UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VANIUS ZAPALOWSKI

Evaluation of Code-based Information to Architectural Module Identification

Thesis presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Prof. Daltro José Nunes
Advisor

Prof. Ingrid Nunes
Coadvisor

Porto Alegre, February 2014

CIP – CATALOGING-IN-PUBLICATION

Zapalowski, Vanius

Evaluation of Code-based Information to Architectural Module Identification / Vanius Zapalowski. – Porto Alegre: PPGC da UFRGS, 2014.

102 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2014. Advisor: Daltro José Nunes; Coadvisor: Ingrid Nunes.

1. Software architecture. 2. Architecture reconstruction. 3. Architecture recovery. 4. Software architecture view. 5. Reverse engineering. I. Nunes, Daltro José. II. Nunes, Ingrid. III. Título.

"If I have seen farther than others,
it is because I stood on the shoulders of giants."
— Sir Isaac Newton

ACKNOWLEDGMENTS

First and foremost, I wish to express my love and gratitude to my parents, Irani and Walter, my fiancee, Paula, for their love and support. Without my parents support, I would not get this accomplishment, neither the ones that will succeed. Also, my fiancée comprehension, encouragement and faith were fundamental to the accomplishment of this work.

I own much to my advisors Professor Daltro Nunes and Professor Ingrid Nunes for their patience, guidance and support through the two years of my work. I am greatly thankful for Daltro's teachings and support during these period. I would like to express my great gratitude to Professor Ingrid for her active interest (that is exemplary to any advisor), methodological way of research, and research and life lessons learnt that I will carry for all my life.

I would like to thank the members of my examining committee, Professors Uirá Kulesza, Karin Becker and Leandro Wives, for reading my dissertation, giving me feedback to improve my research.

I also must thank all my friends for their friendship. They contributed much to define the perspectives I have today and many times helped me to keep the clarity of my thoughts. I specially thank my friends Gabriel Fernandes for his great contribution during all this work and Leonardo Borba for his fellowship during many discussions at Ildo. All the fellows (Jaime, Diego, Jacob, Elen, Sergio, Carlos and Vinicius) of the laboratory 204 of building 67 that made my days happier.

My thanks to the Informatics Institute that supported me in all the ways since my undergraduate to the end of my master in a excellent manner.

Finally, I would like to thank all of the people that the name is not in this text but have helped me in the preparation of this dissertation in any way.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
AVG	Average
BPMN	Business Process Model and Notation
CNW	Class Name Words
CSV	Comma-Separated Values
DAG	Direct Acyclic Graph
DIFF	Difference
DIT	Depth of Inheritance Tree
EC	Expert Committee
EM	Expectation Maximization
IDE	Integrated Development Environment
INT	Interface Usage
Metrics	Eclipse Metrics Plugin Continued
MIN	Minimum
MLOC	Mean Method Size
MMC	Mean Methods Complexity
MVC	Model-View-Controller
NOA	Number of Attributes
NOGS	Number of Getters and Setters
NOM	Number of Methods
NORM	Number of Overridden Methods
NSA	Number of Static Attributes

NSC	Number of Children
NSM	Number of Static Methods
OLIS	OnLine Intelligent Services
RecSys	Recommender System
RoR	Ruby on Rails
SC	Superclass Usage
SD	Standard Deviation
TDD	Test-driven Development
TLOC	Total Lines of Code
WMC	Weighted Methods per Class

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Software architecture plays an important role in the software development, and when explicitly documented, it allows understanding an implemented system and reasoning about how non-functional requirements are addressed. In spite of that, many developed systems lack proper architecture documentation, and if it exists, it may be outdated due to software evolution. The process of recovering the architecture of a system depends mainly on developers' knowledge requiring a manual inspection of the source code. Research on architecture recovery provides support to this process. Most of the existing approaches are based on architectural elements dependency, architectural patterns or source code semantics, but even though they help identifying architectural modules, the obtained results must be significantly improved to be considered reliable. We thus aim to support this task by the exploitation of different code-oriented information and machine learning techniques. Our work consists of an analysis, involving five case studies, of the usefulness of adopting a set of code-level characteristics (or features, in the machine learning terminology) to group elements into architectural modules. The characteristics — mainly source code metrics — that affect the identification of what role software elements play in software architecture are unknown. Then, we evaluate the relationship between different sets of characteristics and the accuracy achieved by an unsupervised algorithm — the Expectation Maximization — in identifying architectural modules. Consequently, we are able to understand which of those characteristics reveal information about the source code structure. By the use of code-oriented information, our approach achieves a significant average accuracy, which indicates the importance of the selected information to recover software architecture. Additionally, we provide a tool to support research on architecture recovery providing software architecture measurements and visualizations. It presents comparisons between predicted architectures and concrete architectures.

Keywords: Software architecture, architecture reconstruction, architecture recovery, software architecture view, reverse engineering.

Avaliação da Relevância de Informações do Código Fonte para Identificar Módulos da Arquitetura de Software.

RESUMO

Arquitetura de software desempenha um importante papel no desenvolvimento de software, quando explicitamente documentada, ela melhora o entendimento sobre o sistema implementado e torna possível entender a forma com que requisitos não funcionais são tratados. Apesar da relevância da arquitetura de software, muitos sistemas não possuem uma arquitetura documentada, e nos casos em que a arquitetura existe, ela pode estar desatualizada por causa da evolução descontrolada do software. O processo de recuperação de arquitetura de um sistema depende principalmente do conhecimento que as pessoas envolvidas com o software tem. Isso acontece porque a recuperação de arquitetura é uma tarefa que demanda muita investigação manual do código fonte. Pesquisas sobre recuperação de arquitetura objetivam auxiliar esse processo. A maioria dos métodos de recuperação existentes são baseados em dependência entre elementos da arquitetura, padrões arquiteturais ou similaridade semântica do código fonte. Embora as abordagem atuais ajudem na identificação de módulos arquiteturais, os resultados devem ser melhorados de forma significativa para serem considerados confiáveis. Então, nesta dissertação, objetivamos melhorar o suporte a recuperação de arquitetura explorando diferentes fontes de informação e técnicas de aprendizado de máquina. Nosso trabalho consiste de uma análise, considerando cinco estudo de casos, da utilidade de usar um conjunto de características de código (*features*, no contexto de aprendizado de máquina) para agrupar elementos em módulos da arquitetura. Atualmente não são conhecidas as características que afetam a identificação de papéis na arquitetura de software. Por isso, nós avaliamos a relação entre diferentes conjuntos de características e a acurácia obtida por um algoritmo não supervisionado na identificação de módulos da arquitetura. Consequentemente, nós entendemos quais dessas características revelam informação sobre a organização de papéis do código fonte. Nossa abordagem usando características de elementos de software atingiu uma acurácia média significativa. Indicando a relevância das informações selecionadas para recuperar a arquitetura. Além disso, nós desenvolvemos uma ferramenta para auxílio ao processo de recuperação de arquitetura de software. Nossa ferramenta tem como principais funções a avaliação da recuperação de arquitetura e apresentação de diferentes visualizações arquiteturais. Para isso, apresentamos comparações entre a arquitetura concreta e a arquitetura sugerida.

Palavras-chave: Arquitetura de Softwar, Reconstrução de Arquitetura, Recuperação de Arquitetura, Visualização de Arquitetura, Engenharia Reversa.

1 INTRODUCTION

Over the last decades, software complexity has grown significantly, given the variety of technologies and the popularization of electronic devices. Thus, handling the software planning and evolution is a complex task. In order to understand what a piece of source code does, usually a manual investigation is performed (GARCIA; IVKOVIC; MEDVIDOVIC, 2013). Consequently, the effort demanded to control the software development is proportional to its size because a manual investigation of the whole software is needed to comprehend the main software concepts. Then, in order to achieve a healthy and controlled software life-cycle to large-scale systems, there are intermediate abstractions between the source code and the real world (HOCHSTEIN; LINDVALL, 2005). Their aim is to represent the source code with less technical details.

One useful software abstraction is the software architecture model (SHAW; CLEMENTS, 2006). It describes the main concepts applied in software through a high-level model. This model captures the concepts applied during the software planning and development phases. It presents these concepts in a comprehensive model increasing the knowledge about software structures. The importance of having software architecture is even more evidence when the software evolves. The lack of documentation makes the developers resort in experts' knowledge, manual investigations or reverse engineering approaches. Performing an architecture recovery based on manual investigations demands effort and experts' knowledge is unreliable. Then, reverse engineering is commonly used to extract at least an initial architectural model to be refined by architects because it reduces the effort needed and is a reliable source of information to recover a software architecture. Moreover, projects that have their architecture documented in conformance with the behavior of the source code have improvements in reuse, construction, evolution, analysis, and management (GARLAN, 2000).

Despite of the importance of having a software architecture updated, a common scenario in software development is the absence of software documentation, which includes architecture. Furthermore, in the cases that a documentation exists, it is usually outdated (KAZMAN; CARRIERE, 1999). Aiming to have reliable documentation about projects, developers use reverse engineering techniques to recover documentation (semi-) automatically. In the particular case of retrieving software architecture from the project source code, there are many approaches proposed, which are related to the architecture recovery research. Such approaches extract information from source code, architectural patterns, runtime executions and any other processes that generate information related to software in order to recovery the concrete architecture (DUCASSE; POLLET, 2009). Architecture recovery research has considered mainly three types of information to identify software architectures: (i) architectural patterns; (ii) architectural element dependencies; and (iii) source code semantics. These three sources of information in fact have relationship with software architecture, but it is unknown the relevance of each of them to software architecture, i.e. which of them have more architectural information. Furthermore, to obtain a software architecture model automatically, architecture recovery approaches are commonly supported by machine learning techniques.

Next, in Section 1.1, we specify the problem that we are addressing pointing out the limitations of related work. In Section 1.2, we present how we handle the problem. Finally, in Section 1.3, we show the structure of this dissertation.

1.1 Problem Statement and Related Work Limitations

Given the benefits of having a software architecture documented and up-to-date, current architecture recovery methods aim to improve the reliability of recovering software architecture (semi-) automatically. Analyzing the work related to software architecture recovery, we identified issues to be improved in this context. These problems are detailed below.

Current software architecture recovery methods are unreliable to recover the software architecture without manual investigation of the target system. In spite of the effort applied to improve the recovered architecture reliability, architecture recovery methods are still predicting inaccurate software architectures. Garcia, Ivkovic and Medvidovic (2013) performed a comparative analysis of six different software architecture recovery methods. They evaluated the selected techniques and concluded that even the top-performing approaches are insufficient to reliably perform an architecture recovery. Additionally, Ducasse and Pollet provided an overview about the software architecture reconstruction techniques (DUCASSE; POLLET, 2009). They analyzed 13 software architecture recovery techniques, detailing their strengths and weaknesses. They also pointed out the inefficiency of bottom-up approaches in extracting architectures. Both studies agree that there is much to be done to improve the software architecture recovery methods in order to reach a satisfactory level of automation in the current methods.

Lack of information related to software architecture. Software architecture is related to many aspects of the software. The current software architecture recovery approaches focus on extracting the three types of information (architectural patterns, architectural element dependencies and source code semantics) commented in the previous section. However, it is unknown which type of information is more related to software architecture because each method uses a different type of information and with a different form of group element in modules. Moreover, software architecture has many other aspects to be considered, such as source code metrics. Furthermore, the combination of different sources of information is also disregarded in current approaches. In fact, Constantinou, Kakarontzas and Stamelos proposed an architecture recovery method using source code metrics to refine the results of obtained from a system dependency graph (CONSTANTINOU; KAKARONTZAS; STAMELOS, 2011). Although, their evaluation was limited, they contributed with the adoption of new attributes of architectural elements related to software architecture.

Lack of architectural views to different purposes. There are many ways to represent software architecture. Depending on the purpose that is being taken into account, the architectural view used should change to highlight particular software interests. Ducasse and Pollet (2009) discussed the need of different forms to represent an architecture. Research on software architecture recovery has its main support on data analysis. However, software architecture main objective is to improve the system comprehension providing a model (usually graphical). The comparison of documented and predicted architectures is complex due to the knowledge demanded to understand both.

Software architecture recovery still is a challenge subject because all problems presented remain with unreliable solutions. Architecture recovery approaches have been proposed to tackle these issues, but there is much to improve in the current approaches in order to achieve a reliable architecture recovery method mainly in the information selection, data relevance and architecture visualization.

1.2 Proposed Solution and Contributions Overview

This dissertation covers an evaluation of different types of code-extract information in order to improve the reliability of architecture recovery processes revealing their relationship with software architecture. Our work provides a generic procedure to evaluate different types of code-extract information using machine learning techniques to identify architectural modules. Moreover, we evaluate our experiments according to consolidated machine learning measures. These measures quantify the relation between software characteristics analyzed and the results achieved by our empirical study.

We list our main contributions below.

1. A standardized procedure to evaluate the relationship between source code information and software architectural modules.
2. Identification of which architectural elements attributes are more significant to recover architectural modules.
3. Architectural views focused on comparison of predicted and documented software architectures improving the understand about software architecture recovery.
4. Point out guidelines of architecture recovery scenario analyzing the importance of types of information, machine learning techniques and evaluations that could contribute to the reliability of the process.

1.3 Outline

In this Section, we present the structure of this dissertation. It is as follows.

In Chapter 2, we present the current scenario of the two main areas related to this dissertation, software architecture and machine learning, detailing their contributions to software engineering and providing foundation to our experiment decisions.

Chapter 3 presents related work. In order to discuss their strengths and limitations, we divided the related work by their type of information extracted to recover the software architecture.

Chapter 4 describes the procedure proposed to evaluate the relevance of the software architectural elements. We set the selected software information to evaluate, the data preparation, the adopted learning process and the evaluation measures.

In Chapter 5, we provide details about the tool developed to support the experiments on software architecture recovery. We present the implemented features related to the architecture recovery process, explaining the three visualizations developed and discuss technical details about the tool.

Chapter 6 details the results achieved by each set of experiments presenting quantitative data to compare the selected source of information. This Chapter bases our arguments to derive our conclusions.

Chapter 7 presents the lessons learned during the experiments. We discuss each step of the evaluation pointing out their effect in the procedure based on the results achieved.

Finally, in Chapter 8, we present the conclusions derived from the whole evaluation process. Additionally, future work related to our conclusions are presented.

2 BACKGROUND

Software architecture recovery aims to obtain an architecture documentation extracting known concepts from source code and often applying machine learning techniques. Thus, this Chapter provides an overview of software architecture, detailing its concepts, benefits and guidelines of architecture recovery in Section 2.1. Additionally, we present an overview of machine learning and aspects of machine learning techniques related to the learning process of architecture recovery in Section 2.2.

2.1 Software Architecture

Since the 1980's, when software architecture foundations began to be studied, to nowadays, software architecture gains attention of different computer science areas. In the academic area, the community has focused on understanding the concepts and principles of different software architectures. They are interested in studying how software architecture improves software quality and contributes to a controlled evolution. On the other hand, the software industry sees software architecture as a way to improve the quality of its products aggregating value to its clients and reducing development and maintenance costs.

The purpose of software architecture is to describe the core structures of systems in a high-level model that clearly presents architectural elements relationships and roles (PERRY; WOLF, 1992; SHAW; CLEMENTS, 2006; CLEMENTS; SHAW, 2009; BASS; CLEMENTS; KAZMAN, 2012). Researchers have been using different ways to define a software architecture, because systems can be defined through many perspectives. However, all definitions typically adopt terms such as relation, elements and structures — since these are in fact the main parts of a software. Because of this controversy surrounding software architecture definition, it is important to state the definition adopted in this work: "The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relation among them, and proprieties of both" (BASS; CLEMENTS; KAZMAN, 2012).

Software architecture groups the main concepts and principles that must be followed to implement good software practices, which are fundamental to the long-term health of the software (HOCHSTEIN; LINDVALL, 2005). In order to improve the software quality maintaining a high-level documentation, there are six contributions of a software architecture to a project: understanding, reuse, construction, evolution, analysis, and management. They have significant impact in the software (GARLAN, 2000). These contributions give evidence to the importance of planning and maintaining an architectural model because software design, development and evolution depend on all these factors.

With the software architecture properly documented, the concepts that guide the development are presented in a high-level model, thus enabling developers to understand such concepts. When good practices are followed, during the software architecture planning and maintenance, the model has elements categorized together according to their similar behaviors or functionalities. This defines how to group elements to form an architectural module.

Similarly to what happens in software design, there are recurrent problems, such as the separation of concerns, that occurs in architectural planning. Then, inspecting architectures of software from similar domains or with similar purpose, it is possible to identify recurrent solutions to the recurrent problems. Based on these recurrent problems and solutions, the term *architectural pattern* (BUSCHMANN; HENNEY; SCHMIDT, 2007) was adopted to specify groups of architectural roles and rules that solve a certain architectural problem. With the intention of spreading the knowledge of architectural patterns, architectural pattern catalogues were created to document these recurrent problems and solutions. These catalogues document many patterns, but many architectural patterns still are undocumented (PERRY; WOLF, 1992). One of the most commonly applied architectural patterns is the layered architecture, illustrated in Figure 2.1. In this pattern, the architectural modules are separated by their roles in the software: (i) the *Data* module handles how to persist the system data; (ii) the *Business* module performs the logical operations; and (iii) *Presentation* module is responsible for the interaction with the users. The communication rules of this pattern restrict the communication between modules to the layer immediately below itself, for instance the *Business* module depends only on *Data* module.

