results with the results respectively obtained from the layering techniques derived from ACDC and ARC.

4.1.2 Analyzed systems

To answer our experimentation questions, we applied our approach on five systems developed in Java. These systems are: Apache Ant, JFreeChart, jEdit, JHotDraw (several versions) and JUnit. All these systems are purported layered systems that are actively maintained and that were analyzed in related works (e.g., (Laval et al., 2013; Scanniello et al. 2010a)):

- Apache Ant is a portable library which allows building Java applications.
- JUnit is a portable framework which supports unit tests in Java.
- JFreeChart is a library that supports developers in displaying various charts in their applications.
- jEdit is an extensible programmer’s text editor.
- JHotDraw is a framework that supports the design of drawing editors.

Table 4.1 indicates some of these systems’ statistics. The first column shows the names of the analyzed projects while the remaining five columns show the number of files, the number of lines of code, the number of packages, the number of packages dependencies for each

[17] During our experiments, we will use the Java implemention of ARCADE available at: http://softarch.usc.edu/wiki/doku.php?id=arcade:start

project and the total [18]weight of package dependencies respectively (i.e. quantity of package dependencies).

Table 4.1 Statistics of the analyzed systems

System	Number of files	LOC (Lines Of Code)	Number of packages	Package dependencies	Total weight of package dependencies
Apache Ant 1.6.2	681	171 491	67	229	2068
JUnit 4.10	162	10 402	28	106	356
JFreeChart 1.0.15	600	222 475	37	243	2313
jEdit 5.0.0	539	148581	34	176	1706
JHotDraw 60b1	498	68509	17	72	1213
JHotDraw 7.0.6	310	51 801	24	89	988
JHotDraw 7.0.7	310	57 020	24	89	972
JHotDraw 7.0.9	472	83642	43	151	1510
JHotDraw 7.2	577	110735	46	199	1958
JHotDraw 7.3	584	112494	46	216	2068
JHotDraw 7.4.1	585	111 239	62	365	2624
JHotDraw 7.5.1	613	119643	65	384	2718
JHotDraw 7.6	680	118 938	65	358	2479

4.1.3 Setting the parameters of the layering algorithms

To run the *TabuLayering* algorithm, we set the maximum number of iterations to 200 and the tabu list length to 10 (i.e., the tabu list records the last ten best packages' moves). To further study the influence of the tabu search parameters on the convergence of the tabu search algorithm, we vary the maximal number of iterations from 100 to 500 by step of 100 while keeping the size of the tabu list at 10. On the other hand, we also vary the size of the tabu list from 10 to 50 by step of 10, while keeping the maximal number of iterations at 200.

To run the *SAHCLayering*, we make *m*[19] vary from 2 to 3. The values of the factors (i.e., *ap* (adjacency factor), *ip* (intra-dependency factor), *sp* (skip-call factor) and *bp* (back-call factor)) can be chosen according to the following five scenarios:

---

[18] This total is the sum of the weights of the package dependencies found within a system.

- Scenario 1: $ap < sp < ip < bp$,
- Scenario 2: $ap = sp < ip < bp$,
- Scenario 3: $ap < ip < sp < bp$,
- Scenario 4: $ap = ip < sp < bp$,
- Scenario 5: $ip < ap < sp < bp$.

Note that choosing the factors values according to scenarios 1 and 2 leads to the generation of layerings with more adjacent and skip-call dependencies than intra-dependencies and back-call dependencies. Scenarios 1 and 2 comply with a reuse-based strategy (Eeles, 2002) since they are likely to produce layerings with a quite high number of skip-calls, which suggests that the most (re)used packages are assigned to bottom layers while the less (re)used packages are put in top layers. In the other hand, choosing the factors values according to the scenarios 3, 4 and 5 allows generating a layering with more adjacent and intra-dependencies than skip-call and back-call dependencies. These scenarios then comply with a responsibility-based strategy (Eeles, 2002) since they are likely to lead to layerings where a package of a given layer uses packages of the same layer to contribute to the same responsibility. In particular, scenario 5 corresponds to a context where an architect might have designed a system with much intra-dependencies in order to address some concerns as portability.

During our experiments, we carried out tests for each of the five scenarios described above, using different setups of factor values for each of these scenarios. We present the results of five significant setups for each scenario. These setups are displayed in Table 4.2. A setup is a quadruple indicating the values of the factors (*ap*, *ip*, *sp* and *bp*) specified for the execution of the layering algorithm. In all the setups, *ap* was set to zero since we reward adjacent dependencies.

---

[19] During the layering recovery process, the algorithm recursively attempts to divide each of the layers of the current partition into *m* layers until the quality LaQ of the so-obtained layering can no longer be improved.

Table 4.2 Setups chosen for the 5 scenarios

	Setup 1				Setup 2				Setup 3				Setup 4				Setup 5			
	Ap	ip	sp	bp	ap	ip	sp	bp	ap	ip	sp	bp	ap	Ip	Bp	bp	ap	ip	sp	bp
Scenario 1	0	2	1	4	0	2	1	8	0	2	1	12	0	2	1	16	0	2	1	20
Scenario 2	0	2	0	4	0	2	0	8	0	2	0	12	0	2	0	16	0	2	0	20
Scenario 3	0	1	2	4	0	1	2	8	0	1	2	12	0	1	2	16	0	1	2	20
Scenario 4	0	0	2	4	0	0	2	8	0	0	2	12	0	0	2	16	0	0	2	20
Scenario 5	1	0	2	4	1	0	2	8	1	0	2	12	1	0	2	16	1	0	2	20

ap: factor adjoined to the adjacent dependencies.

ip: factor adjoined to the intra-dependencies.

sp: factor adjoined to the skip-call dependencies.

bp: factor adjoined to the back-call dependency.

For each system and setup, we run 50 times each layering algorithm (i.e. the *SAHCLayering* and the *TabuLayering* algorithms) and retained the best (lowest layering quality) result.

4.2 EQ1: what is the effect of altering the values of the factors on the convergence of our algorithms?

As indicated above, for each layering algorithm, for each analyzed system and for each of the 5 scenarios' setups, we perform 50 runs. These 50 independent runs respectively generate 50 solutions, each having four dependency layers' dependency attributes values, a number of layers (NBL) and a value expressing the layering quality LaQ. For a given setup, the best layering solution among the 50 generated solutions is the one yielding the lowest quality LaQ. In this section, we are interested by the ability of each layering algorithm to reach the optimal value of the layering quality LaQ.

All the results are reported in Appendix I, p. 177. These results are expressed in terms of the Mean (*Mean*), standard deviation (*StDev*) and minimal value (*Min*) of the layering quality LaQ obtained for each of the analyzed systems when respectively running the *SAHCLayering* and the *TabuLayering* algorithms. Note that, for the sake of brevity, we have limited our analysis to 4 versions of JHotDraw.

Our experiments using the 5 scenarios' setups show that, overall, the setups of the scenarios 1 and 3 are the ones that best fit the layering constraints defined in CHAPTER 2. This indicates that some of the analyzed systems might follow a reuse-based strategy while others follow a responsibility-based strategy. Noteworthy, our experiments show that scenario 2 usually leads to solutions having a very high number of layers (i.e. nearly 7 to 9 layers) and to a number of skip-calls dependencies that outnumbers the adjacent dependencies. Our experiments also show that Scenarios 4 and 5 lead in turn to layerings having a very high number of intra-dependencies; i.e. these scenarios tend to put all dependent packages in the same layer. Hence the resulting layerings do not include neither skip-calls nor back calls dependencies. Such results do not correspond to the real layerings of the analyzed systems. As such, to answer our experimentation question EQ1, we will then focus on the setups of scenarios 1 and 3 to analyze the layering results respectively obtained using the hill climbing algorithm and the tabu search algorithm.

4.2.1 Experimental results with hill climbing

Table 4.3 to Table 4.10 respectively summarize the layering results obtained by applying the *SAHCLayering* on the analyzed systems with the setups of the scenarios 1 and 3. The columns of each of these tables report some descriptive statistics of the layering results' quality (LaQ). These statistics correspond to the mean (*Mean*), the standard deviation (*StDev*) and the minimal value (*Min*) of the distribution comprising the layering results' quality obtained by running the algorithm 50 times for a given system and setup. The minimal value of the distribution indicates the quality of the optimal i.e. best layering solution found among the 50 layered solutions generated for a given setup.

Table 4.3 LaQ variations with the SAHCLayering algorithm applied on Apache 1.6.2

			Mean	**StDev**	**Min**
Apache Ant 1.6.2	**Scenario 1**	***Setup 1***	1035.04	6.90	1026.0
		Setup 2	1126.78	8.72	1120.0
		Setup 3	1182.6	12.71	1163.0

		Setup 4	1226.16	13.09	1216.0
		Setup 5	1290.86	23.86	1262.0
	Scenario 3	***Setup 1***	571.92	2.96	569.0
		Setup 2	619.88	1.40	619.0
		Setup 3	668.58	12.79	664.0
		Setup 4	745.6	99.64	701.0
		Setup 5	789.52	105.13	739.0

Table 4.4 LaQ variations with the SAHCLayering algorithm applied on JUnit 4.10

			Mean	**StDev**	**Min**
Junit 4.10	**Scenario 1**	***Setup 1***	260.7	21.32	234.0
		Setup 2	298.84	18.91	291.0
		Setup 3	341.46	21.78	330.0
		Setup 4	389.88	42.56	366.0
		Setup 5	435.68	44.02	402.0
	Scenario 3	***Setup 1***	166.52	30.01	148.0
		Setup 2	199.9	17.30	184.0
		Setup 3	238.46	26.63	218.0
		Setup 4	272.2	33.38	247.0
		Setup 5	300.92	58.30	263.0

Table 4.5 LaQ variations with the SAHCLayering algorithm applied on JFreeChart 1.0.15

			Mean	**StDev**	**Min**
JFreeChart 1.0.15	**Scenario 1**	***Setup 1***	1941.96	13.80	1938.0
		Setup 2	2439.58	17.73	2436.0
		Setup 3	2907.3	77.25	2864.0
		Setup 4	3196.14	126.63	3071.0
		Setup 5	3425.3	221.44	3174.0
	Scenario 3	***Setup 1***	1447.04	41.16	1406.0

		Setup 2	1698.6	139.21	1562.0
		Setup 3	1786.7	191.92	1631.0
		Setup 4	1843.62	267.78	1660.0
		Setup 5	1908.82	326.50	1684.0

Table 4.6 LaQ variations with the SAHCLayering algorithm applied on jEdit 5.0.0

			Mean	**StDev**	**Min**
jEdit 5.0.0	**Scenario 1**	***Setup 1***	1349.1	48.88	1288.0
		Setup 2	1718.18	60.77	1631.0
		Setup 3	1952.72	49.26	1849.0
		Setup 4	2029.9	50.82	1973.0
		Setup 5	2143.58	60.48	2097.0
	Scenario 3	***Setup 1***	905.44	9.71	893.0
		Setup 2	1157.68	22.34	1120.0
		Setup 3	1254.58	12.40	1231.0
		Setup 4	1345.68	25.35	1316.0
		Setup 5	1418.16	40.31	1361.0

Table 4.7 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 60b1

			Mean	**StDev**	**Min**
JHotDraw 60b1	**Scenario 1**	***Setup 1***	607.2	37.44	587.0
		Setup 2	643.76	18.15	638.0
		Setup 3	708.86	69.89	682.0
		Setup 4	749.9	103.28	726.0
		Setup 5	783.16	30.54	770.0
	Scenario 3	***Setup 1***	389.94	22.72	383.0
		Setup 2	450.94	74.43	423.0
		Setup 3	469.24	30.59	463.0
		Setup 4	551.0	67.30	503.0

		Setup 5	567.64	24.09	543.0

Table 4.8 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.0.7

			Mean	**StDev**	**Min**
JHotDraw 7.0.7	**Scenario 1**	***Setup 1***	569.44	4.96	565.0
		Setup 2	611.16	23.69	580.0
		Setup 3	629.62	41.62	586.0
		Setup 4	646.2	41.86	590.0
		Setup 5	658.74	50.48	594.0
	Scenario 3	***Setup 1***	427.8	77.96	382.0
		Setup 2	442.1	58.35	398.0
		Setup 3	458.08	52.45	414.0
		Setup 4	465.26	40.37	421.0
		Setup 5	503.96	87.01	424.0

Table 4.9 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.4.1

			Mean	**StDev**	**Min**
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	1938.14	33.52	1910.0
		Setup 2	2298.64	98.81	2120.0
		Setup 3	2465.04	179.89	2199.0
		Setup 4	2590.82	241.33	2255.0
		Setup 5	2730.76	268.15	2333.0
	Scenario 3	***Setup 1***	1209.62	69.75	1176.0
		Setup 2	1312.56	59.72	1259.0
		Setup 3	1397.76	143.75	1307.0
		Setup 4	1395.34	97.20	1319.0
		Setup 5	1612.84	364.21	1331.0

Table 4.10 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.6

			Mean	StDev	Min
JHotDraw 7.6	**Scenario 1**	***Setup 1***	1911.9	82.98	1724.0
		Setup 2	2176.8	126.00	1845.0
		Setup 3	2163.52	166.99	1898.0
		Setup 4	2318.04	258.81	1932.0
		Setup 5	2323.56	234.00	1964.0
	Scenario 3	***Setup 1***	1143.54	49.40	1089.0
		Setup 2	1287.92	73.78	1197.0
		Setup 3	1345.7	106.48	1227.0
		Setup 4	1434.12	130.71	1251.0
		Setup 5	1528.16	307.32	1275.0

Table 4.3 to Table 4.10, show that all the setups of scenarios 1 and 3 (except in some cases for the setup 5 of the scenario 3), yield layering results whose layering quality LaQ's means (*Mean*) are very high compared to their associated standard deviation. This allows us to conclude that the Means of the results obtained with these setups generally reflect the true Mean of the considered distributions. For most of these setups, the *SAHCLayering* algorithm then converges in a consistent manner toward the same solution regarding the layering quality. We assume that the variability in the layering quality of the respective results returned using the setup 5 of the scenario 3 can be solved by modifying the *SAHCLayering* so that it supports an intensified search. The latter will allow further exploring the search space by eventually degrading the current solution of a given iteration.

To get a better insight of the *SAHCLayering* convergence, we were interested in finding out the percentage of optimal solutions, i.e., the percentage of solutions whose layering quality LaQ is identical to *Min*. Table 4.11 reports this percentage for each of the scenarios 1 and 3' setups and for each of the analyzed systems. Table 4.11 also indicates the total weight of package dependencies (i.e quantity of package dependencies) of each system. In this table, S_i

indicates the *i*th setup of a given scenario. For a given system, a grey cell contains the maximal factor of optimal solutions found over all the setups of the scenarios 1 and 3.

Table 4.11 Percentage of the optimal solutions per setup and per analyzed system

	Total weight of package dependencies	**Scenario 1**					**Scenario 3**				
		S_1	S_2	S_3	S_4	S_5	S_1	S_2	S_3	S_4	S_5
Apache Ant 1.6.2	2068	4	52	2	50	26	10	52	50	2	22
JUnit 4.10	356	10	14	4	12	8	26	30	38	38	20
JFreeChart 1.0.15	2313	58	96	44	24	20	34	18	28	30	26
jEdit 5.0.0	1706	24	14	18	28	18	26	4	8	18	12
JHotDraw 60b1	1213	72	86	78	90	82	72	72	82	48	46
JHotDraw 707	972	38	24	32	26	26	24	20	22	16	14
JHotDraw 7.4.1	2624	24	10	6	2	4	28	4	2	4	2
JHotDraw 7.6	2479	2	2	2	8	2	32	2	8	4	2

As shown in Table 4.11, JFreeChart 1.0.15 and JHotDraw 60b1 are the systems for which the percentage of optimal solutions is very high for the considered setups. This indicates that the probability of the hill climbing convergence for both of these systems is very high compared to the ones of the other systems. Table 4.11, also reveals that the highest percentages (in grey cells) of solutions are not related to a specific setup. This allows us to conclude that the optimality of the hill climbing is not tied to a specific setup.

To further study the factors that influence the optimality of the layering solutions, we took a look at the size of the systems at hand. It is from this perspective that we used the Spearman correlation to assess the similarity between the size (i.e., total weight of package dependencies) of the analyzed systems and these systems's percentage of optimal solutions found for a given setup. Table 4.12 indicates the corresponding correlation results (*rho*) computed with the online tool available on (Spearman, 2015).

Table 4.12 Correlation between percentage of optimal solution's quality
and the size of the five analyzed systems for each setup

		rho	***p-value***
Scenario 1	Setup 1	-0.23952	0.56777
	Setup 2	-0.29941	0.47126

	Setup 3	-0.26348	0.52837
	Setup 4	-0.47619	0.23294
	Setup 5	-0.49103	0.21661
Scenario 3	Setup 1	0.29941	0.47126
	Setup 2	-0.62277	0.0991
	Setup 3	-0.56288	0.14633
	Setup 4	-0.58684	0.1262
	Setup 5	-0.44312	0.2715

As indicated in Table 4.12, the correlation values (*rho*) of the five systems range from -0.62277 to 0.29941 with p-values much higher than 5%. Hence, we cannot conclude that the relationship between the sizes of the analyzed systems and these systems's percentage of optimal solutions found for a given setup could be considered statistically significant. In other words, we cannot conclude that the optimality of the hill climbing is significantly related to the size of the analyzed system. This calls for more investigations about the factors influencing the optimality of the hill climbing in our context. We aim at performing such investigation in future works.

4.2.2 Experimental results with tabu search

As stated earlier, we run the tabu search with the maximum number of iterations set to 200 and a tabu list's size set to 10. Table 4.13 to Table 4.20 respectively summarize the layering results obtained by applying *TabuLayering* on the analyzed systems with the setups of the scenarios 1 and 3. The columns of each of these tables report some descriptive statistics of the layering results' quality (LaQ). These statistics correspond to the mean (*Mean*), the standard deviation (*StDev*) and the minimal value (*Min*) of the distribution comprising the layering results' quality obtained by running the algorithm 50 times for a given system and setup. The minimal value of the distribution indicates the quality of the optimal layering solution found among the 50 layering solutions generated for a given setup.

Table 4.13 LaQ variations with the TabuLayering algorithm applied on Apache 1.6.2

			Mean	StDev	Min
Apache Ant 1.6.2	**Scenario 1**	***Setup 1***	1034.42	6.98	1026.0
		Setup 2	1126.08	9.13	1120.0
		Setup 3	1179.5	12.94	1168.0
		Setup 4	1230.38	14.27	1216.0
		Setup 5	1285.7	20.50	1257.0
	Scenario 3	***Setup 1***	570.86	0.85	569.0
		Setup 2	620.62	2.76	617.0
		Setup 3	668.32	13.06	662.0
		Setup 4	741.1	44.42	703.0
		Setup 5	804.72	144.43	737.0

Table 4.14 LaQ variations with the TabuLayering algorithm applied on JUnit 4.10

			Mean	StDev	Min
Junit 4.10	**Scenario 1**	***Setup 1***	259.04	22.17	234.0
		Setup 2	300.88	21.15	291.0
		Setup 3	347.98	30.11	330.0
		Setup 4	381.12	17.49	366.0
		Setup 5	430.94	38.69	402.0
	Scenario 3	***Setup 1***	158.66	20.70	148.0
		Setup 2	199.58	21.15	184.0
		Setup 3	241.9	36.18	218.0
		Setup 4	268.06	29.26	247.0
		Setup 5	293.92	41.68	263.0

Table 4.15 LaQ variations with the TabuLayering algorithm applied on JFreeChart 1.0.15

			Mean	**StDev**	**Min**
JFreeChart 1.0.15	**Scenario 1**	***Setup 1***	1943.78	20.92	1938.0
		Setup 2	2439.28	16.24	2436.0
		Setup 3	2913.06	69.40	2864.0
		Setup 4	3207.84	140.88	3071.0
		Setup 5	3367.8	188.67	3174.0
	Scenario 3	***Setup 1***	1445.1	38.97	1406.0
		Setup 2	1682.3	135.74	1562.0
		Setup 3	1866.94	282.00	1631.0
		Setup 4	1797.6	237.34	1660.0
		Setup 5	1950.64	375.70	1684.0

Table 4.16 LaQ variations with the TabuLayering algorithm applied on jEdit 5.0.0

			Mean	**StDev**	**Min**
jEdit 5.0.0	**Scenario 1**	***Setup 1***	1367.88	49.11	1288.0
		Setup 2	1727.88	66.22	1631.0
		Setup 3	1953.08	49.43	1849.0
		Setup 4	2033.02	26.85	1973.0
		Setup 5	2146.5	73.31	2097.0
	Scenario 3	***Setup 1***	903.66	7.30	893.0
		Setup 2	1152.26	14.51	1120.0
		Setup 3	1253.18	12.12	1231.0
		Setup 4	1341.6	23.28	1316.0
		Setup 5	1399.8	37.51	1361.0

Table 4.17 LaQ variations with the TabuLayering algorithm applied on JHotDraw 60b1

			Mean	StDev	Min
JHotDraw 60b1	**Scenario 1**	***Setup 1***	605.1	34.41	587.0
		Setup 2	647.64	22.14	638.0
		Setup 3	696.78	57.61	682.0
		Setup 4	734.94	23.56	726.0
		Setup 5	797.54	85.56	770.0
	Scenario 3	***Setup 1***	390.14	20.97	383.0
		Setup 2	430.84	34.56	423.0
		Setup 3	483.64	75.79	463.0
		Setup 4	535.78	64.12	503.0
		Setup 5	587.46	111.28	543.0

Table 4.18 LaQ variations with the TabuLayering algorithm applied on JHotDraw 707

			Mean	StDev	Min
JHotDraw 707	**Scenario 1**	***Setup 1***	570.42	13.86	565.0
		Setup 2	615.0	30.32	580.0
		Setup 3	627.9	40.58	586.0
		Setup 4	639.92	46.72	590.0
		Setup 5	648.88	47.64	594.0
	Scenario 3	***Setup 1***	414.84	69.73	382.0
		Setup 2	440.66	58.82	398.0
		Setup 3	451.88	40.07	414.0
		Setup 4	478.56	59.18	421.0
		Setup 5	492.4	58.02	424.0

Table 4.19 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.4.1

			Mean	**StDev**	**Min**
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	1938.06	36.26	1910.0
		Setup 2	2308.64	102.27	2120.0
		Setup 3	2419.52	178.34	2199.0
		Setup 4	2564.96	224.25	2263.0
		Setup 5	2690.22	244.54	2278.0
	Scenario 3	***Setup 1***	1194.16	57.48	1176.0
		Setup 2	1300.14	67.71	1259.0
		Setup 3	1357.84	74.90	1307.0
		Setup 4	1530.76	382.03	1319.0
		Setup 5	1656.96	419.15	1331.0

Table 4.20 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.6

			Mean	**StDev**	**Min**
JHotDraw 7.6	**Scenario 1**	***Setup 1***	1907.6	81.89	1724.0
		Setup 2	2143.56	154.27	1845.0
		Setup 3	2190.5	174.07	1899.0
		Setup 4	2264.62	271.15	1932.0
		Setup 5	2378.22	276.20	1964.0
	Scenario 3	***Setup 1***	1142.72	51.50	1089.0
		Setup 2	1290.88	71.39	1197.0
		Setup 3	1352.34	81.84	1227.0
		Setup 4	1399.6	69.86	1292.0
		Setup 5	1530.2	258.06	1275.0

The results displayed by Table 4.13 to Table 4.20 are very close to those obtained by the SAHCLayering algorithm. Overall, Table 4.13 to Table 4.20 show that all the setups of scenarios 1 and 3 (except in some cases for the setup 5 of the scenario 3), yield layering results whose layering quality LaQ's means (*Mean*) are very high compared to their

associated standard deviation. This allows us to conclude that the Means of the results obtained with these setups generally reflect the true Mean of the considered distributions. For most of these setups, the *TabuLayering* algorithm then converges in a consistent manner toward the same solution regarding the layering quality. The variability in the layering quality of the respective results returned using the setup 5 of the scenario 3 can be solved by a better tuning of the parameters used by the *TabuLayering,* i.e., the size of the tabu list and the maximal number of iterations of the algorithm. This aspect is further discussed in the next subsections.

To study the convergence of the *TabuLayering* algorithm, we computed the percentage of the solutions that are optimal, i.e., whose layering quality LaQ is identical to *Min*. Table 4.21 reports this percentage for each of the scenarios 1 and 3‘ setups and for each of the analyzed systems. This table also reports the size of each system (i.e., Total weight of package dependencies of each system). In this table, S_i indicates the *i*th setup of a given scenario. For a given system, a grey cell contains the maximal factor of optimal solutions found over all the setups of the scenarios 1 and 3.

Table 4.21 Percentage of the optimal solutions per setup and per analyzed system

	Total weight of package dependencies	**Scenario 1**					**Scenario 3**				
		S_1	**S_2**	**S_3**	**S_4**	**S_5**	**S_1**	**S_2**	**S_3**	**S_4**	**S_5**
Apache Ant 1.6.2	2068	8	68	50	42	2	14	2	2	38	2
JUnit 4.10	356	22	14	14	12	2	42	34	42	28	26
JFreeChart 1.0.15	2313	40	96	34	22	24	32	26	26	40	20
jEdit 5.0.0	1706	12	16	18	16	22	26	4	14	22	22
JHotDraw 60b1	1213	62	76	82	84	78	64	84	76	66	42
JhotDraw 707	972	46	24	30	38	34	38	28	18	6	8
JHotDraw 7.4.1	2624	38	8	10	2	2	20	6	2	4	8
JHotDraw 7.6	2479	4	8	6	12	4	36	4	8	2	2

Table 4.21 shows that JFreeChart 1.0.15 and JHotDraw 60b1 are the systems for which the percentage of optimal solutions is very high fort the considered setups. This indicates that the probability of the tabu search algorithm convergence for both of these systems is very high compared to the ones of the other systems. Table 4.21 also indicates that the highest

percentages (in grey cells) of solutions are not related to a specific setup. Hence, similar to the *SAHCLayering* algorithm, the optimality of the *TabuLayering* algorithm is not related to a specific setup.

To further study the factors that influence the optimality of the layering solutions, we took a look at the size of the systems at hand. We used the Spearman correlation to assess the similarity between the size (i.e. number of package dependencies) of the analyzed systems and these systems's percentage of optimal solutions found for a given setup. Table 4.22 shows the corresponding correlation results (*rho*) generated by the statistical online tool available at (Spearman, 2015).

Table 4.22 Correlation between percentage of optimal solution's quality and the size of the systems

		Rho	***p-value***
Scenario 1	Setup 1	-0.35714	0.38512
	Setup 2	-0.27545	0.50905
	Setup 3	-0.35714	0.38512
	Setup 4	-0.40719	0.31671
	Setup 5	-0.29277	0.48162
Scenario 3	Setup 1	-0.64286	0.08556
	Setup 2	-0.58684	0.1262
	Setup 3	-0.68265	0.06209
	Setup 4	-0.38095	0.35181
	Setup 5	-0.57835	0.13314

Table 4.2 indicates that the correlation values (*rho*) between the percentages of optimal solution's quality and the size of the systems are very low: they range from -0.68265 to -0.27545 with p-values much higher than 5%. Therefore, we cannot conclude that the relationship between the sizes of the analyzed systems and these systems's percentages of optimal solutions found for a given setup could be considered statistically significant. In other words, we cannot conclude that the optimality of the *TabuLayering* algorithm is significantly related to the size of the analyzed systems. This also calls for more

investigations about the factors influencing the optimality of the *TabuLayering* algorithm in our context. We aim at performing such investigation in future works.

4.2.2.1 Analysis of the influence of the number of iterations

To study the influence of the maximal number of iterations (*Max_It*) on the *TabuLayering* algorithm results, we vary this number from 100 to 500 with a step of 100 while keeping the tabu list size to 10. For each considered scenario (i.e., scenarios 1 and 3), for each setup of these scenarios, and for each value of the maximal number of iterations (*Max_It*), we then run the *TabuLayering* algorithm 50 independent times. The so-obtained results show that for each analyzed system, for a given scenario and for a given setup, each maximal number of iterations (*Max_It*) picked in {100, 200, 300, 400, 500} generally leads to the same best value of LaQ. This trend is always true for the following five systems: JUnit, JHotDraw 60b1, JHotDraw 707, JFreeChart 1.0.15 and JEdit 5.0.0. For these five systems, setting the maximal number of iterations (*Max_It*) to 100 is therefore enough to find the solution yielding the best value of LaQ. Interestingly, apart from JFreeChart 1.0.15, these systems have a small size in terms of package dependencies.

For the bigger systems that are Apache 1.6.2, JHotDraw 7.4.1 and JHotDraw 7.6, there is a few cases which do no consolidate this trend. For Apache 1.6.2 setting *Max_It* to 200 always generates the best value of LaQ for a given setup (i.e., the best LaQ over all the LaQ of the solutions generated when running the *TabuLayering* with *Max_It* varying from 100 to 500). Besides, as expected, when *Max_It* is goes from 100 to 200, the value of LaQ usually improves to reach the best LaQ value found for the setup. Contrariwise, when *Max_it* goes from 200 to 400, the best value of LaQ either remains constant or deteriorates. Finally, when *Max_It* goes from 400 to 500, the best value of LaQ either remains constant or improves to reach the best LaQ value for the setup (found for *Max_It*=200), except for the setup 1 of scenario 1. In the latter, the value of LaQ deteriorates from 1026 to 1031 when *Max_it* goes from 400 to 500.

For JHotDraw 7.4.1 and JHotDraw 7.6, setting *Max_It* to 300 always generates the best value of LaQ for a given setup. As with Apache 1.6.2, other values of *Max_It* can keep the best LaQ stable, or, in a few cases, deteriorates it. In particular, in JHotDraw 7.4.1 varying *Max_It* from 100 to 200 usually allows finding the best LaQ value for the setup as *Max_It*=300 does. But, when *Max_It* goes from 300 to 500, the best value of LaQ either remains constant or deteriorates in a very few cases. It is the case for the setup 4 of scenario 3, where the best value of LaQ is the same (i.e., equals to 1319) when *Max_It* varies from 100 to 400. However, when *Max_It* is set to 500, the value of LaQ deteriorates by going from 1319 to 1321. This possible deterioration of the LaQ value when *Max_It* goes from 300 to 500 also occurred in rare cases with JHotDraw 7.6. But we did not expected a deterioration of LaQ for these higher values of *Max_It* since increasing the number of iterations gives much time to the *TabuLayering* to look for the optimal solution. We assume that this discrepancy is due to the fact that, since Apache 1.6.2, JHotDraw 7.4.1 and JHotDraw 7.6 contain more package dependencies than the other analyzed systems, setting the tabu list size to the same small value used for the smaller systems (i.e., 10) has sometimes driven the *TabuLayering* to cycle within an area comprising sub-optimal solutions.We therefore expect that increasing the size of the tabu list might fix the possible degradation of the LaQ issue encountered by bigger systems.

4.2.2.2 Analysis of the influence of the tabu list

To study the influence of the tabu list's size (*TLS*) on the optimality of the tabu search, we vary this size from 10 to 50 with a step of 10, while keeping the maximal number of iterations to 200. For each considered scenario (i.e., scenarios 1 and 3), for each setup, and for each value of the maximal number of iterations (*TLS*), we then run the *TabuLayering* algorithm 50 independent times. The corresponding layering results show that for each of the analyzed systems, for a given scenario and for a given setup of this scenario, a value of *TLS* chosen in {10, 20, 30, 40, 50} is generally able to find the same best value for LaQ for the so-called setup. This observation is always true for the following 4 of the 8 analyzed systems, namely: JHotDraw 60b1, JHotDraw 707, JFreeChart 1.0.15 and JEdit 5.0.0. For these four

systems, setting the value of *TLS* to 10 is therefore enough to find the best value of LaQ for a given setup (i.e., the best value of LaQ over all the LaQ of the solutions generated when varying *TLS* from 10 to 50 and when using the so-called setup). This conclusion can be explained by the fact that, apart from JFreeChart 1.0.15, these systems have a small size in terms of package dependencies. A small value of *TLS* therefore allows an efficient exploration of the search space.

However, for the other systems that are JUnit 4.10, Apache 1.6.2, JHotDraw 7.4.1 and JHotDraw 7.6, there are a few cases which do no consolidate this trend i.e. for which setting *TLS* to 10 does not lead to the best value of LaQ found for the setup. In particular, for JUnit, there is a single case for which the *TabuLayering* does not follow this observation, namely: for the setup 5 of the scenario 1. For this setup, all the values of *TLS* lead to the same best LaQ value (i.e., 402), except for *TLS*=20 which leads to a less optimal solution with a LaQ of 403. We assume that this discrepancy is purely random since JUnit is a small system and that values of *TLS* that are lower or higher than 20 all lead to the same LaQ value. For the bigger systems Apache 1.6.2, JHotDraw 7.4.1 and JHotDraw 7.6, increasing the value of *TLS* until it reaches a given threshold allows improving the value of LaQ. Beyond this threshold, the best value of LaQ either remains the same or deteriorates. In most cases, this threshold is 20.

For instance, when generating the Apache 1.6.2's solutions for the setup 4 of the scenario 3, the *TabuLayering* finds 703 as the best value of LaQ, with *TLS*=10. *TabuLayering* improves the best value of LaQ from 703 to 701 with *TLS*=20. The value of LaQ then remains the same (i.e. equals to 701) when *TLS* goes from 20 to 30. However, when *TLS* is set to 40 or 50, *TabuLayering* yields a less optimal value of LaQ which deteriorates by going from 701 to 703 for these two values of *TLS*. Likewise, when applied on JHotDraw 7.6 with the setup 5 of the scenario 2 and *TLS* set to 10, the *TabuLayering* finds 1300 as the best value of LaQ. With *TLS* set to 20, *TabuLayering* then improves the best value of LaQ which goes from 1300 to 1275. However, with *TLS* set to 30, the value of LaQ deteriorates by going from 1275 to 1300 and then remains to 1275 when *TLS* is set to 40 or 50. This allows us to

conclude that for bigger systems, setting the size of the tabu list (*TLS*) to 20 is therefore sufficient to find an optimal solution.

4.2.3 Comparison of the two layering algorithms

The averages and standard deviation of LaQ respectively found with the *SAHCLayering* and the *TabuLayering* (calibrated with a maximal number of iterations set to 200 and a tabu list size set to 10) algorithms are very close. As such, these statistics do not provide enough discriminative support to compare the two algorithms. To compare the *SAHCLayering* and the *TabuLayering* algorithms, we therefore start by assessing their ability to reach the optimal quality LaQ for a given setup. Table 4.23 indicates for each analyzed system, the number of setups for which the *SAHCLayering* and the *TabuLayering* algorithms led to the identical optimal LaQ values (*NSIO*). Table 4.23 also reports the number of setups for which the *SAHCLayering* was able to find a better optimal quality LaQ than *TabuLayering* or the other way around (*NSBO*). In this context, the total number of setups that we considered is 10, i.e. the 10 setups of both scenarios 1 and 3.

Table 4.23 The identicality of the optimal quality LaQ

	NSIO	**NSBO**	
		Hill Climbing	**Tabu search**
Apache Ant 1.6.2	4	2	4
JUnit 4.10	10	0	0
JFreeChart 1.0.15	10	0	0
jEdit 5.0.0	10	0	0
JHotDraw 60b1	10	0	0
JHotDraw 7.0.7	10	0	0
JhotDraw 7.4.1	8	1	1
JHotDraw 7.6	8	2	0
Total	70	5	5

As shown in Table 4.23, the sum of NSIO is 70; i.e., out of the 80 optimal values of LaQ that the *SAHCLayering* and the *TabuLayering* have both found for each of the eight analyzed

systems and each of the 10 setups of scenarios 1 and 3, the two algorithms have both found 70 identical optimal LaQ. It means that, the *SAHCLayering* and the *TabuLayering* algorithms find identical optimal solutions for almost all of the setups and for almost all of the analyzed systems. In particular, given each of the ten setups of both scenarios 1 and 3, these two algorithms both find an identical optimal quality LaQ for 5 out of eight systems, i.e. for JUnit 4.10, JFreeChart 1.0.15, jEdit 5.0.0, JHotDraw 60b1 and JHotDraw 7.0.7. For eight of the ten setups of both scenarios 1 and 3, the two algorithms both find an identical optimal quality LaQ for two out of eight systems, namely JHotDraw 7.4.1 and JHotDraw 7.6. For Apache 1.6.2, it is only four of the ten setups of scenarios 1 and 3 that allow generating an identical optimal solution's quality by both the hill climbing and the tabu search algorithms.

As shown in Table 4.23, the NSBO value is 5 for both the *TabuLayering* and the *SAHCLayering* algorithms; i.e., out of the 80 optimal quality LaQ that the tabu search and the hill climbing both have found, there is 5 cases for which the *TabuLayering* was able to find a better optimal LaQ than the *SAHCLayering* and conversely. Regarding the ability to find the optimal quality LaQ, the *TabuLayering* and the *SAHCLayering* are therefore even.

To further investigate the optimality of each of these two algorithms, we took a closer look to the setups for which the two algorithms led to the identical optimal quality LaQ. And for these setups, we compared the percentage of the optimal solutions per setup and per analyzed system respectively yielded by the two algorithms. These percentages are reported in Table 4.11 and Table 4.21. The results reported in these tables show than the *TabuLayering* is able to generate a slightly higher percentage of optimal quality LaQ per setup and per analyzed system than the *SAHCLayering*. However, when comparing each the two algorithms' running times (see APPENDIX I, p. 187), we noticed that, in order to find the optimal quality LaQ, the *TabuLayering* algorithm takes 3 to 4 times longer than the *SAHCLayering* algorithm. For instance, the mean time required by the *TabuLayering* algorithms to perform each of the 50 runs on JHotDraw 7.6 using setup 1 of scenario 1 is 17 seconds. This time is reduced to 5 seconds with the *SAHCLayering*.

These observations allow us to conclude that the *SAHCLayering* is a better compromise than the *TabuLayering* in terms of optimality and running times. This conforms to the findings of other studies that compare Hill climbing to other search-based methods (e.g., (Harman et al., 2002; Mitchell et al., 2002b)). We will therefore rely on the *SAHCLayering* to generate layering results so as to answer the remaining experimentation questions.

4.3 EQ2: What are the values of factors (ap, ip, sp and bp) that best correspond to the common understanding of the layered style?

To answer EQ2, we run the *SAHCLayering* for the setups of the scenarios 1 and 3. In particular, we vary the factor *bp* associated to the back-calls from 1 to 20 by step of 1, while keeping the same values for the other factors *ap*, *ip* and *sp*. For each of the so-obtained setups, we run the *SAHCLayering* algorithm 50 times and keep the solution having the lowest quality LaQ as the best solution.

For the sake of brevity, we only discuss the most relevant setups and we limit our analysis to four versions of JHotDraw. Besides, since we did not have an authoritative decomposition of jEdit, we have eliminated this system from the following analysis. Table 4.24 summarizes these results executing our layering algorithm on the four analyzed systems using 5 setups: Setup 1= (ap=0, ip=1, sp=2, bp=4), Setup 2'= (ap=0, ip=1, sp=2, bp=15), Setup 3= (ap=0, ip=2, sp=1, bp=4), Setup 4'= (ap=0, ip=2, sp=1, bp=15), and Setup 5= (ap=0, ip=2, sp=1, bp=20). Setup 1 and 2 are both setups of scenario 3 while setups 3, 4' and 5 are setups of scenario 1. The first column of Table 4.24 indicates for each setup the values of the factors. For each solution returned by the algorithm, Table 4.24 displays: 1) the layering quality (LaQ); 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. Cells that are greyed in Table 4.24 correspond to the solutions with the highest F-Measure.

Table 4.24 Best results returned by the *SAHCLayering* recovery algorithm

Setup		Ant	JFreeChart 1015	JUnit	JHD.60b1	JHD.707	JHD.741	JHD.76
Setup 1 ap=0, ip=1, sp=2, bp=4	LaQ	569.0	1406.0	148.0	383.0	382.0	1176.0	1089.0
	NL	3	3	3	3	3	3	3
	Adj	1535	1041	235	864	611	1542	1522
	Intra	521	1218	112	335	348	1044	9093
	Skip	0	14	0	4	9	10	6
	Back	12	40	9	10	4	28	42
	F-measure	74	55.41	42	58	29	22	21
Setup 2' ap=0, ip=1, sp=2, bp=15	LaQ	694.0	1653.0	242.0	493.0	426.0	1316.0	1245.0
	NL	4	3	3	3	3	3	3
	Adj	1500	772	226	864	611	1362	1325
	Intra	559	1520	122	335	348	1247	1141
	Skip	0	14	0	4	9	12	7
	Back	9	7	8	10	4	3	6
	F-measure	76	60.27	39	58	29	12	12
Setup 3 ap=0, ip=2 sp=1, bp=4	LaQ	1026.0	1938.0	234.0	587.0	565.0	1910.0	1724.0
	NL	4	3	3	3	4	4	5
	Adj	1567	1373	253	887	655	1622	1581
	Intra	417	572	83	213	218	455	532
	Skip	48	226	4	97	89	396	268
	Back	36	142	16	16	10	151	98
	F-measure	59	46.67	50	82	83	51	60.0
Setup 4' ap=0, ip=2, sp=1, bp=15	LaQ	1204.0	3033.0	357.0	715.0	589.0	2247.0	1924.0
	NL	4	3	4	3	4	4	4
	Adj	1527	1024	234	891	632	1485	1446
	Intra	495	1212	109	239	235	884	779
	Skip	34	39	4	72	104	239	246
	Back	12	38	9	11	1	16	8
	F-measure	67	58.33	53	76	87	69	67
Setup 5 ap=0, ip=2, sp=1, bp=20	LaQ	1262.0	3174.0	402.0	770.0	594.0	2333.0	1964.0
	NL	4	3	4	3	4	5	4
	Adj	1518	779	225	891	632	1480	1446
	Intra	503	1488	119	239	235	885	779
	Skip	36	38	4	72	104	243	46
	Back	11	8	8	11	1	16	8
	F-measure	67	66.10	57	76	87	69	67

As shown by Table 4.24, for JFreeChart, we obtained the best match with the authoritative architecture using setup 5 of scenario 1. JFreeCHart contains several subsystems that are composed 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 factor *bp* is set to a very high value compared to the intra-dependencies factor *ip* (e.g., setup 5).

For Apache Ant, the layering solution that best matches the actual layering of the system is returned using Setup 2 of scenario 3. In general, the most accurate results are produced by our algorithm for this system when using setups where the intra-dependencies factor *ip* is less than the skip-calls factor *sp* (e.g., Setups 1 and 2). This means that the designers of this system have favored intra-dependencies over skip-calls and back-calls. This is consistent with the fact that Apache Ant is a framework that targets different platforms and, thus, portability is one of the concerns that drive its design. Table 4.24 shows that the best setup for JUnit is the setup 5 since it leads to a F-Measure of 57%. However, just as well as Apache, JUnit is a framework designed with portability in mind. We therefore expected that the setups of scenario 3 (e.g., setups 1 or 2) would be the ones yielding the layerings that best match the actual layering of this system. This discrepancy might be caused by the fact that the *SAHCLayering* goes through local optima with some of the setups when applied to Junit.

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 4.24, this is the case for JHotDraw 7.0.7, 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 7.0.7 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 rules (i.e., $sp < ip < bp$).

Based on these observations and 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 undesired dependencies (e.g., back-calls violations) found in each system. To do this, for each solution that best matches the system's architecture (greyed cells in Table 4.24), 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. Figure 4.1 displays the dependencies by type for the best matched solution of each system. JFreeChart have the highest percentage of intra-dependencies (64.33%) relative to the other dependencies.

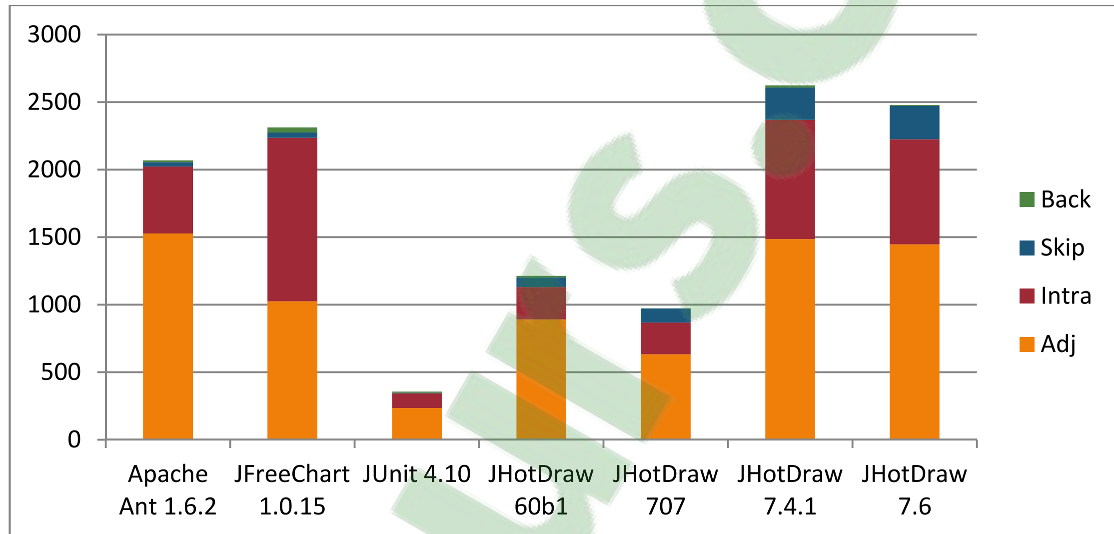


Figure 4.1 Density of dependencies by type for the best matched solution of each system

JHotDraw 6.0.b1 has the lowest percentage of intra-dependencies (17.55%) while JHotDraw 7.0.7 has the lowest percentage of back-calls (0.10%). Figure 4.1 shows that, for the four versions of JHotDraw, the average of the density of undesired dependencies (e.g., back-calls violations) relative to the system size is smaller than the density of undesired dependencies in the two of the three systems (i.e., JUnit and JFreeChart). These findings confirm our hypothesis which is consistent 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. This should be investigated more in future works.

Prior to these experiments, we assumed that the values of the layers' dependency attributes would remain the same when a given threshold of the back-call factor *bp* would be reached, since the more *bp* is high, the more the algorithm tends to aggregate in the same layer the packages involved in cyclic dependencies. So, if a high value of *bp* allows assigning to the

Clicours.com

same layer all the packages involved in cyclic dependencies, then increasing *bp* beyond that value will lead to the same result. Therefore, finding a threshold of *bp* beyond which the layering results remain the same will narrow the range of values that the designer can assign to the factors in order to produce good layering results. Further experiments on nine versions of JHotDraw showed that this threshold is inferior to 28 for this system. However, finding this threshold is no more a primary concern, since the experiments showed that Setup 4' = (*ap*=0, *ip*=2, *sp*=1, *bp*=15) of scenario 1 is the setup that generally produces the best layerings for all the analyzed versions of JHotDraw except one (version 60b1).

4.4 EQ3: How do the layering results evolve across evolutions of a software system and what does it tell 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 similarity of the set of values of the setups factor that yield the layering that matches the known architecture of the system across its evolutions. To investigate these aspects, we study the layering results obtained when analyzing nine versions of JHotDraw using the setup that best corresponds to the common understanding of the layered style, i.e. Setup 4'= (ap=0, ip=2, sp=1, bp=15) of scenario 1 (Section 4.3). Since this setup leads to 50 layering solutions (corresponding to the 50 runs made), in the following, we focus on the best layering solution among these ones; i.e.; the solution that yields the best value of the layering quality LaQ.

4.4.1 Stability of the algorithm

We aim here at investigating if our algorithm produces the same layering for two consecutive versions of the same system when using the same set factor values and giving that there are limited changes between the two analyzed versions. To do so, we compared the layering results for consecutive versions of JHotDraw.

JHotDraw is a Java framework that supports the design of drawing editors. Each of the first versions of JHotDraw including versions 60b1, 7.0.6, 7.0.9, 7.2, 7.3 and 7.4 introduces

substantial changes to the framework (Randelshofer, 2015). These changes include the migration of the framework to support new versions of Java, the introduction of sample programs showing the use of the framework, the reorganization of the framework structure by adding, removing or modifying several packages and classes, the addition of default implementations for some specific functionality of the framework and the redesign of some specific classes. More information on the changes and differences between JHotDraw versions can be found in (Randelshofer, 2015). For example, the evolution of JHotDraw from version 60b1 to version 7.0.1 involved the removal of 14 packages (e.g., org.jhotdraw.applet, org.jhotdraw.contrib) out of 17 packages (i.e., we consider sample programs as part of the framework but we discard the packages containing the test cases) and the addition of 35 packages (e.g., org.jhotdraw.io, org.jhotdraw.xml, net.n3.nanoXML). The changes from version 7.1 to version 7.2 included adding 8 new packages and modifying several packages. Actually, the number of packages increased from 17 for version 60b1 to 24 for version 7.0.6 to 36 for version 7.0.9 to 46 for versions 7.2 and 7.3 to 62 for version 7.4.1 and to 65 for versions 7.5.1 and 7.6. Although the packages from version 7.2 to version 7.3 were the same, substantial modifications were made to the classes and several new classes were added to some packages (e.g., org.jhotdraw.gui.datatransfer) while some classes were moved from a package to another (e.g., three classes of the package org.jhotdraw.gui were moved to the package org.jhotdraw.gui.event).

Accordingly, we limited our analysis of the stability of our layering results to the three last versions of JHotDraw (i.e., 7.4.1, 7.5.1 and 7.6) as they did not introduce major revisions when we compare one version to its previous one. Table 4.25 and Table 4.26 respectively display the results of the comparison of the layerings for versions 7.4.1 and 7.5.1, and the comparison of the layerings for versions 7.5.1 and 7.6. In particular, the first column of these tables shows the layer number, the second and third columns indicate the number of packages per layer for each version and the last column shows the number of common packages to the two versions that are assigned to the same layer. For example, version 7.4.1 has 62 packages while version 7.5.1 has 65 packages; i.e. version 7.5.1 has three new added packages when compared to version 7.4.1. As displayed by Table 4.25, 56 packages among

the 62 common packages of these two versions are assigned to the same layers in our layering results. This corresponds to a ratio of 90.32% which is a very high ratio considering that 29 packages of the 62 common packages were redesigned by adding, removing or modifying their classes. Similarly, versions 7.5.1 and 7.6 have 65 packages among which 63 packages are common to the two versions.

Table 4.25 Comparison of the packages per layer for versions 7.4.1 and 7.5.1

Layer	**Number of assigned packages for JHotDraw 7.4.1**	**Number of assigned packages for JHotDraw 7.5.1**	**Common packages per layer**
4	23	22	21
3	31	29	27
2	7	13	7
1	1	1	1
Total	62	65	56

Table 4.26 shows that 53 packages among these 63 common packages are assigned to the same layers. This corresponds to a ratio of 84.12% which is again a very high score knowing that 20 packages out of the 63 common packages were redesigned by modifying, removing or adding some classes. These results confirm the stability of our layering approach when limited changes have been introduced from a version to a subsequent one.

Table 4.26 Comparison of the packages per layer for versions 7.5.1 and 7.6

Layer	**Number of assigned packages per layer for JHotDraw 7.5.1**	**Number of assigned packages per layer for JHotDraw 7.6**	**Common packages per layer**
4	22	18	18
3	29	31	25
2	13	13	9
1	1	3	1
Total	65	65	53

4.4.2 Similarity of the set of values of the setups factor that yield the layering that matches the known architecture of the system across its revisions

In addition to the assessment of the stability of our approach, we were interested in investigating the similarity of the set factor values that the designer implicitly applies for a given system. In other words, we wanted to check if the degree of compliance with the layered style remains the same through the evolution of the design of a given system. Obviously, we expected that the values of the layers' dependency attributes (Adj, Skip, Intra, Back) and the layering quality LaQ increase with the size (i.e the sum of the package dependencies[20]) of the analyzed system. However, we hypothesized that there should be some similarity between the growth of these values and the growth of the size of the system if the designer kept the same level of conformity to the layered constraints. To investigate this hypothesis, we analyzed the results obtained for 9 versions of JHotDraw.

Table 4.27 displays the size of each analyzed version in terms of the total weight of package dependencies and the values of the layers' dependency attributes and the LaQ that were obtained using the Setup 4'= (*ap*=0, *ip*=2, *sp*=1, *bp*=15) of scenario 1. We used the Spearman correlation coefficient to assess if the relationship between the resulting values and the size of the system can be described using a monotonic function.

Table 4.27 JHotDraw Total weight dependencies and the layers' dependency attributes values obtained using setup 4' of the scenario 1

JHotDraw versions	Total weight of package dependencies	Adjac	Intra	Skip	Back	LaQ
JHotDraw 60b1	1213	891	239	72	11	715.0
JHotDraw 7.0.6	988	647	234	106	1	589.0
JHotDraw 7.0.7	972	634	237	100	1	589.0
JHotDraw 7.0.9	1510	1036	318	150	6	876.0

[20] Recall from Section 2.3 that we derive the dependency from a package P1 to a package P2 is from the relationships, i.e., dependencies between their respective entities. The weight of that package dependency is obtained by summing the weights of the dependencies directed from the entities of package P1 to those of package P2.

JHotDraw 7.2	1958	1315	410	226	7	1151.0
JHotDraw 7.3	2068	1373	455	237	3	1192.0
JHotDraw 7.4.1	2624	1485	884	239	16	2247.0
JHotDraw 7.5.1	2718	1517	971	221	10	2313.0
JHotDraw 7.6	2479	1446	779	246	8	1924.0

We computed the spearman's correlation coefficient *rho* for each measure with the total weight of package dependencies as the size of the system (see Table 4.28). The results for the adjacency (Adjac) and intra-dependency metrics (Intra) yielded high correlation values (*rho* =1 and 0.98 respectively) with a p-value less than 5% (p-value = 0). Even though the correlation between the size of the system and the Skip metric is less strong, it is still statistically significant. In fact, the correlation with the Skip metric is 0.76 with a p-value of 0.015. The correlation results with these three metrics indicate that the relationship between each of these three layering metrics and the size of the system is an increasing monotonic function. We can therefore conclude that there is a relationship between the size of the system and these three layering metrics. However, for the Back metric, the correlation is 0.66 with a p-value higher than 5% (i.e. 0.052). Hence, we couldn't conclude that the relationship between this metric and the size of the system could be considered statistically significant. In other words, we cannot conclude that the number of back-calls violations (Back) is significantly related to the size of the system.

Table 4.28 Correlation values with the size of the system

	Rho	***p-value***
ADJAC	1	0
INTRA	0.98	0
SKIP	0.76	0.015
BACK	0.66	0.52
LaQ	0.99	0

On the other hand, for the layering quality (LaQ) values and the size of the system, the value of spearman's correlation coefficient is 0.99 with a p-value less than 5% (i.e., the p-value equals 0). These results confirm that the relationship between LaQ and the size of the system

is an increasing monotonic function and that it can be considered statistically significant. Hence the correlation indicates that JHotDraw maintains the same overall level of conformity to the layering constraints through its evolution.

4.5 EQ4: is the layering approach performant regarding the size of the system at hand?

To study the performance of our approach, we analyze different versions of JHotDraw. We focus on the running times required to generate the layering results obtained with the Setup 4’= (ap=0, ip=2, sp=1, bp=15) of scenario 1 since, as concluded above (see Section 4.4.2), it is the same setup to which JHotDraw remained faithful over all its versions. Table 4.29 indicates the means (Mean), the standard deviation (StDev) as well as the maximal (Max) running times for each of the versions of JHotDraw over the 50 runs performed for the Setup 4’. This table also reports the running time required to generate the best layering solution (Best) and the total weight of dependencies of each of these versions.

As shown in Table 4.29, all the maximal executions time are below 11 seconds and most of the running times are under 3 seconds, even though the *SAHCLayering* assesses all the neighboring solutions at each of its iterations. This indicates that the *SAHCLayering* algorithm is performant.

However, Table 4.29 also indicates that, for each version of JHotDraw, the execution time can sometimes vary much from an execution to another (see the ***StDev*** column). This is due to the fact that the number of layers in the solutions generated by each run generally varies from 3 to 4 layers in the smaller versions of JHotDraw, and from 4 to 6 layers in the bigger versions of JHotDraw. Besides, the bigger versions of JHotDraw comprise more packages than the smaller ones. Therefore, the number of neighbors assessed by each run varies accordingly, since this number is tight to the number of layers of the solution as well as its number of packages. This explains why, for JHotDraw, the *SAHCLayering* does not always converge in a consistent timely manner.

Table 4.29 Execution times for all the versions of JHotDraw using Setup 4’ of scenario 1

	Total weight of package dependencies	***Execution times (ms)***			
		Mean	***StDev***	***Max***	***Best***
JHotDraw 60b1	1213	81.96	16.39	116.0	78
JHotDraw 7.0.6	988	220.02	20.55	302.0	218
JhotDraw 7.0.7	972	235.78	30.79	308.0	219
JHotDraw 7.0.9	1510	1079.74	155.70	1479.0	1031
JHotDraw 7.2	1958	1656.02	262.10	2185.0	1515
JHotDraw 7.3	2068	1909.62	317.92	2891.0	1699
JHotDraw 7.4.1	2624	5897.1	1322.04	9099.0	5321
JHotDraw 7.5.1	2718	6729.2	1399.13	10430.0	6657
JHotDraw 7.6	2479	6780.0	1333.30	10202.0	5461

To study the evolution of the performance of our algorithm as a system grows, we analyzed the impact of the growth of the size of a system over the execution time. We used the Spearman correlation coefficient to assess whether we can use a monotonic function to describe the relationship between the time required to generate the best layering solution (*Best*) and the size of the system. The correlation results for the execution time yields a high positive correlation values (i.e. *rho* =0.91) with a p-value less than 5% (i.e. p-value = 0.0005). This indicates that the relationship between the execution time of the layering algorithm and the size of the system is an increasing monotonic function.

The size of any analyzed system is always bounded and the execution time of the run leading to the best layering solution reported in Table 4.29 is low (i.e. within a couple of seconds). We can then expect that the execution time of our algorithm will be bounded and reasonable for systems much bigger than the analyzed versions of JHotDraw. This positively anwers our fourth experimentation question (EQ4).

4.6 EQ5: Is the approach more authoritative than other architecture recovery approaches?

To check whether our structural-based recovery approach outperforms other recovery approaches, we have compared our results with those generated using two techniques

implemented by ARCADE (Le et al., 2015), namely: ACDC and ARC. Since ARC and ACDC are clustering techniques and not layering techniques, we have combined each of these two algorithms with our layering technique relying on the *SAHCLayering* algorithm (see Chapter 2). We refer to the resulting techniques as ACDC_L and ARC_L. ACDC and ARC work at the class level, so do ACDC_L and ARC_L. Therefore, to compute the precision, recall and F-Measure of the results generated using ACDC_L and ARC_L, we have used a heuristic that consists in considering that a package *p* has been properly assigned by either of the two techniques to a given layer *L*, if *L* is the layer to which the maximum of classes of *p* has been assigned. Note that our choice to combine ACDC and ARC with a layering technique is motivated by that these two clustering techniques usually generates a number of clusters that is far more higher than the number of packages in the systems at hand. For instance, in the case of Apache Ant for instance, ARC generates 161 clusters while Apache Ant contains 67 packages only.

Unlike us, ARC and ACDC work with dependencies between the entities comprised in the analyzed system as well as the external library used by this system. Hence, when running ACDC_L and ARC_L, we have ignored the dependencies between the analyzed system's entities and the external libraries' entities to work on an even basis. We have run *SAHCLayering*, ACDC_L and ARC_L 50 times using the following setups: Setup 1= (ap=0, ip=1, sp=2, bp=4), Setup 2= (ap=0, ip=2, sp=1, bp=4) and Setup 3= (ap=0, ip=2, sp=1, bp=8). We have performed these experiments on the following systems: Apache, JHotDraw 707, JUnit 4.10 and JFreechart 1.0.15. Overall, these experiments show that our layering technique outperforms the two other layering techniques that are ACDC_L and ARC_L. In other words, our approach outperforms ACDC and ARC combined with the layering. For instance, using Setup 2, the best solution' F-measure value for JHotDraw 707 is 83 with our approach while it is 29 and 16 for ACDC_L and ARC_L respectively. Likewise, using Setup 1, the best solution' F-measure value for Apache is 74 with our approach while it is 47 and 58 for ACDC_L and ARC_L respectively.

One of the reasons explaining why the results of ARC_L and ACDC_L are less good than our approach's ones might lie in the fact that they work at the class-level and not at the package-level as our layering approach does. Indeed, when working at the package-level, the advantage is that each package is already (by default) a cluster of the various responsibilities found in the classes it contains. As a package is assigned to a layer during the layering recovery process, all the classes it contains are therefore assigned to the same layer. On the other hand, when working at the class-level, the classes in the same package are often scattered through different layers to promote adjacencies, at the expense of the cohesion of the package. And even if, prior to the layering, we perform a clustering step either with ARC or ACDC in order to group these classes into clusters, the granularity of the resulting clusters is often much finer than that of the analyzed system's packages (recall that in the case of Apache Ant, ARC generates 161 clusters while Apache Ant contains 67 packages). The general recommendation that emerges from this observation is therefore that when analysing OO systems structured into packages, we should rather perform the layering at the package level instead of the class-level. And if there is no certainty that the packages of the analyzed system are cohesive, before making the layering, we could first remodularize the analyzed system's packages using a technique such as Bavota et al's (Bavota et al., 2013).

This positively answers our fifth experimentation question. However, we felt that this comparison is unfair to these two algorithms as they do not specifically target layered architectures. So, in the next experiments (see Chapter 6), we have tried to find more appropriate approaches for comparison; i.e. approaches that explicitly tackle the layering recovery problem.

4.7 Threats to validity

In this section, we discuss some factors that pose threats to the generalization of our approach.

Conclusion validity: To find out which setups return the most authoritative layered solutions, we compared these solutions to authoritative architectures which come in part from our manual work (Section 4.3). This issue is related to the lack of comparison baselines in the software architecture community. However, this choice was based on our experience and knowledge of the analyzed systems.

Internal validity: being trapped in local optima when running the tabu search or the hill climbing algorithms is another threat to the validity of our work. Indeed, depending on the initial partition—that is generated randomly- the resulting layering solution may vary. Besides, 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, both hill climbing and tabu search cannot guarantee a global optimal solution. However, further experiments carried out in the context of (Boaye Belle et al., 2015) through an exhaustive search confirmed that for the smallest system (JHotDraw 60b1), the two algorithms were 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.

To mitigate this threat, we have to run the algorithms 50 times and kept the solution with the lowest quality as the best solution. However, when running a stochastic algorithm, Arcuri et al. (Arcuri and Briand, 2011) recommend performing a very high number of runs (i.e. thousands of runs) if it is not computationnaly expensive. Nevertheless, when running the *SAHCLayering* 50 times for each of the 5 scenarios' setups (see Section 4.1.3), the total running time of the nine versions of JHotDraw was 26652598 milliseconds (approximatively seven hours and a half). Since these runs were computationnaly expensive, we therefore estimate that 50 runs per setup is a reasonable number of runs in our context. Besides, as pointed out by Harman et al (Harman et al., 2012), the experiments performed using stochastics algorithms are usually repeated 30 to 50 times. This is notably the case with the experiments conducted by Mitchell et al. (Mitchell et al., 2008), and Abdeen et al. (Abdeen et al., 2009).

Another threat to the internal validity of our work is the one concerning the computation of the neighboring solutions. When computing the neighbors of a given solution, we discard the neighbors having less than 3 layers in order to ensure that the algorithm does not generate a solution with very few layers (i.e., one or two layers) and for which the layering quality (LaQ) and the number of back-calls and skip-calls are obviously quite low. The so-obtained solution is poorly structured, since it is close to a monolithic architecture. However, discarding these neighbors might lead to a local optimum. To overcome these issues, we plan to adapt another heuristic-based approach such as the simulated annealing (Kirkpatrick et al., 1983) which avoids being trapped in local minima by tolerating the degradation of a solution according to a given probability.

External validity: The experiment has been conducted on a sample of open source Java systems. While all these systems are known to be layered systems, the observed results may not be generalizable to other systems. 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.

4.8 Chapter summary

In this chapter, we have performed different experiments on five systems so as to assess our layering approach described in CHAPTER 2. To this extent, we have described the experimental setup and implementation. We have also reported and analyzed the results respectively obtained when using the two layering algorithms (i.e. the *SAHCLayering* and the *TabuLayering* algorithms) described in CHAPTER 2 to recover the layering of the analyzed systems. These analyses showed that both the *SAHCLayering* and the *TabuLayering* algorithms are able to find optimal solutions. However, the *SAHCLayering* offers a better compromise in terms of running times. Further experiments carried out with the *SAHCLayering* showed that this algorithm is authoritative, stable and performant.