Indeed, architectural patterns are recurrent solutions, roles and rules that must be respected to organize software structures in a way that they obey good practices and leads to high-quality software. Additionally, they have solutions needed in

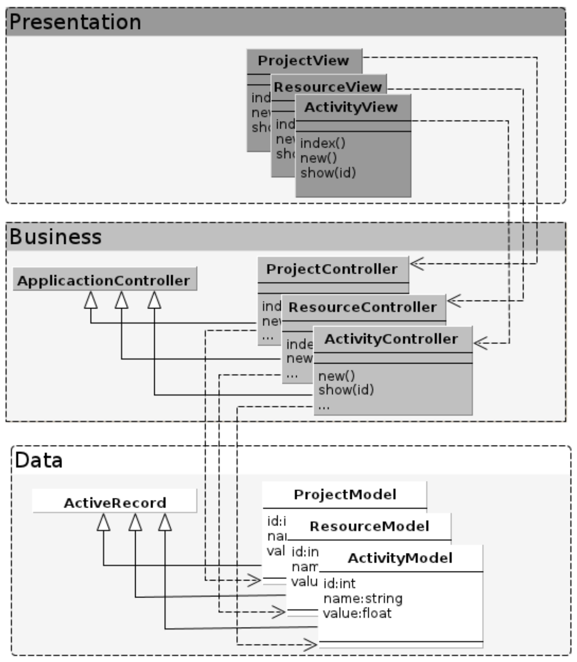


Figure 2.1: System structured with the layered architectural pattern.

common project situations. However, even similar projects have particularities that diverge from a specific architectural pattern. These divergences force an adaptation in the pattern to solve these particularities. Then, these changes generate different architectures and consequently new candidate patterns.

Summarizing, the main objective of a software architecture is document fundamental project decisions in order to be easily and clearly accessed through the software development and evolution. Nowadays, projects have a high complexity. For this reason, it is a common situation when software architecture model is not generated. Even in the cases that it is documented, the model usually is outdated.

2.1.1 Concepts and Terminology

During the thirty years of studies about software architecture, there is no standardized way to refer to important concepts of this subject. Then, to avoid misunderstanding of the objectives addressed by this work, we introduce, in this section, the terms used in the remainder of this dissertation. We adopted as reference to establish our nomenclature the study of Ducasse and Pollet (2009).

Architectural Element

The definition of architectural element usually vary according to the studies objectives. In fact, it could be any part of the software, such as packages, files (ANQUETIL; LETHBRIDGE, 1999) and methods (CORAZZA et al., 2011). In our work, we consider that each class of our case studies is an element of their architectures.

Architectural Module

An architectural module is a group of architectural elements that can be seen as the same according to some criteria. We consider elements that have the same role in the system, as part of modules. For instance, two elements that handle business logic are grouped in the Business module.

Conceptual Architecture

Conceptual architecture represents how the people involved with the software believe the architecture is implemented. It is a model that could diverge from the implemented source code. The documentation form is irrelevant: in a formal document with elements and relations explicit; in a piece of paper; or even just in the architect's mind. So, this concept defines that an architecture exists even not formally. Alternative terms for conceptual architecture are: as-idealized (HARRIS; REUBENSTEIN; YEH, 1995), intended (CARRIERE; KAZMAN, 1998), as-designed (KAZMAN; CARRIERE, 1999), or logical (MEDVIDOVIC; JAKOBAC, 2006) architecture.

Concrete Architecture

The term concrete is related to the real architecture. Concrete architecture means that it reflects what is really coded in the software, i.e. the documented architecture is the concrete architecture when the document agrees with the real modules organization and the communication is correctly represented. In literature, this concept can also be referred to as as-implemented (KAZMAN; CARRIERE, 1999), as-built (HARRIS; REUBENSTEIN; YEH, 1995), realized (CARRIERE; KAZMAN, 1998), or physical (MEDVIDOVIC; JAKOBAC, 2006) architecture.

Architectural View

As software is complex, there are many perspectives to represent different architectural concepts, such as communication or dataflow structure. So, it is possible to view a software from different concerns. Then, an architectural view is a way to organize elements and relations that focus on presenting a group of concerns (STANDARD-1471, 2000). In our study, the architectural concern is the modules view. This view consider software parts that handle different tasks and group the elements according to their tasks in a proper module.

Architectural Pattern

Architectural patterns play the same role as design patterns play in the context of design. The concept is the same, the only difference is the model level of abstraction. Patterns, for this definition, are abstract solutions for recurrent problems in an architectural scope, e.g. Model-View-Controller (MVC) pattern commonly used in the web-context. Architectural patterns are also known as architectural styles.

2.1.2 Software Architecture Recovery

Every software has an architecture, it is not necessarily documented or known, but the architecture is there (BASS; CLEMENTS; KAZMAN, 2012). The elements and their relationship can always be represented through a software architectural view. In some cases the architecture is trivial, as a system with just one element, but it still is an architecture. Then, software architecture is an intrinsic part of the software systems.

Usually, when a software project starts, an architecture is planned to be followed during the development. As a consequence of the product evolution, new features are demanded and, bugs need to be fixed. However, the architecture documentation is frequently not updated to reflect the source code state. Although the process of evolution must be planned and controlled, the time to market and reduction of costs are priority properties in large scale software. The software evolution is inevitable. As a consequence, the gap between architecture documentation and the implemented source code increases because the documentation is rarely updated. This problem creates a divergence between documentation and source code. Consequently, this divergence makes the architecture documentation unreliable to the development team reason about future decisions.

Changes in software are a natural consequence of evolution, but there must be a controlled process of change. Sometimes, due to time constraints and budget, projects do not have a documentation, which forces software engineers to reconstruct the architecture. In the cases that a documentation exists, we must maintain models up-to-date and make changes that respect architectural concepts and decisions (PARNAS, 1994). However, these practices are hard tasks to perform, mainly because of the lack of knowledge about the software.

In order to tackle the lack and divergence of software documentation, techniques have been researched to reduce time and effort dedicated to recover the software architecture (DUCASSE; POLLET, 2009). They are based on the extraction of information from source code, documentation, runtime executions and any other process that generates information related to software to recover the architecture which is implemented.

When a development process is uncontrolled, a phenomenon like architecture erosion is commonly found. An erosion (SILVA; BALASUBRAMANIAM, 2012) in software occurs when a change in architecture is made, and, as a side effect, this change violates a conceptual architectural principle. The introduction of a violation in architecture compromises the maintainability, because the information given by the model is not guided by the architectural principles, what turn the model hard to be followed.

Maintaining a synchronism between architecture and source code manually is a hard task that consumes time and requires a professional with knowledge about the software. Both requirements are precious resources that can not be wasted. Reinforcing this idea, Kazman (KAZMAN; O'BRIEN; VERHOEF, 2001) reported that this task is impossible to undertake only manually.

Aiming to tackle those kind of problems, reverse engineering techniques, such as architecture recovery, have been studied to improve the quality, to provide a high-level of abstraction model, and to reduce the cost (semi-) automating the task of updating software architecture documentation.

Besides the importance of having a software architecture, many projects do not have a documented architecture. Furthermore, if the software architecture exists, its documentation becomes often outdated due to evolution process (KNODEL et al., 2006; CLEMENTS; SHAW, 2009; PASSOS et al., 2010). This fact is confirmed by Kazman and Carriere, who states that "many systems have no documented architecture at all" (KAZMAN; CARRIERE, 1999). When an architectural documentation does not exist, the source code and the knowledge of the people involved are the documentation available. Then, the most reliable approach is to extract architectural information from the source code. So, starting with low-level information and iteratively refining the model to a high-level model is the purpose of architectural recovery techniques, also known as bottom-up techniques. Based on the information provided by the source code, an analysis of similarities is performed in order to group elements forming cluster. As a result of the clustering, a high-level model is built to be checked by architects. These main steps of the architecture recovery process are illustrated in Figure 2.2.

Recovering the architecture from a large-scale software manually is practically impossible. The complexity of the software requires knowledge about all modules and a deep analysis of the relationship between each other. Therefore, recovery processes automate part of this work, inferring information through the extraction of similarities from source code. These techniques turn the architecture recovery feasible when added to architect assistance.

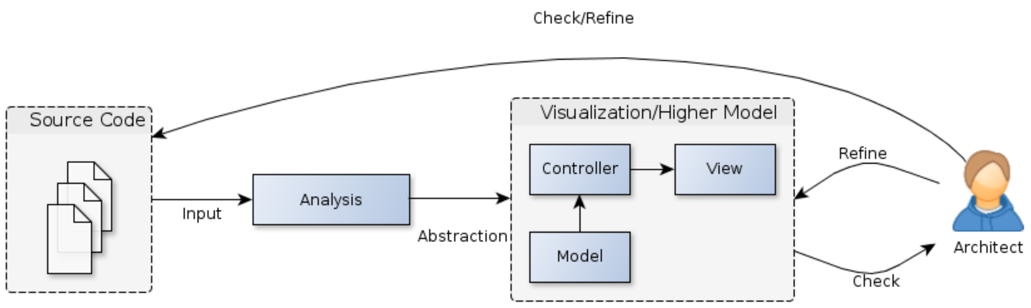


Figure 2.2: Architectural recovery process.

2.2 Machine Learning

Most of the software architecture recovery are based on machine learning techniques to explore similarities among architectural elements. The machine learning techniques thus have great influence in architecture recovery methods, because the learning techniques are responsible for assigning to which architectural modules each element belong. Then, we present next, in Section 2.2.1, an overview of machine learning. In Section 2.2.2, the feature selection process is explained. Finally, in Section 2.2.3, we present the clustering techniques commonly used in architecture recovery methods to the identification of modules.

2.2.1 Overview

The study and construction of systems that learn based on data (MITCHELL, 1999) define the principles of machine learning. Machine learning uses statistical concepts to discover relationships among data. Many problems can be solved based on recognition of data similarities, such as email spam filtering, image recognition, and recommender systems. Seen by a machine learning perspective, architecture recovery is the task of learning which elements belongs to the same architectural module based on software architecture element features (*data*). This means recognizing similarities in input data in order to group elements into modules. Basically, machine learning techniques are classified as *supervised* or *unsupervised*. The techniques of both categories explore the similarities concerning different characteristics of data. These categories differentiate in the type of training dataset needed to perform the learning process. Supervised techniques base its learning process in a classified training dataset, i.e. a training dataset with at least some instances correctly predicted. On the other hand, unsupervised techniques unknown previous results basing its predictions on an unclassified training dataset.

2.2.2 Feature Selection

As machine learning applies statistical concepts to search for similarities in data, the data provided to the learning process is a key factor to discover relevant relationship among data instances. In fact, if we apply a dataset with irrelevant information to the target feature, the relationship extracted by a learning algorithm will be poor. Consequently, the prediction accuracy will be also poor. One example of information quality issue occurs in the text classification task. In this task, a list of texts is given and the classification objective is to group the texts related to the same subject together. So, a dataset is built where each word in the texts is a feature and the words frequency of each text is data for each text. In this dataset, many features are irrelevant, when a word appears only in one text; or correlated, when a word appears together with another one in all occurrences of them.

The dataset number of features affects directly the prediction of the learning processes. If there are few features to be analyzed, the learning process will not extract enough relationship among data (underfitting) (AALST et al., 2010). On the other, if there are too much features generating a excessively complex model (overfitting), the learning process will describe random error or noise instead of a relationship. Other aspect that affects the prediction is the information of each instance. Considering a huge number of instances with irrelevant information, the prediction quality will be also poor. Then, the solution is to search for features that are related to the target variable in order to provide enough data to the learning process.

The feature selection process (DASH et al., 2002) aims to improve the data quality. This process focuses on reducing the number of features selected eliminating redundant features generating a simpler dataset. A simple model has three main benefits to the learning process (GUYON; ELISSEEFF, 2003): (i) the time needed to train a model is reduced because with less complexity the amount of data to process is reduced; (ii) the model interpretability is improved because there are less data to be analyzed by the learning process; and (iii) as the feature selection points out the relevant features, it produces a more generalizable model because the selected features produce less noise and error. Moreover, the feature selection reveals which features are in fact related to the target feature.

There are mainly two steps to perform feature selection. In the first, the set of features under analysis changes using an exhaustive or heuristic approach. Choosing the exhaustive approach is possible to obtain the best subset of all possible as result of the exploitation of all possible subsets. However, this search method demands a higher processing time than the others. The heuristics, such as Variable Neighbourhood Search (MLADENOVIC; HANSEN, 1997) and Best First (KORF,

1993) algorithms, do not guarantee the global best subset, but they usually improve the accuracy significantly with less computational effort. Then, the second step consists of evaluating the selected subsets. This step can be done using a specific learning algorithm, called Wrapper (KOHAVI; JOHN, 1997) approach, or a statistical measure of similarity, called Filter (DASH et al., 2002) approach. Using a specific learning algorithm, the feature selection achieves better results because it evaluates the subsets of the search method using the learning algorithm that will be used to solve a specific problem. Moreover, Wrapper approaches provide results based on the selected algorithm to perform the evaluation, e.g. the results provided using a specific algorithm in Wrapper approaches are not generalizable to other learning algorithms. The limitation of the Wrapper approach is that a classified training set must be available to evaluate the predictions. On the other hand, the filter approach does not need the classified dataset. Nevertheless, filters analyze the statistical relation among data and disregard the relationship with learning process, because it is unknown.

2.2.3 Clustering Techniques

As previously introduced, machine learning algorithms are divided in two categories: supervised and unsupervised. Unsupervised learning aims to discover similar instances in datasets in which the target feature, i.e. the feature that must be predicted has unknown values in all instances of the dataset. These kind of technique focused on finding common structures among the available data instances in order to group them. The lack of previous information associating the source code characteristics and the target feature complicates the process of discovering information solely with the provided characteristics. Due to this reason, unsupervised techniques explore the similarities in the available data, and group them to derive patterns. Similarly, supervised techniques have a target feature whose values must be predicted. However, supervised techniques use a dataset with known target features values, also known as classified training dataset. These techniques use classified training datasets to recognize patterns of the relation between target features and other features. The model built by supervised techniques usually tend to have preciser predictions than unsupervised. This occurs by the learn from the available classified instances. Both categories of machine learning techniques have their accuracy achieved dependent on the selected features and available data. Intuitively, the learning process prediction quality is proportional to the amount of data available to analyze. However, this assumption is false because instances quality also influence the learning process (HAWKINS, 2004). Additionally, how many and which are the features under analysis, which is associated with overfitting and underfitting problems, as we already discussed in previous section.

The clustering techniques are different in terms of the data needed. For instance, in the class of unsupervised algorithms, three algorithms are able to handle mostly all common unsupervised problems. They are K-means (MACQUEEN, 1967), Hierarchical clustering(WARD, 1963) and Expectation Maximization (DEMPSTER; LAIRD; RUBIN, 1977). Each algorithm has peculiarities about how they cluster the dataset instances. Hierarchical clustering initially assumes that each element is a cluster and the main idea of this algorithm is how to merge two clusters. If two clusters are similar enough, they are merged reducing the number of clusters by one. This algorithm needs a stop criteria that can be a similarity threshold or a number of clusters. K-means bases the way of grouping instances in the K previously provided and the elements average distance in the data space. It is usually applied to problems that the number of clusters are known and distribution of elements into clusters is homogeneous. The Expectation Maximization algorithm presents an approach to iterative computation of maximum-likelihood to estimate unknown data. Its name came from the algorithm two steps: an expectation step, in which the expected values of each instance is calculated to the complete data; and maximization step, in which parameters that maximize the expected likelihood from the expected step are determined.

Recovering an architecture from an application is an unsupervised learning problem, because there is no reliable architecture documented to be used as concrete architecture to guide the learning process. In addition, using a dataset from a software whose architecture is known to recover another software architecture whose architecture is unknown may predict a wrong architecture due to the differences that each software have, e.g. a software that adopts MVC architectural pattern used to recover another software whose architecture is in conformance with four-layered architectural pattern find for MVC elements in a four-layered architecture. We thus need to extract the architecture without a classified training dataset. However, we must select a model that fits to the architecture recovery problem appropriately. Our problem has two main aspects. First, the *number of the architectural modules* of the system is *unknown*, i.e. the number of clusters to be identified is unknown. Second, each of the architectural modules (clusters) may have *different sizes*, as architectural elements are distributed unequally into modules, such as the case studies presented in Table 4.5, where each column represent a module. Given these aspects, the algorithm that best fits to this problem is the **Expectation Maximization**. This algorithm is a way of estimating the parameters of mixture models used in statistics to capture properties of sub-populations within an overall population. Additionally, it is one of the most popular algorithms to discover unknown parameters based on likelihood methods. Also, discovering sub-populations matches our objective of recovering an architecture. In order to predict the number of modules, a cross-validation with the EM algorithm is performed. The cross-validation is a technique that iteratively divides the dataset instances in K different dataset: one dataset is used as training dataset and the $K-1$ are used as validation datasets. One iteration of cross-validation uses the training dataset to perform data analysis. Then, it validates the analysis obtained with training dataset on the validation datasets to check the analysis against different instances. The cross-validation performed in EM initializes considering the existence of one cluster and iteratively increments the number of clusters according to the similarity (loglikelihood) of the validation datasets. Based on the number of modules obtained in the cross-validation, the elements prediction is performed with the entire dataset.

2.3 Final Remarks

This Chapter presented an overview of software architecture, detailing the architecture recovery process, and machine learning. Software architecture is an essential documentation to the healthy software life-cycle, increasing the development team software comprehension. Besides the importance of software architecture, projects lack proper architecture documentation. Software architecture recovery aims to improve the automation to have and to maintain an architecture up-to-date. To achieve this objective, recovery processes employ machine learning techniques to extract the architectural concepts automatically. Research on architecture recovery has successfully improved the architectural documentation scenario. However, there are still much to be explored in terms of architectural information extraction and techniques to understand architectural elements similarities.

3 RELATED WORK

Research on architecture recovery has been developed in order to handle the lack of architectural documentation. In this chapter, we present the three main research areas of architecture recovery, dividing the studies by the kind source of information. We also discuss their results and limitations based on their provided results. First, in Section 3.1, we detail the studies that rely on high-level patterns to recover a software architecture. Next, in Section 3.2, dependency-based approaches are presented. Finally, in Section 3.3, we discuss approaches focused on domain comprehension, called semantic-based.

3.1 Pattern-based Approaches

Intuitively, search for architectural patterns in software is a good start to recover software architecture, because patterns are commonly used to plan software. Based on this idea, pattern-based architecture recovery methods aim to find architectural elements in source code that respect the rules of a predefined pattern. We consider pattern-based the architecture recovery methods that do not demand a documented conceptual architecture. Architecture recovery approaches that need architectural patterns set by architects or perform an architecture conformance check against a documented architecture, such as Reflexion Models (MURPHY; NOTKIN; SULLIVAN, 1995), will be disregarded since they fit to the architecture discovery (top-down approach) research area.