CHAPTER 5

LEXICAL AND STRUCTURAL-BASED LAYERING RECOVERY APPROACH

Though the structural-based recovery approach yielded promising results, we conjectured that exploiting additional system's information would enhance these results. In particular, lexical information embedded in the source code vocabulary may help grouping togother packages that participate to the same responsibility before assigning them to layers. Indeed, the lexical information embedded within a software's lexicon is a valuable source of information that enriches the software analysis (Kuhn et al., 2015; Poshyvanyk et al., 2009; Bavota et al., 2013). We therefore present in this chapter a hybrid approach which exploits both the structural and lexical information to recover the layered architectures (phase 3 of the research methodology). To this extent, we rely on the layering rules presented in Chapter 2 to cluster the various responsibilities of the system based on lexical information and we assign the so-recovered clusters to layers using structural information.We formalize both the tasks of recovering responsibilities of the system and of assigning the recovered responsibilities to layers as optimization problems. We respectively solve these problems with search-based algorithms. Besides, to keep our approach language and platform independent, we represent the structural and lexical data of the system under study using the KDM (Knowledge Discovery Metamodel) standard.

The organization of this chapter is as follows: in Section 5.1, we provide an overview of this recovery approach. Section 5.2 is dedicated to the extraction of the structural and lexical facts used as input by our approach. In section 5.3, we explain how we exploited the layering rules to cluster the various responsibilities of the system based on lexical information. Section 5.4 describes the technique we used to assign the so-recovered clusters to layers based on structural information. Finally, Section 5.5 concludes this chapter.

5.1 Overview of the proposed approach

To recover the layering organization of a system, we propose an approach that uses structural and lexical information to recover the layered architecture, at the package level, of an OO system. Figure 5.1 gives an overview of the proposed three-step approach.

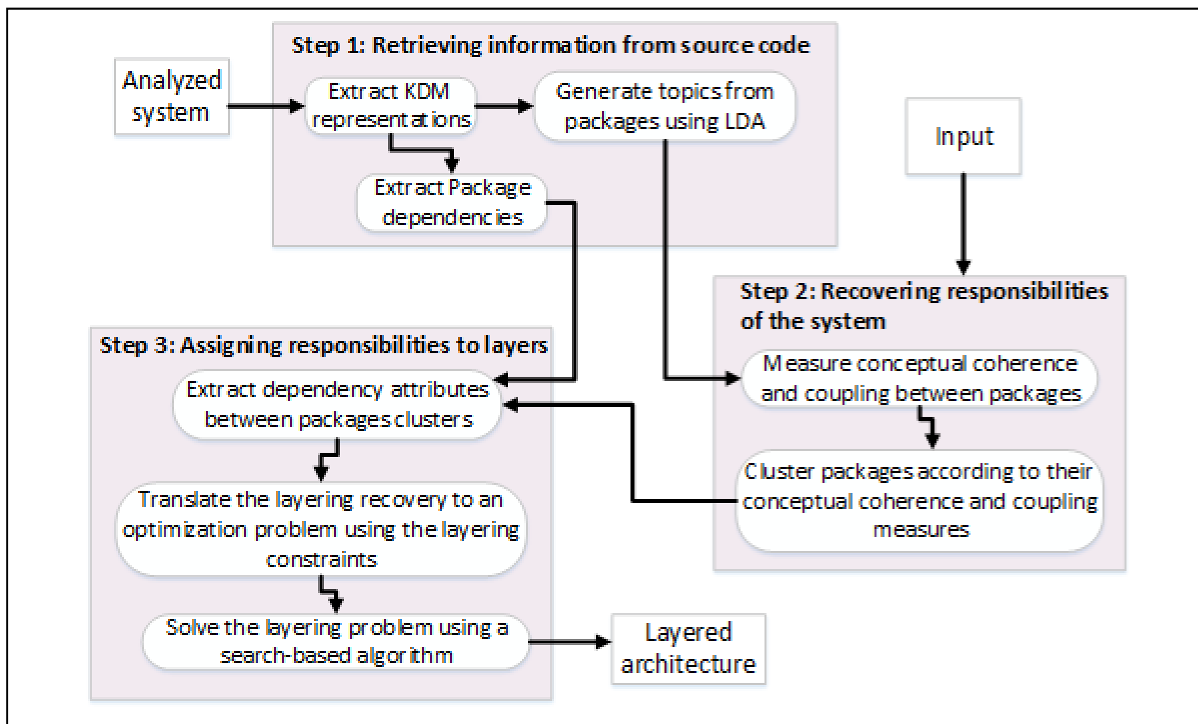


Figure 5.1 Overview of the proposed approach

1- The first step covers the automatic generation of a representation of the system at hand by using the KDM standard and extracts the required facts from that representation. At this step, the structural and lexical facts are retrieved since the focus is on the abstraction and responsibility rules. Regarding the abstraction rules, we specifically rely on the Incremental Layer Dependency rule. Recall that the enforcement of the Layer Abstraction Uniformity rule derives from the enforcement of the Incremental Layer Dependency rule. As explained in Chapter 2, to retrieve the facts, the source code is analyzed to generate platform independent models that are compliant with the KDM (Knowledge Discovery Metamodel). These models are then parsed to extract the lexical and structural information embedded within the system

under study. Since we work at the package level, we aggregate both the lexical and structural information at the package level and we use the aggregated information to assign the packages to the appropriate layers in the two subsequent steps. The structural information is made of the packages and their dependencies. Recall that packages dependencies are derived from dependencies between their respective entities. The lexical information consists in turn of topics computed for each package. We generate these topics by feeding an LDA (Latent Dirichlet Allocation) model (Blei et al., 2003) with the significant keywords derived from the system's source code. These include the identifier names (i.e., names of packages, classes, methods, fields and parameters) as well as the comments found in the source code. We further described the facts extraction step in Section 5.2.

2- The second step of our approach is tasked with the recovery of system's responsibilities. To this end, we rely on the Responsibility rule which states that "*each layer of the system must have a precise meaning, supported by a set of responsibilities, belonging to the same level of abstraction*". The responsibilities of the topmost layer correspond to the overall function of the system as perceived by the final user and the responsibilities of the lower layers contribute to those of the higher layers. Each responsibility can be implemented by a number of packages that are specific to a given domain (e.g., GUI, web interface). To recover such responsibilities, we rely on the lexical facts i.e. the package's topics identified at the previous step to measure the conceptual coupling between packages (CCP). We then use the CCP to express the lexical quality (LQ) of a given partition of the analyzed system into clusters of packages. Using LQ as a fitness function, we translate the recovery of the responsibilities of the system into an optimization problem that we solve using a search-based algorithm. We provide a detailed description of this step in Section 5.3.

3- At the third step of our approach, we aim at recovering the system's layered architecture by assigning the clusters generated in step 2 to the appropriate layers using structural information extracted from the system's KDM models. We then compute dependencies between pairs of clusters from the dependencies between their respective packages. To assign the clusters of packages to layers, we use our structural-based approach as introduced in

Chapter 2; i.e., we translate the assignment of the clusters to layers into an optimization problem based on the minimization of the LaQ fitness function where the dependencies are now referring to dependencies between clusters of packages. We present the details of this step in Section 5.4.

Noteworthy, we perform both the recovery of the responsibilities and their assignment to the layers using search-based algorithms derived from the SAHC (Steepest Ascent Hill Climbing) (Mitchell et al., 2008). As discussed in Chapter 2, we choose to focus on the SAHC because it performs well in the context of large systems (Harman et al., 2002; Mitchell and Mancoridis, 2002a) and it has been successfully used in several approaches. However, the hill climbing is likely to converge toward local optima. To reduce the risk of getting trapped in local optima, we rely on three mechanisms: 1) we perform multiple runs of the hill climbing – this also addresses the stochasticity of the algorithm (see Chapter 2); 2) we randomize its starting point (i.e., the initial partition); and 3) we perform an intensified search to further explore the search space. This third mechanism is also meant to reduce the variability of the respective results obtained when performing multiple runs of a hill climbing based algorithm using the same parameters.

5.2 Facts extraction

The facts extraction step is tasked with retrieving both the lexical and structural facts from the KDM representation of the OO system at hand. To extract the structural facts, we follow the same procedure than the one presented in Chapter 2 (see Section 2.3). In order to extract the lexical facts from the system under study, the KDM models that we generated from the source code are parsed to retrieve and process the meaningful and unique keywords embedded within each package source code. Then are identified meaningful topics in entities of the system using the Latent Dirichlet Allocation (LDA) statistical model (Blei et al., 2003). In the following, we describe LDA, the keywords extraction process and the identification of topics using LDA.

5.2.1 LDA (Latent Dirichlet Allocation)

During development, a software engineer usually embeds her domain knowledge of the system through lexical information disseminated all over the source code (Kuhn et al., 2007, Maskeri et al., 2008). This information appears in the source code as identifier names (i.e., package name, class name, method name, methods' parameters names and variables names) or as comments (Kuhn et al., 2007; Sarkar et al., 2007; Maskeri et al. 2008; Bavota et al 2013; Abebe et al. 2011, Poshyvanyk et al., 2009). Each of these lexical information then conveys the intention of the developers by leaving hints to the meaning of the source code elements in the form of keywords spread over files, functions, data type names and so on (Maskeri et al., 2008). Hence, extracting lexical information from the source code allows enriching software analysis (Kuhn et al., 2007). Of course, this information can only be useful if the developers of the system have followed naming conventions (Müller et al., 1993). Also, naming identifiers sloppily degrades the quality of the linguistic information derived from these identifiers (Kuhn et al., 2007).

A current practice to extract lexical information from the source code is to resort to topic modeling. A topic model is a statistical method that analyzes the words contained in a corpus of documents. This allows extracting the themes that run through these documents, the links between these themes as well as the evolution of these themes over time (Blei et al., 2003). A document can deal with various but related topics. These topics convey the document's intended message to its readers. For instance, topics such as architecture, design and quality attributes can reflect the content of a paper published in a software engineering journal.

One of the most popular topic modeling technique is LDA (Latent Dirichlet Allocation), which is a probabilistic model used in natural language processing to extract a set of latent topics from a corpus of text documents (Maskeri et al., 2008). LDA models each document as a probability distribution over topics and each topic as a probability distribution over the words in the vocabulary (Binkley et al., 2014, Abebe et al., 2011). The estimation of these distributions allows generating the set of T topics used in the corpus of documents as well as

the distribution of these topics in each document (Maskeri et al., 2008). Relying on topic distributions instead of bags of words to represent documents not only decreases the effect of lexical variability but also preserves the semantic structure of the corpus of documents (Yao et al., 2009).

LDA takes as input a word-by-document matrix M = (m_{ij}), where m_{ij} represents the importance of the word w_i in the document d_j. LDA also requires the specification of a set of hyper-parameters that impacts the resulting distributions of topics per document and words per topic. Simply explained, these hyper-parameters are (Maskeri et al., 2008; Panichella et al., 2013):

- α which affects the topic distribution per document: the higher α, the higher the likeliness for every document to be composed of every topic in significant proportions.
- β which affects the word distribution per topic: the higher β, the bigger the set of words per topic.
- T which is the number of topics to be identified from the corpus. With a too high value of T, the same concept can be spread over numerous topics and these ones are then diluted and meaningless (Binkley et al., 2014). As these topics are made of idiosyncratic words, they can become uninterpretable (Steyvers and Griffiths, 2007). Conversely, a small value of T leads to too broad topics i.e. topics which are constituted by keywords coming from multiple concepts and that are hard to discriminate (Steyvers and Griffiths, 2007; Binkley et al., 2014).

Different algorithms have been proposed to estimate topic per document and word per topic distributions. In this thesis, we will use the Gibbs sampling method (Griffiths and Steyvers, 2004) with the speed-up enhancements introduced by (Yao et al., 2009). The Gibbs sampling method uses a Markov Chain Monte Carlo method to converge to its target distributions after N iterations, each iteration consisting in sampling a topic for each word.

5.2.2 Extracting packages'keywords

To extract packages' keywords, we adapt the approach proposed by Maskeri et al. (Maskeri et al., 2008) to mine meaningful keywords from source code files (e.g., class) using identifiers. Unlike them, we work at the package level instead of the file level and we also use comments to extract keywords. Thus, to extract meaningful keywords from the packages' source code, we extend the four-step process of (Maskeri et al., 2008) by introducing a step that enables to process comments in the source code. We therefore proceed as follows:

Step 1: Extracting the facts. This step aims at traversing the KDM source code model to extract linguistic information i.e. identifier names and comments. In our case, identifiers are package names, class names, method names, method signature's parameter names as well as local and global variables names. The comments include entity header comments (i.e comments found in the header of a class, interface or enumerated type) and comments associated to global/local variables. The comments also include method header comments and comments found inside methods (i.e., comments associated to given statements).

Step 2: **Removing whitespaces and punctuation signs from comments**. Comments generally consist of lines of text. A line of text is a set of words separated by characters such as white spaces or punctuation signs. We retrieve the words contained in the comments by splitting their lines according to these characters.

Step 3: Identifier names and comments splitting. A word extracted in the previous steps may be a concatenation of keywords ruled by a common naming convention (e.g., camel case, hungarian) and eventually related to each other by some character (e.g., hyphen and underscore). At this step, we split each word extracted at the previous step into meaningful keywords. For this purpose, we remove characters such as hyphen and underscore from the word and we split it according to the most common naming conventions. For instance, the identifier *getHotelDiscount* (camel case notation) would be split into the following keywords: *get*, *Hotel* and *Discount*. We also convert the so-obtained keywords to lowercase.

This avoids the duplication caused by the presence of the same words in both lower and upper cases.

Step 4: stemming the keywords. Words with the same root can be spelled differently whereas they are related to the same concept (e.g., detail and details). To avoid distinguishing such keywords, we stem the keywords obtained at the previous step to their common roots using the Porter Stemming algorithm (Porter, 1997; Porter, 2006). This algorithm removes suffixes from words resulting from the different inflections of a given word as supported by the syntax of a language. These words include for instance nouns generated by adding a suffix when conjugating a given verb or when putting a word in the plural form. Words such as "*detail*" and "*details*" are therefore both stemmed to "*detail*". Likewise, words such as "*collaboration*" and "*collaborate*" are both stemmed to "*collabor*".

Step 5: filtering the keywords. Some keywords do not convey meaningful information particularly if they are implementation specific terms (e.g., java keywords and java tags). We then use these keywords to build a list of undesired words so as to filter them out. For instance, generic words such as *get*, *set* or *static* are filtered out, as well as keywords containing too little characters (e.g. *a*, *b* or *c*). Trivial words such as "*and*", "*then*" and "*the*" are also filtered words.

5.2.3 Using LDA to generate topics from packages' keywords

Based on the work by Maskeri et al. (Maskeri et al., 2008), we developed an approach to build a package-keyword matrix M that is the main input to LDA. A cell m_{ij} of the matrix M represents the importance of the keyword w_i in the package p_j. We compute the importance factor m_{ij} of the keyword w_i in the package p_j as a weighted sum of the number of occurrences of w_i in the package p_j:

$$m_{ij} = \sum_{lt \in p_j} Weightage(lt) * Frequency(w_i, lt, p_j) \quad (5.1)$$

where *Frequency*(w_i, *lt*, p_j) denotes the number of occurrences of the word w_i in the location type *lt* of the package p_j. *Weightage(lt)* denotes in turn the importance given to the location type *lt* from which the keyword is extracted. In our context, we identify eight different location types: package name, entity (i.e. class, interface or enum) name, method name, method signature's parameter name, (local or global) attribute name, entity header comment, attribute comment, method header comment and method internal comment. Simply put, the weightage is used to give more importance to keywords depending on their location in a given package; e.g., a keyword obtained from a class name is given more importance than a keyword obtained from a method name. The care taken when assigning identifiers to source code items generally decreases all the way from the packages to the comments. This is due to the fact that class-level entities embody domain objects and as such, their identifiers are more suceptible to yield domain words that are important for these entities (Maskeri et al., 2008). The same remark applies to the packages which represent groupings of domain objects.

To illustrate the computation of the importance factor of a word, let us take as example a small application written in Java. This application is named "*XSLT_Transformation"* and its source code is adapted from (XML Transformation, 2015). This application transforms a first XML (XML, 2015) document into another XML document. This application contains a single package named *transformer* comprising two classes respectively named "*XSLTTransformer"* and "*XMLGenerator"*. Figure 5.2 and Figure 5.3 depict the source code of these two classes.

clicours.com

```java
package transformer;

import java.io.File;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.Result;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.Source;
/**
 * This class uses an XSLT file to transform a first XML document into another
 * XML file.
 * @author Boaye Belle Alvine
 * 2015-09-25
 *
 */
public class XMLGenerator {

        private static XMLGenerator uniqueInstance;

        /**
         * Class's default constructor.
         */
        private XMLGenerator(){

        }

        /**
         * Returns the unique instance of the class.
         * @return
         */
        public static XMLGenerator getInstance(){
                if(uniqueInstance == null){
                   uniqueInstance = new XMLGenerator();
                }
                return uniqueInstance;
        }
      /**
       * Uses an XSLT transformation to generate an XML file from another XML
       * file.
       * @param xsltPath
       * @param xMLInputPath
       * @param xMLOutputPath
       */
         public void generateTargetXML(String xsltPath, String xMLInputPath,
                                String xMLOutputPath) {
                try {
                    Source inputXML = new StreamSource(new
                                 File(xMLInputPath).getAbsoluteFile());
                    Source inputXSLT = new StreamSource(new
                                 File(xsltPath).getAbsoluteFile());
                    Result outputXML = new StreamResult(new
                                 File(xMLOutputPath).getAbsoluteFile());
                    Transformer transformer = TransformerFactory.newInstance()
                                      .newTransformer(inputXSLT);
                    transformer.transform(inputXML, outputXML);
                    }
                catch(Exception exception){
                       …
                }
            }
}
```

Figure 5.2 Source code of the XMLGenerator class

```
package transformer;

/**
 * This class executes an XSLT transformation. Its takes three parame-
 * ters specified by the command line to perform this transformation.
 * @author Boaye Belle Alvine
 * 2015-09-25
 *
 */
public class XSLTTransformer {

	public static void main (String args[]) {
		XMLGenerator.getInstance().generateTargetXML(args[0],
		                                             args[1],
		                                             args[2]);
	}
}
```

Figure 5.3 Source code of the XSLTTransformer class

Following the five steps presented earlier, we split and stem each words contained in the two classes of the package *transformer*. The location type *lt* of the words retrieved from this code's package are the following: package name, class name, method name, method signature's parameter name, (local or global) attribute name, class header comment and method header comment. In our example, let us for instance assume that *Weightage* is 3 when the location type is either package name or entity name (i.e class name in this example). We also assume that *Weightage* is 2 when the location types are method name, method signature's parameter name or attribute name. Finally, we assume that *Weightage* is 1 when the location types are either a method internal comment, attribute comment, class header comment or method header comment.

The frequencies corresponding to the occurrences of the stemmed word *transform* in the different locations are the following:

- Frequency(transform, package name, transform) is the frequency of the word "transform" in the location type "package name" within the package named "transform" and it equals 1. This corresponds to the name "*transformer*" beared by the single package of the application. This name is stemmed into "*transform*".

- Frequency(transform, class name, transform) = 1: the identifier of the class "*XSLTTransformer*" is the only class-level name containing the word "*transformer*". After the five-step word processing, this identifier leads to the stemmed word "*transform*".
- Frequency(transform, method name, transform) = 0: in the two classes, no parameter in any method signature has an identifier name containing the stemmed word "*transform*".
- Frequency(transform, method signature, transform) = 0: none of the methods signatures of the two classes's has an identifier name whose processment leads to the stemmed word "*transform*".
- Frequency(transform, attribute name, transform) = 1: the method "*generateTargetXML*" of the class "*XMLGenerator*" has a local attribute bearing the name "*transformer*". The latter is stemmed into "*transform*".
- Frequency(transform, class header comment, transform) = 2: the class "*XMLTransformer*" contains two occurrences of the word "*transformation*" in its header comment. This word is processed into the stemmed word "*transform*".
- Frequency(transform, method header comment, transform) = 1: the method "*generateTargetXML*" of the class "*XMLGenerator*" contains the word "*transformation*" in its header comment. This word is processed into the stemmed word "*transform*".

The importance factor of the stemmed word "*transform*" within the source code of the package "*transformer*" is therefore $m_{11} = 3*1 + 3*1 + 2*1 + 1*2 + 1*1 = 11$.

We follow a similar process to compute the respective importance factors m_{ij} of the other words retrieved from the code of the transformer package.

Using the resulting package-keyword matrix $M = (m_{ij})$, LDA generates: 1) a set of topics extracted from the packages of the analyzed system, 2) the description of each topic as a set of the most probable keywords drawn from the identifier names and comments, as well as 3)

the proportion of each topic in each package. This allows representing the linguistic information embedded in each package source code as a topic proportion vector where each element indicates the proportion of a given topic in the package source code.

5.3 Recovery of the system's responsibilities

A layer's responsibility can be refined into finer responsibilities implemented by a number of packages. Therefore, packages which contribute to the same responsibility should be clustered together so as to be assigned to the same layer. Each of the so-obtained clusters is a cohesive group of packages that contribute to realizing the cluster's responsibility and that contain similar linguistic information. Thus, we rely on packages' linguistic information to recover the responsibilities of the layers. As described in the next subsections, we use this information to define lexical layers' dependency attributes. We then rely on these layers' dependency attributes to translate the recovery of layers' responsibilities into an optimization problem, which we solve using an SAHC.

5.3.1 Measuring linguistic cohesion and coupling of packages

Bavota et al (Bavota et al., 2013), consider that two classes share a conceptual link if their lexical information is similar, i.e., if their responsibilities are similar. We transpose this logic to the package level and assume that two packages are conceptually related if there is a similarity between their respective lexical information. We introduce a conceptual measure called *CCP* (Conceptual Coupling between Packages) to measure the similarity of the lexical information embedded in two packages. The definition of *CCP* is adapted from the conceptual coupling measures *CCM* (Conceptual Coupling between Methods*)* and *CCBC* (Conceptual Coupling Between Classes) proposed by Poshyvanyk et al. (Poshyvanyk et al., 2009). *CCM* and *CCBC* are notably used in (Bavota et al., 2013). *CCM* measures the lexical similarity between two methods and is computed as the cosine similarity between the vectors of these two methods. These vectors are generated from the comments and identifiers comprised in the source code of their respective methods using LSI (Latent Semantic

Indexing) (Deerwester et al., 1990). The *CCBC* between two classes is the average of the *CCM* between their respective methods (Poshyvanyk et al., 2009; Bavota et al., 2013). The *CCM* and *CCBC* work on method and class levels and are computed using LSI. Unlike them, the *CCP* is a higher-level (package) measure that we compute based on LDA. We compute the conceptual coupling between two packages *i* and *j* as the cosine similarity between their corresponding topic proportion vectors:

$$CCP(pi, pj) = \frac{\overrightarrow{Pi}.\overrightarrow{Pj}}{\left\|\overrightarrow{Pi}\right\| \left\|\overrightarrow{Pj}\right\|} \quad (5.2)$$

In this equation, $\overrightarrow{Pi.}$ and $\overrightarrow{Pj}$ are respectively the topic proportion vectors of the packages p_i and p_j while $\|\overrightarrow{x}\|$ represents in turn the Euclidian norm of the vector x. The value of *CCP* belongs to the interval [0, 1]. The more the lexical information comprised in the vectors $\overrightarrow{Pi.}$ and $\overrightarrow{Pj}$ are similar, the higher (closer to 1) is *CCP*. The CCP measure is symmetrical. This means that: $CCP(p_k, p_h) = CCP(p_h, p_k)$.

In order to illustrate the computation of the values of *CCP*, let us for instance consider 3 packages from JHotDraw 7.0.7. By applying the process described in the previous subsection, we identify sample topics in these 3 packages, the keywords associated to each of these topics as well as the proportion (i.e., probability) of these topics in each package. Table 5.1 reports these data[21]. In particular, its greyed cells correspond to dominant topics in each package. Note that, for comprehension purposes, we have reported the topics' keywords in this table under their original form (i.e., not stemmed). In JHotDraw, the documentation of the packages *net.n3.nanoxml* and *nanoxml* indicates that they have conceptually related responsibilities. The *CCP* between these two packages is 0.999. This *CCP* value is very high (i.e., very close to 1). This indicates that these two packages share very similar lexical information. This is consistent with the fact that their respective responsibilities are similar.

[21] These results were generated using the LDA parameters 0.2 as α, 0.1 as β, 10 as the number of topics and 1000 as the number of iterations.

Contrariwise, the *CCP* between the packages *net.n3.nanoxml* and *org.jhotdraw.undo* is 0.00017, i.e., very close to 0. This is not surprising given the lack of cohesion between the responsibilities of these two packages. Likewise, the very low *CCP* value (i.e. 0.00034) between *nanoxml* and *org.jhotdraw.undo* is consistent with the lack of cohesion between their respective responsibilities.

Table 5.1 Sample topics extracted from three packages of JHotDraw 7.0.7

	Topics keywords	**Topics proportion in each package**		
		nanoxml	**net.n3.nanoxml**	**org.jhotdraw.undo**
Topic 2	edit, change, value, property, remove, listen, figure, fire, composite	0.218E-3	0.496E-4	0.914
Topic 3	xml, system, element, attribute, namespace, reader, resolve, entity, value	0.998	0.999	0.105E-3
Topic 4	action, event, value, application, app, perform, init, project, listen	0.113E-3	0.257E-4	0.084
Topic 7	draw, init, editor, create, event, value, info, tool, add	0.318E-3	0.722E-4	0.459E-3

We rely on the *CCP* measure to recover the responsibilities of the analyzed system. In particular, we use this measure to group the system's packages into a set of clusters exhibiting high lexical cohesion and low lexical coupling between the clusters. To this end, we follow the same notations adopted by (Mitchell et al., 2008) when naming modularity measures. Hence, we refer to the lexical cohesion within a cluster i (of packages) as μ_i. Furthermore, we refer to the lexical inter-coupling between 2 distinct clusters c_i and c_j as $\varepsilon_{i,j}$.

We express μ_i as the average lexical cohesion between all the unordered pairs of packages p_k, p_h comprised in a cluster i. Let n be the number of packages of the cluster c_i. By leveraging the symmetricity of *CCP*, we can compute μ_i as follows:

$$\mu i = 2 * \frac{\sum_{k=1}^{n-1}\sum_{h=k+1}^{n} CCP(p_k, p_h)}{\frac{n(n-1)}{2}} = \frac{4}{n(n-1)} \sum_{k=1}^{n-1}\sum_{h=k+1}^{n} CCP(p_k, p_h) \quad (5.3)$$

We then obtain the following simplified version of μ_i:

$$\mu i = \frac{2}{n(n-1)} \sum_{k=1}^{n} \sum_{h=1,h\neq k}^{n} CCP(p_k, p_h) \tag{5.4}$$

We express $\varepsilon_{i,j}$ as the average lexical coupling between all the ordered pairs of packages p_k, p_h respectively belonging to two distinct clusters i and j. Considering that n and m are the respective number of packages of the clusters c_i and c_j, we compute $\varepsilon_{i,j}$ as follows:

$$\varepsilon i, j = \frac{1}{n * m} \sum_{p_k \epsilon c_i} \sum_{p_h \epsilon j} CCP(p_k, p_h) \tag{5.5}$$

Note that, since CCP is symmetric, $\varepsilon_{i,j}$ is also symmetric. Hence, $\varepsilon_{i,j} = \varepsilon_{ji}$.

5.3.2 A fitness function to assess the linguistic quality

We propose a fitness function called *LQ* (Lexical Quality) to measure the quality of the linguistic information (i.e., conceptual cohesion) within the clusters of a given system. LQ is adapted from the fitness function *MQ* in (Mitchell et al., 2008). Our fitness function *LQ* relies on packages' linguistic information. The aim of *LQ* is not only to reward conceptual cohesion within clusters but also to penalize inter-clusters conceptual coupling. Following the same logic as (Mitchell et al., 2008), we define the conceptual cluster factor CCF_i of a cluster c_i as the normalized ratio between the lexical cohesion μ_i within the cluster i and the lexical coupling $\varepsilon_{i,j}$ between the cluster c_i and every other cluster c_j. This allows us to express the lexical quality *LQ* of a system as the sum of all CCF_i factors of all the clusters generated from its packages.

$$LQ = \sum_{i=1}^{k} CCF_i \tag{5.6}$$

$$CCF_i = \begin{cases} o & \mu_i = 0 \\ \dfrac{\mu_i}{\mu_i + \sum_{\substack{j=1 \\ j\neq i}}^{k} \varepsilon_{i,j}} & \text{otherwise} \end{cases} \tag{5.7}$$

We formulate the recovery of responsibilities as an optimization problem whose fitness function is *LQ*. Thus, recovering the responsibilities of the analyzed system consists in finding a system's partition for which LQ is maximal. We do so, as explained in the next section, by applying the SAHC (Steepest Ascent Hill Climbing).

5.3.3 Using the hill climbing to recover layers' responsibilities

Algorithm 5.1 illustrates a high level view of *SAHCClustering*, our version of the SAHC for the recovery of layers' responsibilities. *SAHCClustering* uses LQ as a fitness function.

Algorithm 5.1 A high level view of the clustering algorithm

```
Algorithm SAHCClustering
Input: initialClusteredSystem, improvementTrials
Output: ClusteredSolution
  1.  bestSolution ← initialClusteredSystem
  2.  while (TRUE) {
  3.      bestNeighbor  ← NULL // bestNeighbor's LQ is set to  -∞ by default
  4.      neighborList ← computeAllNeighbors(bestSolution)
  5.      for (neighbor in neighborList){
  6.          if (LQ(neighbor) > LQ(bestNeighbor))
  7.              bestNeighbor ← neighbor
  8.      } //end for
  9.      if(LQ(bestNeighbor)  > LQ(bestSolution))
  10.         bestSolution ← bestNeighbor
  11.     else {
  12.         improvedNeighbor ← improve(bestNeighbor, improvementTrials)
  13.         if (LQ(improvedNeighbor )  > LQ(bestSolution))
  14.             bestSolution ← improvedNeighbor
  15.         else
  16.             END WHILE LOOP
  17.     } //end if
  18. }//end while loop
  19. return bestSolution
```

This algorithm starts by an initial random partition where the system's packages are randomly assigned to a set of clusters. We compute this initial partition by relying on the nodes and edges of the module dependency graph representing the input system. This initial

partition is considered as the best solution of the algorithm (line 1). In the following iterations (lines 2 to 18), all the neighboring solutions are created (line 4) and evaluated using their quality LQ (line 6). A neighbor solution is created by moving a single package from a cluster to a distinct one. The neighbor having the highest value of LQ is considered as the best neighbor of the iteration (lines 5 to 8). This neighbor is accepted as the new best solution if its fitness is higher than that of the best solution (lines 9 and 10). Otherwise, the algorithm starts an intensified search aiming at finding another neighbor that is better than the current best neighbor (line 12). This new neighbor is accepted as the new best solution if its fitness is higher than the one of the best solution (lines 13 to 14). During the intensified search, the algorithm performs a number of iterations (i.e., the *improvementTrials* parameter) (line 12) to compute the neighbors of the best solution and eventually accepts its degradation so as to further explore the search space. The algorithm stops if the best solution can no longer be improved (lines 15 and 16).

As explained in Chapter 2, the randomness of the initial partition of a hill climbing based algorithm such as the *SAHCClustering* can drive it to generate solutions that do not converge always toward the same sub-optimal solution. Running this algorithm many times is a way to address its stochasticity (e.g., Mitchell et al. (Mitchell et al., 2006) and Saeidi et al. (Saeidi et al., 2015). Hence, the solution having the best quality (i.e. the highest quality in our context) among the so-generated solutions is kept as the best solution.

5.4 Assigning responsibilities to layers

5.4.1 The layering of responsibilities as an optimization problem

To assign the clusters (i.e., responsibilities) resulting from the previous step to layers, we follow a process similar to the one used in Chapter 2. Hence, we translate the problem of assigning the clusters to layers into an optimization problem using the layers' dependency attributes and constraints introduced in Chapter 2, with the sole difference that they now apply to the cluster level instead of the package level.

Given two layers of a layered architecture respectively referred as layer *i* and layer *j*, we now compute the dependency going from layer *i* to layer *j* by summing the weights of the dependencies directed from each cluster of layer *i* to each cluster of layer *j*. Thus, we compute the layering quality *LaQ* of assigning the clusters of a system to a set of *n* layers as follows:

$$LaQ = \sum_{i=1}^{n} ILQ(i) \tag{5.8}$$

$$ILQ(i) = ap * AdjacencyUse(i, i-1) + ip * IntraUse(i) + sp * \sum_{j=i-2}^{1} SkipUse(i,j) + bp * \sum_{j=i+1}^{n} BackUse(i,j) \tag{5.9}$$

Where:

- *n* is the number of layers;
- *AdjacencyUse(i,i-1)* [22] denotes the number of dependencies directed from layer *i* to its adjacent lower layer *i-1*;
- *IntraUse(i)* indicates the number the dependencies within layer *i*;
- *SkipUse(i,j)* denotes the number of skip-calls directed from layer *i* to layer *j*;
- *BackUse(i,j)* indicates the number of back-calls directed from layer *i* to layer *j*; and
- *ap*, *ip*, *sp* and *bp* are respectively the factors adjoined to adjacent dependencies, intra-dependencies, skip-call dependencies and back-call dependencies.

The lower LaQ is, the better the assignment of clusters to layers is. To find the system's layering that minimizes LaQ, we rely on the algorithm described in the next section.

---

[22] When computing these metrics, we consider that the layers are numbered in ascending order from bottom to top.

clicours.com

5.4.2 Using the hill climbing to assign clusters to layers

To assign the clusters to layers, we build an algorithm called *SAHCLayeringCl* that applies the SAHC technique. This algorithm is very similar to the *SAHCLayering* algorithm (see Chapter 2) with the sole differences that it applies to the cluster level instead of the package level and supports a further exploration of the search space. In particular, the *SAHCLayeringCl* algorithm aims at producing an optimal 3-layered solution from an analyzed system. This algorithm starts from an initial partition consisting of a set of 3 layers which are randomly assigned the system's clusters resulting from the clustering step (see Section 5.3). The *SAHCLayeringCl* algorithm then recursively attempts to divide each of these 3 layers into *m* layers until the LaQ value of the so-obtained layering can no longer be improved. This layering refinement is motivated by the fact that the analyzed system might have more than 3 layers. The *SAHCLayeringCl* algorithm uses the *HCOptimizationCl* algorithm described by Algorithm 5.2 to optimize the content of each layered solution according to a given neighborhood definition.

The *HCOptimizationCl* algorithm is similar to the *HCOptimization* introduced in Chapter 2. However, unlike the *HCOptimization*, the *HCOptimizationCl* works at the cluster level and performs an intensified search to further explore the search space. The *HCOptimizationCl* takes as input: 1) an initial layered partition; 2) the appropriate neighborhood to use when computing the neighboring solutions; 3) the values of the factors (*ap*, *ip*, *sp* and *bp*) assigned to each kind of layering dependencies; and 4) the number of iterations (i.e., the *improvementTrials* parameter) used to further explore the search space. The initial partition taken as input by the *HCOptimizationCl* is then considered as the best solution of the algorithm (line 1).

In the following iterations (lines 2 to 18), the algorithm computes all the neighboring solutions (line 4) and evaluates them based on their quality LaQ (line 6). A neighbor solution is computed by moving a single cluster from a layer to another one. The neighbor which has the lowest value of LaQ is considered as the best neighbor of the iteration (lines 6 to 8). This

neighbor is accepted as the new best solution if its fitness is lower than that of the best solution (lines 10 to 12). Otherwise, the algorithm performs an intensified search in order to find another neighbor that is better than the current best neighbor (lines 11 to 14). This new neighbor is accepted as the new best solution if its fitness is higher than that of the best solution (lines 13 to 15). During this intensified search, the algorithm performs a number of iterations (i.e., the *improvementTrials* parameter) to compute the neighbors of the best solution and eventually accepts its degradation so as to further explore the search space.

Algorithm 5.2 Hill climbing based optimization algorithm

```
Algorithm HCOptimizationCl
Input: inputPartition, neighborhood,  factors, improvementTrials
Output: LayeredSolution
   1.  currentSolution ← inputPartition
   2.  while (TRUE){
   3.      bestNeighbor  ← NULL // bestNeighbor's LC is set to  +∞ by default
   4.      neighborList ← computeAllNeighbors(currentSolution, neighborhood)
   5.      for (neighbor in neighborList){
   6.          if (LC(neighbor, factors)  <  LC(bestNeighbor, factors))
   7.              bestNeighbor ← neighbor
   8.      }//end for
   9.      if (LC(bestNeighbor, factors)  <  LC(bestSolution, factors))
   10.         bestSolution ← bestNeighbor
   11.     else {
   12.         improvedNeighbor ← improve(bestNeighbor, neighborhood ,
                                              improvementTrials, factors)
   13.         if (LC(improvedNeighbor , factors)  <  LC(bestSolution, factors))
   14.             bestSolution ← improvedNeighbor
   15.         else
   16.             END WHILE LOOP
   17.     }// end if
   18. }//end while loop
   19. return currentSolution
```

The algorithm stops the iterative process if the best solution cannot be improved anymore (lines 15 to 17).

Note that the specificities of the *SAHCLayering* discussed in Chapter 2 have been preserved when defining the *SAHCLayering*Cl algorithm. In particular, the latter (as well as the *HCOptimizationCl*) also uses two definitions of the neighborhood. These definitions remain the same as in the Chapter 2, except that they now apply at the cluster level instead of the package level. Furthermore, as explained in Chapter 2, the literature (e.g., Mitchell et al. (Mitchell et al., 2006) and Saeidi et al. (Saeidi et al., 2015)) generally overcome the stochasticity of the hill climbing, by running this algorithm many times. In this case, the solution having the best quality (i.e. the lowest quality in our context) among the so-generated solutions is usually kept as the best solution. Since *SAHCLayeringCl* algorithm (and the *HCOptimizationCl* algorithm) is derived from the hill climbing, we also run it many times to keep the best solution out of the ones yielded by the different runs. Finally, as computing the quality of each partition from scratch can be time-consuming for very large systems, we also compute LaQ incrementally.

5.5 Chapter summary

In this chapter, we have proposed a second approach which exploits a subset of the layered style rules, namely the responsibility and abstraction rules, to recover the layered architecture of object-oriented systems. This hybrid approach first relies on linguistic information to recover the responsibilities of the analyzed system into a set of cohesive clusters. This approach then exploits structural information to assign these responsibilities/clusters to layers. To this end, this hybrid approach translates both the recovery of responsibilities and their assignment to layers into optimization problems that are respectively solved using a search-based algorithm.

CHAPTER 6

EVALUATING THE LEXICAL AND STRUCTURAL-BASED LAYERING RECOVERY APPROACH

In this chapter, we present and discuss the results of our structural and lexical based layering approach (Chapter 5) when applied to four analyzed systems (phase 3 of the research methodology). In Section 6.1, we describe the experimental design. In Section 6.2, we present and discuss the results obtained with our hybrid approach and we also compare these results to those generated using two other approaches: our structural-based layering approach (CHAPTER 2 and CHAPTER 4) and the Lattix tool (Sangal et al., 2005a; Sangal et al., 2005b). In Section 6.3, we outline some threats which may affect the validity of the results of our hybrid approach. We summarize this chapter in Section 6.4.

6.1 Experimental design

In this section, we provide a description of the tool we developed to support our approach and present the systems analyzed during the validation of this approach. We also describe the experimentation questions that guided our experimentations as well as the experimental settings.

6.1.1 Implementation

We experimented our approach using a tool that we developed within the Eclipse™ environment. This tool which is an improvement of ReALEITY (presented in Chapter 3) comprises three modules. The first one is a fact extractor built atop of the MoDisco (Modisco, 2015) open source tool which allows the generation of a KDM representation of the system under study. The extractor parses the KDM representation in order to extract: 1) the system's packages and their dependencies; and 2) the identifiers' names and the comments from each package. In particular, the extractor relies on the Porter Stemming algorithm implementation available at (PorterStemmer, 2015) to process these keywords. It

then uses the so-obtained keywords to create the LDA input matrix and derives packages topics using the LDA implementation provided by the Mallet API (Mallet, 2015). Our second module which provides an implementation of the SAHC technique, uses LQ (lexical quality) as a fitness function to cluster packages. This second module generates a module dependency graph where nodes and edges respectively represent clusters of packages and dependencies between the clusters. This module dependency graph enables the creation of an initial partition that serves as the input of the third module of our tool. This third module provides an implementation of our layering algorithm. As such, it uses a SAHC technique to assign clusters to layers using the LaQ (layering quality) as a fitness function. Noteworthy, the tool supports the interaction with the user by allowing her to modify the recovered layers through the re-assignement of nodes to different layers.

6.1.2 Dataset

During our experimentations, we applied our approach on the following four open source systems developed in Java: Table 6.1 provides some statistics describing these systems.

Table 6.1 Statistics of the analyzed systems

Systems	Number of files	Lines Of Code	Number of Packages	Packages dependencies	Number of extracted keywords
Apache Ant 1.6.2	681	171 491	67	229	4134
JFreechart 1.0.15	600	222 475	37	243	1522
JHotDraw 7.0.7	310	57 020	24	89	1177
JUnit 4.10	162	10 402	28	107	625

6.1.3 Experimentation questions

The purpose of the experiments we carried out on the systems described above was to investigate the following experimentation questions:

1. EQ1: To which extent do the clusters generated by the lexical clustering match the responsibilities provided by the system under study?

2. EQ2: is the accuracy of the layering results increased by the use of lexical information?
3. EQ3: is our approach more authoritative than a well established layering recovery approach?

For EQ1, we make the assumption that a system's package structure reflects its responsibilities and their relationships if this system is well modularized. In this regard, we study the results generated by our lexical clustering through the comparison between the clusters and the package structure as implemented in the system at hand. To this extent, we obtain the package structure from the analyzed system's source folder structure. Hence, in order to provide an answer to EQ1, we investigate two sub-questions, namely: i) in which cases do the lexical clusters correspond to the analyzed system's package structure? And ii) in which cases the lexical clusters do not correspond to the the analyzed system's package structure?

We provide an answer to these two sub-questions by comparing, for each analyzed system, the best clustering result (i.e., the one having the highest quality LQ) over all the LDA combinations used in our experiments (see section 6.1.4) to the system's package structure. To perform this comparison, we introduce a measure named Clusters to Structure Conformance (C2SC). More specifically, for a given system, we compute the C2SC measure as the percentage of the clustered packages that match the package structure implemented by the designers. We compute C2SC as follows:

$$C2SC = \frac{\sum_{i=1}^{NC} \text{MaxMatch}(C_i)}{NP} \tag{6.1}$$

Where *NP* refers to the number of packages of the analyzed system while *NC* denotes the number of clusters returned by the lexical clustering. *MaxMatch*(C_i) designates in turn the size (i.e. number of packages) of the largest subset of the cluster C_i for which the packages conform to the package structure of the system. Note that there are two cases in which we consider that *n* packages contained in the same cluster conform to the package structure. These cases are: 1) the *n* packages are at the same level in the package structure and have the

same parent container, or 2) one of the *n* packages is the direct parent (i.e., the direct container) of the other (*n-1*) packages.

Let us take for instance the clustering depicted by Figure 6.1. In this example, the system at hand comprises ten packages. All packages that belong to the cluster 1 conform to the package structure of the system (first-case conformance). Therefore, *MaxMatch(Cluster 1) = 2*. However, when considering all the packages belonging to cluster 2, we notice that, together, these packages do not correspond to the structure of the system. *MaxMatch(Cluster 2) =2* for this cluster, since the size of the largest subset of packages that matches the package structure of the system is 2; and since "input.format.text.*" is the direct parent of "input.format.text.utf8", this correspond to the second case of conformance. Similarly to Cluster 2, when considering all the packages belonging to cluster 3, we notice that, together,

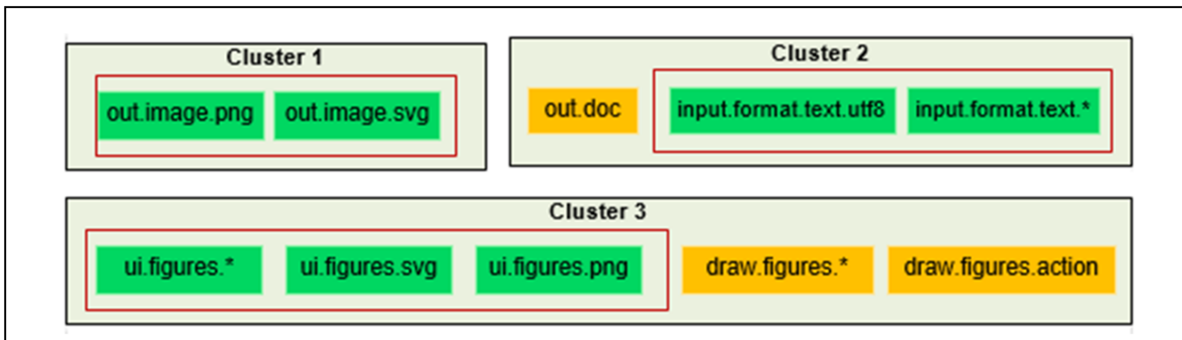


Figure 6.1 Example of clustering to illustrate the computation of C2SC

these packages do not match the structure of the system. In particular, Cluster 3 contains two subsets of packages and when considered separately, each of these subsets is consistent with the system's structure (i.e., second-case conformance). Therefore, *MaxMatch(Cluster 3)* equals the number of packages of the subset which has the larger size; in this case, this number is 3. Hence, for this sample clustering, C2SC = (2 + 2 + 3)/10 = 0.7. This value of C2SC indicates that 70% of the clustered packages match the package structure of the system.

Note that C2SC is a conservative measure by construction. When computing C2SC based on the size of the largest subset of each cluster for which packages match the package structure rather than the sum of the sizes of all the subsets of each cluster for which packages match the package structure, we implicitly reduce its value. For instance, in the cluster 3 of Figure 6.1, the computation of C2SC only considers the 3 matches met by the subset {*ui.figures.*, ui.figures.svg, ui.figures.png*} instead of also taking into account the 2 matches of the subset {*draw.figures.*, draw.figures.action*} which would have brought to 5 the total number of matches within cluster 3.

Besides, based on the assumption that the reasons that explain the correspondance or absence of correspondance between the clustering results and the package structure might notably be inherent to the nature of the topics generated by LDA, we examine the resulting clusters in the light of the obtained topics. To this end, we classify the topics based on the topic distribution analysis proposed in (Kuhn et al., 2007). The latter identifies various patterns for the distribution of the topics. Among these patterns, the ones that are relevant to our work are the following (Kuhn et al., 2007):

- **Well-encapsulated topic**: it is a topic which corresponds to system parts (i.e., clusters of the packages in our context). This topic is distributed over one or multiple parts of the system, including a significant portion of the source code within those parts. A well-encapsulated topic that is limited to a single part of the system is called a **solitary topic**.
- **Cross-cutting topic**: it is a topic that spans multiple parts of the system while covering one or very few elements within each part. This topic is orthogonal to the system parts.
- **Octopus topic**: it is a well-encapsulated topic that dominates a single part of the system (i.e., a solitary topic) but it also spans other parts of the system as a cross-cutting topic.

Regarding EQ2, we aim at assessing the extent to which the use of lexical information to cluster packages into responsibilities prior to their assignment to layers affects the

layering results. In order to provide an answer to this question, we compare the results of the hybrid layering approach (to which we now refer as the LI approach) described in Chapter 5, which uses both lexical and structural information, with the SAHC based layering approach presented in Chapter 2 (to which we now refer as the NLI approach). The latter solely exploits structural information to reconstruct the architecture of layered systems. To provide a fair comparision between the two approaches, we add an intensified search to this second approach (i.e. to the NLI approach). We rely on the harmonic mean (F-measure) (Baeza-Yates and Ribeiro-Neto, 1999) of the precision and recall to assess both the correctness and completeness of each layering solution compared to the authoritative decomposition. APPENDIX II (p. 199) reports all the authoritative decompositions used to compute the harmonic mean. Similar to our experiments with the NLI approach (see Chapter 4), we rely on the precision and recall as defined in (Scanniello et al., 2010a) to compute the harmonic mean. Hence, we compute the precision as the number of packages that our tool correctly assigned divided by the total number of packages that our tool assigned. We compute the recall as the number of packages that our tool correctly assigned divided by the number of packages assigned to layers in the authoritative decomposition. The precision and recall might differ when there is a package that is not related to others. In this case, the tool cannot assign this package to any layer while it is assigned in the authoritative decomposition.

However, to be able to perform a fair comparison with other existing layering approaches (e.g., the ones implemented by the Lattix tool (Sangal et al., 2005a)) which require an input from the user to recursively refine the layers of the analyzed systems, we adapted the way we compute precision and recall as follows. We first establish a matching between the layers of the resulting decomposition and those of the authoritative ones; a layer is at most matched to a layer with which it shares the higher number of packages. Based on these layer matchings, we then compute the precision and recall as described above.

To answer EQ3, we will compare our hybrid layering approach to a well established layering recovery approach (the Lattix tool (Sangal et al., 2005a; Sangal et al., 2005b)). We will use

the harmonic mean (F-measure) as described above to compare our layering results with the results obtained using Lattix.

6.1.4 Experimental settings

We performed our experiments on a machine with an Intel(R) Core i7-3778 CPU @3.40GHz and 16 GB of RAM. We followed a trial-and-error procedure and also relied on previous works (e.g., (Panichella et al., 2013; Binkley et al., 2014)) to set the parameters of our experiments. We therefore chose the parameters of LDA as follows:

- We vary the number of topics T from 10 to 50 topics, increasing its value by a step of 5.
- We vary the values of α (topic distribution per package) and β (word distribution per topic) from 0.1 to 1, increasing its value by a step of 0.225.
- We set to 1000 the number of iterations N of the Gibbs Sampling method.

For each system at hand, we therefore run the clustering algorithm using 225 combinations of LDA parameters. Each solution produced by the clustering algorithm using a given combination of the LDA parameters is then given as input to the layering algorithm.

The clustering and layering algorithms described in Chapter 5 both rely on the SAHC. It is also the case for *SAHCLayering* layering algorithm that we presented in Chapter 2 (that we now enhance with an intensified search) to which we compare our layering results. Since the SAHC is a stochastic algorithm, for each combination of the LDA parameters and for each setup of layering factors, we respectively perform 50 independent runs of each of the three algorithms. At the end of the execution of each algorithm, the solution yielding the best quality among the 50 generated solutions is kept as the best solution. We also set the *improvementsTrials* parameter to 50; i.e., we perform 50 iterations during the intensified search. Remind that once any of the two layering algorithms has generated a first layering, it then recursively tries to divide each generated layer into m other layers. During the respective execution of the two layering algorithms, we make m vary from 2 to 3 by step of 1.

We run the hybrid layering algorithm described in Section 5.4 as well as the SAHC-based layering algorithm described in Chapter 2 using the five following setups of the factors *ap*, *ip*, *sp* and *bp*:

1. *Setup 1* = (*ap*=0, *ip*=1, *sp*=2, *bp*=4),
2. *Setup 2*= (ap=0, ip=1, sp=2, bp=15)
3. *Setup 3* = (*ap*=0, *ip*=2, *sp*=1, *bp*=4)
4. *Setup 4*= (ap=0, ip=2, sp=1, bp=15)
5. *Setup 5* = (*ap*=0, *ip*=2, *sp*=1, *bp*=20).

To comply with the incremental layer dependency (ILD) property, we set the factor *ap* to zero in all the five setups. Note that we chose these setups according to the previous experiments that we conducted in Section 4.3. Remark that setups 1 and 2, on one hand, and setups 3, 4 and 5, on the other hand, vary depending on the value they assign to the back-call factor β and they are meant to analyze the extent to which the back-calls are tolerated in the analyzed systems. In particular, the factors values specified in setups 1 and 2 lead to the generation of a layering where the number of adjacent and intra-dependencies outnumbers the number of skip-calls and back-calls. These setups are suitable for systems that foster portability at the expense of the reuse. Contrariwise, the factors values specified in setups 3, 4 and 5 allow generating layerings where the number of adjacent and skip-call dependencies outnumber the number of intra-dependencies and back-calls. Setups 3, 4 and 5 are therefore a good fit for systems complying with a reuse-based layering strategy. Let us remind that, according to this strategy, the most (re)used packages are assigned to the lowest layers.

6.2 Experimental results

In this section, we present and discuss the results generated by our hybrid recovery approach when applied to the analyzed systems.

6.2.1 Analysis of the clusters of packages (EQ1)

We analyzed the clusters of packages that the second step of our approach (i.e., the recovery of the system's responsibilities) generated. This analysis revealed that the lexical clustering generally produce meaningful groupings of packages, i.e., most of the clusters are conceptually cohesive clusters as confirmed by the manual analysis of the source code as well as the documentation of the analyzed systems. In particular, the clusters divide the analyzed systems into manageable parts which usually comprise 2 to 3 packages except for the Apache system. The latter yields a few larger clusters. The lexical clusters generally exhibit cohesive responsibilities. In addition, these clusters often group together packages involved in cyclic dependencies. For instance, in the case of JHotDraw, some of the resulting clusters are c_1={*org.jhotdraw.gui.**, *org.jhotdraw.gui.event*} and c_2={*net.n3.nanoxml*, *nanoxml*}. In the case of JFreeChart, some of the resulting clusters are c'_1={*org.jfree.chart.demo*, *org.jfree.chart.servlet*} and c'_2={*org.jfree.chart.title*, *org.jfree.chart.block*}.

Our analysis of the lexical clusters also revealed that these clusters do not always match the package structure of the system at hand. This observation particularly applies to JUnit. The computation of the C2SC measure that we presented in the previous section indicates that the percentage of matches between the resulting clusters and the package structure are 88.6% for JFreeChart, 62.5% for JHotDraw, 52.2% for Apache Ant and 35.7% for JUnit. However, the performance of the clustering algorithm is not necessarily at the origin of the non-conformance of the lexical clusters with the package structure. The lexical clustering does not always correspond to the package structure (Kuhn et al., 2007). We therefore investigated the facts/reasons behind the (non-) conformance between the resulting lexical clusters and the package structure. This investigation led us to the identification of various categories of factors influencing the creation of clusters that match or not the system's package structure. In case of conformance between the clusters and the system's package structure, these factors are:

1. The presence of cohesive responsibilities (i.e. well distributed responsibilities) and the use of an appropriate vocabulary within packages. This allows deriving well-encapsulated topics leading to the clustering of cohesive packages;
2. The non-utilization of thresholds to filter the LDA results which enables the creation of undesired clusters (i.e., non-cohesive) but that match the package structure. This encloses the two following other facts: i) the non-utilization of a threshold indicating whether a keyword belongs to its associated topic; and ii) the non-utilization of a threshold indicating whether the proportion of a given topic inside a package should be considered or not when measuring the conceptual coupling between packages.

We identified the following factors leading to the non-conformance between the clusters and the system's package structure:

1. A poor distribution of responsibilities among packages;
2. The presence of cross-cutting topics which is due to: i) the presence of cross-cutting concerns; or ii) an improper LDA calibration;
3. The presence of octopus topics;
4. The presence of solitary topics.

For each system, Table 6.2 reports the distribution of the C2SC measure according to these factors. This table also reports, for each system, the values of the LDA parameters (T, α, and β) that generated the best clustering results.

6.2.1.1 When do the lexical clusters match the package structure?

When packages have cohesive responsibilities and appropriate vocabulary

When analyzing the clusters that conform to the package structure, we noticed that most of these clusters respectively comprise packages sharing cohesive responsibilities. Moreover, the packages' cohesion emerges in their respective lexical information. The latter is usually captured by LDA under the form of well-encapsulated topics. The clusters containing packages that yield a high proportion of such topic(s) are generally cohesive. In the system

Table 6.2 Distribution of packages according to the identified factors

<table>
<tr><th colspan="3"></th><th>JHotDraw
$T=10$, $\alpha=0.55$, $\beta=0.1$</th><th>Apache Ant
$T=20$, $\alpha=0.1$, $\beta=0.325$</th><th>JUnit 4.10
$T=15$, $\alpha=0.32$, $\beta=0.55$</th><th>JFreeChart
$T=25$, $\alpha=0.1$, $\beta=0.55$</th></tr>
<tr><td rowspan="3">Matching (C2SC distribution)</td><td>Cohesive responsibilities and appropriate vocabulary</td><td>Well-encapsulated topics</td><td>50</td><td>52.2</td><td>35.7</td><td>82.9</td></tr>
<tr><td rowspan="2">Not using thresholds to filter LDA results</td><td>Absence of keywords threshold</td><td>12.5</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Absence of topics threshold (Solitary topics)</td><td>0</td><td>0</td><td>0</td><td>5.7</td></tr>
<tr><td colspan="3">C2SC Total</td><td>62.5</td><td>52.2</td><td>35.7</td><td>88.6</td></tr>
<tr><td colspan="7"></td></tr>
<tr><td rowspan="5">No matching (% of packages)</td><td colspan="2">responsibilities are poorly distributed among packages</td><td>16.7</td><td>0</td><td>42.9</td><td>5.7</td></tr>
<tr><td rowspan="2">Cross-cutting topics</td><td>Cross-cutting concerns</td><td>0</td><td>3</td><td>0</td><td>0</td></tr>
<tr><td>Improper LDA calibration</td><td>20.8</td><td>32.8</td><td>14.3</td><td>0</td></tr>
<tr><td colspan="2">Octopus topics</td><td></td><td>3</td><td>0</td><td>5.7</td></tr>
<tr><td colspan="2">Solitary topics</td><td>0</td><td>9</td><td>7.1</td><td>0</td></tr>
</table>

Apache Ant, an example of a well-encapsulated topic is T_{Select}=*{selector, add, select, filename, basedir, verify, algorithm, cach, valu}*. The latter has the respective high proportions 0.952 and 0.960 in the vectors of the packages *ant.types.selectors.**, *ant.types.selectors.modifiedselector*. These two packages have been assigned to the same cluster during the clustering process. In JHotDraw, an example of a well-encapsulated topic is the following: T_{App}= *{action, evt, valu, applic, app, perform, init, project, listen}*. The latter has respectively the proportions 0.987 and 0.999 in the vectors of the packages contained in the cluster *{app.**, *app.action}*. This cluster is cohesive judging from the documentation of the packages it contains.

When not using thresholds to filter LDA results leads to unexpected matches

We observed in very rare cases, an unexpected match between the resulting lexical clusters and the package structure. In particular, with the JHotDraw system, the execution of LDA notably results in the generation of the topic T_{obu}={*edit, valu, chang, property, redo, undo, add, listen, composit*} which is dominant in the packages *io*, *beans* and *undo*. These packages are assigned to the same cluster and they all match the package structure. The clustering of *beans* and *undo* together seems logic since they both offer a support for property change

listeners (Randelshofer, 2015). However, since *io* does not seem to share common responsibilities with the packages *beans* and *undo*, the assignment of *io* to the same cluster comprising these two packages seems problematic. A further investigation of the factors behind the creation of the cluster {*io, beans, undo*}, showed that it is only the keyword "*add* " of the topic T_{obu} which appears, in relatively small proportions, in the vocabulary of the package *io*. To solve this issue we could define a threshold value indicating whether a keyword belongs to its associated topic or not; i.e., when a keyword has a probability value that is less than the threshold value then that keyword is not an indicator of that topic (Maskeri et al., 2008).

We also found an instance of an unexpected match in JFreeChart. In this instance, the packages *chart.panel* and *chart.event* are grouped together by the clustering algorithm. But even if both of these packages correspond to the package structure, their lexical information are orthogonal since all the topics they have in common appear in both of them in very small proportions (i.e. in a proportion <0.05). Each of these two packages has a distinct dominant topic that does not appear in the other packages of JFreeChart; i.e., both packages have a solitary topic. In this case, the cluster comprising the two packages would not have been created if we had set a threshold on the proportions of topics to be considered when computing the conceptual coupling between packages.

6.2.1.2 When do the lexical clusters not match the package structure?

When the responsibilities are poorly distributed among packages

A poor distribution of a system's responsibilities can lead to the scattering of similar responsibilities throughout packages belonging to different parts of the package hierarchy. In this case, the clusters created by the clustering algorithm can group together these packages exhibiting similar responsibilities into cohesive clusters. This might lead to a mismatch with the package structure. In the case of JHotDraw for instance, the two packages *nanoxml* and *net.n3.nanoxml* are assigned to the same cluster which exhibits cohesive responsibilities in regard to the parsing of XML files. However, these two packages do not belong to the same

package hierarchy. This suggests that a refactoring might be done to either put these two packages in the same hierarchy or to merge their classes so as to create a single XML parser package. The designers of JHotDraw have preferred the second option since in the subsequent versions of JHotDraw (7.1 or higher), the package *nanoxml* does no longer exist and its content has been moved to *net.n3.nanoxml* (Randelschofer, 2015). Likewise, in JUnit 4.10, our lexical clustering returns the cluster {*org.junit.*, junit.framework*}. Since *org.junit.** provides JUnit core classes and annotations and corresponds to *junit.framework* in JUnit 3.x (JUnit, 2015), both of these packages then implement similar responsibilities. They are therefore appropriately clustered together by our clustering algorithm. Nevertheless, the designers of JUnit have chosen to keep apart the legacy *junit.framework* namespace used with JUnit 3.x.

When a topic spans many packages of the system

A topic that spans many parts of the system while covering very few elements within each part is called a cross-cutting topic (Kuhn et al., 2007). In our context, cross-cutting topics can either be the results of cross-cutting concerns or of a LDA calibration problem related to an improper setting of the number of topics *T*. When cross-cutting topics capture a high proportion of the lexical information embedded in some packages, the algorithm generally assigns these packages to the same cluster.

This is the case for Apache Ant with the cross-cutting topic T_{Image}={*oper, draw, add, execut, imag, instr, transform, rectangle, height*}, which derives from a cross-cutting concern (i.e., manipulating an image object). The topic T_{Image} is dominant in the vectors of the packages *ant.types.optional.image* and *ant.taskdefs.optional.image*. This led to the assignment of these two packages to the same cluster C_{Image} therefore clashing with the package structure. The placement of these two packages in distinct spots of the package structure is justified by the fact that they respectively embody the image related types and the task that allows performing image manipulation operations on existing images (Apache, 2015).

In the cases of Apache Ant and JHotDraw, the discrepancy between the best clustering results and the package structure are mostly due to cross-cutting topics that result from an improper LDA calibration and specifically from an inappropriate setting of the number of topics *T*. When the value of *T* is too low, it leads to topics that capture too many concepts i.e. concepts coming from different parts of the system. The predominance of such a topic(s) within the respective topic proportion vectors of a subset of packages generally leads to the assignment of these packages to the same cluster. However, the so-obtained cluster lacks cohesion and does not even address cross-cutting concerns. For instance, the presence of cross-cutting topics derived from the Apache Ant's lexical information notably leads to the creation of the cluster C_1={*ant.util.facade, ant.taskdefs.optional.extension.resolvers*} which lack cohesion and causes a mismatch with the package structure. In the C_1's packages, the most dominant topic is the cross-cutting topic T_{13}={*vaj, event, messag, tool, finish, pattern, project, util, send*}. However, this topic does not convey meaningful cross-cutting information.

When octopus topics lead to a mismatch with the package structure

In JFreeChart, the octopus topic T_{Time}={*millisecond, period, time, calendar, zone, bound, domain, includ, year*} is found in high proportions in *data.time.** (0.8) and in very few proportions in *chart.axis* (0.087) and *data.time.ohlc* (0.09). As the package *data.time.ohlc* is already assigned to the cohesive cluster {*data.xy, data.time.ohlc*}, this prevents the creation of a cluster containing both *data.time.** and *data.time.ohlc*. Since *chart.axis* and *data.time.** have both in common T_{Time} and the proportions of the other topics they have in common are insignificant (i.e. <0.05) in both packages, the clustering algorithm then assigns them to the same cluster even if they convey mostly unrelated responsibilities. This is also the case in Apache Ant, where the packages *ant.taskdefs.optional.sitraka.bytecode.attributes* and *ant.taskdefs.optional.metamata* are clustered together due to the presence of an octopus topic.