Tzerpos and Holt proposed the *A*lgorithm for *C*omprehension-*D*riven *C*lustering (ACDC) (TZERPOS; HOLT, 2000) based implementation level patterns observed on the manual inspection of systems source code. They noticed seven aggregation patterns used to compose implementation elements into architectural modules. Their seven patterns aggregate architectural elements into the same module considering: (i) elements of a same file; (ii) elements of the same folder; (iii) splitting task in more than one architectural element, e.g. the implementation (.c files) and definition (.h files) of C language; (iv) independent elements; (v) elements frequently used considering the whole system; (vi) elements that depend on many other elements; and (vii) dependency subgraph with a dominator element. ACDC iteratively applies these patterns in a system source code to compose architectural elements forming architectural modules. Recently, a comparative analysis of architecture recovery approaches (GARCIA; IVKOVIC; MEDVIDOVIC, 2013) reported that ACDC is one of the top architecture recovery methods. However, as stated by the authors, the list of patterns proposed was incomplete. There are more code level patterns to be discovered to increment the algorithm in order to improve it. Furthermore, the patterns proposed were extracted based on the manual architecture recovery of the systems used to evaluate their approach. This compromises the generalization of the results to other software that were not developed using the same structures of the ACDC case studies.

Exploring another kind of pattern to recover software architectures, Constantinou et al. (CONSTANTINOU; KAKARONTZAS; STAMELOS, 2011) analyzed *a suite of design metrics* to identify rules among architectural modules. They focused only on four-layered-structured architectures (User Interface, Controllers, Business Logic and Infrastructure layers) of open-source projects. Their process of architecture recovery consists of extracting a suite of source code metrics and a dependency *directed acyclic graph* (DAG) from the source code to identify the layers that comprise the software architecture according to the hierarchical communication among architectural elements. More specifically, the DAG defines an initial architecture grouping the architectural elements that depends on the same other group of elements. As the DAG derives too many layers, the authors developed a refinement process to merge DAG layers based on the correlation of clusters' design metrics proposed by Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994). The refinement iteratively calculates the correlation of the design metrics to merge the layers until only the four pre-defined layers remain. These identified layers and metrics define rules of metrics thresholds for each architectural layer, using the JRip, a machine learning algorithm to extract classification rules. They concluded that the metrics evaluated in fact have impact to identify the architectural layers. However, the assumption that the system architecture follows the four-layered architecture strongly limits their study because open-source projects use a variety of architectural patterns (STOL; AVGERIOU; BABAR, 2010).Consequently, their approach does not recover an architecture, but only classifies elements into one of the four layers of the pattern that is assumed to be followed.

In both approaches, the usage of patterns to recover software architecture depends on the generality of the patterns chosen to recover architectures, i.e. it achieves more effectiveness in extract software architecture if the patterns adopted by a method have general aspects of software architecture, such as dependency rules. On the other hand, if a method uses patterns that restrict architecture recovery, such as stating a fixed number of modules, it limits the architecture recovery to the projects that in fact follow a type of architecture. All pattern-based approaches will have issues of pattern generalization because software organization varies accordingly domain and purpose. Also, address all software variations considering only a set of pattern is impossible given the software complexity of software.

3.2 Dependency-based Approaches

The software architecture organization focus on communication rules which are the core part of software architecture. Aiming to have a software structure, dependency-based approaches investigate similarities between elements to create modules based on metrics derived from coupling and cohesion. Next, we describe the work based on dependency approaches to architecture recovery.

With the aim of extracting dependency similarities among architectural elements, Mancoridis et al. developed a tool called Bunch (MANCORIDIS et al., 1998, 1999). This tool provides an objective function to the software decomposition problem. It uses a modularization quality metric, also created by the authors, applying clustering techniques to organize architectural elements in modules. The proposed modularization metric is based on an architectural module analysis of the intra-connectivity (when an element of a module depends on an element from the same module) and the inter-connectivity (when an element of a module depend on an element of another module) of the system modules. Then, to decide to which module each element belongs, Bunch iteratively applies a genetic algorithm to all architectural elements calculating the intra-connectivity and inter-connectivity of the architectural modules in each iteration. Based on the mutations made by the genetic algorithm, it evaluates the prediction using the quality metrics and maintains the solution with the best result. As occurs in Constantinou et al. study, this method to group elements in module produces too many modules. Then, to reduce the number of modules, a hierarchical clustering algorithm is applied in order to merge similar modules based on their degree of similar intra-connectivity dependencies. In general, Mancoridis et al. method provides inaccurate results. This occurs because they analyzed only the communication of software architecture lacking information about architectural elements roles. In addition, their recovered architecture is composed of layers, such as the initial architecture of Constatinou et al. study, and there are many other architectural patterns that do not rely on this type of organization, such as MVC.

Xiao and Tzerpos investigated the extraction of dynamic dependencies at software runtime (XIAO; TZERPOS, 2005). In their study, they analyzed the architecture recovery improvements applying different clustering algorithms to static and dynamic dependencies. In addition, they ran experiments with different types of element dependency degree filters to reduce the search scope of the dependency graphs. They concluded that the dynamic dependencies provide a better model using different weights according to runtime dependencies between elements. However, this extraction of runtime dependencies relies on the exercised zones of the source code, i.e. if parts of systems have little activation during runtime execution, the data collected with respect to their element dependencies is proportional to the activation of these code parts. This limits their approach to only the exercised parts of source code.

In both studies, a high-level structure of the architectural organization is provided as result of the architecture recover methods. However, this kind of approach lacks information about the roles that architectural elements play in the software. Consequently, the architecture comprehension gain is limited due to the necessity of manual investigation of elements to understand what each module in fact does. Additionally, problems related to architecture drift and erosion (MEDVIDOVIC; TAYLOR, 2010) are related to violation of communication rules, which results in an inaccurate grouping of elements that drift or erode the architecture. It occurs because the classification of elements is only based on the element dependencies.

3.3 Semantic-based Approaches

Usually, the planning phase of software involves defining a glossary of terms to be used in the development to name software elements, such as methods and classes. Using ideas from information retrieval, semantic-based approaches map the problem of retrieve information from documents to the architecture recovery. This kind of approaches extracts textual content from architectural elements and evaluates their similarities in terms of words to group them into modules.

A pioneer work on semantic-based architecture recovery was performed by Anquetil and Lethbridge (ANQUETIL; LETHBRIDGE, 1999). They proposed an approach to extract subsystems of an industrial case study using only textual information. Their hypothesis rely on the fact that architectural elements names follow a pattern, e.g. the suffix of names of elements that handle data is *data*. They investigated how the architectural elements names can be partitioned to provide meaningful information about their functionalities. They performed an evaluation analyzing four parts of architectural elements: (i) file names, (ii) file comments, (iii) procedure names, and (iv) variables names. The authors performed experiments with a set of proposed algorithms to fragment the textual content of architectural elements in order to cluster the architectural elements with the same words. To cluster architectural elements with the similar keywords, they used a simple algorithm that group elements based on the occurrence of words, i.e. given two architectural elements named *UserData* and *AdminData*, they are grouped by the word *Data* . They concluded that the file comments outweigh all the other source of information analyzed mainly because of the larger amount of information provided by comments compared to the others sources of information.

The system vocabulary was also analyzed by Kuhn et al. who proposed the Semantic Cluster (KUHN; DUCASSE; GÍRBA, 2007). They measured element similarities with Latent Semantic Indexing, an information retrieval technique to identify patterns, to build a matrix of elements correlation. They used this matrix as input to a hierarchical clustering algorithm in order to group elements. As their main goal, they improved software comprehension because their approach enrich the knowledge retrieved extracting linguistic information and suggesting labels to the modules recovered. This idea is similar to Anquetil and Lethbridge, but using sophisticated word retrieval algorithms and clustering algorithms for documentation indexing.

Recently, Corazza et al. (CORAZZA; DI MARTINO; SCANNIELLO, 2010; CORAZZA et al., 2011) proposed a method to recover software architecture based on semantic information, called Zone-Based Recovery (ZBR). As the two others semantic-based approaches, ZBR considers an architectural element as a document with words. For ZBR word indexing algorithm, each architectural element has six document zones: class names, attribute names, function names, parameter names, comments, and function bodies. Given these zones, the word indexing assigns different weights according to the zone a word resides. These zone weights are set using the Expectation Maximization algorithm that analyzes all document zones and attribute a determined weight to each of the six zones. Then, to group the architectural elements, ZBR uses a hierarchical clustering algorithm with the cosine similarity measure.

These semantic-based studies provided significant results in terms of supporting the human understanding of the system domain. However, the correction is associated with an adequate use of file name patterns, which occurs in the case studies evaluated in all studies. In addition, the architectural model recovered by these approaches lacks information about architecture rules, e.g. how architectural modules communicate. Furthermore, the behavior of architectural elements is disregarded, e.g. the name given to an architectural element does not necessarily reflect what it actually does.

3.4 Final Remarks

In this Chapter, we introduced the main researches developed to architecture recovery detailing the information used as input of each of them and analyzed their strengths and weaknesses. Most of the studies focus on an objective, such as comprehension or extraction of high-level structure, and their source of information are directly related to their objective. Analyzing the studies, architecture recovery still is a field of study that needs improvement, as discussed here and also in the of comparative analysis presented elsewhere (GARCIA; IVKOVIC; MEDVIDOVIC, 2013; MAQBOOL; BABRI, 2007). In fact, recent approach exploited information, such as semantic, to improve the accuracy and quality results of the recovered architectures.

4 PROCEDURE TO ANALYZE CODE-ORIENTED INFORMATION FOR ARCHITECTURE RECOVERY

In the previous chapter, we discussed many approaches whose aim is to recover software architecture. Each approach assumes that a particular information is useful to identify architecture models and uses it to extract an architecture. However, there is no in-depth study that evaluates and compares the usefulness of different kinds of information that can be collected from the source code. In particular, code metrics have been exploited in solely one approach. We therefore, as introduced in the introduction, performed an study that makes this evaluation and comparison. We describe the procedure adopted to perform experiments, detailing all aspects that involve their execution. In Section 4.1, we specify the steps of our procedure namely; information selection, dataset preparation, learning process and result analysis. Next, in Section 4.2, we present two types of features, which were analyzed in our experiments. As our evaluation is empirical, we present the case studies used in the experiments in Section 4.3. Finally, in Section 4.4, we discuss the threats to validity of our study.

4.1 Procedure

In this Section, we detail each step of the procedure to evaluate the relevance of different information extracted from the source code to identify the architectural modules. Broadly, we first select the types of information that we consider candidates for this purpose. Second, we choose case studies whose architectures we can manually retrieve and classify their elements according to the architectural module each element belongs to. Third, we apply an unsupervised machine learning technique to verify the results achieved for predicting the case studies modules. Finally, we measure the obtained results. This sequence of main steps of our approach is presented in the Figure 4.1. Furthermore, in the next sections, we detail all these steps of procedure emphasizing the importance of each to the whole process.

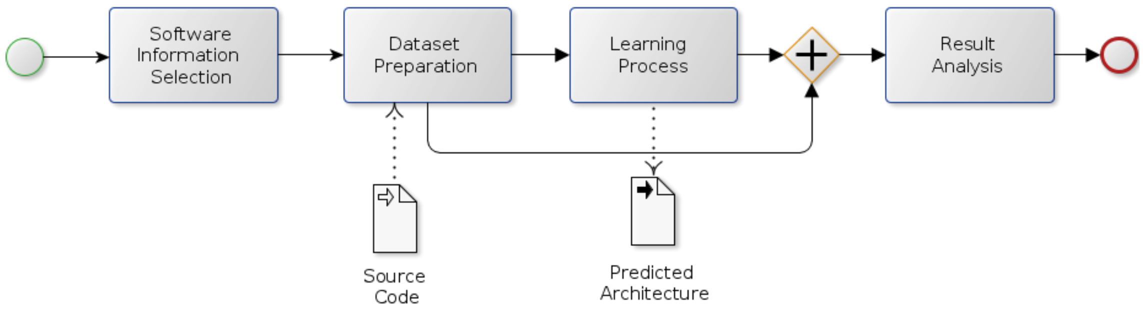


Figure 4.1: Main procedure activities of our evaluation method presented in BPMN.

4.1.1 Information Selection

The first step of our study consists of selecting what kind of information will be extracted that may predict architectural information of a system. This selection is based on the available literature, for instance existing design metrics and architectural element dependencies, or intuition, as our experiments evaluates whether this intuition is meaningful. In particular, we adopted as a source: the suite of design metrics proposed by Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994); other metrics, such as the number of getters and setters of a source code element; and architectural element dependencies explored largely in the architecture recovery literature (MANCORIDIS et al., 1999; XIAO; TZERPOS, 2005). We will introduce how we handle all the selected source of information in detail in the Section 4.2.

4.1.2 Dataset Preparation

To evaluate the model prediction of an unsupervised machine learning technique, we need a classified dataset, what means the concrete architecture of a system documented. Therefore, case studies must be selected and prepared. We selected five case studies, presented in Section 4.3. This preparation involves two derived substeps: a manual architecture recovery and data extraction. These substeps are detailed as follows and the organization of these activities is illustrated in Figure 4.2.

4.1.2.1 Manual Architecture Recovery

To evaluate the correctness of the architectural module identification, the concrete architecture must be explicitly documented, i.e. an architecture must be manually recovered and in conformance with the actual behavior of the system. Although the unsupervised algorithm used in our experiments only needs an unclas-

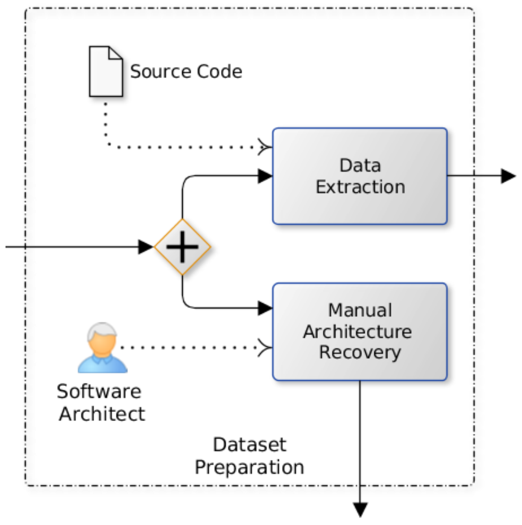


Figure 4.2: Dataset preparation.

sified training dataset, the architectural modules (target feature) must be known to evaluate learning effectiveness to the identification of modules. So, we performed a manual inspection of the source code of each case study in order to recover its architecture. In addition, we used the documentation available of each case study and contacted the developers involved in the development of the case studies, when possible, to validate the architecture derived from the manual recovery.

4.1.2.2 Data Extraction

Additionally to know the target characteristic, the unclassified training dataset content must be extracted from the system to the learning process recognize similarities in these data. In order to provide data to the machine learning technique, we used: the Eclipse IDE [1]; its associated Eclipse Metrics Plugin Continued,[2] also known as Metrics2; and the Classycle Plugin[3]. These tools allow to export the data extracted, therefore facilitating its manipulation. Although Metrics2 extracts almost all the selected features related to design metrics, some of them had to be implemented as an extension of this tool. In particular, we implemented five characteristics, namely *class name words*, *superclass usage*, *interface usage*, *mean method size*, and *number of getters and setters.*

4.1.3 Learning Process

In this step, we performed a feature selection in the classified dataset when the relevance of each feature is unknown, in order avoid overfitting and underfitting.

[1] `http://www.eclipse.org`
[2] `http://sourceforge.net/projects/metrics2`
[3] `http://classycleplugin.graf-tec.ch`

The feature selection used to evaluate the relevance of features is the Wrapper approach as long as we know the target feature, recovered manually in the last activity of dataset preparation, and have a specified algorithm to perform the architecture recovery. If the relevance of features is known, we run the unsupervised machine learning technique with the provided dataset directly without feature selection, as the learning process activity illustrated in Figure 4.3. We selected the Expectation Maximization algorithm to execute the learning process, because it matches the scenario of the architecture recovery process. In particular, we perform our experiments using the Expectation Maximization implementation of the Weka tool (HALL et al., 2009).

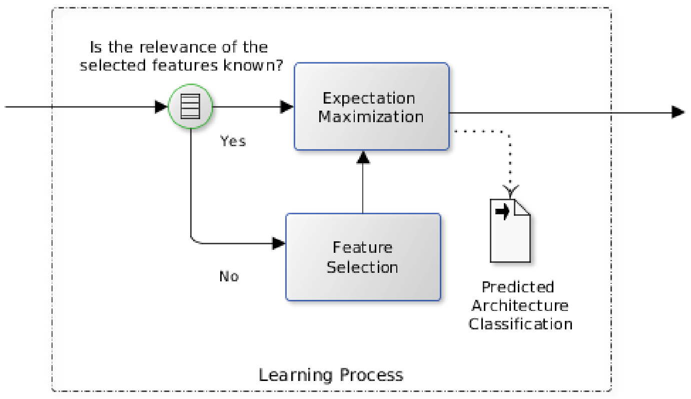

Is the relevance of the selected features known?
Expectation Maximization
Yes
Feature Selection
No
Predicted Architecture Classification
Learning Process

Figure 4.3: Learning process.

4.1.4 Results Analysis

To provide a quantitative comparison of the information sources, we used the general purpose metrics to evaluate multi-class prediction of machine learning algorithms defined by Sokolova and Lapalme (SOKOLOVA; LAPALME, 2009). Based on the quality achieved by the EM algorithm, we calculate the metrics to each information source and also during the feature selection. Using these metrics, we standardise the results in order to compare them. The metrics definitions are given next following the notation: K is the set of proposed architectural modules, i is a module such that $i \in K$, $|K|$ represents the cardinality of K, tp_i are the true positives of i, tn_i are the true negatives of i, fp_i are the false positives of i, and fn_i are the false negatives of i.