When solitary topics lead to a mismatch with the package structure

In JUnit, the packages assigned to the cluster {*org.junit.rules, org.junit.internal.runners.statements*} have respectively high proportions of the solitary

topics T_6 and T_8. The proportion of T_6 within *org.junit.rules* is 0.761. Besides, the proportions of T_6 is insignificant (<0.05) within the other packages. The proportion of T_8 within *org.junit.internal.runners.statements* is 0.97. The only other package in which T_8 is not insignificant is *org.junit.rules* where its proportion is very low i.e. 0.2. We therefore expected that the algorithm will respectively assign these two packages to singletons clusters. However, since the CCF (the Conceptual Cluster Factor as defined in Chapter 5) of singleton clusters is zero, and both packages have the same topic T_8 in common, the algorithm assigns them to the same cluster. This is meant to create a cluster having a CCF value higher than zero, which increases the overall value of LQ.

6.2.2 Analysis of the best layering results per system and setup (EQ2)

As indicated in Section 6.1, we have performed 50 independent runs for each of the algorithms applying the SAHC. This ensures that the results reported in this thesis are not fortuitous. Thus, we have executed the LI approach (i.e. the layering hybrid approach presented in Chapter 5 and which uses both lexical and structural information) 50 independent times. In other words, given each best solution generated by our clustering algorithm using a given LDA combination (i.e. the best solution among the 50 solutions generated for a given LDA combination among the 225 combinations), and for each of the five considered setup, we have executed the LI approach 50 independent times. Likewise, for each of these setups, we have also run the NLI approach (i.e. the layering approach presented in Chapter 2 which does not rely on lexical information) 50 times.

Table 6.3 indicates the corresponding average and standard deviation of the Precision, Recall and F-Measure for each analyzed system and each of the five setups factor. Table 6.3 shows that, for some setups, the mean and standard deviation relative to the LI are not always higher than the NLI's ones: this is notably the case for JFreeChart (e.g., setup 1). Furthermore, this Table also shows that the fluctuations of the precision, recall and F-measure are much higher for the LI than the NLI. The explanation to these two observations lies in the fact that with the NLI, for a given setup, the average and standard deviation are computed based on 50

runs. With the LI, these layers' dependency attributes are in turn computed based on 50*225 runs (i.e. 50 runs for each of the 225 LDA combinations' best clustering results). The independent runs of the lexical clustering may help improve or deteriorate the layering results obtained with the LI approach depending on the quality of the resulting clusters. And as we have discussed in Section 6.2.1, an inappropriate calibration of LDA can lead to very poor clustering results and therefore to very sub-optimal layering solutions. We further discuss the calibration of LDA in the Section 6.2.4.

Table 6.3 Average and standard deviation of the layering results obtained with and without lexical information for the 3 factors setups

			JHotDraw		**Apache Ant**		**JUnit 4.10**		**JFreeChart**	
			LI	NLI	LI	NLI	LI	NLI	LI	NLI
Setup 1 ap = 0, ip =1, sp = 2, bp = 4	Precision	Average	64.78	69.83	74.14	72.89	55.84	54.64	48.75	52.17
		Standard Dev.	9.49	6.10	5.50	1.32	7.49	4.16	6.04	5.93
	Recall	Average	64.78	69.83	74.14	72.89	55.84	54.64	46.11	52.17
		Standard Dev.	9.49	6.10	5.50	1.32	7.49	4.16	5.71	5.93
	F-measure	Average	64.78	69.83	74.14	72.89	55.84	54.64	47.39	52.17
		Standard Dev.	9.49	6.10	5.50	1.32	7.49	4.16	5.87	5.93
Setup 2 ap = 0, ip = 1, sp = 2, bp = 15	Precision	Average	63.28	68.33	72.09	74.53	65.26	61.14	47.02	58.11
		Standard Dev.	9.68	4.6	6.95	1.45	6.68	9.83	4.79	5.27
	Recall	Average	63.28	68.33	72.09	74.53	65.26	61.14	44.47	58.11
		Standard Dev.	9.68	4.6	6.95	1.45	6.68	9.83	4.53	5.27
	F-measure	Average	63.28	68.33	72.09	74.53	65.26	61.14	45.71	58.11
		Standard Dev.	9.68	4.6	6.95	1.45	6.68	9.83	4.65	5.27
Setup 3 ap =0, ip =2, sp = 1, bp = 4	Precision	Average	79.45	66.33	66.61	61.70	49.47	50.35	45.48	46.57
		Standard Dev.	14.73	15.14	7.47	1.36	6.90	4.90	4.98	1.44
	Recall	Average	79.45	66.33	66.61	61.70	49.47	50.35	43.02	46.57
		Standard Dev.	14.73	15.14	7.47	1.36	6.90	4.90	4.71	1.44
	F-measure	Average	79.45	66.33	66.61	61.70	49.47	50.35	44.21	46.57
		Standard Dev.	14.73	15.14	7.47	1.36	6.90	4.90	4.84	1.44
Setup 4 ap =0, ip =2, sp = 1, bp = 15	Precision	Average	77.74	69.83	63.68	66.98	58.57	50.50	48.54	53.31
		Standard Dev.	14.36	13.11	11.58	2.34	7.93	3.38	5.88	7.80
	Recall	Average	77.74	69.83	63.68	66.98	58.57	50.50	45.91	53.31
		Standard Dev.	14.36	13.11	11.58	2.34	7.93	3.38	5.56	7.80
	F-measure	Average	77.74	69.83	63.68	66.98	58.57	50.50	47.19	53.31
		Standard Dev.	14.36	13.11	11.58	2.34	7.93	3.38	5.71	7.80
Setup 5 ap =0, ip =2, sp = 1, bp = 20	Precision	Average	74.84	69.25	59.62	66.56	60.06	51.07	47.75	59.77
		Standard Dev.	15.10	12.95	12.09	2.62	7.78	3.96	5.33	8.05
	Recall	Average	74.84	69.25	59.62	66.56	60.06	51.07	45.16	59.77
		Standard Dev.	15.10	12.95	12.09	2.62	7.78	3.96	5.04	8.05
	F-measure	Average	74.84	69.25	59.62	66.56	60.06	51.07	46.42	59.77
		Standard Dev.	15.10	12.95	12.09	2.62	7.78	3.96	5.18	8.05

Given the 50 independent runs that respectively generate a layering solution, we consider the solution having the lowest LaQ as the best layering result. Table 6.4 shows the best layering results obtained for the analyzed systems respectively with the LI approach, and with the NLI

approach. For each setup, Table 6.4 reports these results in terms of precision, recall, F-measure and number of layers in the resulting layered architectures. The greyed cells indicate the F-measure values of the best results obtained using the two approaches (LI and NLI).

Table 6.4 Layering results with and without lexical information for the 5 setups of factors

		JHotDraw		**Apache Ant**		**JUnit 4.10**		**JFreeChart**	
		LI	NLI	LI	NLI	LI	NLI	LI	NLI
Setup 1 ap = 0, ip =1, sp = 2, bp = 4	**Precision**	75	70.83	79.10	74.62	60.71	57.14	57.14	57.14
	Recall	75	70.83	79.10	74.62	60.71	57.14	54.05	54.05
	F-measure	75	70.83	79.10	74.62	60.71	57.14	55.55	55.55
	Nb layers	3	3	3	3	3	3	3	3
Setup 2 ap = 0, ip = 1, sp = 2, bp = 15	**Precision**	66.66	70.83	83.58	76.11	75	60.71	62.85	62.85
	Recall	66.66	70.83	83.58	76.11	75	60.71	59.45	59.45
	F-measure	66.66	70.83	83.58	76.11	75	60.71	61.10	61.10
	Nb layers	3	3	3	4	3	3	3	3
Setup 3 ap =0, ip =2, sp = 1, bp = 4	**Precision**	91.66	87.5	74.62	59.70	53.57	53.57	51.42	48.57
	Recall	91.66	87.5	74.62	59.70	53.57	53.57	48.64	45.94
	F-measure	91.66	87.5	74.62	59.70	53.57	53.57	49.99	47.22
	Nb layers	4	4	3	4	3	3	3	3
Setup4 ap =0, ip =2, sp = 1, bp = 15	**Precision**	91.66	87.5	73.13	68.65	53.57	53.57	57.14	60
	Recall	91.66	87.5	73.13	68.65	53.57	53.57	54.05	56.75
	F-measure	91.66	87.5	73.13	68.65	53.57	53.57	55.55	58.32
	Nb layers	4	4	3	3	4	3	3	3
Setup 5 ap =0, ip =2, sp = 1, bp = 20	**Precision**	91.66	87.5	73.13	67.16	64.28	57.14	57.14	68.57
	Recall	91.66	87.5	73.13	67.16	64.28	57.14	54.05	64.86
	F-measure	91.66	87.5	73.13	67.16	64.28	57.14	55.55	66.68
	Nb layers	4	4	4	4	3	3	3	3

Overall, Table 6.4 (see the greyed cells) shows that the LI approach yields high F-measure values for most of the analyzed systems; i.e., the average of the F-measure values is around 70% with a maximum value that reaches 91.66%. The recovered architectures are therefore very close to the authoritative decompositions of the analyzed systems. Of course, we need more experiments in order to consider generalizing this trend. The number of layers comprised in these results varies from 3 to 4 layers[23]. Time-wise, the LI approach needs on average, depending on the systems, 3 to 125s while the NLI requires from 2 to 11s. Noteworthy, since our focus is on the quality of the results so neither approach was particularly optimized for time efficiency.

---

[23] Though the returned number of layers is low, the search space for the solution is as large as L^P where L is the number of layers and P the number of packages.

Figure 6.2 illustrates as an example, an excerpt of the recovered layering of Apache Ant 1.6.2 which comprises three layers. Note that this layering was built using setup 2, 15 as the topics number, 0.325 as α and 0.55 as β. In this layering, the uppermost layer encompasses the packages which implement the tasks supported by the system. The middle layer contains the packages in charge with the core functionalities as well as the definitions of the supported types. The lowest layer comprises in turn the packages handling the utilities functionalities.

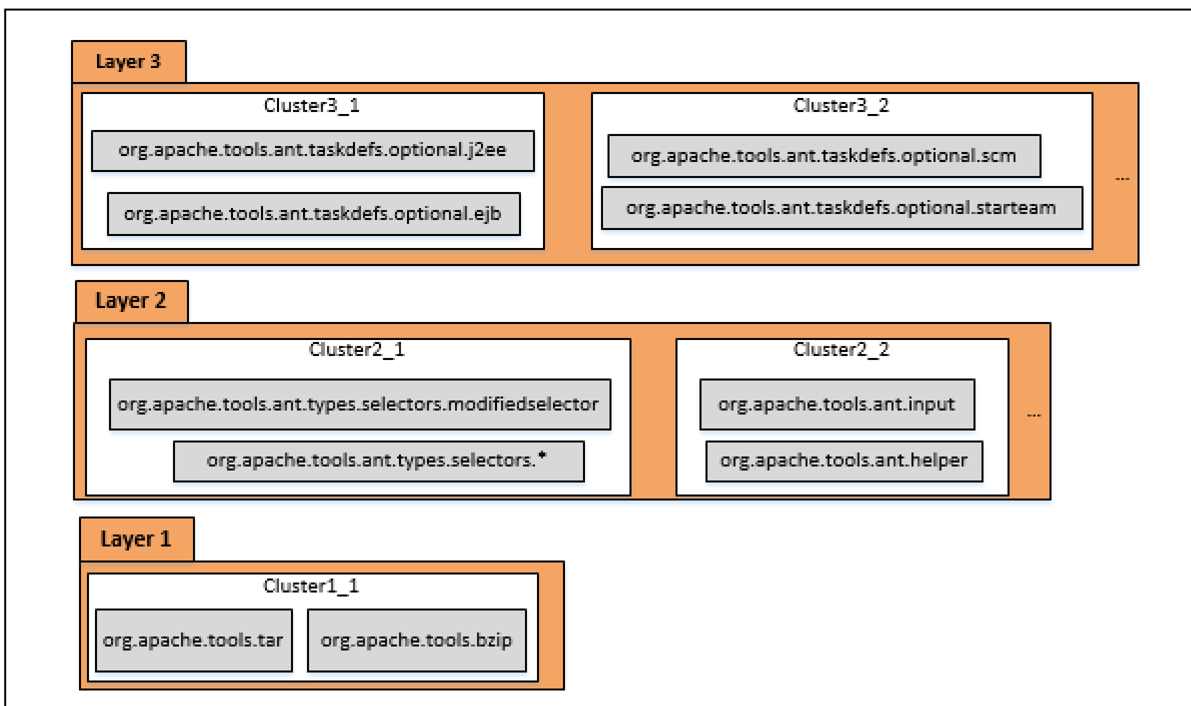


Figure 6.2 An excerpt of the recovered layering of Apache Ant 1.6.2

The best recovered layering for both Apache Ant and JUnit is obtained using setup 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 systems are frameworks that target different platforms and, thus, portability is one of the concerns that drive their design. Using the LI approach and setup 1, the F-measure increased from 60.71% to 75% for JUnit and from 76.11% to 83.58% for Apache Ant.

JHotDraw is a system proposed as an example for a well-designed framework that targets reuse as it supports designers in building their own drawing editors. Thus, we assumed that setups which favored adjacent dependencies and skip-calls over intra-dependencies and back-calls (e.g., setups 4 and 5) would produce the correct layering of JHotDraw. This assumption proved to be true since, as displayed by Table 6.4, using structural information only (NLI) or combining it with lexical information (LI), the most accurate layering is returned with such setups (i.e. setups 3, 4 and 5). Besides, combining both the lexical and structural information produces a more accurate layering for these setups (F-measure value of 91%).

For software systems such as JFreeChart that include a high number of cyclic dependencies, we expected that setups that penalize enough back-calls and skip-calls compared to intra-dependencies (as in setups 1 and 2) would allow to build the most accurate layering result; i.e. we assume that packages involved in a cyclic dependency contribute to the same responsibility and setups 1 and 2 will assign these packages to the same layer. This assumption is corroborated by the LI approach with which the most accurate layering is generated with setup 2 (F-measure value of 61.10%). However, as displayed in Table 6.4, using structural information only, the most accurate layering for JFreeChart is returned by setup 5 (F-measure value of 66.68%). Therefore, when using only structural information, unless we give a very high value to the back-call factor *bp* compared to intra-dependency factor *ip*, and penalize enough intra-dependencies compared to skip-calls, some packages that contribute to the same responsibility are assigned to different layers. Besides, the layering results of JFreeChart also show that the LI approach produces better or identical results than the NLI approach for three out of the five setups (i.e. setups 1, 2 and 3). However, using the two remaining setups (i.e setups 4 and 5), the LI fails to outperform the NLI approach. This might be due to the calibration of the LDA parameters used when running the LI approach. As discussed earlier, an inappropriate calibration of LDA can lead to very poor clustering results and therefore to very sub-optimal layering solutions.

Based on the results yielded by all setups on all the analyzed systems, the LI approach provide a slight amelioration to the layering results compared to the NLI approach for all the

analyzed systems except JFreeChart. These are interesting results that suggest a positive answer to our second experimentation question.

6.2.3 Comparison with another layering recovery approach (EQ3)

To check whether our hybrid recovery approach (i.e. LI) outperforms other recovery approaches, we have compared the results of our approach to those generated using the Lattix tool (Sangal et al., 2005a; Sangal et al., 2005b). In particular, Lattix is a tool which enables the visualization of a system's organization under the form of a dependency structure matrix (DSM). Briefly, Lattix assigns strongly connected packages to the same layer. Depending on the quality of the system at hand, this might generate architectures with either very few layers or too many layers. Lattix proposes various algorithms to support the architecture recovery process. To recover the layered architectures of the analyzed systems, we used the Lattix's reachability algorithm (Sangal et al., 2005a) and we made a comparison between its layering results and those obtained with our LI approach. We focus on the reachability algorithm because it supports architecture discovery by identifying layers and independent subsystems[24].

It is worth pointing out that, to the best of our knowledge, our work is the first to make such a comparison with other layering approaches. In spite of the existence of a number of layering approaches, the associated tools are generally not available. And even when they are, their configuration as well as their use to analyze the same systems is very tricky since they usually have their own specific internal representations of the system's data.

For each analyzed system, Table 6.5 indicates the high level results obtained using Lattix, the LI approach and the NLI approach in terms of precision, recall and F-measure. For three out of the four analyzed systems (i.e., JHotDraw, Apache Ant and JUnit), the LI approach

---

[24] http://cp.lattix.com/help/v9/usermanual/algorithms

outperforms Lattix. This is due to the fact that the lexical clustering used in our LI approach yields a better grouping of the responsibilities of these systems than the one based on cyclic dependencies as performed by Lattix (i.e., Lattix puts packages involved in cyclic dependencies in the same layer). Besides, the NLI approach, which does not include the lexical clustering, outperforms Lattix for the same systems (i.e., JHotDraw, Apache Ant and JUnit). For JFreeChart, Lattix outperforms the LI and NLI approaches. This is due to the fact that JFreeChart comprises a high number of cyclic dependencies. However, we assume that increasing the gap between the *ip* and *bp* factors in setup 2 would produce a more accurate layering for JFreeChart. We plan to investigate this in future work.

The weak results of LI and NLI in JFreeChart in comparison to Lattix, do not allow us to generalize our findings that LI and NLI perform better than this tool. This calls for more experiments on more systems to identify the additional conditions explaining these results.

Table 6.5 Precision, recall and F-measure using the Lattix, the NLI and the LI approaches

		Precision	***Recall***	***F-Measure***
JHotDraw	**LI**	91.66	91.66	91.66
	NLI	87.5	87.5	87.5
	Lattix	66.66	66.66	66.66
Apache Ant	**LI**	83.58	83.58	83.58
	NLI	76.11	76.11	76.11
	Lattix	68.65	68.65	68.65
JUnit 4.10	**LI**	75	75	75
	NLI	60.71	60.71	60.71
	Lattix	57.14	57.14	57.14
JFreeChart	**LI**	61.10	61.10	61.10
	NLI	68.57	64.86	66.68
	Lattix	74.28	74.28	74.28

Notice that both the results of Lattix and our approach can be improved through interactions with users (i.e., users may give some hints or constraints on the assignment of packages). Thus, the two approaches could complement each other.

6.2.4 Influence of the calibration of the LDA parameters on the layering results

As Binkley et al. pointed out (Binkley et al., 2014), the optimal calibration of the LDA parameters is hard to achieve. For this reason, we considered a broad range of LDA parameters. We therefore carried out our experiments using different combinations of the LDA parameters: 225 combinations to be more specific. Some of these LDA combinations yield better layering results than those generated by the structural-based layering technique i.e. the NLI approach. Let us for instance consider the setup 2 as well as the three systems for which this setup is expected to yield the most accurate layering results—namely JUnit, Apache Ant, and JFreeChart. The corresponding percentages[25] of the LDA configurations that produce better results than the structural approach are 100%, 36.88% and 8.44%, respectively. These percentages indicate that, in the case of JUnit, the LI approach systematically outperforms the NLI approach regardless of the considered LDA combination used in these experiments. This trend is less pronounced in the two other systems (i.e. Apache Ant, and JFreeChart) for which the percentages are much lower. In the case of JHotDraw, the F-measure of NLI was already really high (87.5%). For this system, the percentage of the LDA combinations that produce better results than the NLI approach is 17.33% for each of the setups 3, 4 and 5. Even though these different percentages yielded by the studied systems are encouraging, they also indicate that there is still a room for improvement in terms of the calibration of the LDA parameters.

We observed that many of the best F-measure, when using LI, came from solutions that did not have the lowest LaQ. For instance, considering LI and the best setups for each system, we recorded F-measures of 85, 82, 72 and 96% respectively for Apache, JUnit, JFreeChart, and JHotDraw. These numbers represent improvements (i) over the results of LI reported in Table 6.4 (up to 6% for JHotDraw and Apache, and 22% for JFreeChart and JUnit for most of the 5 setups), and (ii) over the best results of NLI (more than 16%, in average for most of

---

[25] These percentages are computed based on the best layering solution found over the 50 multiple runs performed for the same setup.

the 5 setups). In other words, the potential of the key idea used in LI, which is grouping together in a meaningful way some packages before the layering, is not fully translated in the results when only solutions with the best LaQ are considered. This means that, while our results are promising, future work is definitely needed to further investigate the best LDA configurations. Interestingly, a first preliminary analysis indicates that a small set of LDA values is enough to produce the best F-measure for all the systems at hand. These parameters are the following: Topics in {30, 40}, α in {0.325, 0.775} and β in {0.1, 0.55, 0.775}.

6.3 Threats to validity

As Wohlin et al. explain in (Wohlin, 2012), the notion of validity refers to the trustworthiness of the results as well as to the extent to which the results are true and not biased by the subjectivity of the researchers. In the literature, many schemes have been proposed to classify the different aspects of validity and threats to validity. In this classification, we focus on some validity's aspects widely used in the literature (e.g., (Wohlin, 2012; Kitchenham, 2004; Yin, 2009)), namely: the conclusion validity, the internal validity and the external validity.

The Internal Validity threat: it concerns the examination of causal relationships. In this regard, one factor that might threaten the internal validity of our study is the quality of the linguistic information used when carrying out the experminents. The quality of our results is tightly coupled to the quality of the linguistic information retrieved in the analyzed systems. In our case, we can expect good linguistic information from the analyzed systems whose quality of linguistic information is good enough. The former statement is notably corroborated by (Corazza et al., 2016) who also analyzed the lexicons of various systems including JHotDraw and JFreeChart. Besides, for this study, we selected the SAHC as a simple search-based technique that has been used by other researchers in the domain. But, it is a technique more susceptible than others (such as Simulated Annealing, Tabu etc.) to get stuck in local optima. We plan, in future work, to explore other options. The stochasticity of the hill climbing is another threat to the validity of our approach. The randomness of the hill

climbing can drive it to produce a different solution from one run to another. We mitigated this threat by performing 50 independent runs of each algorithm for each analyzed system.

The external validity threat: it concerns the applicability/generalization of the effects observed in the study to other studies. At this level, note that our preliminary experiments rely on a number of parameters which are not always easy to calibrate. Besides, our experiments involve a small subset (four) of open-source systems that were known to be layered systems. This hinders the applicability of our approach to other contexts and call for more experiments with different systems (e.g., industrial legacy systems) and deeper exploration of the parameters, especially those of LDA. To overcome this issue, we plan to investigate the use of parameter control based algorithms that allow setting initial values of the algorithm parameters and automatically adjusting these values during the run (Smith, 2008; Karafotias et al., 2014). Finally, one issue that might impede the generalizability of our approach is its performance. However, we have mitigated this issue by analyzing medium size systems such as Apache Ant and JFreeChart. In addition, the fact that our approach is able to cluster and layer the analyzed systems within seconds indicates that it is a performant approach.

The Conclusion validity threat: it concerns the relationships we draw from the analysis of our data. One factor that threatens the conclusion validity of our work is the way we evaluated the quality of our layering results. The lack of ground-truth architectures or of experts able to produce these architectures impedes the comparison or the assessment of the techniques used by these approaches (Garcia et al., 2013; Saiedi et al., 2015). We therefore assessed the quality of the recovered layerings by relying on authoritative decompositions coming from two sources. These sources are the following: published analyzed systems' authoritative architectures for 2 systems and manual decompositions coming from us. This might lead to issues with bias and competency. However, our experience and knowledge of the analyzed systems mitigate this threat. In addition, we made the authoritative decompositions used in our experiments available at (Authoritative decomposition, 2015). Hence, those who access the repository might therefore comment on its content. Their

suggestions might then be very helpful to adjust the content of the repository. In future work, we plan to use these decompositions to help populating a repository that will house the authoritative architectures used in all our ongoing experiments. Another factor that might threaten the conclusion validity is the fact that we rely on the C2SC measure to assess the similarity between our clustering results and the package hierarchy. Nevertheless, as explained in Section 6.1, The C2SC measure is a very strict measure of such resemblance. The formulation of this measure could therefore affect some of the conclusions drawn from our experiments. The usage of less conservative and more nuanced measures in future work will provide different perspectives.

6.4 Chapter summary

In this chapter, we have assessed our structural and lexical based layered architecture recovery approach (Chapter 5) by performing different experiments on four open-source systems. To this extent, we have described the experimental calibration and implementation, as well as the results obtained when applying this hybrid recovery approach on the four studied systems. We have also compared these results to those obtained using our structural-based approach, on one side, and another layering recovery technique, on the other side. These experiments showed that our hybrid layering approach yielded good preliminary results and proved for three out of four of the analyzed systems that a structural and lexical based layering technique can outperform a structural-based one as well as another well established layering technique. These experimentations also showed that our approach can capture a system's responsibilities into manageable and conceptually cohesive clusters as corroborated by the manual analysis of the source code and the documentation of the analyzed systems.

CONCLUSION AND FUTURE WORKS

Conclusion

The architectural reconstruction is a reverse engineering activity aiming at recovering the decisions made when developing a system and that are currently missing because they are either unknown (lack of documentation, departure of original developers) or recent (due to changes in the system). There is a large body of work dedicated to the software architecture reconstruction. Most of it relies on clustering, which is a common used technique to reconstruct architecture. However, most of these approaches target specific languages and systems and do not use a standard representation of the data of the system under analysis. Therefore, the resulting tools are not able to interoperate with each other. Besides, this ad hoc aspect of the existing approaches hinders the reproducibility of the techniques they used. Moreover, most of the layering recovery approaches attempt to recover the layered architecture by relying on heuristics to resolve cyclic dependencies or to layer modules according to the number of their fan-in and fan-out dependencies. However, these heuristics are rarely based on layering rules and may result in architectures that are too permissive with layering violations.

To tackle the limitations of these approaches, we have proposed an approach that exploits the rules of the layered style to recover the architectures of object oriented layered systems. The contribution of this thesis lies in the proposal of an approach having the following advantages: 1) it proposes a set of layers' dependency attributes and constraints embodying a minimal set of rules that a layered system must conform to; 2) it formalizes the layering recovery problem as an optimization problem; 3) it is a language and platform independent since it relies on the OMG's standard for software modernization (i.e., the KDM specification standard); and 4) it is implemented as a tool that is available for the software engineering community in the form of an Eclipse plugin.

Software systems are usually built by combining and composing architectural styles. Thus, proposing a set of layers' dependency attributes and constraints that convey the essence of a minimal set of layering rules allows defining a recovery approach that leverages the architectural style that was used to create the analyzed layered systems.

By translating the layered architecture recovery problem into an optimization problem, we exploit the set of layers' dependency attributes and constraints derived from the layering rules to propose a recovery approach that reflects how strictly the architect enforced the layering rules when designing her system. Moreover, this translation enables to model the layered architecture recovery problem as a quadratic assignment problem (QAP). The latter is a well-established combinatorial optimization formulation which has been used to model problems such as layout design or resource allocation. This translation also allows avoiding the reliance on heuristics to resolve cyclic dependencies while recovering the layered architectures. The reliance on these heuristics might result in: 1) architectures comprising very few layers if the heuristics rely for instance on highly connected modules; or 2) architectures comprising too many layers if the heuristics rely for instance on the resolution of cyclic dependencies.

Furthermore, the reliance on the OMG's standard for software modernization (i.e., the KDM specification standard) makes our approach language and platform independent. This standard provides a common interchange format that supports the representation of the analyzed systems' artifacts in a standardized manner, thereby ensuring the interoperability between resulting tools.

To automate our approach, we implement it as a tool that is available for the software engineering community in the form of an Eclipse plugin. This tool provides the following main functionalities: 1) extracting facts from the analyzed software system and using a standard and platform-independent representation of the system; 2) performing the layering using these facts; 3) visualizing the resulting layering architecture and related layers'

dependency attributes; and 4) refining the so-obtained layering results. Our tool is useful to understand and document layered systems and to detect layering violations.

Note that the proposed recovery approach was notably published at the Software Engineering and Knowledge Engineering Conference (Boaye Belle et al., 2013), the European Conference on Software Architecture (Boaye Belle et al., 2015), and at the Information and Software Technology Journal (Boaye Belle et al., 2016).

Limitations

In this section, we present some limitations of our work.

Local optimum issue

To recover the layered architecture of software systems, we relied on two search-based algorithms, namely: the tabu search and the hill climbing algorithms. A current issue in the reliance on search-based algorithms such as these two is the likeliness to find local optimal solutions. To mitigate this limit, we have: 1) used a random partition as the starting point of each of our algorithms—the resulting layering solution may vary from a run to another; 2) we have run each of these algorithms multiple times with the same parameters and kept the solution yielding the best results as the best solution; and 3) in the case of the hill climbing algorithm, we have performed an intensified search to further explore the search space. To overcome the non-optimality issue, another solution would have been to adapt another heuristic-based approach such as the simulated annealing (Kirkpatrick et al., 1983) which avoids being trapped in local minima by tolerating the degradation of a solution according to a given probability.

Clicours.COM

Calibration of the algorithms used to recover the layered architectures

Our experiments rely on a number of parameters which calibration is not always obvious and they involve a few open-source systems that were known to be layered systems. This impedes the generalizability of our approach and call for more experiments with different systems and deeper exploration of the parameters, especially those of LDA. To overcome this calibration issue, a solution would be to investigate the use of parameter control based algorithms that allow setting initial values of the algorithm parameters and automatically adjusting these values during the run (Smith, 2008; Karafotias et al., 2014). Note that two techniques allowing setting the parameters of meta-heuristics have been proposed so far: parameter tuning and parameter control (Smith, 2008; Arcuri and Fraser, 2013). Parameter tuning consists at finding the adequate values of parameters before running the algorithm and at keeping these same values during the execution. Parameter control aims in turn at assigning initial values to the parameters and at automatically adjusting these values during runtime. Parameter control presents many advantages over parameter tuning: it is less time-consumming (Smith, 2008) and alleviates the user from the burden of finding the appropriate parameters' values before running the algorithm (Karafotias et al 2014). As future work, we then intend to investigate parameters control based techniques.

The coverage of a subset of rules

In this thesis, we have exploited the rules related to the abstraction and the responsibility in order to recover layered architectures. However, we did not exploit other identified rules. Recall that during our analysis of the layered style, three other rules have emerged, namely: the transversality and the protection against variations rules.

Combining the abstraction and the responsibility rules with the transversality and the protection against variations rules may lead to a more robust recovery approach. In particular, using the transversality rule may help handling issues related to library components also called omnipresent modules. Library components obscure the system's

structure if considered during the decomposition of the system (Müller et al., 1993). These components could have been clustered into a vertical layer (called transversal layer in (Clements et al., 2003)) to ease the recovery of the layered architecture.

The use of a limited number of sources of information

In this thesis, we have used two types of sources of information to perform the recovery: the structural and the lexical ones. While these sources of information constituted valuable input to the recovery process, other sources of information could have been plugged in the recovery process to enrich the quality of its outcomes. Among them are the human expertise and the dynamic information (Ducasse and Pollet, 2009). The human expertise is one of the most valuable sources of information when recovering architecture since it allows guiding, validating and improving the recovery results. The dynamic information allows understanding the behaviour of a system by providing a good insight to its runtime nature (Ducasse and Pollet, 2009). Even though the dynamic information is rarely used to recover static views such as the layered view, it could have helped refining the relationships between the software entities involved in the recovery process.

Future works

In this section, we describe some possible extensions to our work.

Recovery of the architectures of domain-specific systems

A possible extension of our work could be to experiment on domain-specific systems and find out if particular setups (i.e., a set of factors) are related to specific domains or classes of systems. Each specific domain may emphasize on a specific range of quality attributes. The diverses natures of these quality attributes might lead to a relaxation or on the contrary to a tightening of the application of the rules of the layered style. In this context, the availability of such domain-specific systems and some architecture description of these systems is a

challenging issue. To carry out this extension of our work, the taxonomy of software types by Forward and Lethbridge (Forward and Lethbridge, 2008) could be helpful since it notably compiles a list of application domains.

Recover architectures compliant to other architectural styles

To widen the scope of our thesis beyond the layered style, we intend to adapt it so as to recover architectures compliant to other architectural styles. To this end, to recover the architectures compliant to a given architectural style, we could follow the same process than with the layered style. Hence, we could analyze the target architectural style to extract the set of rules that it embodies. Then, we could derive attributes and constraints from these rules. The attributes and constraints could therefore be used to formalize the recovery of this architectural style's compliant architectures into an optimization problem. Works such as (Tibermacine et al., 2011; Buschmann et al., 2007; Zdun and Avgeriou, 2008) provide good insights on the architectural constraints and could therefore be helpful in this formalization process.