Definition 1 (Average Accuracy) *The average accuracy measures the correctness of each identified module and distinctness from the others modules. It evaluates the correct predictions, true positives (tp) and true negatives (tn), of the modules prediction. In fact, the average accuracy is a way to measure the per-module effec-*

tiveness of the classifier. Then, the formula to calculate the average accuracy is as follows.

$$Average\ Accuracy = \frac{\sum\limits_{i=1}^{K} \frac{tp_i+tn_i}{tp_i+tn_i+fp_i+fn_i}}{|K|}$$

Definition 2 (Average Precision) *Focusing on evaluating per-module precision, the average precision measures only the agreement between the prediction and the concrete architecture for each module. It considers only the cases where the prediction and the correct classification agree. The formula to calculate the average precision is as follows.*

$$Average\ Precision = \frac{\sum\limits_{i}^{K} \frac{tp_i}{tp_i+fp_i}}{|K|}$$

Definition 3 (Average Recall) *Calculating the average recall, we obtain how the prediction method correctly predicts the modules. Trust in the number of modules predicted improve significantly the results quality. To calculate the average recall, we consider the true positives and false negative of each module as the formula above describes.*

$$Average\ Recall = \frac{\sum\limits_{i=1}^{K} \frac{tp_i}{tp_i+fn_i}}{|K|}$$

Definition 4 (Average fMeasure) *The average fMeasure combine the average precision and average recall to provide one metrics that comply the overall correctness and the modules prediction quality. It provides a relationship between positive predictions and those given by a classifier based on a per-module average. The formula to calculate the average fMeasure is as follows.*

$$Average\ fMeasure = \frac{2 * average\ precision * average\ recall}{(average\ precision + average\ recall)}$$

4.2 Features

Software systems have many characteristics that can be measured during their planning, development and maintenance phases. Therefore, there are a variety of data to be collected about software characteristics. We selected characteristics of architectural elements that may characterise a group of elements to be analyzed. We extracted three types of element dependencies: Direct, Indirect and External. These dependencies are detailed in Section 4.2.1. Next, in Section 4.2.2, we describe the source code metrics, detailing their purpose and relationship with concrete software architecture modules, and the elements names similarity. Both of them relate element roles in the systems, because metrics present behavior similarities and dependencies show the communication rules.

4.2.1 Component Dependency

Communication rules are core parts of software architectures because software development good practices are based on coupling and cohesion of elements (ALLEN; KHOSHGOFTAAR, 1999). So, understanding how elements interact is an important issue to have a comprehension of software architecture. The dependencies between software elements are usually represented as a direct graph (GANSNER; NORTH, 2000). In this graph, elements are nodes and the use relationship is represented by the edges between two elements, where the edge source is are close to the element that uses the other one. Figure 4.4 presents an example of a graph dependency of the *WeatherResponse* and *ForecastResponse* elements of OLIS case study. In this figure, the *WeatherResponse* uses five other internal elements, the *ForecastResponse* uses six internal elements and both use one external element.

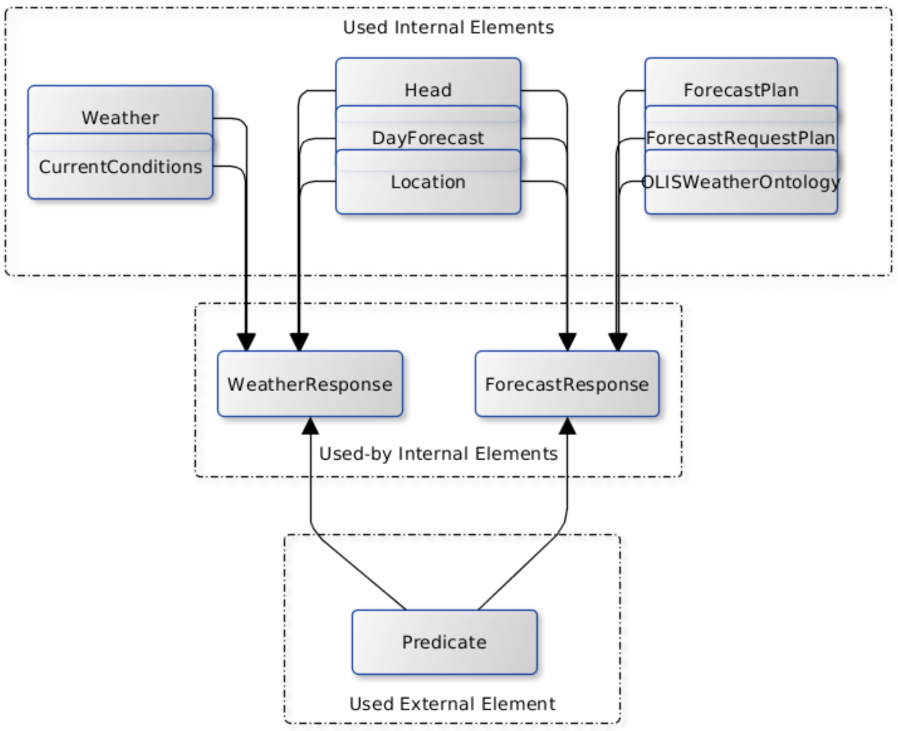


Figure 4.4: Graph dependency of *WeatherResponse* and *ForecastResponse* architectural elements.

This section details the three types of architectural element dependencies used as source of information of the learning process. In addition, this section presents the way the dependency graphs are used in the learning process. Section 4.2.1.1 presents the direct dependency between architectural elements. Next, the inverse dependency is detailed in Section 4.2.1.2. Finally, in Section 4.2.1.3, we discuss the external dependencies.

4.2.1.1 Direct

The direct dependency occurs when an element directly depends on another element. This type of dependency shows the hierarchy between project elements, because this type of dependency considers only the project internal dependencies. In Figure 4.4, an example of direct dependency occurs between *WeatherResponse* and *Weather*, where the *WeatherResponse* element needs the *Weather* to perform its task. In order to use the direct dependency relationship with machine learning algorithms, we represented the graph as a matrix, where columns are used architectural elements and rows are all architectural elements. The matrix cells are filled with a binary representation, where 0 means that the row element is not dependent on the column element and 1 that exists a dependency between them, where the row element uses the column element. The direct dependencies presented in Figure 4.4 are represented as a matrix in Table 4.1.

Commonly, the elements dependencies have a relationship with the role that architectural elements play in an architecture. Principles of modularization and decoupling are applied to structure modules in a way the dependency between modules are reduced. Moreover, by applying reuse practices, such as importing files or inheritance, projects source code that have similar dependencies tend to perform the same role in a software. We can see this relationship of architectural role and direct element dependencies in Table 4.1, *WeatherResponse* and *ForecastResponse* are elements from the same architectural module, Agent module, which depend on *Head*, *DayForecast* and *Location* elements.

4.2.1.2 Inverse

The inverse dependency focuses on the used elements. It is exactly the opposite of the direct dependency. This dependency reveals by which elements a particular element is used. In fact, the matrix representation is exactly the transposed matrix of the direct dependency, where the used elements correspond to the rows and the elements that use them correspond to the columns. The inverse dependency of Figure 4.4 is represented as matrix in Table 4.2.

Similarly to the direct dependency, the inverse dependency matrix carries information about patterns of communication that may derive communication rules to define a group of elements of a same module. In Table 4.2, the highlighted cells show this kind of pattern. The elements *Head*, *Location* and *DayForecast* belong to the same architectural module, Model.

Table 4.1: Direct dependency matrix highlighting *WeatherResponse* and *ForecastResponse*

Element	***Weather***	***Forecast Response***	***Weather Response***	***Current Conditions***	***Head***	***Location***	***Day Forecast***	***Forecast Plan***	***Forecast RequestPlan***	***OLISWeather Ontology***
Weather Response	1	0	0	1	**1**	**1**	**1**	0	0	0
Forecast Response	0	0	0	0	**1**	**1**	**1**	1	1	1
...	...	...	...	...	...	...	...	...	...	...

Table 4.2: Inverse dependency matrix highlighting *WeatherResponse* and *ForecastResponse*.

Element	***WeatherResponse***	***ForecastResponse***	**. . .**
Weather	1	0	. . .
ForecastResponse	0	0	. . .
WeatherResponse	0	0	. . .
CurrentConditions	1	0	. . .
Head	**1**	**1**	. . .
Location	**1**	**1**	. . .
DayForecast	**1**	**1**	. . .
ForecastPlan	1	0	. . .
ForecastRequestPlan	1	0	. . .
OLISWeatherOntology	1	0	. . .
. . .	. . .	. . .	. . .

4.2.1.3 External

The direct and invert dependencies disregard communication with external module, such as Application Programming Interface (API) or libraries. However, these external communications also help in the task of discovering communication patterns, because it is common to have external modules that handle recurrent activities, such as the database communication implemented by the Hibernate framework.

In fact, the representation of external dependencies is equal to direct dependencies with a difference that external elements never will use one of the project elements, what means that it will not be present in the rows in a matrix representation.

4.2.2 Metrics and Labels

Metrics and Labels are architectural element characteristics, i.e. a characteristic is any attribute that can be extracted from the source code elements. Metrics are countable characteristics, such as the number of methods of an architectural elements. Labels are semantic characteristics, such as the five most frequent words in class names of a software. In addition, these characteristics may contribute to the identification of the architectural modules and their elements. Some of these characteristics are metrics, but their use is different from the use to evaluate the architecture quality, where extracted values have an associated semantics that indicate “good” and “bad” values. Instead, when recovering an architecture, metrics describe elements properties and possibly help in identifying a correlation among metric values.

With the aim of selecting a set of source code characteristics that impact in the architecture recovery process, we adopted those that are significant to source code or to architectural elements from the existing literature. Based on the literature, we selected fifteen characteristics, from which twelve are numerical and three are binary, such as the metric suite proposed by Chidamber and Kemerer (CHIDAMBER; KEMERER, 1994), file names of the Antequil's study (ANQUETIL; LETHBRIDGE, 1999) (which in Java means the class name), and the metrics associated with a rationale of their relationship with the role of architectural elements (the present study allows confirming this intuition). Table 4.3 shows the selected characteristics with their types and from where they come. Additionally, we describe each of the selected characteristics and a rationale that justifies why they may be useful to recover architectures next.

Class Name Words (CNW) are the words present in a class name, which in our case correspond to the words present in an architectural element. This information provides an understanding of the application domain, because its syntax makes the code legible. Additionally, similarities in the elements names indicates to which architectural module a class belongs, such as the suffix of its name, which relates the responsibility of a particular class (e.g. the suffix *BusinessService*). Evaluating all class name words contained in an application would bring too much variability and singularity, so we consider in this characteristic up to the five most frequent words in the whole application. In fact, this frequency selection generates five features to this characteristic (CNW1-5).

Superclass Usage (SC) is the list of all superclasses, from the direct superclass above to the `Object` class, that an architectural element has. As class name words, this characteristic retrieves information related to the application domain. In addition, it also provides structural knowledge, as classes performing similar tasks inherit methods from the same superclasses. To extract similarities among superclasses of the elements, a group of elements must share a domain or a behavior. As above, we consider a characteristic each one of the five most frequent superclasses of the system generating five features (SC1-5).

Interface Usage (INT) is the list of interfaces that an architectural element implements. Similarly to superclass usage, this characteristic relates common tasks of architectural elements. It also considers up to the five most frequent interfaces used in the application creating five features (INT1-5). Note that

Table 4.3: Set of selected characteristics.

Name	**Type**	**Source**
Depth of Inheritance Tree	Numerical	Chimdaber and Kramer (CHIDAMBER; KEMERER, 1994)
Class Name Words	Binary	Antequil (ANQUETIL; LETHBRIDGE, 1999)
Superclass Usage	Binary	
Interfaces Usage	Binary	
Number of Attributes	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Number of Children	Numerical	Chimdaber and Kramer (CHIDAMBER; KEMERER, 1994)
Number of Getters and Setters	Numerical	
Number of Methods	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Number of Overridden Methods	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Number of Static Attributes	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Number of Static Methods	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Total Lines of Code	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Class Methods Mean Size	Numerical	Henderson-Sellers (HENDERSON-SELLERS, 1995)
Weighted Method per Class	Numerical	Chimdaber and Kramer (CHIDAMBER; KEMERER, 1994)
Mean Methods Complexity	Numerical	Chimdaber and Kramer (CHIDAMBER; KEMERER, 1994)

class name words, superclass usage, and interface usage characteristics depend on developers using appropriate names.

Depth of Inheritance Tree (DIT) is the distance from the top-most class, i.e. the class from which all other classes derive, which is the `Object` class in Java. The DIT values start in zero to the top-most class and, for each level of class inheritance, the DIT value increases by one unit. For instance, if a class `Telephone` directly extends `Object`, the **DIT** value of `Telephone` is one. And if a class `Mobile` extends `Telephone`, `Mobile` has the **DIT** value of two. This characteristic extracts therefore the level of specialization of a element.

Number of Children (NSC) is the total of direct subclasses of a class. This characteristic extracts information related to the structural organization of the system quantifying the number of classes that directly depend on another under measurement. Consequently, elements with high number of children have a more generic role in the system than those with low number of children.

Number of Overridden Methods (NORM) is the class number of redefined methods when the class evaluated is a child class. This metric measures how much a child class differentiates itself from its parent class. It provides structural information of how specific a element behaves.

Number of Attributes (NOA) is the number of attributes of a class. In software applications, data (or domain) classes tend to have more attributes than other classes. Consequently, this characteristic indicates the architecture role of an element, as it occurs in data classes.

Number of Static Attributes (NSA) is the sum of static attributes of a class. It provides information related to the role that elements play in the architecture as static attributes commonly characterises classes that state definitions.

Number of Methods (NOM) is the number of methods of a class. Counting the number of methods gives insights about the quantity of processing a class handles. For example, elements in a business module of a layered architecture has higher number of methods than elements of other modules.

Number of Static Methods (NSM) counts the number of statics methods of a class. Static methods occur in classes that state definition, similarly to the number of static attributes. It is usually related to business rules that are enclosed to the domain.

Weighted Methods per Class (WMC) is the sum of the McCabe Cyclomatic Complexity (MCCABE, 1976) of all methods of a class. The cyclomatic complexity counts the number of paths linearly independent in the source code decision tree. This characteristic measures the amount of processing handled by an element in order to group them by their quantity of processing.

Mean Methods Complexity (MMC) is the average of the McCabe Cyclomatic Complexity of architectural element methods, i.e. establishing a relation between weighted methods per class and number of methods. This characteristic differentiates elements handling a lot of complexity in fewer methods, such as Utils elements, from elements with many methods handling large complexity, usually Model or Data elements.

Number of Getters and Setters (NOGS) is the sum of getters and setters of a class, more specifically, the sum of methods whose name starts with *get* or *set*. Data classes have more readable and writeable attributes than other classes, and therefore more getters and setters. Consequently, a high number of getters and setters also indicates the architecture role of an element.

Mean Method Size (MLOC) is the average lines of code of the class methods excluding blank and comment lines. Similar mean method size leads to elements that share a common behavior, e.g. high mean method size indicates that a class has much logic inside it, consequently it may be a business elements.

Total Lines of Code (TLOC) is the sum of lines of code of the class excluding blank and comment lines. As architectural elements that handle processing may have complex algorithms, they have more lines of code than others. In addition, data elements have fewer lines of code, because their methods are mostly getters and setters.

4.3 Case Studies

This section details information about each case study selected for our experiments providing aspects of the case studies that influences in the architecture recovery process. We selected five case studies that are presented next.

Eclipse Metrics Plugin Continued (Metrics) is an Eclipse IDE plugin, which extracts a suite of metrics related to object-oriented good practices from projects developed using Eclipse. It automatically extracts and exports the design metrics implemented.

Expert Committee (EC) is an agent-based conference management system for the web domain developed to support the paper submission and reviewing processes from conferences and workshops.

Table 4.4: Selected case studies.

Case Study	***#Files***	***#Packages***	***#Lines***	***#Modules***	***Pattern***
Metrics	150	16	15674	4	Extended MVC
EC	195	38	11796	5	Layered
OLIS	212	30	11437	5	Layered
RecSys	334	64	22871	6	Heterogeneous
Port	399	51	41654	12	Heterogeneous

OnLine Intelligent Services (OLIS) is a reactive multi-agent system that provides several personal services to users, such as calendar and events announcement. The user defines services needed in a web-based interface.

Recommender System (RecSys) is a desktop-based recommender system that aims to recommend items based on their properties and preferences defined by the user.

Port [4] is an industrial case study that provides a web-based interface to manage the ship's cargo control in a port. It is integrated to a set of systems that perform the whole management of a port.

We selected these case studies because they have different properties. First, they have different sizes: small (less than 20K lines of code), medium (less than 40K lines of code) and large (more than 40K lines of code). Case studies sizes are relevant to our experiment because the learning algorithm derives statistical information of the data available. As the applications are different in terms of domain and purpose, they adopt different architecture patterns, such as MVC and the layered architecture. Third, they have different numbers of modules. Finally, we could recover their architecture manually in a relatively easy way in order to prepare our classified training dataset, as we managed to contact four of the five case study developers — except Metrics, all case studies had their manual recovered architecture validated by the development team. Table 4.4 presents detailed information of each case study (number of files, number of packages, number of lines of code and number of architectural modules).

To apply an unsupervised algorithm, more specifically the EM, to our case studies, we extracted the concrete architecture of each case study. *EC*, *OLIS*, *Port* and *RecSys* had their architecture extracted by using their available documentation and also analyzing their source code. For all applications but Metrics, we validated the architecture proposed based on our manual investigation with the development team involved with each case study. As *Metrics* is an open-source project and has

[4]Name omitted for confidentiality.