Use the recovery results to initiate a refactoring process

Software systems are practically built by combining and composing architectural styles. However as a software system ages, its as-built architecture progressively deviates from the initial style that guided its design. This is notably due to the increasing presence of violations of the style constraints. These violations obscure the structure of the system's architecture and they make it deviate from the quality attributes supported by the architectural style that guided its design. As pointed out by Schmidt et al (Schmidt et al., 2012), this erosion of the system's design makes it brittle, immobile, viscous, opaque, rigid and leads to a clash with the system's conceptual architecture. To realign the conceptual architecture with the as-built one, a refactoring process aiming at getting rid of these violations should be initiated. For this purpose, a recovery process allowing the generation of the system's as-built architecture should be performed first. This recovery process will also allow, in our context, to spot

different architectural violations such as the back-calls and the skip calls, and propose refactoring operations accordingly.

APPENDIX I

LAYERING RESULTS OBTAINED WITH THE STRUCTURAL-BASED APPROACH

Results obtained with the structural *SAHCLayering* algorithm (Chapter 4)

Table 6 LaQ variations with the SAHCLayering algorithm applied on Apache 1.6.2

			Mean	StDev	Min
Apache Ant 1.6.2	Scenario 1	*Setup 1*	1035.04	6.90	1026.0
		Setup 2	1126.78	8.72	1120.0
		Setup 3	1182.6	12.71	1163.0
		Setup 4	1226.16	13.09	1216.0
		Setup 5	1290.86	23.86	1262.0
	Scenario 2	*Setup 1*	458.52	19.43	436.0
		Setup 2	647.2	20.29	628.0
		Setup 3	809.92	15.85	790.0
		Setup 4	908.0	6.67	894.0
		Setup 5	954.24	6.63	942.0
	Scenario 3	*Setup 1*	571.92	2.96	569.0
		Setup 2	619.88	1.40	619.0
		Setup 3	668.58	12.79	664.0
		Setup 4	745.6	99.64	701.0
		Setup 5	789.52	105.13	739.0
	Scenario 4	*Setup 1*	17.68	38.35	0.0
		Setup 2	19.12	62.28	0.0
		Setup 3	63.2	129.25	0.0
		Setup 4	34.44	98.21	0.0
		Setup 5	42.56	90.83	0.0
	Scenario 5	*Setup 1*	21.6	15.00	4.0
		Setup 2	35.6	33.72	4.0
		Setup 3	77.88	215.18	4.0
		Setup 4	131.16	310.80	6.0
		Setup 5	177.26	407.56	8.0

Table 7 LaQ variations with the SAHCLayering algorithm applied on JUnit 4.10

			Mean	StDev	Min
Junit 4.10	Scenario 1	*Setup 1*	260.7	21.32	234.0
		Setup 2	298.84	18.91	291.0
		Setup 3	341.46	21.78	330.0
		Setup 4	389.88	42.56	366.0
		Setup 5	435.68	44.02	402.0

	Scenario 2	***Setup 1***	132.56	1.14	132.0
		Setup 2	209.04	5.20	204.0
		Setup 3	267.08	16.93	248.0
		Setup 4	311.68	12.284	294.0
		Setup 5	369.52	34.05	336.0
	Scenario 3	***Setup 1***	166.52	30.01	148.0
		Setup 2	199.9	17.30	184.0
		Setup 3	238.46	26.63	218.0
		Setup 4	272.2	33.38	247.0
		Setup 5	300.92	58.30	263.0
	Scenario 4	***Setup 1***	14.4	22.86	0.0
		Setup 2	23.92	36.62	0.0
		Setup 3	37.56	62.31	0.0
		Setup 4	33.84	71.31	0.0
		Setup 5	38.4	77.52	0.0
	Scenario 5	***Setup 1***	36.74	43.66	7.0
		Setup 2	52.32	72.80	7.0
		Setup 3	52.5	74.78	7.0
		Setup 4	59.3	91.99	7.0
		Setup 5	81.9	129.79	7.0

Table 8 LaQ variations with the SAHCLayering algorithm applied on JFreeChart 1.0.15

			Mean	**StDev**	**Min**
JFreeChart 1.0.15	**Scenario 1**	***Setup 1***	1941.96	13.80	1938.0
		Setup 2	2439.58	17.73	2436.0
		Setup 3	2907.3	77.25	2864.0
		Setup 4	3196.14	126.63	3071.0
		Setup 5	3425.3	221.44	3174.0
	Scenario 2	***Setup 1***	1182.96	12.28	1160.0
		Setup 2	1859.52	34.92	1808.0
		Setup 3	2343.8	59.12	2278.0
		Setup 4	2763.56	80.09	2692.0
		Setup 5	3005.68	177.41	2870.0
	Scenario 3	***Setup 1***	1447.04	41.16	1406.0
		Setup 2	1698.6	139.21	1562.0
		Setup 3	1786.7	191.92	1631.0
		Setup 4	1843.62	267.78	1660.0
		Setup 5	1908.82	326.50	1684.0
	Scenario 4	***Setup 1***	21.88	17.94	4.0
		Setup 2	37.96	23.45	4.0
		Setup 3	140.44	336.87	4.0
		Setup 4	195.6	375.73	4.0
		Setup 5	302.32	488.64	4.0
	Scenario 5	***Setup 1***	95.98	210.85	8.0
		Setup 2	221.88	300.44	8.0
		Setup 3	373.16	433.17	8.0
		Setup 4	403.74	510.56	8.0

		Setup 5	645.5	513.24	8.0

Table 9 LaQ variations with the SAHCLayering algorithm applied on jEdit 5.0.0

			Mean	**StDev**	**Min**
jEdit 5.0.0	**Scenario 1**	***Setup 1***	1349.1	48.88	1288.0
		Setup 2	1718.18	60.77	1631.0
		Setup 3	1952.72	49.26	1849.0
		Setup 4	2029.9	50.82	1973.0
		Setup 5	2143.58	60.48	2097.0
	Scenario 2	***Setup 1***	817.04	40.79	760.0
		Setup 2	1393.4	45.83	1272.0
		Setup 3	1665.16	67.75	1632.0
		Setup 4	1908.72	72.57	1836.0
		Setup 5	2034.8	140.41	1960.0
	Scenario 3	***Setup 1***	905.44	9.71	893.0
		Setup 2	1157.68	22.34	1120.0
		Setup 3	1254.58	12.40	1231.0
		Setup 4	1345.68	25.35	1316.0
		Setup 5	1418.16	40.31	1361.0
	Scenario 4	***Setup 1***	7.2	11.05	0.0
		Setup 2	9.84	19.23	0.0
		Setup 3	12.36	21.41	0.0
		Setup 4	12.52	18.27	0.0
		Setup 5	18.16	40.32	0.0
	Scenario 5	***Setup 1***	14.26	14.64	5.0
		Setup 2	14.66	14.13	5.0
		Setup 3	14.76	14.34	5.0
		Setup 4	18.54	17.97	5.0
		Setup 5	18.12	30.80	5.0

Table 10 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 60b1

			Mean	**StDev**	**Min**
JHotDraw 60b1	**Scenario 1**	***Setup 1***	607.2	37.44	587.0
		Setup 2	643.76	18.15	638.0
		Setup 3	708.86	69.89	682.0
		Setup 4	749.9	103.28	726.0
		Setup 5	783.16	30.54	770.0
	Scenario 2	***Setup 1***	114.6	18.38	112.0
		Setup 2	207.32	31.34	198.0
		Setup 3	281.92	25.63	276.0
		Setup 4	343.36	22.44	336.0
		Setup 5	391.16	12.22	384.0
	Scenario 3	***Setup 1***	389.94	22.72	383.0
		Setup 2	450.94	74.43	423.0
		Setup 3	469.24	30.59	463.0

		Setup 4	551.0	67.30	503.0
		Setup 5	567.64	24.09	543.0
	Scenario 4	***Setup 1***	24.08	30.79	2.0
		Setup 2	40.32	66.34	2.0
		Setup 3	72.2	116.21	2.0
		Setup 4	28.6	51.30	2.0
		Setup 5	41.68	55.39	2.0
	Scenario 5	***Setup 1***	27.22	36.52	3.0
		Setup 2	42.9	71.34	3.0
		Setup 3	62.92	113.28	3.0
		Setup 4	49.62	96.84	3.0
		Setup 5	83.96	190.07	3.0

Table 11 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.0.7

			Mean	**StDev**	**Min**
JHotDraw 7.0.7	**Scenario 1**	***Setup 1***	569.44	4.96	565.0
		Setup 2	611.16	23.69	580.0
		Setup 3	629.62	41.62	586.0
		Setup 4	646.2	41.86	590.0
		Setup 5	658.74	50.48	594.0
	Scenario 2	***Setup 1***	254.28	4.69	248.0
		Setup 2	328.76	10.18	320.0
		Setup 3	345.12	10.26	336.0
		Setup 4	357.16	14.45	344.0
		Setup 5	365.48	24.27	348.0
	Scenario 3	***Setup 1***	427.8	77.96	382.0
		Setup 2	442.1	58.35	398.0
		Setup 3	458.08	52.45	414.0
		Setup 4	465.26	40.37	421.0
		Setup 5	503.96	87.01	424.0
	Scenario 4	***Setup 1***	55.16	121.58	0.0
		Setup 2	47.92	143.67	0.0
		Setup 3	80.16	203.69	0.0
		Setup 4	63.12	175.89	0.0
		Setup 5	31.8	41.77	0.0
	Scenario 5	***Setup 1***	181.14	158.20	4.0
		Setup 2	225.6	201.02	4.0
		Setup 3	338.34	307.19	4.0
		Setup 4	334.0	267.19	5.0
		Setup 5	404.6	321.85	4.0

Table 12 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.4.1

			Mean	**StDev**	**Min**
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	1938.14	33.52	1910.0
		Setup 2	2298.64	98.81	2120.0

		Setup 3	2465.04	179.89	2199.0
		Setup 4	2590.82	241.33	2255.0
		Setup 5	2730.76	268.15	2333.0
	Scenario 2	***Setup 1***	846.64	6.03	840.0
		Setup 2	1367.92	18.78	1332.0
		Setup 3	1739.56	110.50	1628.0
		Setup 4	1875.52	169.82	1720.0
		Setup 5	1902.24	152.82	1764.0
	Scenario 3	***Setup 1***	1209.62	69.75	1176.0
		Setup 2	1312.56	59.72	1259.0
		Setup 3	1397.76	143.75	1307.0
		Setup 4	1395.34	97.20	1319.0
		Setup 5	1612.84	364.21	1331.0
	Scenario 4	***Setup 1***	25.28	25.46	0.0
		Setup 2	60.16	171.95	0.0
		Setup 3	117.76	286.35	0.0
		Setup 4	179.48	393.07	0.0
		Setup 5	267.12	388.78	0.0
	Scenario 5	***Setup 1***	111.34	246.72	2.0
		Setup 2	162.06	314.51	2.0
		Setup 3	434.8	592.17	2.0
		Setup 4	537.02	751.70	5.0
		Setup 5	735.86	752.00	9.0

Table 13 LaQ variations with the SAHCLayering algorithm applied on JHotDraw 7.6

			Mean	**StDev**	**Min**
JHotDraw 7.6	**Scenario 1**	***Setup 1***	1911.9	82.98	1724.0
		Setup 2	2176.8	126.00	1845.0
		Setup 3	2163.52	166.99	1898.0
		Setup 4	2318.04	258.81	1932.0
		Setup 5	2323.56	234.00	1964.0
	Scenario 2	***Setup 1***	914.76	11.32	872.0
		Setup 2	1337.0	28.92	1286.0
		Setup 3	1635.44	119.25	1378.0
		Setup 4	1615.0	148.99	1408.0
		Setup 5	1662.24	246.68	1410.0
	Scenario 3	***Setup 1***	1143.54	49.40	1089.0
		Setup 2	1287.92	73.78	1197.0
		Setup 3	1345.7	106.48	1227.0
		Setup 4	1434.12	130.71	1251.0
		Setup 5	1528.16	307.32	1275.0
	Scenario 4	***Setup 1***	34.2	43.63	0.0
		Setup 2	75.08	112.16	0.0
		Setup 3	149.24	246.71	0.0
		Setup 4	235.72	391.58	0.0
		Setup 5	331.6	541.36	0.0
	Scenario 5	***Setup 1***	87.8	136.83	2.0

		Setup 2	332.72	388.23	2.0
		Setup 3	774.64	651.61	2.0
		Setup 4	1002.6	806.77	2.0
		Setup 5	1010.24	876.30	5.0

Results obtained with the structural *TabuLayering* algorithm (Chapter 4)

Table 14 LaQ variations with the TabuLayering algorithm applied on Apache 1.6.2

			Mean	**StDev**	**Min**
Apache Ant 1.6.2	**Scenario 1**	***Setup 1***	1034.42	6.98	1026.0
		Setup 2	1126.08	9.13	1120.0
		Setup 3	1179.5	12.94	1168.0
		Setup 4	1230.38	14.27	1216.0
		Setup 5	1285.7	20.50	1257.0
	Scenario 2	***Setup 1***	460.08	20.49	426.0
		Setup 2	647.0	20.17	638.0
		Setup 3	811.56	18.60	798.0
		Setup 4	907.64	8.98	894.0
		Setup 5	954.64	5.61	942.0
	Scenario 3	***Setup 1***	570.86	0.85	569.0
		Setup 2	620.62	2.76	617.0
		Setup 3	668.32	13.06	662.0
		Setup 4	741.1	44.42	703.0
		Setup 5	804.72	144.43	737.0
	Scenario 4	***Setup 1***	17.84	38.27	0.0
		Setup 2	20.36	63.00	0.0
		Setup 3	28.56	82.57	0.0
		Setup 4	23.08	76.45	0.0
		Setup 5	33.8	99.88	0.0
	Scenario 5	***Setup 1***	11.78	12.13	4.0
		Setup 2	69.54	186.21	4.0
		Setup 3	170.66	401.52	6.0
		Setup 4	124.16	309.90	6.0
		Setup 5	101.86	245.72	6.0

Table 15 LaQ variations with the TabuLayering algorithm applied on JUnit 4.10

			Mean	**StDev**	**Min**
Junit 4.10	**Scenario 1**	***Setup 1***	259.04	22.17	234.0
		Setup 2	300.88	21.15	291.0
		Setup 3	347.98	30.11	330.0
		Setup 4	381.12	17.49	366.0
		Setup 5	430.94	38.69	402.0
	Scenario 2	***Setup 1***	134.24	3.01	132.0
		Setup 2	210.84	5.48	204.0

		Setup 3	266.24	15.73	248.0
		Setup 4	311.68	10.00	298.0
		Setup 5	362.84	28.94	338.0
	Scenario 3	*Setup 1*	158.66	20.70	148.0
		Setup 2	199.58	21.15	184.0
		Setup 3	241.9	36.18	218.0
		Setup 4	268.06	29.26	247.0
		Setup 5	293.92	41.68	263.0
	Scenario 4	*Setup 1*	16.6	24.34	0.0
		Setup 2	18.04	29.70	0.0
		Setup 3	28.4	58.35	0.0
		Setup 4	38.56	70.09	0.0
		Setup 5	37.64	74.56	0.0
	Scenario 5	*Setup 1*	49.42	54.82	7.0
		Setup 2	43.5	57.49	7.0
		Setup 3	57.56	93.34	7.0
		Setup 4	74.06	101.07	7.0
		Setup 5	51.3	93.28	7.0

Table 16 LaQ variations with the TabuLayering algorithm applied on JFreeChart 1.0.15

			Mean	**StDev**	**Min**
JFreeChart 1.0.15	**Scenario 1**	*Setup 1*	1943.78	20.92	1938.0
		Setup 2	2439.28	16.24	2436.0
		Setup 3	2913.06	69.40	2864.0
		Setup 4	3207.84	140.88	3071.0
		Setup 5	3367.8	188.67	3174.0
	Scenario 2	*Setup 1*	1184.56	13.14	1160.0
		Setup 2	1846.0	34.58	1808.0
		Setup 3	2374.4	173.39	2278.0
		Setup 4	2762.56	63.37	2704.0
		Setup 5	3004.28	177.86	2878.0
	Scenario 3	*Setup 1*	1445.1	38.97	1406.0
		Setup 2	1682.3	135.74	1562.0
		Setup 3	1866.94	282.00	1631.0
		Setup 4	1797.6	237.34	1660.0
		Setup 5	1950.64	375.70	1684.0
	Scenario 4	*Setup 1*	19.52	13.93	4.0
		Setup 2	32.44	27.22	4.0
		Setup 3	125.16	293.17	4.0
		Setup 4	292.44	478.27	4.0
		Setup 5	286.68	463.85	4.0
	Scenario 5	*Setup 1*	121.3	237.07	8.0
		Setup 2	250.0	366.71	8.0
		Setup 3	451.92	446.72	8.0
		Setup 4	418.14	490.13	8.0
		Setup 5	466.82	458.09	8.0

Table 17 LaQ variations with the TabuLayering algorithm applied on jEdit 5.0.0

			Mean	StDev	Min
jEdit 5.0.0	**Scenario 1**	***Setup 1***	1367.88	49.11	1288.0
		Setup 2	1727.88	66.22	1631.0
		Setup 3	1953.08	49.43	1849.0
		Setup 4	2033.02	26.85	1973.0
		Setup 5	2146.5	73.31	2097.0
	Scenario 2	***Setup 1***	814.52	34.09	772.0
		Setup 2	1385.48	53.48	1272.0
		Setup 3	1666.12	67.40	1632.0
		Setup 4	1938.64	133.94	1840.0
		Setup 5	2017.36	117.60	1960.
	Scenario 3	***Setup 1***	903.66	7.30	893.0
		Setup 2	1152.26	14.51	1120.0
		Setup 3	1253.18	12.12	1231.0
		Setup 4	1341.6	23.28	1316.0
		Setup 5	1399.8	37.51	1361.0
	Scenario 4	***Setup 1***	4.0	4.44	0.0
		Setup 2	7.0	10.82	0.0
		Setup 3	6.4	9.68	0.0
		Setup 4	10.08	15.51	0.0
		Setup 5	8.0	16.59	0.0
	Scenario 5	***Setup 1***	13.96	13.81	5.0
		Setup 2	15.96	14.60	5.0
		Setup 3	20.04	23.49	5.0
		Setup 4	13.56	13.48	5.0
		Setup 5	22.66	46.26	5.0

Table 18 LaQ variations with the TabuLayering algorithm applied on JHotDraw 60b1

			Mean	StDev	Min
JHotDraw 60b1	**Scenario 1**	***Setup 1***	605.1	34.41	587.0
		Setup 2	647.64	22.14	638.0
		Setup 3	696.78	57.61	682.0
		Setup 4	734.94	23.56	726.0
		Setup 5	797.54	85.56	770.0
	Scenario 2	***Setup 1***	114.6	18.38	112.0
		Setup 2	203.2	19.62	198.0
		Setup 3	282.48	25.66	276.0
		Setup 4	338.68	8.37	336.0
		Setup 5	389.08	11.30	384.0
	Scenario 3	***Setup 1***	390.14	20.97	383.0
		Setup 2	430.84	34.56	423.0
		Setup 3	483.64	75.79	463.0
		Setup 4	535.78	64.12	503.0

		Setup 5	587.46	111.28	543.0
	Scenario 4	***Setup 1***	20.04	31.24	2.0
		Setup 2	35.4	58.90	2.0
		Setup 3	47.12	68.02	2.0
		Setup 4	61.6	95.79	2.0
		Setup 5	57.88	92.76	2.0
	Scenario 5	***Setup 1***	25.48	37.67	3.0
		Setup 2	27.62	43.57	3.0
		Setup 3	33.58	59.57	3.0
		Setup 4	62.94	117.39	3.0
		Setup 5	62.4	132.27	3.0

Table 19 LaQ variations with the TabuLayering algorithm applied on JHotDraw 707

			Mean	**StDev**	**Min**
JHotDraw 707	**Scenario 1**	***Setup 1***	570.42	13.86	565.0
		Setup 2	615.0	30.32	580.0
		Setup 3	627.9	40.58	586.0
		Setup 4	639.92	46.72	590.0
		Setup 5	648.88	47.64	594.0
	Scenario 2	***Setup 1***	254.56	5.85	248.0
		Setup 2	332.64	26.50	320.0
		Setup 3	349.0	20.49	336.0
		Setup 4	356.6	11.78	344.0
		Setup 5	370.56	31.55	348.0
	Scenario 3	***Setup 1***	414.84	69.73	382.0
		Setup 2	440.66	58.82	398.0
		Setup 3	451.88	40.07	414.0
		Setup 4	478.56	59.18	421.0
		Setup 5	492.4	58.02	424.0
	Scenario 4	***Setup 1***	32.8	89.67	0.0
		Setup 2	43.04	120.09	0.0
		Setup 3	20.12	32.99	0.0
		Setup 4	34.12	53.36	0.0
		Setup 5	46.24	66.12	0.0
	Scenario 5	***Setup 1***	195.66	162.14	4.0
		Setup 2	295.22	255.47	4.0
		Setup 3	294.74	245.56	4.0
		Setup 4	341.24	345.06	4.0
		Setup 5	387.38	303.85	4.0

Table 20 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.4.1

			Mean	**StDev**	**Min**
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	1938.06	36.26	1910.0
		Setup 2	2308.64	102.27	2120.0
		Setup 3	2419.52	178.34	2199.0

		Setup 4	2564.96	224.25	2263.0
		Setup 5	2690.22	244.54	2278.0
	Scenario 2	***Setup 1***	850.64	6.35	840.0
		Setup 2	1370.08	18.40	1340.0
		Setup 3	1740.84	111.96	1620.0
		Setup 4	1852.48	155.22	1716.0
		Setup 5	1921.84	175.12	1734.0
	Scenario 3	***Setup 1***	1194.16	57.48	1176.0
		Setup 2	1300.14	67.71	1259.0
		Setup 3	1357.84	74.90	1307.0
		Setup 4	1530.76	382.03	1319.0
		Setup 5	1656.96	419.15	1331.0
	Scenario 4	***Setup 1***	18.2	23.14	0.0
		Setup 2	44.64	77.00	0.0
		Setup 3	129.96	294.79	0.0
		Setup 4	105.52	242.40	0.0
		Setup 5	289.64	453.69	0.0
	Scenario 5	***Setup 1***	72.0	192.78	2.0
		Setup 2	222.38	423.00	2.0
		Setup 3	486.64	579.22	2.0
		Setup 4	874.4	701.02	9.0
		Setup 5	1024.4	814.09	2.0

Table 21 LaQ variations with the TabuLayering algorithm applied on JHotDraw 7.6

			Mean	**StDev**	**Min**
JHotDraw 7.6	**Scenario 1**	***Setup 1***	1907.6	81.89	1724.0
		Setup 2	2143.56	154.27	1845.0
		Setup 3	2190.5	174.07	1899.0
		Setup 4	2264.62	271.15	1932.0
		Setup 5	2378.22	276.20	1964.0
	Scenario 2	***Setup 1***	919.84	11.42	872.0
		Setup 2	1340.16	38.14	1282.0
		Setup 3	1652.52	141.60	1384.0
		Setup 4	1647.32	192.97	1406.0
		Setup 5	1674.08	265.81	1398.0
	Scenario 3	***Setup 1***	1142.72	51.50	1089.0
		Setup 2	1290.88	71.39	1197.0
		Setup 3	1352.34	81.84	1227.0
		Setup 4	1399.6	69.86	1292.0
		Setup 5	1530.2	258.06	1275.0
	Scenario 4	***Setup 1***	23.24	40.24	0.0
		Setup 2	82.84	109.51	0.0
		Setup 3	162.32	231.43	0.0
		Setup 4	199.6	411.87	0.0
		Setup 5	447.04	628.26	0.0
	Scenario 5	***Setup 1***	126.68	293.12	2.0
		Setup 2	408.06	524.60	2.0

		Setup 3	672.3	660.45	2.0
		Setup 4	865.08	815.88	2.0
		Setup 5	1304.3	888.09	21.0

Time Results obtained with the *SAHCLayering* algorithm

Table 22 Running times yielded by the SAHCLayering algorithm applied on Apache 1.6.2

			Mean (ms)	**StDev (ms)**	**Max (ms)**
Apache Ant 1.6.2	**Scenario 1**	*Setup 1*	3746.58	66.23	3873.0
		Setup 2	3730.9	74.85	3949.0
		Setup 3	3691.22	183.63	4622.0
		Setup 4	3691.48	325.86	4907.0
		Setup 5	3736.96	422.56	5027.0
	Scenario 3	*Setup 1*	3608.32	559.43	4013.0
		Setup 2	3712.68	351.90	3919.0
		Setup 3	3787.0	285.58	4002.0
		Setup 4	3718.4	304.81	3973.0
		Setup 5	3695.8	255.40	3935.0

Table 23 Running times yielded by the SAHCLayering algorithm applied on JUnit 4.10

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JUnit 4.10	**Scenario 1**	*Setup 1*	261.52	59.51	350.0
		Setup 2	248.68	60.71	460.0
		Setup 3	234.96	43.23	362.0
		Setup 4	248.96	54.38	354.0
		Setup 5	259.76	55.06	348.0
	Scenario 3	*Setup 1*	246.74	54.96	344.0
		Setup 2	227.06	42.36	343.0
		Setup 3	224.24	37.59	328.0
		Setup 4	245.52	52.93	375.0
		Setup 5	261.1	64.38	406.0

Table 24 Running times yielded by the SAHCLayering algorithm applied on JFreeChart 1.0.15

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JFreeChart 1.0.15	**Scenario 1**	*Setup 1*	878.92	103.22	1374.0
		Setup 2	904.88	190.81	1829.0
		Setup 3	1037.54	235.59	1391.0
		Setup 4	1033.14	217.01	1512.0
		Setup 5	1206.16	228.39	1847.0
	Scenario 3	*Setup 1*	1002.14	212.41	1388.0

		Setup 2	1095.0	237.01	1452.0
		Setup 3	1092.32	226.19	1416.0
		Setup 4	1111.74	230.68	1451.0
		Setup 5	1084.22	210.35	1405.0

Table 25 Running times yielded by the SAHCLayering algorithm applied on jEdit 5.0.0

			Mean (ms)	**StDev (ms)**	**Max (ms)**
jEdit 5.0.0	**Scenario 1**	***Setup 1***	839.1	63.03	1137.0
		Setup 2	843.2	82.66	1180.0
		Setup 3	887.84	128.90	1126.0
		Setup 4	902.38	159.75	1156.0
		Setup 5	922.36	182.55	1156.0
	Scenario 3	***Setup 1***	763.82	100.54	844.0
		Setup 2	789.58	83.28	859.0
		Setup 3	839.16	140.20	1109.0
		Setup 4	833.74	159.99	1127.0
		Setup 5	792.9	157.66	1109.0

Table 26 Running times yielded by the SAHCLayering algorithm applied on JHotDraw 60b1

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JHotDraw 60b1	**Scenario 1**	***Setup 1***	86.5	23.35	182.0
		Setup 2	79.9	14.33	120.0
		Setup 3	81.98	16.70	115.0
		Setup 4	77.42	11.86	114.0
		Setup 5	80.82	15.01	115.0
	Scenario 3	***Setup 1***	86.96	18.14	117.0
		Setup 2	85.76	17.99	116.0
		Setup 3	82.46	16.06	120.0
		Setup 4	79.8	16.26	164.0
		Setup 5	77.8	11.27	117.0

Table 27 Running times yielded by the SAHCLayering algorithm applied on JHotDraw 7.0.7

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JHotDraw 7.0.7	**Scenario 1**	***Setup 1***	239.78	27.46	339.0
		Setup 2	251.26	82.07	761.0
		Setup 3	237.98	70.86	698.0
		Setup 4	233.38	34.53	340.0
		Setup 5	233.38	33.46	381.0
	Scenario 3	***Setup 1***	184.42	43.37	280.0
		Setup 2	172.92	40.29	253.0
		Setup 3	179.18	40.88	229.0
		Setup 4	201.62	41.74	299.0
		Setup 5	233.12	43.78	313.0

Table 28 Running times yielded by the SAHCLayering algorithm applied on JHotDraw 7.4.1

			Mean (ms)	StDev (ms)	Max (ms)
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	5194.34	680.72	5717.0
		Setup 2	5518.14	307.22	7348.0
		Setup 3	5765.66	774.37	7705.0
		Setup 4	5806.32	1403.99	9314.0
		Setup 5	5196.78	1528.51	7360.0
	Scenario 3	***Setup 1***	4362.84	989.98	5821.0
		Setup 2	5134.92	807.11	5876.0
		Setup 3	4928.98	888.77	5795.0
		Setup 4	4712.96	872.39	5779.0
		Setup 5	4644.14	968.44	7392.0

Table 29 Running times yielded by the SAHCLayering algorithm applied on JHotDraw 7.6

			Mean (ms)	StDev (ms)	Max (ms)
JHotDraw 7.6	**Scenario 1**	***Setup 1***	5757.02	286.47	7580.0
		Setup 2	5805.56	301.91	7652.0
		Setup 3	6227.34	845.86	7870.0
		Setup 4	6699.94	1466.01	9713.0
		Setup 5	6275.0	1272.69	9653.0
	Scenario 3	***Setup 1***	5111.86	1018.82	6245.0
		Setup 2	5565.3	769.92	6103.0
		Setup 3	5234.12	939.13	6331.0
		Setup 4	5075.76	976.60	6446.0
		Setup 5	5064.2	1275.56	9632.0

Time Results obtained with the *TabuLayering* algorithm

Table 30 Running times yielded by the TabuLayering algorithm applied on Apache 1.6.2

			Mean (ms)	StDev (ms)	Max (ms)
Apache Ant 1.6.2	**Scenario 1**	***Setup 1***	10693.82	69.73	10809.0
		Setup 2	10681.14	85.45	10889.0
		Setup 3	10632.62	85.57	10837.0
		Setup 4	10779.1	573.84	14700.0
		Setup 5	10632.72	827.80	14762.0
	Scenario 3	***Setup 1***	9081.1	1798.64	10330.0
		Setup 2	10180.26	563.69	10347.0
		Setup 3	10068.24	933.88	10665.0
		Setup 4	10133.92	961.27	10694.0
		Setup 5	10221.86	834.77	10701.0

Table 31 Running times yielded by the TabuLayering algorithm applied on JUnit 4.10

			Mean (ms)	StDev (ms)	Max (ms)
JUnit 4.10	**Scenario 1**	***Setup 1***	985.18	248.57	1318.0
		Setup 2	890.52	215.85	1286.0
		Setup 3	909.38	226.41	1281.0
		Setup 4	810.24	157.08	1203.0
		Setup 5	921.34	234.74	1262.0
	Scenario 3	***Setup 1***	879.34	207.61	1183.0
		Setup 2	836.8	186.26	1175.0
		Setup 3	786.22	166.03	1203.0
		Setup 4	925.52	264.36	1656.0
		Setup 5	866.66	226.22	1260.0

Table 32 Running times yielded by the TabuLayering algorithm applied on JFreeChart 1.0.15

			Mean (ms)	StDev (ms)	Max (ms)
JFreeChart 1.0.15	**Scenario 1**	***Setup 1***	2988.02	395.53	4933.0
		Setup 2	3040.26	733.87	6654.0
		Setup 3	3605.4	886.76	5105.0
		Setup 4	3778.64	1029.84	6510.0
		Setup 5	4039.94	1002.84	6775.0
	Scenario 3	***Setup 1***	3415.98	784.84	4575.0
		Setup 2	3502.12	835.38	4636.0
		Setup 3	3451.48	860.05	4901.0
		Setup 4	3511.64	833.96	4871.0
		Setup 5	3857.04	878.92	4855.0

Table 33 Running times yielded by the TabuLayering algorithm applied on jEdit 5.0.0

			Mean (ms)	StDev (ms)	Max (ms)
jEdit 5.0.0	**Scenario 1**	***Setup 1***	2954.06	369.69	3996.0
		Setup 2	2965.66	334.48	3981.0
		Setup 3	3145.5	493.94	4012.0
		Setup 4	3271.08	543.41	3982.0
		Setup 5	3249.92	673.52	3996.0
	Scenario 3	***Setup 1***	2601.58	426.87	2967.0
		Setup 2	2770.66	75.09	2964.0
		Setup 3	2800.86	749.16	4027.0
		Setup 4	3186.72	664.93	4027.0
		Setup 5	3000.38	766.21	4044.0

Table 34 Running times yielded by the TabuLayering algorithm applied on JHotDraw 60b1

			Mean (ms)	StDev (ms)	Max (ms)
JHotDraw 60b1	**Scenario 1**	***Setup 1***	329.82	79.02	500.0

		Setup 2	308.54	71.74	469.0
		Setup 3	300.42	60.47	438.0
		Setup 4	293.32	57.89	436.0
		Setup 5	305.7	67.99	453.0
	Scenario 3	***Setup 1***	331.06	76.68	484.0
		Setup 2	296.68	58.75	438.0
		Setup 3	307.62	66.54	437.0
		Setup 4	288.56	47.93	437.0
		Setup 5	300.44	61.48	438.0

Table 35 Running times yielded by the TabuLayering algorithm applied on JHotDraw 7.0.7

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JHotDraw 7.0.7	**Scenario 1**	***Setup 1***	922.36	158.57	1747.0
		Setup 2	893.0	86.99	1246.0
		Setup 3	894.56	95.07	1163.0
		Setup 4	875.78	48.50	1122.0
		Setup 5	904.38	103.37	1340.0
	Scenario 3	***Setup 1***	904.38	103.37	1340.0
		Setup 2	605.52	155.60	938.0
		Setup 3	600.26	146.16	836.0
		Setup 4	760.14	164.45	1151.0
		Setup 5	858.18	170.22	1200.0

Table 36 Running times yielded by the TabuLayering algorithm applied on JHotDraw 7.4.1

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JHotDraw 7.4.1	**Scenario 1**	***Setup 1***	15313.14	2406.65	16898.0
		Setup 2	17290.4	1967.78	23199.0
		Setup 3	18450.8	3107.33	23715.0
		Setup 4	18705.64	4774.69	30082.0
		Setup 5	18787.88	5864.13	32220.0
	Scenario 3	***Setup 1***	13095.66	2967.96	16430.0
		Setup 2	14478.92	2510.08	16431.0
		Setup 3	14364.2	2654.82	16274.0
		Setup 4	14077.0	2822.98	16445.0
		Setup 5	13827.18	3009.08	16649.0

Table 37 Running times yielded by the TabuLayering algorithm applied on JHotDraw 7.6

			Mean (ms)	**StDev (ms)**	**Max (ms)**
JHotDraw 7.6	**Scenario 1**	***Setup 1***	17485.86	991.62	24263.0
		Setup 2	17602.28	882.14	23515.0
		Setup 3	20912.78	3807.53	31407.0
		Setup 4	20622.02	4966.91	32625.0
		Setup 5	20300.54	5758.22	34545.0
	Scenario 3	***Setup 1***	14375.86	3152.10	17320.0
		Setup 2	15505.16	2732.70	17537.0

		Setup 3	14904.36	3002.13	17506.0
		Setup 4	14232.36	3269.34	17506.0
		Setup 5	13946.46	3309.86	17476.0

Stability between the respective layerings of JHotDraw 7.4.1 and JHotDraw 7.5.1

Table 38 Stability between the respective layerings of JHotDraw 7.4.1 and JHotDraw 7.5.1

JHotDraw 7.4.1	**JHotDraw 7.5**	**Package Assigned to the same LAYER in the two versions**	**Ratio of packages assigned to the SAME LAYER**
Package/ assigned layer	**Package/ assigned layer**		
1. net.n3.nanoxml /Layer1	net.n3.nanoxml /Layer1	YES	1/1
2. org.jhotdraw.beans /Layer2	org.jhotdraw.annotations /Layer2	YES	7/7
3. org.jhotdraw.geom /Layer2	org.jhotdraw.beans /Layer2	YES	
4. org.jhotdraw.util.* /Layer2	org.jhotdraw.util.* /Layer2	YES	
5. org.jhotdraw.gui.plaf.* /Layer2	org.jhotdraw.gui.plaf.* /Layer2	YES	
6. org.jhotdraw.xml.* /Layer2	org.jhotdraw.gui.fontchooser /Layer2	YES	
7. org.jhotdraw.gui.fontchooser /Layer2	org.jhotdraw.xml.* /Layer2	YES	
8. org.jhotdraw.gui.datatransfer /Layer2	org.jhotdraw.net /Layer2	YES	
9. org.jhotdraw.gui.* /Layer3	org.jhotdraw.gui.event /Layer2	YES	27/31
10. org.jhotdraw.gui.event /Layer3	org.jhotdraw.gui.filechooser /Layer2	NO	
11. org.jhotdraw.color /Layer3	org.jhotdraw.geom /Layer2	YES	
12. org.jhotdraw.gui.plaf.palette /Layer3	org.jhotdraw.xml.css /Layer2	YES	
13. org.jhotdraw.draw.* /Layer3	org.jhotdraw.gui.datatransfer Layer2	YES	
14. org.jhotdraw.draw.decoration /Layer3	org.jhotdraw.util.prefs /Layer2	YES	
15. org.jhotdraw.draw.io /Layer3	org.jhotdraw.app.osx /Layer3	YES	
16. org.jhotdraw.draw.layouter /Layer3	org.jhotdraw.gui.* /Layer3	YES	
17. org.jhotdraw.draw.liner /Layer3	org.jhotdraw.gui.plaf.palette.*	YES	

	/Layer3		
18. org.jhotdraw.undo /Layer3	org.jhotdraw.text /Layer3	YES	
19. org.jhotdraw.draw.text /Layer3	org.jhotdraw.draw.* /Layer3	YES	
20. org.jhotdraw.samples.teddy.io /Layer3	org.jhotdraw.draw.decoration /Layer3	YES	
21. org.jhotdraw.draw.handle /Layer3	org.jhotdraw.draw.handle /Layer3	YES	
22. org.jhotdraw.draw.tool /Layer3	org.jhotdraw.draw.io /Layer3	YES	
23. org.jhotdraw.draw.connector /Layer3	org.jhotdraw.draw.layouter /Layer3	YES	
24. org.jhotdraw.app.*/ Layer3	org.jhotdraw.samples.teddy.rege x /Layer3	YES	
25. org.jhotdraw.draw.locator /Layer3	org.jhotdraw.draw.event /Layer3	YES	
26. org.jhotdraw.draw.event /Layer3	org.jhotdraw.draw.connector /Layer3	YES	
27. org.jhotdraw.app.action.* /Layer3	org.jhotdraw.draw.liner /Layer3	YES	
28. org.jhotdraw.app.action.edit /Layer3	org.jhotdraw.app.* /Layer3	YES	
29. org.jhotdraw.io /Layer3	org.jhotdraw.app.action.edit /Layer3	YES	
30. org.jhotdraw.util.prefs /Layer3	org.jhotdraw.draw.locator /Layer3	NO	
31. org.jhotdraw.app.action.windo w /Layer3	org.jhotdraw.draw.tool /Layer3	YES	
32. org.jhotdraw.draw.print /Layer3	org.jhotdraw.app.action.* /Layer3	YES	
33. org.jhotdraw.text /Layer3	org.jhotdraw.app.action.window /Layer3	YES	
34. org.jhotdraw.app.osx /Layer3	org.jhotdraw.app.action.app /Layer3	YES	
35. org.jhotdraw.xml.css /Layer3	org.jhotdraw.draw.text /Layer3	NO	
36. org.jhotdraw.samples.teddy.reg ex /Layer3	org.jhotdraw.io /Layer3	YES	
37. org.jhotdraw.samples.teddy.tex t /Layer3	org.jhotdraw.undo /Layer3	YES	
38. org.jhotdraw.samples.odg.geo m /Layer3	org.jhotdraw.app.action.file /Layer3	YES	

39. org.jhotdraw.net /Layer3	org.jhotdraw.draw.print /Layer3	NO	
40. org.jhotdraw.samples.svg.actio n /Layer4	org.jhotdraw.samples.odg.geom /Layer3	YES	21/23
41. org.jhotdraw.samples.color /Layer4	org.jhotdraw.samples.teddy.io /Layer3	YES	
42. org.jhotdraw.samples.net.* /Layer4	org.jhotdraw.samples.teddy.text /Layer3	YES	
43. org.jhotdraw.samples.odg.figur es /Layer4	org.jhotdraw.color /Layer3	YES	
44. org.jhotdraw.samples.svg.figur es /Layer4	org.jhotdraw.draw.action /Layer4	YES	
45. org.jhotdraw.samples.pert.* /Layer4	org.jhotdraw.samples.draw /Layer4	YES	
46. org.jhotdraw.samples.pert.figur es /Layer4	org.jhotdraw.samples.net.figure s /Layer4	YES	
47. org.jhotdraw.samples.svg.gui /Layer4	org.jhotdraw.samples.svg.action /Layer4	YES	
48. org.jhotdraw.samples.teddy.act ion /Layer4	org.jhotdraw.samples.svg.figure s /Layer4	YES	
49. org.jhotdraw.app.action.file /Layer4	org.jhotdraw.samples.pert.figure s /Layer4	NO	
50. org.jhotdraw.samples.odg.* /Layer4	org.jhotdraw.samples.svg.gui /Layer4	YES	
51. org.jhotdraw.samples.odg.io Layer4	org.jhotdraw.samples.teddy.* Layer4	YES	
52. org.jhotdraw.samples.svg.io /Layer4	org.jhotdraw.samples.svg.io /Layer4	YES	
53. org.jhotdraw.samples.svg.* /Layer4	org.jhotdraw.samples.pert.* /Layer4	YES	
54. org.jhotdraw.samples.teddy.* /Layer4	org.jhotdraw.samples.svg.* /Layer4	YES	
55. org.jhotdraw.samples.mini Layer4	org.jhotdraw.samples.odg.* Layer4	YES	
56. org.jhotdraw.samples.draw Layer4	org.jhotdraw.samples.mini Layer4	YES	
57. org.jhotdraw.samples.net.figur es /Layer4	org.jhotdraw.samples.odg.io	YES	

	/Layer4		
58. org.jhotdraw.draw.action /Layer4	org.jhotdraw.samples.net.* /Layer4	YES	
59. org.jhotdraw.samples.odg.actio n /Layer4	org.jhotdraw.gui.plaf.palette.col orchooser /Layer4	YES	
60. org.jhotdraw.app.action.view /Layer4	org.jhotdraw.samples.odg.figure s /Layer4	YES	
61. org.jhotdraw.app.action.app /Layer4	org.jhotdraw.samples.odg.action /Layer4	NO	
62. org.jhotdraw.samples.font /Layer4	org.jhotdraw.app.action.view /Layer4	YES	
63.	org.jhotdraw.samples.teddy.acti on /Layer4		
64.	org.jhotdraw.samples.font /Layer4		
65.	org.jhotdraw.samples.color /Layer4		

Stability between the respective layerings of JHotDraw 7.5.1 and JHotDraw 7.6

Table 39 Stability between the respective layerings of JHotDraw 7.5.1 and JHotDraw 7.6

JHotDraw 7.5.1	**JHotDraw 7.6**	**Package Assigned to the same LAYER in the two versions**	**Ratio of packages assigned to the SAME LAYER**
Package/ assigned layer	**Package/ assigned layer**		
1. net.n3.nanoxml/ Layer1	org.jhotdraw.gui.plaf.*/ Layer1	YES	1/1
2. org.jhotdraw.annotations /Layer2	org.jhotdraw.gui.fontchooser /Layer1	**N/A**	9/12
3. org.jhotdraw.beans /Layer2	net.n3.nanoxml/ Layer1	YES	
4. org.jhotdraw.util.*/ Layer2	org.jhotdraw.xml.* /Layer2	YES	
5. org.jhotdraw.gui.plaf.*/ Layer2	org.jhotdraw.beans /Layer2	NO	

6. org.jhotdraw.gui.fontchooser /Layer2	org.jhotdraw.util.* /Layer2	NO	
7. org.jhotdraw.xml.* /Layer2	org.jhotdraw.color /Layer2	YES	
8. org.jhotdraw.net /Layer2	org.jhotdraw.geom /Layer2	YES	
9. org.jhotdraw.gui.event /Layer2	org.jhotdraw.gui.* /Layer2	YES	
10. org.jhotdraw.gui.filechooser /Layer2	org.jhotdraw.gui.plaf.palette.*/ Layer2	YES	
11. org.jhotdraw.geom /Layer2	org.jhotdraw.text /Layer2	YES	
12. org.jhotdraw.xml.css /Layer2	org.jhotdraw.gui.event /Layer2	NO	
13. org.jhotdraw.gui.datatransfer /Layer2	org.jhotdraw.gui.datatransfer /Layer2	YES	
14. org.jhotdraw.util.prefs /Layer2	org.jhotdraw.net /Layer2	YES	
15. org.jhotdraw.app.osx /Layer3	org.jhotdraw.gui.filechooser /Layer2	YES	25/29
16. org.jhotdraw.gui.* /Layer3	org.jhotdraw.util.prefs /Layer2	NO	
17. org.jhotdraw.gui.plaf.palette.* /Layer3	org.jhotdraw.app.* /Layer3	NO	
18. org.jhotdraw.text /Layer3	org.jhotdraw.app.action.* /Layer3	NO	
19. org.jhotdraw.draw.*/ Layer3	org.jhotdraw.app.action.edit /Layer3	YES	
20. org.jhotdraw.draw.decoration/ Layer3	org.jhotdraw.draw.connector /Layer3	YES	
21. org.jhotdraw.draw.handle /Layer3	org.jhotdraw.draw.event /Layer3	YES	
22. org.jhotdraw.draw.io /Layer3	org.jhotdraw.draw.layouter /Layer3	YES	
23. org.jhotdraw.draw.layouter /Layer3	org.jhotdraw.draw.tool /Layer3	YES	
24. org.jhotdraw.samples.teddy.regex /Layer3	org.jhotdraw.draw.text /Layer3	YES	
25. org.jhotdraw.draw.event /Layer3	org.jhotdraw.xml.css /Layer3	YES	
26. org.jhotdraw.draw.connector /Layer3	org.jhotdraw.samples.teddy.regex /Layer3	YES	
27. org.jhotdraw.draw.liner /Layer3	org.jhotdraw.draw.* /Layer3	YES	
28. org.jhotdraw.app.* /Layer3	org.jhotdraw.draw.handle /Layer3	YES	
29. org.jhotdraw.app.action.edit /Layer3	org.jhotdraw.draw.locator /Layer3	YES	
30. org.jhotdraw.draw.locator /Layer3	org.jhotdraw.app.action.window /Layer3	YES	
31. org.jhotdraw.draw.tool /Layer3	org.jhotdraw.draw.io /Layer3	YES	
32. org.jhotdraw.app.action.* /Layer3	org.jhotdraw.undo /Layer3	YES	
33. org.jhotdraw.app.action.window /Layer3	org.jhotdraw.draw.liner /Layer3	YES	
34. org.jhotdraw.app.action.app /Layer3	org.jhotdraw.samples.color /Layer3	YES	
35. org.jhotdraw.draw.text /Layer3	org.jhotdraw.app.action.app/ Layer3	YES	
36. org.jhotdraw.io /Layer3	org.jhotdraw.draw.decoration /Layer3	YES	
37. org.jhotdraw.undo/ Layer3	org.jhotdraw.svg.gui /Layer3	YES	

38. org.jhotdraw.app.action.file /Layer3	org.jhotdraw.io Layer3	YES	
39. org.jhotdraw.draw.print /Layer3	org.jhotdraw.app.osx /Layer3	YES	
40. org.jhotdraw.samples.odg.geom /Layer3	org.jhotdraw.app.action.file /Layer3	YES	
41. org.jhotdraw.samples.teddy.io /Layer3	org.jhotdraw.gui.plaf.palette.colorchooser /Layer3	YES	
42. org.jhotdraw.samples.teddy.text /Layer3	org.jhotdraw.draw.gui /Layer3	YES	
43. org.jhotdraw.color /Layer3	org.jhotdraw.draw.print /Layer3	NO	
44. org.jhotdraw.draw.action /Layer4	org.jhotdraw.samples.teddy.io /Layer3	YES	18/21
45. org.jhotdraw.samples.draw /Layer4	org.jhotdraw.samples.teddy.text /Layer3	YES	
46. org.jhotdraw.samples.net.figures /Layer4	org.jhotdraw.samples.font /Layer3	YES	
47. org.jhotdraw.samples.svg.action /Layer4	org.jhotdraw.samples.odg.geom /Layer3	YES	
48. org.jhotdraw.samples.svg.figures /Layer4	org.jhotdraw.app.action.view /Layer4	YES	
49. org.jhotdraw.samples.pert.figures Layer4	org.jhotdraw.draw.action /Layer4	YES	
50. org.jhotdraw.samples.svg.gui /Layer4	org.jhotdraw.samples.net.* /Layer4	**N/A**	
51. org.jhotdraw.samples.teddy.* /Layer4	org.jhotdraw.samples.svg.io /Layer4	YES	
52. org.jhotdraw.samples.svg.io /Layer4	org.jhotdraw.samples.pert.figures /Layer4	YES	
53. org.jhotdraw.samples.pert.* /Layer4	org.jhotdraw.samples.odg.* /Layer4	YES	
54. org.jhotdraw.samples.svg.* /Layer4	org.jhotdraw.samples.svg.figures /Layer4	YES	
55. org.jhotdraw.samples.odg.* /Layer4	org.jhotdraw.samples.mini /Layer4	YES	
56. org.jhotdraw.samples.mini /Layer4	org.jhotdraw.samples.odg.figures /Layer4	YES	
57. org.jhotdraw.samples.odg.io /Layer4	org.jhotdraw.samples.svg.* /Layer4	YES	
58. org.jhotdraw.samples.net.* /Layer4	org.jhotdraw.samples.draw /Layer4	YES	
59. org.jhotdraw.gui.plaf.palette.colorchooser /Layer4	org.jhotdraw.samples.pert.* /Layer4	NO	
60. org.jhotdraw.samples.odg.figures /Layer4	org.jhotdraw.samples.teddy.*/ Layer4	YES	
61. org.jhotdraw.samples.odg.action /Layer4	org.jhotdraw.samples.teddy.action /Layer4	YES	
62. org.jhotdraw.app.action.view /Layer4	org.jhotdraw.samples.net.figures /Layer4	YES	
63. org.jhotdraw.samples.teddy.action /Layer4	org.jhotdraw.samples.odg.io /Layer4	YES	
64. org.jhotdraw.samples.font /Layer4	org.jhotdraw.samples.svg.action /Layer4	NO	

65. org.jhotdraw.samples.color /Layer4	org.jhotdraw.samples.odg.action /Layer4	NO	

APPENDIX II

MANUAL DECOMPOSITIONS OF THE ANALYZED SYSTEMS

Manual decomposition of Apache 1.6.2

```
Layer3: org.apache.tools.ant.taskdefs.*
Layer3: org.apache.tools.ant.taskdefs.compilers
Layer3: org.apache.tools.ant.taskdefs.condition
Layer3: org.apache.tools.ant.taskdefs.cvslib
Layer3: org.apache.tools.ant.taskdefs.email
Layer3: org.apache.tools.ant.taskdefs.rmic
Layer3: org.apache.tools.ant.taskdefs.optional.*
Layer3: org.apache.tools.ant.taskdefs.optional.ccm
Layer3: org.apache.tools.ant.taskdefs.optional.clearcase
Layer3: org.apache.tools.ant.taskdefs.optional.depend.*
Layer3: org.apache.tools.ant.taskdefs.optional.depend.constantpool
Layer3: org.apache.tools.ant.taskdefs.optional.dotnet
Layer3: org.apache.tools.ant.taskdefs.optional.ejb
Layer3: org.apache.tools.ant.taskdefs.optional.extension.*
Layer3: org.apache.tools.ant.taskdefs.optional.extension.resolvers
Layer3: org.apache.tools.ant.taskdefs.optional.i18n
Layer3: org.apache.tools.ant.taskdefs.optional.ide
Layer3: org.apache.tools.ant.taskdefs.optional.image
Layer3: org.apache.tools.ant.taskdefs.optional.j2ee
Layer3: org.apache.tools.ant.taskdefs.optional.javacc
Layer3: org.apache.tools.ant.taskdefs.optional.jdepend
Layer3: org.apache.tools.ant.taskdefs.optional.jlink
Layer3: org.apache.tools.ant.taskdefs.optional.jsp.*
Layer3: org.apache.tools.ant.taskdefs.optional.jsp.compilers
Layer3: org.apache.tools.ant.taskdefs.optional.junit
Layer3: org.apache.tools.ant.taskdefs.optional.metamata
Layer3: org.apache.tools.ant.taskdefs.optional.net
Layer3: org.apache.tools.ant.taskdefs.optional.perforce
Layer3: org.apache.tools.ant.taskdefs.optional.pvcs
Layer3: org.apache.tools.ant.taskdefs.optional.scm
Layer3: org.apache.tools.ant.taskdefs.optional.script
Layer3: org.apache.tools.ant.taskdefs.optional.sitraka.*
Layer3: org.apache.tools.ant.taskdefs.optional.sitraka.bytecode.*
Layer3: org.apache.tools.ant.taskdefs.optional.sitraka.bytecode.attributes
Layer3: org.apache.tools.ant.taskdefs.optional.sos
Layer3: org.apache.tools.ant.taskdefs.optional.sound
Layer3: org.apache.tools.ant.taskdefs.optional.splash
Layer3: org.apache.tools.ant.taskdefs.optional.ssh
Layer3: org.apache.tools.ant.taskdefs.optional.starteam
Layer3: org.apache.tools.ant.taskdefs.optional.unix
Layer3: org.apache.tools.ant.taskdefs.optional.vss
Layer3: org.apache.tools.ant.taskdefs.optional.windows

Layer2: org.apache.tools.ant.*
```

```
Layer2: org.apache.tools.ant.filters.*
Layer2: org.apache.tools.ant.filters.util
Layer2: org.apache.tools.ant.helper
Layer2: org.apache.tools.ant.input
Layer2: org.apache.tools.ant.launch
Layer2: org.apache.tools.ant.listener
Layer2: org.apache.tools.ant.loader
Layer2: org.apache.tools.ant.util.*
Layer2: org.apache.tools.ant.util.depend.*
Layer2: org.apache.tools.ant.util.depend.bcel
Layer2: org.apache.tools.ant.util.facade
Layer2: org.apache.tools.ant.util.optional
Layer2: org.apache.tools.ant.util.regexp
Layer2: org.apache.tools.ant.types.*
Layer2: org.apache.tools.ant.types.resolver
Layer2: org.apache.tools.ant.types.selectors.*
Layer2: org.apache.tools.ant.types.selectors.modifiedselector
Layer2: org.apache.tools.ant.types.optional.*
Layer2: org.apache.tools.ant.types.optional.depend
Layer2: org.apache.tools.ant.types.optional.image

Layer1: org.apache.tools.bzip2
Layer1: org.apache.tools.mail
Layer1: org.apache.tools.tar
Layer1: org.apache.tools.zip
```

Manual decomposition of JUnit 4.10

```
Layer3: junit.textui
Layer3: org.junit.experimental.*
Layer3: org.junit.experimental.categories
Layer3: org.junit.experimental.max
Layer3: org.junit.experimental.results
Layer3: org.junit.experimental.runners
Layer3: org.junit.experimental.theories.*
Layer3: org.junit.experimental.theories.internal
Layer3: org.junit.experimental.theories.suppliers

Layer2: junit.runner
Layer2: junit.extensions
Layer2: org.junit.internal.*
Layer2: org.junit.internal.builders
Layer2: org.junit.internal.matchers
Layer2: org.junit.internal.requests
Layer2: org.junit.internal.runners.*
Layer2: org.junit.internal.runners.model
Layer2: org.junit.internal.runners.rules
Layer2: org.junit.internal.runners.statements
Layer2: org.junit.matchers
Layer2: org.junit.rules
Layer2: org.junit.runner.*
```

```
Layer2: org.junit.runner.manipulation
Layer2: org.junit.runner.notification
Layer2: org.junit.runners.*
Layer2: org.junit.runners.model

Layer1: junit.framework
Layer1: org.junit.*
```

Manual decomposition of JHotDraw 60b1

```
Layer3: org.jhotdraw.applet
Layer3: org.jhotdraw.application
Layer3: org.jhotdraw.samples.javadraw
Layer3: org.jhotdraw.samples.minimap
Layer3: org.jhotdraw.samples.net
Layer3: org.jhotdraw.samples.nothing
Layer3: org.jhotdraw.samples.pert

Layer2: org.jhotdraw.standard
Layer2: org.jhotdraw.figures
Layer2: org.jhotdraw.contrib.*
Layer2: org.jhotdraw.contrib.dnd
Layer2: org.jhotdraw.contrib.html
Layer2: org.jhotdraw.contrib.zoom

Layer1: org.jhotdraw.framework
Layer1: org.jhotdraw.util.*
Layer1: org.jhotdraw.util.collections.jdk11
Layer1: org.jhotdraw.util.collections.jdk12
```

Manual decomposition of JHotDraw 707

```
Layer4: org.jhotdraw.samples.draw
Layer4: org.jhotdraw.samples.net.*
Layer4: org.jhotdraw.samples.net.figures
Layer4: org.jhotdraw.samples.pert.figures
Layer4: org.jhotdraw.samples.svg.action
Layer4: org.jhotdraw.samples.pert.*
Layer4: org.jhotdraw.samples.svg.figures
Layer4: org.jhotdraw.samples.svg.*

Layer3: org.jhotdraw.app.*
Layer3: org.jhotdraw.app.action
Layer3: org.jhotdraw.draw.action
Layer3: org.jhotdraw.draw.*
Layer3: org.jhotdraw.undo
```

```
Layer2: org.jhotdraw.beans
Layer2: org.jhotdraw.geom
Layer2: org.jhotdraw.gui.event
Layer2: org.jhotdraw.gui.datatransfer
Layer2: org.jhotdraw.io
Layer2: org.jhotdraw.util.*
Layer2: org.jhotdraw.util.prefs
Layer2: org.jhotdraw.gui.*
Layer2: org.jhotdraw.xml

Layer1: net.n3.nanoxml
Layer1: nanoxml
```

Manual decomposition JHotDraw 7.4.1

```
Layer4: org.jhotdraw.samples.color
Layer4: org.jhotdraw.samples.draw
Layer4: org.jhotdraw.samples.font
Layer4: org.jhotdraw.samples.mini
Layer4: org.jhotdraw.samples.net.*
Layer4: org.jhotdraw.samples.net.figures
Layer4: org.jhotdraw.samples.odg.*
Layer4: org.jhotdraw.samples.odg.action
Layer4: org.jhotdraw.samples.odg.figures
Layer4: org.jhotdraw.samples.odg.geom
Layer4: org.jhotdraw.samples.odg.io
Layer4: org.jhotdraw.samples.pert.*
Layer4: org.jhotdraw.samples.pert.figures
Layer4: org.jhotdraw.samples.svg.*
Layer4: org.jhotdraw.samples.svg.action
Layer4: org.jhotdraw.samples.svg.figures
Layer4: org.jhotdraw.samples.svg.gui
Layer4: org.jhotdraw.samples.svg.io
Layer4: org.jhotdraw.samples.teddy.*
Layer4: org.jhotdraw.samples.teddy.action
Layer4: org.jhotdraw.samples.teddy.io
Layer4: org.jhotdraw.samples.teddy.regex
Layer4: org.jhotdraw.samples.teddy.text

Layer3: org.jhotdraw.app.*
Layer3: org.jhotdraw.app.action.*
Layer3: org.jhotdraw.app.action.app
Layer3: org.jhotdraw.app.action.edit
Layer3: org.jhotdraw.app.action.file
Layer3: org.jhotdraw.app.action.view
Layer3: org.jhotdraw.app.action.window
Layer3: org.jhotdraw.app.osx
Layer3: org.jhotdraw.gui.*
Layer3: org.jhotdraw.gui.datatransfer
```

```
Layer3: org.jhotdraw.gui.event
Layer3: org.jhotdraw.gui.fontchooser
Layer3: org.jhotdraw.gui.plaf.*
Layer3: org.jhotdraw.gui.plaf.palette
Layer3: org.jhotdraw.samples.color
Layer3: org.jhotdraw.draw.*
Layer3: org.jhotdraw.draw.action
Layer3: org.jhotdraw.draw.connector
Layer3: org.jhotdraw.draw.decoration
Layer3: org.jhotdraw.draw.event
Layer3: org.jhotdraw.draw.handle
Layer3: org.jhotdraw.draw.io
Layer3: org.jhotdraw.draw.layouter
Layer3: org.jhotdraw.draw.liner
Layer3: org.jhotdraw.draw.locator
Layer3: org.jhotdraw.draw.print
Layer3: org.jhotdraw.draw.text
Layer3: org.jhotdraw.draw.tool
```

```
Layer2: org.jhotdraw.io
Layer2: org.jhotdraw.geom
Layer2: org.jhotdraw.net
Layer2: org.jhotdraw.text
Layer2: org.jhotdraw.undo
Layer2: org.jhotdraw.beans
Layer2: org.jhotdraw.xml.*
Layer2: org.jhotdraw.xml.css
```

```
Layer1: org.jhotdraw.util.*
Layer1: org.jhotdraw.util.prefs
Layer1: net.n3.nanoxml
```

Manual decomposition of JHotDraw 7.6

```
Layer4: org.jhotdraw.samples.color
Layer4: org.jhotdraw.samples.draw
Layer4: org.jhotdraw.samples.font
Layer4: org.jhotdraw.samples.mini
Layer4: org.jhotdraw.samples.net.*
Layer4: org.jhotdraw.samples.net.figures
Layer4: org.jhotdraw.samples.odg.*
Layer4: org.jhotdraw.samples.odg.action
Layer4: org.jhotdraw.samples.odg.figures
Layer4: org.jhotdraw.samples.odg.geom
Layer4: org.jhotdraw.samples.odg.io
Layer4: org.jhotdraw.samples.pert.*
Layer4: org.jhotdraw.samples.pert.figures
Layer4: org.jhotdraw.samples.svg.*
Layer4: org.jhotdraw.samples.svg.action
Layer4: org.jhotdraw.samples.svg.figures
Layer4: org.jhotdraw.samples.svg.gui
```

```
Layer4: org.jhotdraw.samples.svg.io
Layer4: org.jhotdraw.samples.teddy.*
Layer4: org.jhotdraw.samples.teddy.action
Layer4: org.jhotdraw.samples.teddy.io
Layer4: org.jhotdraw.samples.teddy.regex
Layer4: org.jhotdraw.samples.teddy.text

Layer3: org.jhotdraw.app.*
Layer3: org.jhotdraw.app.action.*
Layer3: org.jhotdraw.app.action.app
Layer3: org.jhotdraw.app.action.edit
Layer3: org.jhotdraw.app.action.file
Layer3: org.jhotdraw.app.action.view
Layer3: org.jhotdraw.app.action.window
Layer3: org.jhotdraw.app.osx
Layer3: org.jhotdraw.gui.*
Layer3: org.jhotdraw.gui.datatransfer
Layer3: org.jhotdraw.gui.event
Layer3: org.jhotdraw.gui.fontchooser
Layer3: org.jhotdraw.gui.plaf.*
Layer3: org.jhotdraw.gui.plaf.palette
Layer3: org.jhotdraw.gui.filechooser
Layer3: org.jhotdraw.gui.plaf.palette.colorchooser
Layer3: org.jhotdraw.samples.color
Layer3: org.jhotdraw.draw.*
Layer3: org.jhotdraw.draw.action
Layer3: org.jhotdraw.draw.connector
Layer3: org.jhotdraw.draw.decoration
Layer3: org.jhotdraw.draw.event
Layer3: org.jhotdraw.draw.handle
Layer3: org.jhotdraw.draw.io
Layer3: org.jhotdraw.draw.layouter
Layer3: org.jhotdraw.draw.liner
Layer3: org.jhotdraw.draw.locator
Layer3: org.jhotdraw.draw.print
Layer3: org.jhotdraw.draw.text
Layer3: org.jhotdraw.draw.tool
Layer3: org.jhotdraw.draw.gui

Layer2: org.jhotdraw.io
Layer2: org.jhotdraw.geom
Layer2: org.jhotdraw.net
Layer2: org.jhotdraw.text
Layer2: org.jhotdraw.undo
Layer2: org.jhotdraw.beans
Layer2: org.jhotdraw.xml.*
Layer2: org.jhotdraw.xml.css

Layer1: org.jhotdraw.util.*
Layer1: org.jhotdraw.util.prefs
Layer1: net.n3.nanoxml
```

Manual decomposition of JFreeChart 1.0.15

```
Layer3: org.jfree.chart.demo
Layer3: org.jfree.chart.editor
Layer3: org.jfree.chart.servlet
Layer3: org.jfree.chart.needle
Layer3: org.jfree.chart.panel
Layer3: org.jfree.chart.plot.dial

Layer2: org.jfree.chart.*
Layer2: org.jfree.chart.annotations
Layer2: org.jfree.chart.axis
Layer2: org.jfree.chart.block
Layer2: org.jfree.chart.encoders
Layer2: org.jfree.chart.entity
Layer2: org.jfree.chart.event
Layer2: org.jfree.chart.labels
Layer2: org.jfree.chart.plot.*
Layer2: org.jfree.chart.renderer.*
Layer2: org.jfree.chart.renderer.category
Layer2: org.jfree.chart.renderer.xy
Layer2: org.jfree.chart.title

Layer1: org.jfree.chart.util
Layer1: org.jfree.chart.imagemap
Layer1: org.jfree.chart.urls
Layer1: org.jfree.data.*
Layer1: org.jfree.data.category
Layer1: org.jfree.data.function
Layer1: org.jfree.data.gantt
Layer1: org.jfree.data.general
Layer1: org.jfree.data.io
Layer1: org.jfree.data.contour
Layer1: org.jfree.data.jdbc
Layer1: org.jfree.data.statistics
Layer1: org.jfree.data.time.*
Layer1: org.jfree.data.time.ohLAQ
Layer1: org.jfree.data.xml
Layer1: org.jfree.data.xy
```

LIST OF REFERENCES

Abdeen, H., Ducasse, S., Sahraoui, H., et Alloui, I. 2009. «Automatic package coupling and cycle minimization». In *Working Conference on Reverse Engineering (WCRE)*. (Lille, France, October 2009), p. 103-112. IEEE.