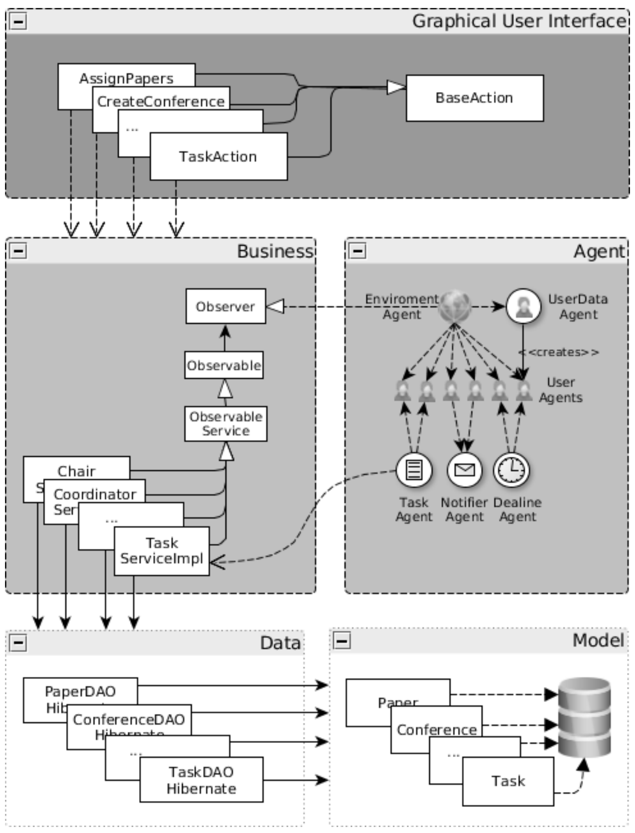


Figure 4.5: Expert Committee layered architecture.(NUNES, I.; NUNES, C.; CIRILO, E.; KULESZA, U.; LUCENA, C., 2013)

Table 4.5: Number of elements in each case studies modules.

Case Study	***M1***	***M2***	***M3***	***M4***	***M5***	***M6***	***M7***	***M8***	***M9***	***M10***	***M11***	***M12***
Metrics	63	43	39	5								
EC	54	49	40	26	26							
OLIS	65	50	40	37	19							
RecSys	97	91	56	55	32	3						
Port	113	94	49	49	23	23	12	12	10	8	5	1

its development opened to the community, we recovered its architecture based on manual inspection and available documentation. Based on the recovered architectures, we classified the architectural elements into different modules. For instance, the *EC* follows a layered pattern with two extra modules, as depicted in Figure 4.5. Its layers are: *Data*, which is the module responsible for persisting the domain entities of the system; *Business*, which is the module that processes information; *User Interface (UI)*, which is the module that focuses on the presentation of the processed information to the end-user; *Agent*, which is the module that provides autonomous behavior; and *Model*, which is the module that focuses on the representation of domain entities involved in the system.

One aspect that can only be analyzed after the concrete architectures recovered is the distribution of elements in the architectural modules of the applications. By analyzing their architectures, we observed significant differences in architectural aspects, as shown in Table 4.5. The elements distribution in the architectural modules of *OLIS* is mainly concentrated in the business module. The same occurs with the *EC* distribution, but it has a lower concentration of elements in the business module compared to *OLIS*. Following the same behavior, the module with more elements in Metrics is also the business module. In addition, *Metrics* has another interesting point in the distribution of elements, which is the low number of data elements. It occurs because the *Metrics* persistence entities are mainly the source code of the projects. Our other two case studies, *RecSys* and *Port*, have more complex architectures than the others. Each one has more modules with specific behavior and their element distribution is concentrated in four modules in *RecSys* and in six modules in *Port*.

4.4 Threats to Validity

Empirical studies essentially have threats to validity because the empirical investigation basis its conclusions on observation or experience. Then, to assure that an experience in fact reflects the cause consequence derived, facts that reduce or eliminate the influence of the threats in the study must be performed. This section presents the threats to validity and the actions applied to eliminate those threats of our study dividing them by: threats in the observed variables, presented in Section 4.4.1; and in the generalization of the study, presented in Section 4.4.2.

4.4.1 Construct Validity

Studies basis its decisions believing on assumptions in order to achieve their hypothesis. So, if the study base has threads to validity, all the scientific method applied is compromised. So, in our study, our construction variables are the characteristics selected to extract architectural similarities, selected by us, even though we had the help of their own developers, and the case studies concrete architecture,recovered manually by us. Next, we detail both construction threats to validity and present actions applied to the procedure to minimize their effects.

Features relevance to software architecture. The features selected to our study are also the base to our conclusions and may not reflect the architectural behavior of the case studies elements. However, the features evaluated are commonly used in the literature to reach conclusions about software quality and system modularization. In addition, applying a feature selection process using an exhaustive approach, when needed, highlight the features relevance, by selecting and presenting their contribution to the characterization of architectural modules.

Correctness of the concrete architecture recovered. The case studies concrete architecture is a main point of our work. The manual architecture recovery process was performed by the people involved with the software and the experience in the software architecture scenario influence the manual recovery result. To increase the confidence of this process, the manual recovery of each case study selected performed by the people that were directly involved in the development of each case study. The Port case study, our industrial case, it is recovered by the software architect that is responsible by the case study. Moreover, a re-checking of the manual architecture recovery in order to validate the correction of the concrete architecture provided by the people involved with the case studies development. Recently, Garcia et al. provided an analysis of how the manual architecture recovery should be performed (GARCIA et al., 2013). Almost all step related in their study were performed in our manual architecture recovery process.

4.4.2 External Validity

One problem of the empirical validation is the generalization of the results based on the data provided, or even the data provided could be insufficient to derive conclusions about a problem. In our work, we selected five case studies to perform our experiments. We consider the level of difference in their architectures aiming to evaluate different types of architectures, as we presented in Section 4.3. Furthermore, applying a method to a restricted domain of applications affect the results generalizability. To decrease this threat impact, the selected case studies are from different domains. This avoid a domain-driven result. Moreover, our case studies are from different sizes and built by different development teams. Furthermore, we used an industrial case.

4.5 Final Remarks

In this Chapter, we provided a detailed description of our experiment settings, detailing each relevant point of the procedure. This description provide the definitions of the overall procedure steps, the features under analysis, the case studies used as empirical data and the threats to validity of our study. After specifying how we conducted our experiments, we present the results achieved applying our procedure to the selected case studies.

5 ARCHVIZ: A TOOL TO SUPPORT ARCHITECTURE RECOVERY RESEARCH

In order to evaluate the accuracy achieved by architecture recovery approaches, the recovered architecture must be matched against a concrete architecture, which must be available. To compare these two architectures, evaluation metrics, such as those presented in Section 4.1.4, are applied. As a consequence, evaluating and comparing of architecture recovery methods is not trivial, because studies use divergent measurements to evaluate its performance. With the aim of providing support to architecture recovery research and standardize the evaluation of architecture recovery methods, we developed the ArchViz tool, which is presented in this chapter.

Additionally to the data support that scientific studies must provide, graphical representation of a recovered architecture brings simple visualization of the data. Comparison views of software architecture are needed in order to facilitate the comprehension of software architecture recovery approaches. Representing software architectures in different levels of abstraction improves the understand (DUCASSE; POLLET, 2009). Consequently, using different abstraction levels to compare between concrete and conceptual architectures also improve the comprehension of the architecture recovery process. Then, ArchViz tool uses the entered data to evaluate the architecture recovery to generate visualizations to compare the concrete architecture against the predicted architecture in three different levels of abstraction.

This Chapter thus introduces ArchViz, a tool developed to support research on architecture recovery. In Section 5.1, we detail the ArchViz features, showing how it supports the comparison of two architecture views, such as conceptual vs. concrete and concrete vs. extracted. Furthermore, in Section 5.2, we describe the three visualizations provided by the tool. Then, in Section 5.3, we present technical details of the ArchViz implementation.

5.1 Features

ArchViz provides functionalities to support the architecture recovery process evaluation and visualization. It provides a methodological form of evaluation of the architecture recovery processes. Aiming to support data handling, ArchViz has features to import, export and manage systems. Additionally, to evaluate the data, the tool also calculates and shows metrics detailed in Section 4.1.4. Furthermore, three architectural visualizations are automatically generated. Next, we detail each of these introduced features.

1. **Import Case Study.** Importing case study allows to add a case study to ArchViz. It uses two comma-separated values (CSV) containing all the architectural information needed to build the two architecture views. Two files must be provides, the first containing information about the architectural elements, and the second specifying the communication between these elements. The first file must have the elements *name*, *package*, *concrete module* and *predicted module*. The second file has lines with pairs of columns with name and package architectural elements. The first pair of the second file is the element that uses the followed elements in a line. Importing case studies is fundamental to the tool be used. Because manually handling the data input of large projects demands much time.

2. **Case Study Architecture Management.** In ArchViz, it is also possible to manage the architectural elements, changing their modules, names and predictions. Furthermore, information related to the different levels of abstraction is available. There is information related to modules, such as number of modules and modules sizes, and elements, such as element dependencies, level of abstraction.

3. **Export Case Study.** An exporting of the architectural information based on the communication between architectural elements was implemented. Projects in ArchViz can be exported as a communication matrix where the architectural elements are the rows and the direct and inverse dependencies are the columns.

4. **Prediction Metrics.** To support a standardized evaluation of the recovery process, we implemented four metrics *Average Accuracy*, *Average Precision*, *Average Recall* and *Average fMeasure*. This metrics are automatically calculated to each case study added to ArchViz providing the quality of the recovered architecture.

5. **Visualizations.** We implemented three visualizations that improve the understanding of the concrete architecture and help match the predicted archi-

tecture against the concrete architecture. We detail each of the implemented visualizations next.

5.2 Visualizations

Large-scale software is an abstraction that is complex to represent in a understandable way. Then, a software architecture visualization helps people involved with software to understand the main concepts applied in their applications using a high-level representation. Most of the software architecture recovery approaches focus on presenting metrics to evaluate their results and they do not provide graphical visualizations of their results. Actually, humans handle models and abstractions in a easier way than just analysis purely data (KEIM et al., 2008). So, we identify three visualizations that can improve the comparison and understanding of recovered architectures and implemented them in ArchViz.

The remainder of this section presents the three visualizations of the ArchViz tool, discussing their objectives. First, in Section 5.2.1, a two-dimensional graph that aims to increase the understanding of the predicted architecture evaluation metrics is presented. Next, in Section 5.2.2, we detail a module dependency visualization comparison that highlights the modules sizes and dependency among modules. Then, in Section 5.2.3, the traditional element dependency graph is explained.

5.2.1 Treemap

Treemap is a two dimensional hierarchy graph created by Shneiderman (SHNEIDERMAN, 1992) to analyze the use of hard disks, where, similarly to software architecture, the disk folders hierarchy represents categories and files are leaf elements that belong to a folder. A common problem to visualize the data is to represent the relevance of more than two attributes in one graph. In the hard disk usage, for example, the files have a parent folder and size, what means that to visualize both attributes using a Cartesian coordinate system we need to plot one graph for the file usage and one graph for the folder usage. Shneiderman proposed a tree organization structure where each element is represented as a rectangle and attributes can be defined as colors, sizes or hierarchy position of the rectangles. Then, the hard disk usage can be represented in one graph where the folders are bigger rectangles with file elements inside, and the size represents the amount of disk usage.

As in hard disk usage, the software architecture has a hierarchical structure – architectural elements belong to modules. So, we mapped the software architecture using the treemap representation to understand the predicted results of the concrete and conceptual architectures representing the concrete and the recovered architecture in a single graph to visualize the prediction results. Figure 5.1 is a sample of the treemap visualization, where the outer rectangles are the concrete architectural modules, the inner rectangles are the architectural elements with their name, and the architectural elements colors are assigned according to the predicted module to which they belong. The concrete module names, in the upper right of Figure 5.1, are from the manual architecture recovery. In the case when the architecture predicted matches 100% against the concrete architecture, all outer rectangles are colored by only one color and each outer rectangles have a different color from the others. Figure 5.1 illustrates a scenario in which the recovered architecture differs from the concrete architecture. As can be seen in this figure, the ideal case currently do not occur, the outer rectangles major color defines its predicted architectural module, i.e. in Figure 5.1, the lower right corner correspond to the Data module and the upper right corner corresponds to the UI module.

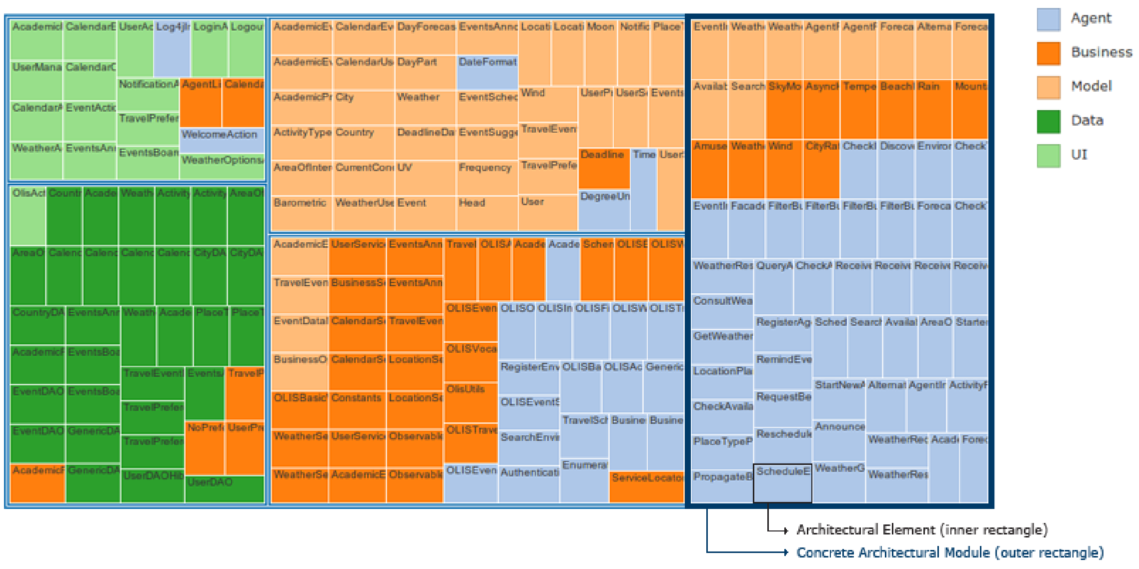


Figure 5.1: Treemap visualization of software architecture prediction.

Analysing Figure 5.1, it clearly shows some of the modules predicted and the assignment distribution of predicted architectural elements to the concrete modules. Furthermore, the evaluation metrics are also presented in this representation, since the concentration of color in the concrete module means that the accuracy of module is high, and if a module color is scattered in the graph its accuracy is low.

Additionally, the treemap visualization comprises concrete and recovered architecture allowing a visual comparison of the architectural measures extracted from a architecture recovery process.

5.2.2 Modules Graph

The module dependency visualization is a coarse-grained view that aims to understand a big picture of the system. It is the most common architecture view used, where the architectural modules are represented as nodes and the communication among them as edges. Having this representation improves the understanding because it handles the main system concepts, presenting in a summarized picture the architectural modules and how they communicate to each other, as presented in Figure 5.2, which shows an example of layered pattern.

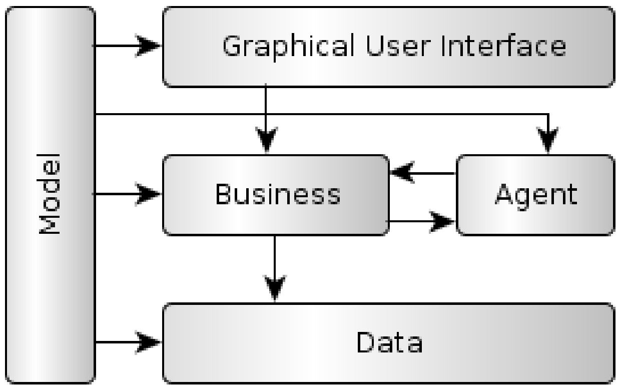


Figure 5.2: Example of a typical architecture model.

This traditional model present in a high-level the main architectural modules and communication between them. However, it lacks software details needed to compare this model against a recovered architecture. Furthermore, it undertakes architectural information that can be represented in a visualization, such as the intense of the dependency among two modules. Analysing the representation of OLIS architecture in Figure 5.2, it impossible to identify the intensive of the dependencies among modules. Usually,the module sizes represents just the existence of architectural modules and they are not related to the importances of the modules in the system. So, we implemented the module visualization with modifications from the usual visualization. The same architecture presented in Figure 5.2 is represented in ArchViz as shown in Figure 5.3. In ArchViz, the modules are defined by their size and color. Their colors characterizes each module role and their size is proportional to the number of architectural elements it has. Also, the modules have labels with the architectural role and number of elements that they have. The edges represent the communication among modules in way that when an edge is red means that the red module is using the opposite module. Additionally, the thickness of the edge

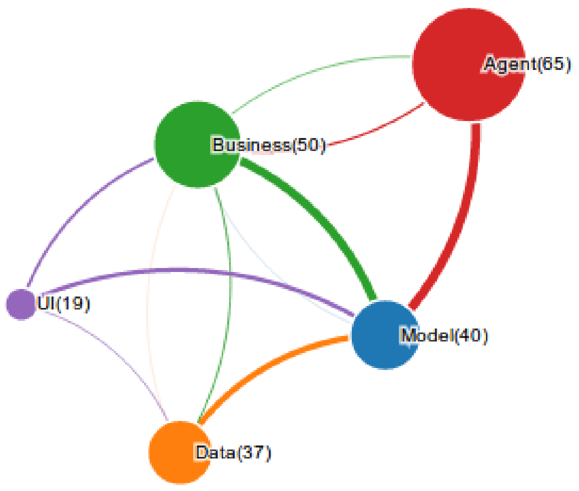


Figure 5.3: Modules dependencies software architecture graph

is proportional to the level of dependency the relationship has, e.g. the relationship among modules Agent and Model is greater than the Agent and Business in the OLIS architecture.

5.2.3 Elements Graph

Architectural element dependencies is the finer-grained architectural model that can be represented. It presents the dependencies among architectural elements classifying them into architectural modules. In the element visualization implemented in our tool, the graph represents architectural elements as nodes and they have the color of the their modules. The edges are colored by the module that uses another module, similarly to modules visualization. This representation disregards the intra-module dependencies to reduce the number edges in the visualization. As an example, we present the Expert Committee concrete architecture representation using the elements visualization in Figure 5.4, where the five modules, the inter-modules dependencies and the 195 elements of the system are represented.

5.3 Implementation

We built the ArchViz as a web-based application, so it is plataform-independent because the only requirement to use it is an Internet browser. To develop our tool, we followed the Model-View-Controller architectural pattern commonly used for the development web applications.