Abebe, S. L., and Tonella, P. 2011. «Towards the extraction of domain concepts from the identifiers». In *Reverse Engineering (WCRE), 2011 18th Working Conference on*. (Limerick, Ireland, October 2011)p. 77-86. IEEE.

Suryn, W., A. Abran et A. April. 2003. «ISO/IEC SQuaRE: The second generation of standards for software product quality». In *International Conference on Software Engineering and Applications* (ICSEA). (Marina del Rey, CA, USA, 2003).

Abran, A, Moore, J. W., Bourque, P. et Dupuis, R. 2004. *Guide to the Software Engineering Body of Knowledge*: Edition-SWEBOK. IEEE Computer Society.

Abran, A. 2010. *Software metrics and software metrology*. John Wiley & Sons.

Alshayeb, M., Naji, M., Elish, M. O., et Al-Ghamdi, J. 2011. «Towards measuring object-oriented class stability». *Software, IET*, vol. 5, n° 4, p. 415-424.

Andreopoulos, B., An, A., Tzerpos, V., et Wang, X. 2007. «Clustering large software systems at multiple layers». *Information and Software Technology*, vol. 49, n° 3, p. 244-254.

Andritsos, P., and Tzerpos, V. 2005. «Information-theoretic software clustering». *IEEE Transactions on Software Engineering.*, vol. 31, n° 2, p. 150–165.

Anquetil, N., et Lethbridge, T. C. 1999. «Recovering software architecture from the names of source files». *Journal of Software Maintenance*, vol. 11, n° 3, p. 201-221.

Apache: < http://ant.apache.org/ >. Accessed in January 2016.

Arcuri, A., and Briand, L. 2011. «A practical guide for using statistical tests to assess randomized algorithms in software engineering». In *International Conference on Software Engineering (ICSE)*. (Waikiki, Honolulu, Hawai, may 2011), p. 1-10. IEEE.

Arcuri, A., and Fraser, G. (2013). «Parameter tuning or default values? An empirical

investigation in search-based software engineering». *Empirical Software Engineering*, vol. 18, n° 3, p. 594-623.

Authoritative decomposition : <https://sites.google.com/a/etsmtl.net/kdmauthoritativedecompositions/>. Accessed in April 2016

Avgeriou, P., et Zdun, U. 2005. «Architectural patterns revisited–a pattern». In *10th European Conference on Pattern Languages of Programs* (EuroPlop).

Bachmann, F., Bass, L., Carriere, J., Clements, P. C., Garlan, D., Ivers, J., ... et Little, R. 2000. *Software architecture documentation in practice: Documenting architectural layers*. Special Report CMU/SEI-2000-SR-004, Software Eng. Inst., Carnegie Mellon University.

Bachmann, Felix, Len Bass et Robert Nord. 2007. *Modifiability tactics*. DTIC Document.

Baeza-Yates, R., and Ribeiro-Neto, B. 1999. *Modern information retrieval* (Vol. 463). New York: ACM press.

Bass, Len, Paul Clements and Rick Kazman. 2003. *Software Architecture in Practice*. Addison-Wesley, second edition.

Bass, Len, Paul Clements and Rick Kazman. 2012. *Software architecture in practice*. Addison-Wesley, third edition.

Battiti, R., and Tecchiolli, G. 1994. «The reactive tabu search ». *ORSA journal on computing*, Vol. 6, n° 2, p. 126-140.

Bavota, G., De Lucia, A., Marcus, A., et Oliveto, R. 2013. «Using structural and semantic measures to improve software modularization ». *Empirical Software Engineering*, vol. 18, n° 5, p. 901-932.

Bhattacharya, S., and Perry, D. E. 2005. «Predicting architectural styles from component specifications». In *5th IEEE/IFIP Working Conference on Software Architecture (WICSA)*. (Pittsburgh, Pennsylvania, USA, November 2005), p. 231-232. IEEE.

Bianchi, L., Dorigo, M., Gambardella, L. M., et Gutjahr, W. J. 2009. «A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing: an*

international journal, vol. 8, n° 2, p. 239-287.

Binkley, D., Heinz, D., Lawrie, D., et Overfelt, J. 2014. «Understanding LDA in source code analysis». In *Proceedings of the 22nd International Conference on Program Comprehension*. (Hyderabad, India, june 2014), p. 26-36. ACM.

Bittencourt, R. A., and Guerrero, D. D. S. 2009. «Comparison of graph clustering algorithms for recovering software architecture module views». In *Proc of European Conference on Software Maintenance and Reengineering*. (Kaiserslautern, Germany, may 2009), p. 251-254. IEEE CS Press.

Blei, D. M., Ng, A. Y., et Jordan, M. I. 2003. «Latent dirichlet allocation». *The Journal of Machine Learning Research*, vol. 3, p.993-1022.

Blum, C., and Roli, A. 2003. «Metaheuristics in combinatorial optimization: Overview and conceptual comparison». *ACM Computing Surveys (CSUR)*, 35(3), 268-308.

Boaye Belle, A. , El Boussaidi, G., Desrosiers, C., et Mili, H. 2013. «The layered architecture revisited: Is it an optimization problem». In *Proceedings of the 25th International Confference on Software Engineering and Knowledge Engineering (SEKE)*. (Boston, Massachussets, june 2013), p. 344-349.

Boaye Belle, A., El Boussaidi, G., et Mili, H. 2014. «Recovering software layers from object oriented systems». In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. (Lisbonne, Portugal, april 2014), p. 1-12. IEEE.

Boaye Belle, A. , El Boussaidi, G., et Mili, H. 2015. «ReALEITY: Recovering softwAre Layers from objEct-orIenTed sYstems». Poster In *Proceedings of the 12th Working IEEE / IFIP Conference on Software Architecture (WICSA)*. (Montreal, Canada, may 2015).

Boaye Belle, A., El Boussaidi, G., Desrosiers, C., Kpodjedo, S., et Mili, H. 2015. «The Layered Architecture Recovery as a Quadratic Assignment Problem». In *European Conference in Software Architecture (ECSA)*. (Dubrovnik, Croatia, September 2015), p. 339-354. Springer International Publishing.

Boaye Belle, A. B., El Boussaidi, G., et Kpodjedo, S. 2016. «Combining lexical and Structural information to reconstruct software layers». *Information and Software Technology*, vol. 74, p. 1-16.

Bourque, P., et Fairley, R. E. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

Büchi, M., et Weck, W. 1999. *The greybox approach: When blackbox specifications hide too much*. Technical Report 297, Turku Center for Computer Science.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., et Stal, M.. 1996. *Pattern-oriented software architecture, volume 1: A system of patterns*. John Wiley and Sons.

Buschmann, F., and Henney, K. (2003). «Explicit Interface and Object Manager». In EuroPLoP. (Irsee, Germany, june 2003), p. 207-220.

Buschmann, F., Henney, K., et Schmidt, D.C. 2007. *Pattern-Oriented Software Architecture. On Patterns and Pattern Languages*, vol. 5. John Wiley & sons.

Cai, Y., Wang, H., Wong, S., et Wang, L. 2013. «Leveraging design rules to improve software architecture recovery». In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. (Vancouver, Canada, June 2013), p. 133-142. ACM.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., et Little, R. 2002. *Documenting software architectures: views and beyond*. Pearson Education.

Comella-Dorda, S., Wallnau, K., Seacord, R. C., et Robert, J. 2000. *A survey of legacy system modernization approaches*. DTIC Document.

Constantinou, E., Kakarontzas, G., et Stamelos, I. 2011. «Towards open source software system architecture recovery using design metrics». In *Informatics (PCI), 15th Panhellenic Conference on* (Kastoria, Greece, September 2011). p. 166-170. IEEE.

Constantinou, E., Kakarontzas, G., et Stamelos, I. 2015. «An automated approach for noise identification to assist software architecture recovery techniques». *Journal of Systems and Software*, vol. *107*, 142-157.

Corazza, A., Di Martino, S., Maggio, V., & Scanniello, G. 2016. Weighing lexical information for software clustering in the context of architecture recovery. *Empirical Software Engineering*, vol 21, n° 1, p. 72-103.

Cunningham, W. H., Pulleyblank, W. R., et Schrijver, A. 1998. *Combinatorial optimization*, vol. 605. New York: Wiley.

De Lucia, A., Penta, M. D., Oliveto, R., Panichella, A., et Panichella, S. 2012. «Using IR methods for labeling source code artifacts: Is it worthwhile? ». In *Program Comprehension (ICPC), IEEE 20th International Conference on*. (Passau, Germany, june 2012), p. 193-202. IEEE.

De Oliveira Barros, M., de Almeida Farzat, F., et Travassos, G. H. 2015. «Learning from optimization: A case study with Apache Ant». *Information and Software Technology*, vol. 57, 684-704.

De Silva, L., and Balasubramaniam, D. 2012. «Controlling software architecture erosion: A survey». *Journal of Systems and Software*, *85*(1), 132-151

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., et Harshman, R. 1990. «Indexing by latent semantic analysis». *Journal of the American society for Information science*, vol. 41, n° 6, p. 391.

Dijkstra, E. W. 1968. The structure of the "THE"multiprogramming system. Commun. *ACM* vol. 11, n° 5.

Ding, L., and Medvidovic, N. 2001. «Focus: A light-weight, incremental approach to software architecture recovery and evolution». In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*. (Amsterdam, Netherlands, august 2001), p. 191-200. IEEE.

Ducasse, S., and Pollet, D. 2009. «Software architecture reconstruction: A process-oriented taxonomy». *Software Engineering, IEEE Transactions on*, vol. *35*, n° 4, p. 573-591.

Eeles, P. 2002. Layering Strategies. *Rational Software White Paper*.

El Boussaidi, G., Boaye Belle, A., Vaucher, S., Mili, H. 2012. «Reconstructing Architectural Views from Legacy Systems». In *19th Working Conference on Reverse Engineering (WCRE)*. (Kingston, Canada, october 2012), p. 345-354. IEEE.

Falleri, J. R., Denier, S., Laval, J., Vismara, P., et Ducasse, S. 2011. «Efficient retrieval and ranking of undesired package cycles in large software systems ». In *Objects, Models,*

Components, Patterns. (Zurich, Switzerland, June 2011), p. 260-275. Springer Berlin Heidelberg.

Fielding, R. T. 2000. «Architectural styles and the design of network-based software architectures». Doctoral dissertation, University of California, Irvine.

Forward, A., & Lethbridge, T. C. (2008, October). «A taxonomy of software types to facilitate search and evidence-based software engineering». In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*. (Richmond Hill, Ontario, Canada), p. 179-191. ACM.

Gamma, E., Helm, R., Johnson, R., et Vlissides, J.. 1994. *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Garcia, J., Popescu, D., Mattmann, C., Medvidovic, N., et Cai, Y. 2011. «Enhancing architectural recovery using concerns». In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. (Washington, DC, USA), p. 552-555. IEEE Computer Society.

Garcia, J., Ivkovic, I., et Medvidovic, N. 2013. «A comparative analysis of software architecture recovery techniques». In *Automated Software Engineerin (ASE), IEEE/ACM 28th International Conference on*. (Palo Alto, California November 2013), p. 486-496. IEEE.

Garlan, D., and Shaw, M. 1993. «An introduction to software architecture». *Advances in software engineering and knowledge engineering,* vol. 1, p. 1-40.

Garlan, D. 2000. «Software architecture: a roadmap». In *Proceedings of the Conference on the Future of Software Engineering.* (Limerick, Ireland, june 2000), p. 91-101. ACM.

Glover, F. 1989. «Tabu search-part I». *ORSA Journal on computing*, vol. 1, n° 3, p. 190-206.

Glover, F., Laguna, M. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

GraphViz: < http://www.graphviz.org/>. Accessed in April 2016.

Griffiths, T. L., and Steyvers, M. 2004. «Finding scientific topics». In *Proceedings of the National academy of Sciences of the United States of America*, 101(Suppl 1). P. 5228-5235.

Harman, M., Hierons, R. M., et Proctor, M. 2002. «A New Representation And Crossover Operator For Search-based Optimization Of Software Modularization». In *Proceedings of the Genetic and Evolutionary Computation Conference*. Vol. 2. (New York, USA , june 2002), p. 1351-1358.

Harman, M. 2007. «The current state and future of search based software engineering». In *Future of Software Engineering*. p. 342-357. IEEE Computer Society.

Harman, M., McMinn, P., De Souza, J. T., et Yoo, S. 2012. «Search based software engineering: Techniques, taxonomy, tutorial». *In Empirical Software Engineering and Verification,* p. 1-59. Springer Berlin Heidelberg.

Harris, D. R., Reubenstein, H. B., et Yeh, A. S. 1995. «Recognizers for extracting architectural features from source code». In *Reverse Engineering, Proceedings of 2nd Working Conference on*. (Toronto, Canada, july 1995). p. 252-261. IEEE.

Harrison, N. B., Avgeriou, P., et Zdun, U. 2007. «Using patterns to capture architectural decisions». *Software, IEEE*, vol. *24*, n° 4, 38-45.

Hassan, A. E., et Holt, R. C. 2002. «Architecture recovery of web applications». In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*. (Orlando, Florida, USA, may 2002), p. 349-359. IEEE.

Hautus, E. 2002. «Improving Java software through package structure analysis». In *the 6th IASTED International Conference Software Engineering and Applications*. (Cambridge, Massachusetts, USA, November 2002).

Hofmeister, C., Nord, R., et Soni, D. 2000. *Applied software architecture*. Addison-Wesley Professional.

Holland, J. H. 1975. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press.

Holt, R. 2002. «Software architecture as a shared mental model». In *Proceedings of the ASERC Workhop on Software Architecture*.

ISO/IEC, ISO/IEC DIS 19506. Information technology – Object Management Group Architecture-Driven Modernization (ADM) – Knowledge Discovery Meta-model (KDM), v1.1 (Architecture-Driven Modernization),

<http://www.iso.org/iso/catalogue_detail.1128htm?csnumber=32625>, 2009, ISO/IEC. p. 302.

jEdit: <http://www.jedit.org/>. Accessed in January 2016.

JFreeChart: < http://www.jfree.org/jfreechart/ >. Accessed in January 2016.

JHotDraw: <http://www.randelshofer.ch/oop/jhotdraw/ >. Accessed in January 2015.

JUnit: <http://junit.org/>. Accessed in January 2016.

Karafotias, G., Hoogendoorn, M., et Eiben, Á. E. 2015. «Parameter Control in Evolutionary Algorithms: Trends and Challenges». *IEEE Trans. Evolutionary Computation*, vol. 19, n° 2, p. 167-187.

Kazman, R., Woods, S. G., et Carrière, S. J. 1998. «Requirements for integrating software architecture and reengineering models: CORUM II». In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. (Honolulu, USA, October 1998), p. 154-163. IEEE.

Kazman, R., and Carrière, S. J. 1999. «Playing detective: Reconstructing software architecture from available evidence». In *Automated Software Engineering*, vol. *6*, n°2, p. 107-138.

Kienle, H. M., and Müller, H. A. 2010. «Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation». *Science of Computer Programming*, vol. *75*, n° 4, p. 247-263.

Kim, J. S., and Garlan, D. 2010. «Analyzing architectural styles». *Journal of Systems and Software*, vol. 83, n° 7, p. 1216-1235.

Kirkpatrick, S., and Vecchi, M. P. 1983. «Optimization by simmulated annealing». *Science*, vol. 220, n° 4598, p. 671-680.

Kitchenham, B. 2004. «Procedures for performing systematic reviews». *Keele, UK, Keele University*, vol. 33, n° 2004, p. 1-26.

Kleinberg, J. M. 1999. «Authoritative sources in a hyperlinked environment». *Journal of the ACM (JACM)*, vol. 46, n° 5, p. 604-632.

Koschke, R. 2009. «Architecture reconstruction». In *Software Engineering*, p. 140-173. Springer Berlin Heidelberg.

Kruchten, P. B. (1995). «The 4+ 1 view model of architecture». *Software, IEEE,* vol. 12, n° 6, p. 42-50.

Kuhn, A., Ducasse, S., et Gírba, T. 2007. «Semantic clustering: Identifying topics in source code». *Information and Software Technology*, vol. 49, n° 3, p. 230-243.

Laguë, B., Leduc, C., Le Bon, A., Merlo, E., et Dagenais, M. 1998. «An analysis framework for understanding layered software architectures». In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC).* (Ischia, Italy, june 1998), p. 37-44. IEEE.

Larman, Craig. 2005. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Pearson Education India.

Laval, J., Anquetil, N., Bhatti, U., et Ducasse, S. 2013. «OZONE: Layer Identification in the presence of Cyclic Dependencies». *Science of Computer Programming*, vol. 78, n° 8, p. 1055-1072.

Le, D. M., Behnamghader, P., Garcia, J., Link, D., Shahbazian, A., et Medvidovic, N. 2015. «An empirical study of architectural change in open-source software systems». In *Proceedings of the 12th Working Conference on Mining Software Repositories*. (Florence, Italy, may 2015), p. 235-245. IEEE Press.

Le Métayer, D. 1998. «Describing software architecture styles using graph grammars». *Software Engineering, IEEE Transactions on*, vol. 24, n° 7, p. 521-533.

Lehman, M. M., and Belady, L. A. 1985. *Program evolution: processes of software change*. Academic Press Professional, Inc..

Loiola, E. M., de Abreu, N. M. M., Boaventura-Netto, P. O., Hahn, P., et Querido, T. 2007. «A survey for the quadratic assignment problem». In *European Journal of Operational Research*, vol. *176*, n° 2, p. 657-690.

Lung, C. H., Zaman, M., et Nandi, A. 2004. «Applications of clustering techniques to

software partitioning, recovery and restructuring». *Journal of Systems and Software*, vol. 73, nº 2, p. 227-244.

Maletic, J. I., and Marcus, A. 2001. «Supporting program comprehension using semantic and structural information». In *Proceedings of the 23rd International Conference on Software Engineering*. (Toronto, Canada, may 2001), p. 103-112. IEEE Computer Society.

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y. F., et Gansner, E. R. 1998. «Using Automatic Clustering to Produce High-Level System Organizations of Source Code». In *Proceedings of the International Workshop on Program Comprehension (IWPC)*. Vol. 98. (Ischia, Italy, june 1998), p. 45-52.

Mancoridis, S., Mitchell, B. S., Chen, Y., et Gansner, E. R. 1999. «Bunch: A clustering tool for the recovery and maintenance of software system structures». In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM).* (Oxford, England, UK, August 30 - September 3, 1999), p. 50-59. IEEE.

Manning, C. D., Raghavan, P., et Schütze, H. 2008. *Introduction to information retrieva*l, vol. 1, nº 1, p. 496. Cambridge: Cambridge university press.

Maqbool, O., and Babri, H. A. 2007. «Hierarchical clustering for software architecture recovery». *Software Engineering, IEEE Transactions on*, vol. 33, nº 11, p. 759-780.

Martin, R. C. 2000. «Design principles and design patterns». *Object Mentor*, vol. 1, p. 34.

Maskeri, G., Sarkar, S., et Heafield, K. 2008. «Mining business topics in source code using latent dirichlet allocation». In *Proceedings of the 1st India software engineering conference*. (Tucson, Arizona, USA, September 2008), p. 113-120.

Medvidovic, N., and Jakobac, V. 2006. «Using software evolution to focus architectural recovery». *Automated Software Engineering*, vol. *13*, nº 2, p. 225-256.

Mendonça, N. C., and Kramer, J. 1996. «Requirements for an effective architecture recovery framework». In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*. (San Francisco, CA, USA, October, 1996), p. 101-105. ACM.

Meyer, B. 1988. *Object-oriented software construction*, (vol. 2, p. 331-410). New York: Prentice hall.

Mikkonen, T., Pitkänen, R., et Pussinen, M. 2004. «On the role of architectural style in model driven development». In *Proceedings of the First European Workshop Software Architecture*. (St Andrews, UK, May 2004), p. 74-87. Springer Berlin Heidelberg.

Misevicius, A. 2005. «A tabu search algorithm for the quadratic assignment problem». *Computational Optimization and Applications*, vol. *30*, n° 1, p. 95-111.

Mitchell, B., Traverso, M., et Mancoridis, S. 2001. «An architecture for distributing the computation of software clustering algorithms». In *Proceedings of Software Technology and Engineering Practice*. (August 2001), p. 181-190. IEEE.

Mitchell, B. S., and Mancoridis, S. 2002a. «Using Heuristic Search Techniques To Extract Design Abstractions From Source Code». In *Proceedings of the IEEE International Conference on Software Maintenance*. (Montreal, Canada, October 2002), p. 1375-1382.

Mitchell, B. S. 2002b. *A heuristic search approach to solving the software clustering problem*. Doctoral dissertation. Drexel University.

Mitchell, B. S., and Mancoridis, S. 2006. «On the automatic modularization of software systems using the bunch tool». *Software Engineering, IEEE Transactions on*, vol. 32, n° 3, 193-208.

Mitchell, B. S., and Mancoridis, S. 2008. «On the evaluation of the Bunch search-based software modularization algorithm». *Soft Computing*, vol. 12, n° 1, p. 77-93.

MoDisco: < http://www.eclipse.org/MoDisco/>. Accessed in March 2016.

Müller, H. A., Orgun, M. A., Tilley, S. R., et Uhl, J. S. 1993. «A reverse-engineering approach to subsystem structure identification». *Journal of Software Maintenance: Research and Practice*, vol. 5, n° 4, p. 181-204.

Liam O'Brien, Christoph Stoermer, et Chris Verhoef. 2002. *Software architecture reconstruction: Practice needs and current approaches*. Technical report. DTIC Document.

OMG. *Architecture-driven modernization: Transforming the enterprise*. Technical report, 2007.

OMG Specifications: <http://www.omg.org/>. Accessed in November 2015.

Oracle: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html >. Accessed in April 2016.

Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., et De Lucia, A. 2013. «How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms». In *Proceedings of the 2013 International Conference on Software Engineering*. (San Francisco, CA, may 2013), p. 522-531. IEEE Press.

Pardalos, P. M., et Wolkowicz, H. (Eds.). 1994. «Quadratic Assignment and Related Problems». *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 16, p. 1-42.

Paris, M. 2004. « Reuse-based layering: a strategy for architectural frameworks for learning technologies ». In *Proceedings of the Fourth IEEE International Conference on Advanced Learning Technologies (ICALT)*. (Joensuu, Finland, august 2004), p. 455-459. IEEE.

Pérez-Castillo, R., De Guzman, I. G. R., et Piattini, M. 2011. «Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems». *Computer Standards et Interfaces*, vol. 33, n° 6, p. 519-532.

Porter, M. F. 1980. «An algorithm for suffix stripping». *Program*, vol. 14, n° 3, p. 130-137.

Porter, M. F. 2006. «An algorithm for suffix stripping». *Program: electronic library and information systems*, vol. 40, n° 3, p. 211-218.

PorterStemmer: <http://tartarus.org/~martin/PorterStemmer/java.txt>. Accessed in April 2016.

Poshyvanyk, D., Marcus, A., Ferenc, R., et Gyimóthy, T. 2009. «Using information retrieval based coupling measures for impact analysis». *Empirical software engineering*, vol. 14, n° 1, p. 5-32.

Pressman, R. S. 2005. *Software engineering: a practitioner's approach*. Palgrave Macmillan.

ReALEITY: <https://sites.google.com/a/etsmtl.net/realeity_v1_0/home>. Accessed in January 2015.

Riva, C. 2002. «Architecture reconstruction in practice». In *IEEE/IFIP Working Conference on Software Architecture (WICSA)*. (Montreal, Canada, august 2002), p. 159-173. Springer US.

Russell, S., and Norvig, P. 2002. *Artificial intelligence: a modern approach*. 2nd Edition. Prentice Hall Series.

Saeidi, A. M., Hage, J., Khadka, R., et Jansen, S. 2015. «A search-based approach to multi-view clustering of software systems». In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*. (Montreal, Canada, march 2015), p. 429-438. IEEE.

Sangal, N., Jordan, E., Sinha, V., et Jackson, D. 2005a. «Using dependency models to manage complex software architecture ». In *ACM Sigplan Notices*, vol. 40, n° 10, p. 167-176. ACM.

Sangal, N., et Waldman, F. 2005b. «Dependency models to manage software architecture». In *the Journal of Defense Software Engineering*.

Sarkar, S., Rama, G. M., et Kak, A. C. 2007. «API-based and information-theoretic metrics for measuring the quality of software modularization». *Software Engineering, IEEE Transactions on*, vol. 33, n° 1, p. 14-32.

Sarkar, S., Maskeri, G., et Ramachandran, S. 2009. «Discovery of architectural layers and measurement of layering violations in source code». *Journal of Systems and Software*, vol. 82, n° 11, p. 1891-1905.

Sartipi, K. 2003. «Software architecture recovery based on pattern matching». In *Proceedings of the International Conference on Software Maintenance (ICSM).* (Amsterdam, The Netherlands, September 2003), p. 293-296. IEEE.

Scanniello, G., D'Amico, A., D'Amico, C., et D'Amico, T. 2010a. «Architectural layer recovery for software system understanding and evolution». *Software: Practice and Experience*, vol. 40, n° 10, p. 897-916.

Scanniello, G., D'Amico, A., D'Amico, C., et D'Amico, T. 2010. « Using the kleinberg algorithm and vector space model for software system clustering». In *the International Conference on Program Comprehension (ICPC)*. (Braga, Portugal, july 2010), p. 180-189. IEEE.

Schmidt, D. C., Stal, M., Rohnert, H., et Buschmann, F. 2000. *Patterns for concurrent and distributed objects*. Pattern-oriented software architecture.

Schmidt, F., MacDonell, S. G., et Connor, A. M. 2012. «An automatic architecture reconstruction and refactoring framework». In *Software Engineering Research, Management and Applications*. p. 95-111. Springer Berlin Heidelberg.

Schrijver, A. 2003. «Combinatorial optimization: polyhedra and efficiency» (Vol. 24). *Springer Science et Business Media*.

Seacord, R. C., Plakosh, D., et Lewis, G. A. 2003. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

Shaw, M., and Garlan, D. 1996. *Software architecture: perspectives on an emerging discipline* (Vol. 1, p. 12). Englewood Cliffs: Prentice Hall.

Shaw, M., and Clements, P. 1997. «A field guide to boxology: Preliminary classification of architectural styles for software systems». In *Computer Software and Applications Conference*. p. 6-13. IEEE.

Skorin-Kapov, J. 1990. «Tabu search applied to the quadratic assignment problem». *ORSA Journal on computing*, vol. 2, n° 1, p. 33-45.

Smith, J. E. 2008. «Self-adaptation in evolutionary algorithms for combinatorial optimisation». *In Adaptive and Multilevel Metaheuristics*, p. 31-57. Springer Berlin Heidelberg.

Sneed, H. M. 2005. «Estimating the costs of a reengineering project». In *Reverse Engineering, 12th Working Conference on*. (Pittsburgh, Pennsylvania, USA, November 2005), p. 9. IEEE.

Sommerville, I. 2007. *Software Engineering*. Addison-Wesley, eight edition.

Sora, I., Glodean, G., et Gligor, M. 2010. «Software architecture reconstruction: An approach based on combining graph clustering and partitioning». In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on.* (Timisora, Romania, may 2010), p. 259-26. IEEE.

Spearman: < http://www.socscistatistics.com/tests/spearman>. Accessed in november 2015.

Steyvers, M. and Griffiths, T. 2007. Probabilistic Topic Models in Latent Semantic Analysis: A Road to Meaning, Landauer, T. and Mc Namara, D. and Dennis, S. and Kintsch, W., eds. Lawrence Erlbaum.

Stoermer, C., O'Brien, L., et Verhoef, C. 2003. «Moving towards quality attribute driven software architecture reconstruction». In *Proceedings of the 10th Working Conference on Reverse Engineering*. (Victoria, Canada, November 2003), p. 46. IEEE.

Szyperski, C.A.: *Component Software*. Addison Wesley, 1998.

Taillard, E. 1991. «Robust Taboo Search for the Quadratic Assignment Problem». *Paral. Comput.,* vol. 17, p. 443–455.

Tamzalit, D., and Mens, T. 2010. «Guiding architectural restructuring through architectural styles ». In *Engineering of Computer Based Systems (ECBS),* 17th *IEEE International Conference and Workshops on*. (Prague, Czech Republic, may 2010), p. 69-78. IEEE.

Tibermacine, C., Sadou, S., Dony, C., et Fabresse, L. 2011. «Component-based specification of software architecture constraints ». In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. (Boulder, Colorado, USA, june 2011), p. 31-40. ACM.

Tzerpos, V., and Holt, R. C. 1997. «The orphan adoption problem in architecture maintenance». In *Proceedings of the Fourth Working Conference on Reverse Engineering*. (Amsterdam, The Netherlands, October 1997), p. 76-82. IEEE.

Tzerpos, V., and Holt, R. C. 2000. «ACDC: An algorithm for comprehension-driven clustering». In *Proceedings of the Working Conference on Reverse Engineering*. (Brisbane, Australia, November 2000), p. 258. IEEE.

Ulrich, W. M., and Newcomb, P. 2010. *Information systems transformation: architecture-*

driven modernization case studies. Morgan Kaufmann.

Volter, M., Kircher, M., et Zdun, U. 2005. *Remoting patterns*. John Wiley et Sons.

Wiggerts, T. A. 1997. «Using clustering algorithms in legacy systems remodularization». In *Proceedings of the Fourth Working Conference on Reverse Engineering*. (Amsterdam, The Netherlands, October 1997), p. 33-43. IEEE.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., et Wesslén, A. 2012. *Experimentation in software engineering*. Springer Science et Business Media.

XML: <http://www.w3.org/XML/ >. Accessed in January 2016.

XML transformation: <http://www.microhowto.info/howto/process_an_xml_document_using_an_xslt_stylesheet_in_java.html >. Accessed in January 2016.

Xu, R., and Wunsch, D. 2005. «Survey of clustering algorithms». *Neural Networks, IEEE Transactions on*, vol. 16, n° 3, p. 645-678.

Yan, H., Garlan, D., Schmerl, B., Aldrich, J., et Kazman, R. 2004. «Discotect: A system for discovering architectures from running systems». In *Proceedings of the 26th International Conference on Software Engineering*. (Edinburgh, United Kingdom, may 2004), p. 470-479. IEEE Computer Society.

Yao, L., Mimno, D., et McCallum, A. 2009. «Efficient methods for topic model inference on streaming document collections». In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. (Paris, France, june 2009), p. 937-946). ACM.

Yin, R. K. 2009. *Case study research: Design and methods*. Thousand Oaks, fourth edition.

Zdun, U., and Avgeriou, P. 2008. «A catalog of architectural primitives for modeling architectural patterns». *Information and Software Technology*, vol. 50, n° 9, p. 1003-1034.

Zhang, Q., Qiu, D., Tian, Q., et Sun, L. 2010. «Object-oriented software architecture recovery using a new hybrid clustering algorithm». In *Fuzzy Systems and Knowledge Discovery (FSKD), Seventh International Conference on*. (Yantai, China, 2010),

Vol.6. p. 2546-2550. IEEE.

Zest: <http://www.eclipse.org/gef/zest/ >. Accessed in march 2016.

Zimmermann, H. (1980). «OSI Reference Model--The ISO Model of Architecture for Open Systems Interconnection». *IEEE Transactions on Communications*, vol. 28, nº 4, p.425-432.