To develop our tool, we chose Ruby as programming language because of the support of the Ruby on Rails[1] (RoR) web application framework and our experience

[1]`http://rubyonrails.org`

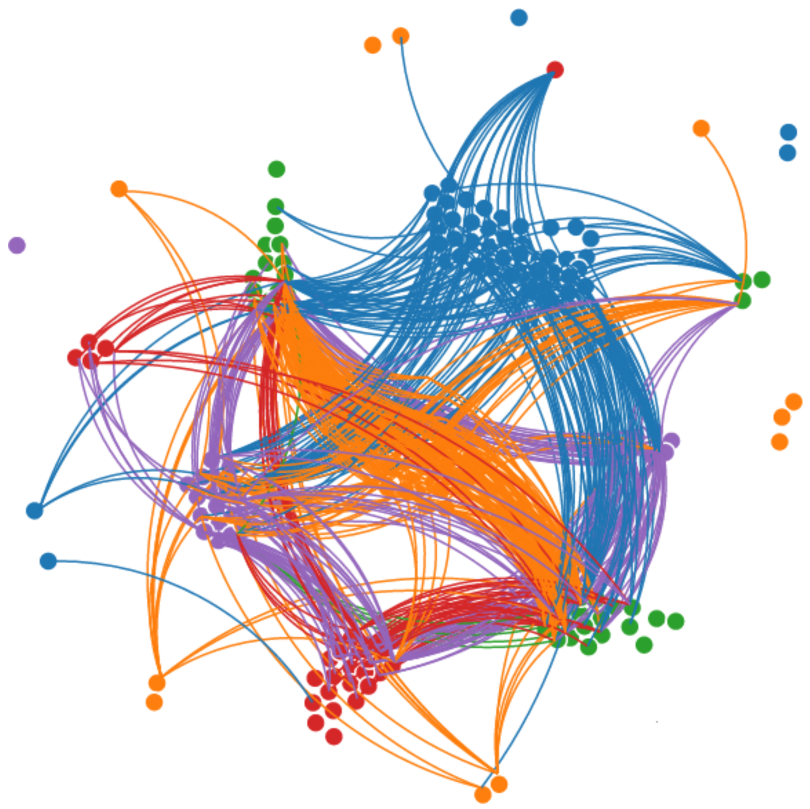


Figure 5.4: Elements dependencies visualization of Expert Committee software architecture.

in this language. Additionally, to develop our models, we followed a test-drive development (TDD), which provides automatic suite of tests. Furthermore, during the development, we controlled our code version with Git[2].

Our tool is available online as a Beta version in the address `http://archviz.herokuapp.com` to be used as support to architecture recovery process. This Beta version has limited resources in the Heroku [3] deployment server. Moreover, as ArchViz still is a Beta version, it is not indexed in the search engines yet.

5.4 Final Remarks

In this Chapter, we presented our tool to support the architecture recovery process, ArchViz, explaining its functionalities, detailing the visualizations developed and providing technical information about how it was developed. Additionally, we deployed ArchViz Beta version available online. Furthermore, ArchViz contributes to the improvement of the architecture recovery process comprehension since it provides evaluation measures and architectural visualizations automatically.

[2]`http://git-scm.com`
[3]`http://www.heroku.com`

In the next chapters, we use ArchViz visualizations and metrics to present results and to derive conclusions from our experiments.

6 EXPERIMENTS

In this chapter, we present the experiments performed following the settings detailed previously. Thus, the results achieved by the subsets of features selected are provided considering all the case studies. Additionally, we evaluated a way to combine the two selected features in one dataset. Based on the analysis of the results, we present the subset of features that achieves the best accuracy using the Expectation Maximization to recover the software architecture. We focus on presenting the results based on the average accuracy, since this metric provides a summarized evaluation of the correctness and distinctness of each module predicted. Moreover, Appendix A has the evaluation of precision, average precision, average recall and average fMeasure of all the significant subsets evaluated in this chapter.

Our experiments are divided according to the features selected to be evaluated. In Section 6.1, we detail the use of architectural element dependencies. Next, we analyze the relevant subsets of the several experiments performed using metrics and labels in Section 6.2. Focusing on improving the accuracy achieved, in Section 6.3, we combine element dependencies, metrics and labels features in a unique dataset.

6.1 Element Dependency

Many architectural patterns base their concepts in the elements communication rules, such as layered modules and MVC pattern. Our experiments using element dependencies aim to derive patterns of system communication rules to group them in modules. It is unknown if the Expectation Maximization algorithm is able to extract dependency patterns. Moreover, an investigation about the contribution of each type of dependencies must be performed in order to extract the types of dependency that contribute more to the identification of architectural modules.

Thus, to present the experiment results related to element dependencies, we divided our analysis following a three part structure. First, we analyze, in Section 6.1.1, the results obtained with the subset that contains the three types of element dependencies. Then, in Section 6.1.2, we investigate the subsets that perform best to each case study. Concluding, in Section 6.1.3, we present the subsets that achieved the best accuracies given all case studies in order to discover which types of dependencies have the most influence to recover the architecture correctly.

6.1.1 All Dependencies Results

Our experiments related to element dependencies are presented in Table 6.1 showing the average accuracy achieved by all investigated subsets. Additionally, Table 6.1 presents the average accuracy (*AVG*), the standard deviation (*SD*) and minimum accuracy (*MIN*) of all case studies. Each row of Table 6.1 corresponds to one subset of selected features of element dependencies, where the features selected are explicitly shown in the subset column. The first row containing a subset has all type of dependencies selected (Direct, Inverse and External). The remainder rows are all the six other combinations of selection of element dependencies features. Then, considering that all types of element dependencies are relevant to the architecture recovery process, we investigate the subset with all dependencies. The architecture recovery using dependencies among all elements, the subset Direct, Inverse and External of Table 6.1, provided an average accuracy range of 52.7% – 75.7% considering all case studies. Moreover, it achieved an average accuracy of 66.9% taking in count all the case studies. In the last row of Table 6.1 (Average by Case Study), the average accuracy of each case study is presented.

Table 6.1: Accuracy of dependencies subsets

Subset	***Metrics***	***OLIS***	***Port***	***EC***	***RecSys***	***AVG***	***SD***	***MIN***
Direct, Inverse and External	**0.527**	**0.757**	0.676	0.630	0.754	**0.669**	**0.096**	**0.527**
External	0.487	0.537	0.690	**0.733**	0.779	0.645	0.127	0.487
Inverse	0.477	0.487	**0.748**	0.623	0.481	0.563	0.120	0.477
Direct	0.473	0.573	0.666	0.626	0.754	0.618	0.105	0.473
Inverse and External	0.310	0.579	0.633	0.685	0.290	0.500	0.186	0.290
Direct and Inverse	0.475	0.492	0.283	0.510	**0.786**	0.509	0.180	0.283
Direct and External	0.377	0.712	0.283	0.435	0.635	0.489	0.179	0.283
Average by Case Study	0.447	0.591	0.568	0.606	0.640			

6.1.2 Individual Case Studies Results

Each case study selected has particularities, such as domain and architectural style. In order to capture the influence of these specificities of each case study to the architecture recovery process, we investigated the best subsets of each of them using the element dependencies features. We also investigate the subsets that achieve the best accuracy to each case study to verify whether there is a group of selected features that is present in all of them. In Table 6.1, the best accuracies to each case study are highlighted in boldface. These accuracies vary from 52.7% to 78.6%. Analyzing the individual best accuracies, *Metrics* has the lowest and it is the unique that achieves an accuracy lower than 70% even with its best subset. Additionally, the best subsets, except the one with all features, cause skewness in the accuracies, i.e. a best subset privileges one case study characteristics causing a poor accuracy in another. For instance, selecting the Direct and Inverse subset, *RecSys* achieves 78.6% and on the other hand *Port* achieves 28.3%. This skewness in the average accuracies is a consequence of case studies particularities.

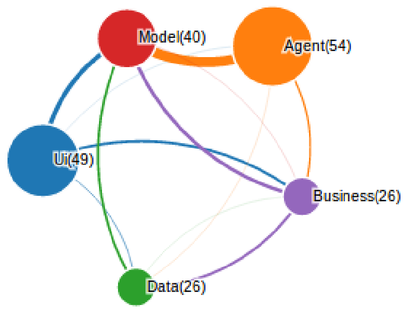
There are four different best subsets to the five case studies — *Metrics* and *OLIS* achieve their best accuracy with the subset containing all element dependencies. Actually, all the best subsets share the Inverse dependency features. However, the subset with only the Inverse dependency feature selected performs best only to Port, what means that the other four case studies have complementary information in the Direct and External dependencies.

Furthermore, the average accuracy per case study, presented in the last row of Table 6.1, show a poor extraction of architectural information from the Metrics, 44.7%, and similar extraction from the other four, in a range from 56.8% to 64%. This is correlated to the Metrics number of architectural elements. Essentially, the four other case studies are at least 30% greater than Metrics, since metrics has 150 architectural elements and the next greater case study has 195 elements.

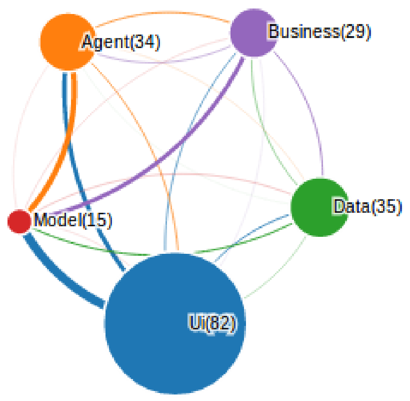
6.1.3 Selecting the Element Dependencies Best Subset

An exhaustive search method allows us to find the best subset of all possible. In the element dependencies case, we performed a feature selection using all possible subsets of the three types of element dependencies, Table 6.1 presents all the seven possible subsets of dependencies features, to verify the relevance of each type of dependency and obtain the best subset.

Curiously, all subsets with two features selected perform near 50% of average accuracy. They achieved the worst average accuracies, even compared to the subset with only one feature selected. This occurs by the singularity of the case studies. As



(a) Concrete architecture.



(b) Predicted architecture using all element dependencies.

Figure 6.1: Concrete and predicted architecture of Expert Committee.

presented in the Section 6.1.2, we found four best subsets to five case studies with only Inverse dependency shared by three of the case studies.

Surprisingly, the subset containing all features performs best considering the average accuracy all case studies. Evaluating all the three measures in Table 6.1, the best subset has an average accuracy greater than all others subsets (66.9%), the lowest standard deviation (9.6%) and the higher minimum accuracy (52.7%). The Direct, Inverse and External subset achieves an average accuracy of 66.9% without any human interaction, only based on the dependencies information extracted from the projects. This fact reinforces that all dependency types have singular information about the architecture of their systems and must be considered in an architecture recovery process. To illustrate the results achieved by the subset of dependencies with the three features, we detail the Expert Committee case study presenting the predicted architecture using the module dependencies visualization in Figure 6.1. The architecture recovered using dependencies retrieved the five modules of Expert Committe, as can be seen in Figure 6.1(b). Moreover, the gross dependencies of each view are similar, in Figures 6.1(a) and 6.1(b), indicating that the recover process extracted the element dependencies similarities.

6.2 Metrics and Labels

In this Chapter, we present our experiment results related to the use of metrics and labels features. As we adopted an exhaustive approach to compare the different subsets of metrics and labels — almost 147 thousand subsets — we focus on a representative group of the experiment results, from which we will draw conclusions.

The presentation of results of metrics and labels are divided three main parts. We first analyze, in Section 6.2.1, the results obtained with the subset that contains all selected metrics and labels features. Then, in Section 6.2.2, we present the subsets that achieved the best accuracies for each case study. Finally, in Section 6.2.3, we investigate the subsets that achieved the best accuracies across all case studies, so that we can identify the metrics and labels that most contribute to recover the architecture.

6.2.1 All Metrics and Labels Use Analysis

We present, in Table 6.2, the accuracy achieved by different metrics and labels subsets for each application. We also show the average accuracy (AVG), standard deviation (SD) and the minimum accuracy (MIN). Rows are split into three parts: (i) accuracy of the set with all selected metrics and labels (ALL); (ii) accuracy of the subsets with the best accuracy for each application — these subsets are named with the first letter of the application name; and (iii) the top 23 best subsets considering the average of all applications. These data strongly indicate that the analyzed source code metrics and labels are able to reveal information of how the software architecture is structured. The trivial case uses all the source code characteristics selected (without a feature selection process), and with this set is possible to group elements into architectural modules with an accuracy ranging from 59.3% to 89% without any human intervention. Furthermore, considering all case studies, we obtained an average accuracy of the 70.8%, as presented in the subset ALL of Table 6.2.

Given the exhaustive method of search applied during the experiments phase, we rank all the generated subsets according to their accuracy to better understand the relationship between different sets of metrics and labels, and the accuracy they achieve. The ALL subset is ranked in the 2536^{th} position considering all subsets, according to the average accuracy of all analyzed case studies. The position of this subset in the ranking corroborates the relevance of all selected features, as the accuracy achieved with the ALL subset is better than 98% of all possible subsets generated using metrics and labels. Note, however, that the ALL subset has high variance in the accuracy achieved for each application, as shown by its standard deviation (13.6%) presented in Table 6.2.

6.2.2 Individual Case Studies

In order to investigate whether exists a relationship between the set of source code metrics and labels selected and the architectural pattern adopted by each case

Table 6.2: Accuracy of metrics and labels subsets.

Subset	***Metrics***	***OLIS***	***Port***	***EC***	***RecSys***	***AVG***	***SD***	***MIN***
ALL	0.593	0.890	0.623	0.818	0.615	0.708	0.136	**0.593**
M	**0.770**	0.826	0.766	0.818	0.526	0.741	0.123	**0.526**
O	0.505	**0.930**	0.784	0.732	0.808	0.752	0.156	**0.505**
P	0.360	0.854	**0.891**	0.856	0.745	0.741	0.220	**0.360**
E	0.342	0.839	0.803	**0.879**	0.649	0.702	0.219	**0.342**
R	0.336	0.892	0.638	0.500	**0.832**	0.639	0.231	**0.336**
Bst1	0.730	0.896	0.791	0.826	0.765	**0.801**	**0.063**	**0.730**
Bst2	0.717	0.909	0.695	0.821	0.695	0.767	0.095	0.695
Bst3	0.733	0.906	0.795	0.834	0.668	0.787	0.092	0.668
Bst4	0.731	0.890	0.739	0.869	0.667	0.779	0.096	0.667
Bst5	0.720	0.900	0.664	0.821	0.793	0.779	0.091	0.664
Bst6	0.707	0.911	0.807	0.830	0.662	0.783	0.099	0.662
Bst7	0.720	0.901	0.801	0.828	0.661	0.782	0.094	0.661
Bst8	0.660	0.900	0.686	0.858	0.725	0.766	0.107	0.660
Bst9	0.653	0.894	0.805	0.832	0.745	0.786	0.091	0.653
Bst10	0.653	0.906	0.759	0.824	0.772	0.783	0.093	0.653
Bst11	0.700	0.898	0.652	0.841	0.663	0.751	0.111	0.652
Bst12	0.653	0.905	0.806	0.831	0.649	0.769	0.114	0.649
Bst13	0.642	0.894	0.721	0.846	0.667	0.754	0.111	0.642
Bst14	0.633	0.890	0.678	0.861	0.665	0.745	0.120	0.633
Bst15	0.633	0.890	0.779	0.869	0.789	0.792	0.101	0.633
Bst16	0.673	0.917	0.630	0.838	0.798	0.771	0.118	0.630
Bst17	0.629	0.894	0.844	0.823	0.629	0.764	0.126	0.629
Bst18	0.624	0.890	0.844	0.823	0.742	0.785	0.104	0.624
Bst19	0.607	0.903	0.874	0.824	0.651	0.772	0.134	0.607
Bst20	0.607	0.901	0.812	0.834	0.666	0.764	0.123	0.607
Bst21	0.607	0.909	0.679	0.831	0.767	0.758	0.120	0.607
Bst22	0.607	0.890	0.781	0.846	0.649	0.754	0.123	0.607
Bst23	0.607	0.900	0.723	0.840	0.776	0.769	0.112	0.607

study, we identified which subsets of features lead to the best accuracy in predicting each application architecture. As each application has architectural singularities and domain specificities, we verify the influence of the different features in the prediction of the recovered information.

In Table 6.2, we observe a subset of features for each application that achieves high accuracy, which ranges from 77% to 93%. However, the best subset for one application leads to a poor result for another. For example, the best subset for *Port* achieves an accuracy of 89.1%, while the accuracy achieved by this subset for *Metrics* is 36%, as shown in the subset P of Table 6.2. This variance can also be observed in the standard deviation of each of these subsets. Also, the lowest accuracies are always associated with the *Metrics* case study. This is the smallest case study of our experiment, and therefore the limited data available makes it more difficult to extract precise information from the selected features.

After identifying the best subsets for each application, we analyze the features contained in each subset. The features of each of these best subsets are depicted in Table 6.3. Our results indicate that the architectural pattern adopted has insignificant influence in the accuracy, and accuracy difference is due to the frameworks used (so that classes and interfaces are extended and implemented), the patterns used in class names, and patterns to create attributes and methods. From our twenty nine features, six (highlighted in boldface in Table 6.3) are shared by all best subsets: mean method complexity (MMC), total lines of code (TLOC), top two class name words (CNW1, CNW2), top two interfaces usage (INT1, INT2). However, if we ignore *Metrics* for the reasons said above, the number of attributes (NOA), number of static methods (NSM) as well as all class name words and implemented interfaces, are all shared. This means that the nomenclature pattern used in class names are more meaningful to all other applications than to *Metrics* — as the latter has fewer classes, the vocabulary adopted is smaller. The same occurs with the implemented interfaces.

Table 6.3: Best characteristic subsets by case study.

Subset	***MMC***	***DIT***	***WMC***	***NSC***	***NORM***	***NOA***	***NSA***	***NOM***	***NSM***	***NOGS***	***TLOC***	***MLOC***	***CNW1***	***CNW2***	***CNW3***	***CNW4***	***CNW5***	***SC1***	***SC2***	***SC3***	***SC4***	***SC5***	***INT1***	***INT2***	***INT3***	***INT4***	***INT5***	***size***
E	**x**		x		x	x	x		x	x	**x**	x	**x**	**x**	x	x	x						**x**	**x**	x	x	x	19
M	**x**	x		x						x	**x**		**x**	**x**									**x**	**x**				9
O	**x**		x	x	x	x	x	x	x		**x**		**x**	**x**	x	x	x						**x**	**x**	x	x	x	19
P	**x**	x	x	x		x	x	x	x	x	**x**		**x**	**x**	x	x	x	x	x	x	x	x	**x**	**x**	x	x	x	25
R	**x**	x		x		x			x	x	**x**	x	**x**	**x**	x	x	x						**x**	**x**	x	x	x	18

6.2.3 Selecting the Best Subset of Metrics and Labels

Our goal with this experiment is evaluate whether source code metrics and labels can be used in the architecture recovery process, and if so, identify if a particular subset of them that should be used in this process. As shown above, our selected metrics and labels can be used to recover software architectures with high accuracy, and in order to verify the existence of this particular subset, we investigated all the subsets with a better average accuracy than the average accuracy obtained by the *ALL*.

From the 2535 subsets of features better than the ALL subset, most of them are skewed, i.e. the accuracy achieved across all case studies has high variance, as it is also the case of the best subsets for each case study, which were presented in the previous section. We thus excluded these skewed results because we are interested in subsets that achieve an overall good result (not only for one case study). To exclude these skewed subsets, we take into account only the subsets that improve the accuracy for all applications in comparison with the accuracies achieved by the *ALL* subset. As result, we identified 23 subsets with a diversity of source code metrics and labels present in each of them, as depicted in Table 6.4.

Analysing these subsets, which have a better minimum accuracy than the *ALL* subset (Table 6.4), 12 features occur in more than 70% of the subsets: number of overridden methods (NORM), average method size (MLOC), all class name words (CNW1-5) and all interface usage (INT1-5). Even though the accuracy achieved by all the 23 subsets for all applications are improved with respect to the *ALL* subset, almost all of them are also associated with high standard deviation, which indicates that the accuracy achieved for some of the case studies is much worse than for others. The exception is the *Bst1* subset, which has the highest average accuracy, around 80%, and also the lowest standard deviation, 6.3%.

In order to visualize how good *Bst1* subset predicts the architecture, we present in Figures 6.2 and 6.3 the treemap prediction visualization of the *OLIS* (best accuracy) and *Metrics* (worst accuracy) case studies, respectively. The remainder case studies has a graph similar to that of the *OLIS*.

Table 6.4: Analysis of characteristic presence in subsets better than the ALL subset.

Subset	*MMC*	*DIT*	*WMC*	*NSC*	*NORM*	*NOA*	*NSA*	*NOM*	*NSM*	*NOGS*	*TLOC*	*MLOC*	*CNW1*	*CNW2*	*CNW3*	*CNW4*	*CNW5*	*SC1*	*SC2*	*SC3*	*SC4*	*SC5*	*INT1*	*INT2*	*INT3*	*INT4*	*INT5*	*Size*
Bst1	x	x		x		x		x		x		x	x	x	x	x	x						x	x	x	x	x	17
Bst2	x				x		x		x			x						x	x	x	x	x	x	x	x	x	x	15
Bst3	x	x	x	x		x			x	x		x	x	x	x	x							x	x	x	x		16
Bst4		x	x			x	x	x			x		x	x	x	x	x						x	x	x	x	x	16
Bst5	x		x		x		x		x			x						x	x	x	x	x	x	x	x	x	x	16
Bst6	x			x	x	x	x				x	x						x	x	x	x	x	x	x	x	x	x	17
Bst7	x		x		x		x					x						x	x	x	x	x	x	x	x	x	x	15
Bst8	x				x	x	x	x		x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	21
Bst9	x	x	x				x		x	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	21
Bst10			x	x			x		x			x	x	x	x	x	x						x	x	x	x	x	15
Bst11		x	x	x	x	x		x	x				x	x	x	x	x						x	x	x	x	x	17
Bst12				x	x	x		x			x		x	x	x	x	x						x	x	x	x	x	15
Bst13			x		x	x		x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	22
Bst14	x				x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	23
Bst15	x	x		x	x	x		x	x			x	x	x	x	x	x						x	x	x	x	x	18
Bst16		x		x	x	x						x	x	x	x	x	x						x	x	x	x	x	15
Bst17		x	x		x	x	x	x	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	23
Bst18	x		x		x	x	x		x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	23
Bst19	x	x			x		x	x			x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	22
Bst20	x				x		x	x			x	x	x	x	x	x		x	x	x	x		x	x	x	x		18
Bst21	x		x	x		x	x	x	x		x	x	x	x	x	x							x	x	x	x		17
Bst22	x			x	x			x		x	x	x	x	x	x	x	x						x	x	x	x	x	17
Bst23		x			x		x	x				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	20
Count	15	10	11	10	17	14	15	13	12	8	9	17	19	19	19	19	16	13	13	13	13	12	23	23	23	23	20	
Freq	0.65	0.43	0.48	0.43	0.74	0.61	0.65	0.57	0.52	0.35	0.39	0.74	0.83	0.83	0.83	0.83	0.70	0.57	0.57	0.57	0.57	0.52	1.00	1.00	1.00	1.00	0.87	

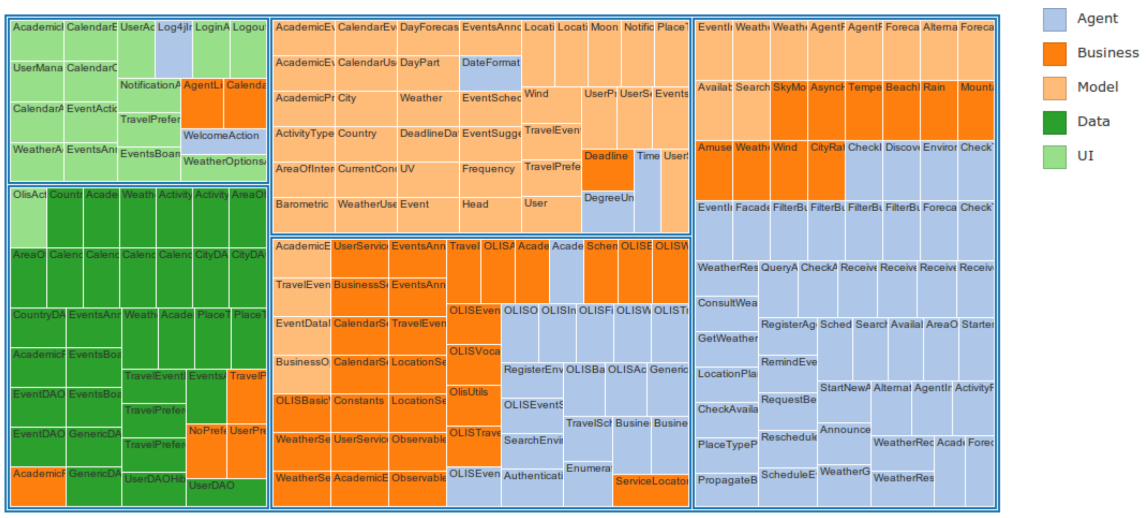


Figure 6.2: Treemap of Bst1 subset prediction to OLIS.

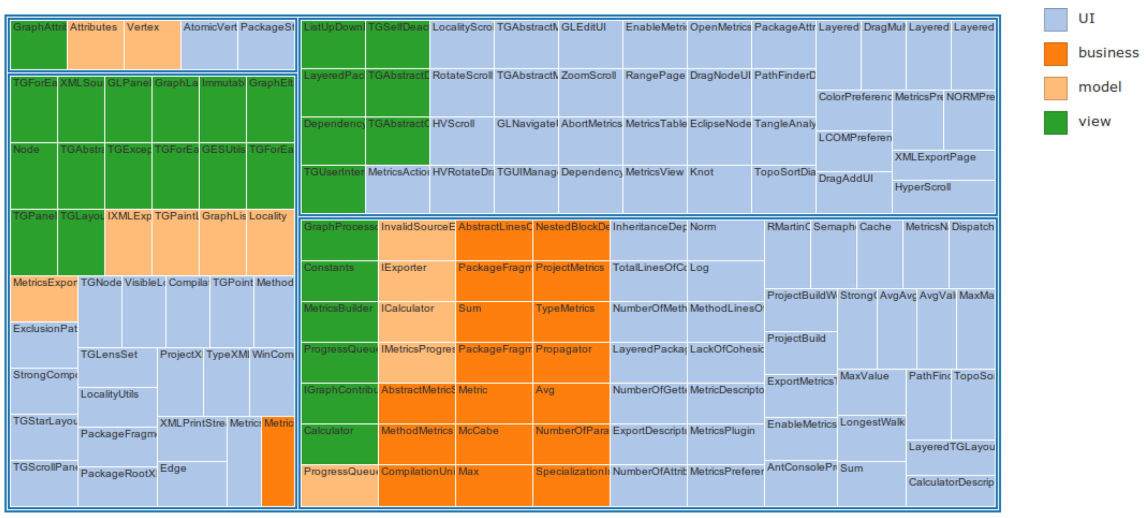


Figure 6.3: Treemap of Bst1 subset prediction to Metrics.

6.3 Combining Source Code Features

Our purpose with the experiments performed in Sections 6.1 and 6.2 was to investigate the information relevant to the architectural elements communication and behavior respectively. The objective of this section is to verify if merging the features improves the architecture recovery accuracy.

Our approach to combine the features is to add the architectural modules predicted by element dependencies to the metrics and labels features dataset. We produced a matrix with the element dependency modules as features, which means that when the dependencies features subset predicts five modules, the appended dataset will have all the metrics and labels features with five more features at the last columns of the dataset. The combined dataset follows the features schema depicted in Table 6.5, where metrics and labels features are placed first in the dataset, and the modules predicted using the element dependencies, represent as MOD in the table, are added at the end of the dataset. A MOD feature represents the presence or not of an architectural element in the module predicted by element dependencies. Its value is binary, zero when absent and one when present. We choose to add dependencies prediction to metrics and labels instead of the opposite, because the element dependencies generate too many features, almost 400 features in the smallest case, what practically makes the metrics and labels features insignificant to the learning process.

Table 6.5: Combined features dataset schema.

Elements	***MMC***	***DIT***	...	***INT4***	***INT5***	***MOD1***	***MOD2***	...
element1	2	3	...	1	1	1	0	...
element2	10	7	...	0	0	0	1	...
element3	5	4	...	0	0	1	0	...
...	...	...	...	...	...	...	...	...

Analysing the results of both sources of information, the accuracy achieved with metrics and labels overcomes that achieved with element dependencies. Only RecSys achieved a similar accuracy considering the both best subsets of each type of features. In Table 6.6, we present comparative data regarding all previously presented best subsets (Bst1; Direct, Inverse and External) and the subset with the features combination. Moreover, the average of all case studies accuracies considering each subset clearly show that metrics and labels perform best than element dependencies and than combining both features. Combining both features improves the results compared to element dependencies and decreases results of the metrics and labels considering the average of all case studies accuracies. Also, element dependencies always have the worst results comparing the three subsets presented in Table 6.6.

Although it is not an expressive gain, Recsys achieves an improvement combining both features in one dataset. Analysing the six Recsys modules, we observe that Recsys concrete architecture has two bigger modules, Business and Domain presented in Figure 6.4(a). The learning process using element dependencies allows to identify Business, Domain, Model and View modules, but it merges Core and Data modules to Busines module, as illustrated in Figure 6.4(b). On the other hand, metric and labels features allows to identify Domain, Business, View, Data and Model. Using metrics and labels, the learning process merges part of all modules and Core module with Domain module, as presented in Figure 6.4(c). Combining

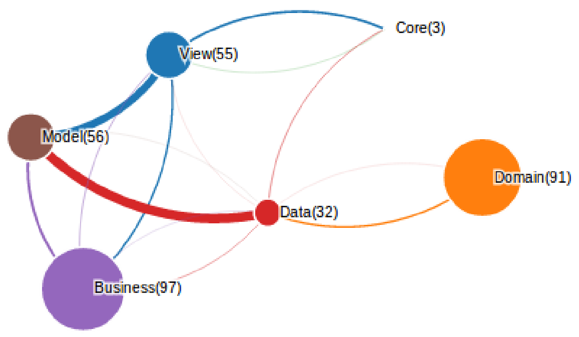
Table 6.6: Subsets comparison: Bst1; direct, inverse and external; and combined features.

Case Study	***Bst1***	***Direct, Inverse and External***	***Combined Features***
OLIS	**0.896**	0.757	0.879
Expert	**0.826**	0.630	0.793
RecSys	0.765	0.754	**0.790**
Port	**0.791**	0.676	0.617
Metrics	**0.730**	0.527	0.530
Average	**0.801**	0.669	0.722

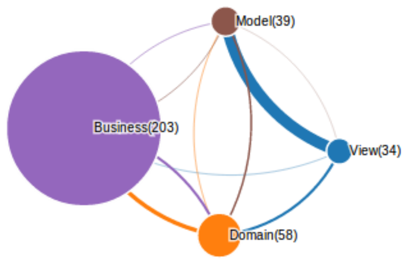
both, Figure 6.4(d), we achieved a better accuracy since, in Recsys case, each type of features focus on identifies one different part of the system.

6.4 Final Remarks

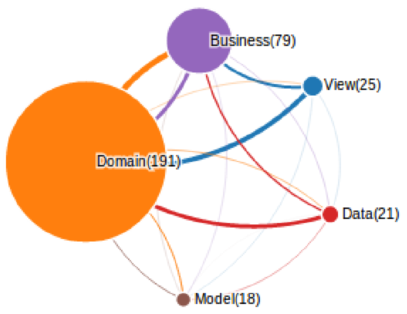
We conducted a series of experiments in order to evaluate the relevance of the features selected to identifying architectural modules. This Chapter was dedicated to provide relevant data analysis. Furthermore, we revealed the results achieved by each type of feature selected and combined them in order to compare all of them. In addition, the presented data provides evidences for the discussion presented in the next chapter.



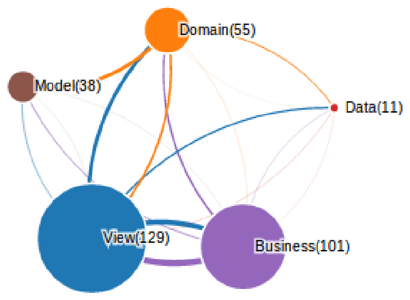
(a) Concrete architecture.



(b) Predicted architecture using Direct, Inverse and External element dependencies.



(c) Predicted architecture using Bst1 subset.



(d) Predicted architecture using the combined features subset.

Figure 6.4: Combining RecSys features to improve accuracy.

7 LESSONS LEARNED

In the previous chapter, we focused on presenting the collected data and pointed out observations we extracted from it. Then, in this chapter, we provide a deeper analysis of the results of this study and discuss the lessons learned during the study based on the experiments performed.

We detail the lessons learned in each step of the procedure specified in Chapter 4. In Section 7.1, we discuss the relation of the case studies with the results achieved. Next, we analyze the effects of the learning process adopted to our study considering the architecture recovery problem in Section 7.2. The analysis of the features selected in our study is provided in Section 7.3. Finally, an overall discussion about the experiments and procedure adopted is presented in Section 7.4.

7.1 Case Studies

The case studies used in a empirical study affect significantly the evaluation of the features relevance to software architecture. Moreover, based on the results achieved by the experiments, we derive conclusions to guide the construction of architecture recovery methods and to understand the relationship between the case studies and software architecture.

As we presented in Tables 6.1 and 6.4, different subsets of features influence differently in the result due to the individual characteristics of each case study as Constantinou, Kakarontzas and Stamelos suggested (CONSTANTINOU; KAKARONTZAS; STAMELOS, 2011) in their future work. By comparing the case studies, we observed that case studies have a different number of modules, their modules have different quantity of elements and unequal distribution of architectural elements. This variability increases the difficulty of extracting proper architectural information without previous information, such as patterns or a documented conceptual architectural. Despite of these differences, our evaluation procedure achieved significant results for *OLIS*, *RecSys*, *EC*, and *Port* case studies, as illustrated with the predicted treemap presented in Figure 6.2. Moreover, based on the accuracy of the architecture recovered prediction of the Bst1 subset using the Expectation Maximization algorithm, four of five modules of the *OLIS* case studies are clearly identified improving the understanding about the system without any previous knowledge. *OLIS* is similar to the other except *Metrics* that had a limited accuracy due to its limited size. Another important fact is the distribution of elements in the software modules. RecSys has five modules with few elements that are hard to be predicted using machine learning.

Furthermore, the manual architecture recovery of the case studies gave us insights, which were confirmed by the results achieved, about the relation of metrics, labels and element dependencies with software architecture. Also, the manual investigation provided insights about three new features related to architectural elements: (i) superclass usage; (ii) interface usage; and (iii) number of getters and setters of architectural elements. Until now, these features were disregarded by the architecture recovery methods and actually they have relation with software architecture — two of them are present in subset of features that achieved the best results in our experiments. Our study provided an analysis of features of the selected case studies, but manual investigations of more projects must be performed in order to identify more possible features.

7.2 Learning Process

The learning process fits well to our purpose, as it was expected, since the feature selection evaluated the data relevance and the unsupervised learning process adopted recovered cluster with unequal sizes. However, there are some points to be detailed to provide a clear analysis of how the feature selection and the Expectation Maximization handle the relation between the features selected and the software architecture.

Performing a feature selection over the selected features served to discover the best subsets of features. Furthermore, this process revealed the relation between the features selected and the software architecture. Measuring the accuracy of each subset evaluating all the case studies, we quantify the relevance of each subset of features to the architecture recovery. Additionally, the application of features selection improved the achieved results reducing the number of features used to metrics and labels, and state the importance of all types of dependencies to element dependencies features.

Another point that must be discussed is the influence of the case studies in the learning process. As discussed early, the case studies have many differences, such as number of lines of code, adopted architectural patterns and number of architectural modules. These differences affect the quantity of information that an application provides to the learning process, mainly the number of architectural elements. As we use an unsupervised technique guided by similarities among data, the Expectation Maximization prediction depends on how many architectural elements exist to recognize similarities during the learning process and obtain a better prediction of architectural modules. In Table 4.4, we present data related to the case study sizes, showing that *Metrics* is the smallest case study analyzed. Consequently, it has the worst accuracy compared to the others projects, and also the minimum accuracy with the *Bst1* subset and the subset with all element dependencies, as presented in Table 6.6. Furthermore, investigating the accuracies of the two largest case studies, *Port* and *RecSys*, we observe that their variation range is smaller than the *Metrics* case study. This indicates that the quantity of the architectural elements influence the learning process, i.e. the application of machine learning techniques to recover architectures is recommended mainly to medium-large applications in terms of the number of architectural elements.

Another fact related to the learning process is the recovery of modules from the case studies. Differently from the others learning algorithms, the Expectation Maximization algorithm handle clusters with different sizes since its based on statistics to derive modules. Then, recovering the larger modules is easier because they have more data related to them, consequently more data to recognize similarities in the case when this similarities exist. An example of this recognition of larger modules is presented in Figure 6.4, where the concrete architecture, Figure 6.4(a), has four large modules and two smal modules. The *Bst1* and the dependencies features, Figures 6.4(c) and 6.4(b), recovered the four main modules. On the other hand, the smalles module, Core, is not recovered since the algorithm needs more similar elements to identify that a group of elements have enough similarity to create a new module.

7.3 Features

Our study evaluated the results of two types of features in order to reveal the relevance of them to identify the software architecture module. As the results show, both types of features extracted significant information to projects architecture, both achieved at least 52.3% of average accuracy considering all case studies. The architectural element dependencies are commonly used in the literature to recover the architecture. However, the metrics and labels features evaluated overcame the element dependencies.

Analysing the results presented in Table 6.6, the results achieved by features of metrics and labels perform better than element dependencies in all case studies. In fact, the difference between the average accuracies of both types of features usage, 13.2%, reinforce superior performance of metrics and labels compared to the element dependencies.

Next, we provide a deeper discussion about the results related architectural element dependency features, in Section 7.3.1, metrics and labels, in Section 7.3.2. In addition, an analysis of the combination of both features is provided in Section 7.3.3.

7.3.1 Element Dependencies

Despite of the element dependencies have a worst performance than metrics and labels, its results also have significance to the architecture recovery process. An average accuracy of 66.9% means that it contributes to retrieve software architecture information, as showed in the RecSys. Comparing the use of all element dependencies and the *ALL* subset, presented in Table 7.1, we observe that, in the larger case studies, element dependencies perform better than the usage of all metrics and labels. It suggests that the element dependencies features demand more data to establish similarities.

Table 7.1: Average accuracy of ALL and elements dependencies subsets

Case Study	**Number of Elements**	**ALL**	**All Dependencies**	**DIFF**
Port	399	0.623	**0.676**	0.053
RecSys	334	0.615	**0.754**	0.139
OLIS	212	**0.890**	0.757	-0.133
EC	195	**0.818**	0.630	-0.188
Metrics	150	**0.593**	0.527	-0.066
AVG		**0.708**	0.669	-0.039
STD		0.136	**0.096**	-0.040

Furthermore, comparing the use of all features from both types of information selected, we see the improvement achieved by the feature selection. Then, aiming to improve the results of element dependencies, more elements dependency features may be evaluated. The three of dependency features investigated in our study are the main features related to communication of architectural elements, as they cover the three types of possible element dependencies. However, more dependency properties should be explored in order to found similarities among software architectural elements.

7.3.2 Metrics and Labels

First, we reiterate that the results presented in Section 6.2 indicate a strong relationship between the metrics and labels analyzed and the identification of architectural modules. Even when the subset is composed of all metrics and labels, the high accuracies achieved, demonstrate that all analyzed metrics and labels have contribution to group elements into modules. Even though the best subset of features contain only part of the metrics and labels evaluated, the presence of each feature in at least 8 of the best 23 subsets analyzed, presented in the *count* row of Table 6.4, indicates that each individual feature contributes to achieve a better accuracy in some of these 23 subsets, thus providing architectural information.

In order to understand why the overall best subset had unselected metrics or labels, we investigated the occurrence of each feature in the subsets with better results than the trivial subset. Analysing the frequency of features in the best subsets, presented in the last row of Table 6.4, we observed that the most important feature related to the software architecture is INT1–INT4 (i.e. the top 4 most implemented interfaces should be considered), given their presence in all best subsets. In addition, no other analyzed feature is present in all best subsets. Despite that, the other features also provide architectural information. Analysing the subsets *Bst1–Bst7* in Table 6.4, an interesting selection pattern occurs: superclass usage and class name words mutually exclude each other. This indicates that these features are correlated to each other, providing similar statistical information related to the architecture. Additionally, in most of the subsets, class name words contributes more than superclass usage. Moreover, a similar pattern occurs in the *Bst1–Bst10* subsets with the depth of inheritance and number of overridden methods features. These selection patterns justify the unselected features in the overall best subset.

A simple analysis may lead, considering the best subsets for individual case studies, to the wrong conclusion that the overall best subset consists of the following features: mean method complexity, total lines of code, top most two class name words, and top most two interface usages. This subset is the intersection of all individual best subsets presented in Table 6.3. However, performing an exhaustive investigation, we concluded differently — the set with these features has worst accuracy than the top 23 subsets. The difficult in derive information from the source code metrics and labels is connected to the complexity of relationship between the metrics and labels analyzed and the architectural modules.

Bst1 subset achieves the best overall accuracy. Its high accuracy is independent from the unselected features, which indicates that this set of metrics and labels has the maximum shared architectural information considering all case studies and the minimum correlation among them, as can be seen in Table 6.4. Consequently, mean method complexity, depth of inheritance, number of children, number of attributes, number of methods, number of getters and setters, average methods size, top most five class name words and top most five interface usages are considered the most relevant features of all analyzed metrics and labels related to software architecture. Furthermore, this subset has the highest average accuracy and the lowest standard deviation what reinforce the relevance of this subset. Additionally, each features in the *Bst1* subset contributes with an specific property, possibly with the rationale described in Section 4.2.

7.3.3 Combining Features

The combination of features intuitively seems to be an improvement to the data quality because the each main concept of software architecture are present one of the two types of features analyzed. However, combine the communication information provided and the behavior information provided by metrics and label did not improve the results, at least using the combination used in our study as presented in Table 6.6. In fact, combining features is not trivial, for instance the element dependencies have a variable number of features and usually many features compared to the fixed model of the metrics and labels features. Mainly, it may be the reason for why the software architecture recovery methods use only one type of information retrieved from the source code to recover the software architecture. Actually, the only combination of features performed by the architecture recovery approaches is use of architectural patterns to guide the recovering process. It occurs mainly due to the complexity to merge different features, which still is a challenge.

7.4 Overall Analysis

As introduced in Section 2.1, a software architecture states the rules of what elements do and how they interact. Our study focused on how to group architectural elements based on their characteristics, as the result of applying a machine learning algorithm is a set of identified modules and to which module each element belongs. This identification of each module is performed without the specification of a predefined number of modules, as it is the case of existing approaches (KUHN; DUCASSE; GÍRBA, 2007; CONSTANTINOU; KAKARONTZAS; STAMELOS, 2011). We evaluated information related to roles and communication in order to cover both architectural concepts. Using only metrics and labels, there is lack of information retrieved related to communication of architectural elements. On the other hand, considering only element dependencies, architectural roles information are disregarded. Our attempt to combine both failure since the accuracy achieved using both types of characteristics is worse than just metrics and labels. However, metrics and labels achieved a significant accuracy alone.

Our experiments presented the best subset of each type of features selected concluding that metrics and labels are the most appropriate type of features to be applied considering the procedure adopted and the selected case studies. Additionally, our study is one of the first studies that compare the relevance of source code metrics to recover software architectural modules. Moreover, our study revealed the strong relationship between software architecture and metrics and labels.

Recently, Garcia, Ivkovic and Medvidovic performed a study considering a new evaluation metric to architecture recovery (GARCIA; IVKOVIC; MEDVIDOVIC, 2013). They proposed the Cluster-to-Cluster (c2c) comparison providing a ranges of level of similarity comparing nine architecture recovery algorithms. They stated three levels of matching: (i) moderate matching, from 40% to 60%; (ii) strong matching, from 60% to 80%; and (iii) a very strong matching from 80% to 100%. Their metric is basically a precision of each module predicted. Our similar measure of evaluation is the average precision of each case study. The average precision presented in the Appendix A defines our case studies average precision as at least moderate in all case studies considering Bst1, all element dependencies and combined features subsets. Furthermore, the Bst1 subset has the average precision of 50.1% to Metrics, 53.5% to Port, 55% to RecSys, 67% to Expert Committee and 78.8% to OLIS — Bst1 subset thus moderately matches three and strongly matches two of the case studies selected.

7.5 Final Remarks

In this chapter, we analyzed the experiments results deriving the main lessons learned to identify architectural modules using machine learning. We presented a detailed discussion about each part of our procedure highlighting the conclusions of each step and their relation with the results obtained. Firstly, an analysis of the relationship between the case studies characteristics and the accuracy achieved by the subsets analyzed. Additionally, the manual investigation importance is discussed since the architecture recovery approaches still must be improved in terms of extraction of information. Then, the impact of the learning process in our results and guidelines recommendations to the use of the Expectation Maximization was discussed. We also analyzed the selected features relevance to software architecture aiming to present the reasons of why metrics and labels achieve a high accuracy. We conclude giving an overview of the improvements obtained, highlighting the points related to the whole procedure adopted.

8 CONCLUSION

The software complexity increases due to the growing computer science evolution techniques to build a software, such as programming languages and hardware variety. Consequently, the effort demanded to maintain the software documentation in conformance with the concrete architecture is also increasing. Software architecture comes to avoid the technological specificities in the documentation capturing the concepts applied to build a software in a high-level model. Despite of the benefits of having a documented software architecture, the architectural models commonly do not exist or diverge from the concrete architecture. This occurs because the task of maintaining a software architecture model up-to-date is complex. Then, aiming to tackle this lack of architectural documentation, software architecture recovery research focuses on automating the extraction of software architectures based on source code information automatically. In order to improve the software architecture recovery scenario, machine learning techniques have being applied in sources of software data to automate the process of recovering software architecture.

In our study, we presented an evaluation of the software architecture recovery sources of information applying an unsupervised machine learning technique. We specified a procedure to evaluate the relevance of each source of architectural information. We performed this evaluation considering five case studies, two different types of architectural elements characteristics and five statistical metrics. As a results of our approach, we achieved a high accuracy, 80.1%, considering the best subset of features obtained through the feature selection process performed. An advantage of our approach is that the accuracy achieved was obtained performing a process that can be totally automated. Additionally, due to the lack of support to measure and understand the architecture recovery results, we developed ArchViz — a web-based tool to provide automatic evaluation of the architecture recovery results. Furthermore, it provides three different visualizations of the concrete and predicted architecture to support understand of the architecture recovery process.

8.1 Contributions

Given the results presented in this dissertation, we list below our main contributions.

Architecture Recovery Using Machine Learning Guidelines. We specified, in Chapter 4, the generic steps to develop a method to recover software architecture automatically based on machine learning techniques. We set our steps parameters — software information selection, dataset preparation, learning process and results analysis — to evaluate the relevance of the features analyzed based on the performance of our experiments. Given the results achieved, we provided empirical evidences of the effectiveness of our guidelines to build a method to recover software architecture using the parameters set and the features analyzed.

Architectural Elements Characteristics Comparison. There are many architecture recovery methods, but most of them use only one type of software information. In addition, they do not provide an evaluation of the architectural elements information that could be considered following their approaches. Aiming discover which type of source code information is more relevant to identify software architecture modules, we performed a comparative evaluation of the two main concepts related to software architecture (communication and role). Additionally, we combined both types of characteristics in a dataset aiming to provide an improved dataset with both information related to software architecture. However, our experiments presented the best performance using just a subset of the evaluated source code metrics and labels.

Metrics and Labels to Identify Architectural Modules. Most of the current software architecture recovery methods disregarded source code metrics and labels. Thus, we performed a deep evaluation and analysis about the importance of source code metrics and labels to identify architectural modules. We concluded by our experiments that there is a strong relationship between them. The selected features (metrics and labels) achieved better results in all case studies, compared to the commonly used element dependencies features.

Architecture Recovery Tool Support [1]. Due to the lack of standardization and visualizations of architecture recovery methods. We provided a tool to support the architecture recovery research, ArchViz. Our tool provides a standardized evaluation of the architecture recovery processes given the concrete architecture and the predicted architecture. It automatically calculates the statistical

[1]`http://archviz.herokuapp.com`

metrics from the information retrieval context used to evaluate the prediction quality of unsupervised approaches. Additionally, ArchViz provides visualizations of concrete and predicted architectures. It facilitates the comparison between sources of information and learning processes providing quantitative measures. It improves the recovery process understand with architectural representation of conceptual and concrete architectural models.

8.2 Future Work

Our work is an initial step towards an automatic approach to manage software architecture based on source code comprising the architecture evolution and an architecture recovery complete approach. In order to apply the lessons learned with respect to architecture recovery and use the contributions presented in this dissertation to in fact recover software architecture, future work is related to this study listed below.

An Architecture Recovery Method. Our procedure of evaluation achieved relevant results to architecture recovery. Based on the procedure parameters presented in Chapter 4, an architecture recovery method can be developed — an integrated method that handles the extraction of features, the learning process and the predicted architecture evaluation. This architecture recovery method would be an extension of ArchViz, it already performs the evaluation of the predicted architecture, with all the procedure implemented.

Investigation of Architectural Element Characteristics. As presented by Garcia, Ivkovic and Medvidovic (GARCIA; IVKOVIC; MEDVIDOVIC, 2013), there are much to be improved in the architecture recovery process and a key improvement factor is the software information selection. Architectural elements have many characteristics disregarded by the software architecture recovery studies, such as the code metrics presented in this dissertation. We aimed to improve the set of characteristics with our study, but many others must be evaluated to achieve a more robust and reliable architecture recovery method.

Specific Purpose Expectation Maximization Algorithm. An analysis of the Expectation Maximization algorithm influence in the results must be performed in order to have a better understanding of the learning process and proposed specific purpose changes in the algorithm. In particular, a modification in EM algorithm to accept weight in the used features may help to combine different types of features and also improve the results achieved.

Evaluation of Architectural Modules Roles Capturing the idea that similar systems have similar class profiles (OLIVEIRA et al., 2013). In our work, we provided an analysis of source code metrics, labels and dependencies relevance considering the Expectation Maximization. Thus, a deeper analysis about recurrent software architecture modules, such as Data modules, may lead to class profiles related to a specific modules. It could improve the understanding about the relationship between the selected features and architectural modules.

In summary, we evaluated different sources of information relevance to software architecture recovery. In fact, the research on architecture recovery still is a challenge since the architecture recovery methods still are depend on the human assistance. Establishing the relationship between sources of information (metrics, labels and dependencies) and software architecture, our work provide evidences to future architecture recovery methods to increase their reliability.

REFERENCES

AALST, W. et al. Process mining: a two-step approach to balance between underfitting and overfitting. **Journal of Software & Systems Modeling**, [S.l.], v.9, n.1, p.87–111, 2010.

ALLEN, E.; KHOSHGOFTAAR, T. Measuring coupling and cohesion: an information-theory approach. In: INTERNATIONAL SOFTWARE METRICS SYMPOSIUM, Boca Raton, FL, USA. **Proceedings. . .** [S.l.: s.n.], 1999. p.119–127.

ANQUETIL, N.; LETHBRIDGE, T. C. Recovering software architecture from the names of source files. **Journal of Software Maintenance: Research and Practice**, [S.l.], v.11, n.3, p.201–221, May 1999.

BASS, L.; CLEMENTS, P.; KAZMAN, R. In: **Software Architecture in Practice**. [S.l.]: Pearson Education, 2012. p.3–24. (SEI Series in Software Engineering).

BUSCHMANN, F.; HENNEY, K.; SCHMIDT, D. In: **Pattern Oriented Software Architecture**: on patterns and pattern languages. [S.l.]: John Wiley & Sons, 2007. p.25–64. (Wiley Series in Software Design Patterns).

CARRIERE, S. J.; KAZMAN, R. The Perils of reconstructing architectures. In: INTERNATIONAL WORKSHOP ON SOFTWARE ARCHITECTURE, New York, NY, USA. **Proceedings. . .** ACM, 1998. p.13–16. (ISAW ’98).

CHIDAMBER, S. R.; KEMERER, C. F. A Metrics Suite for Object Oriented Design. **Journal of Transactions on Software Engineering**, Piscataway, NJ, USA, v.20, n.6, p.476–493, June 1994.

CLEMENTS, P.; SHAW, M. "The Golden Age of Software Architecture" Revisited. **Journal of IEEE Software**, [S.l.], v.26, n.4, p.70–72, 2009.

CONSTANTINOU, E.; KAKARONTZAS, G.; STAMELOS, I. Towards Open Source Software System Architecture Recovery Using Design Metrics. In: PANHEL-

LENIC CONFERENCE ON INFORMATICS. **Proceedings. . .** [S.l.: s.n.], 2011. p.166–170.

CORAZZA, A.; DI MARTINO, S.; SCANNIELLO, G. A Probabilistic Based Approach towards Software System Clustering. In: FOURTEENTH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings. . .** [S.l.: s.n.], 2010. p.88–96.

CORAZZA, A. et al. Investigating the use of lexical information for software system clustering. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings. . .** [S.l.: s.n.], 2011. p.35–44.

DASH, M. et al. Feature Selection for Clustering — A Filter Solution. In: INTERNATIONAL CONFERENCE ON DATA MINING. **Proceedings. . .** [S.l.: s.n.], 2002. p.115–122.

DEMPSTER, A.; LAIRD, N.; RUBIN, D. Maximum likelihood from incomplete data via the EM algorithm. **Journal of the Royal Statistical Society**, [S.l.], v.39, n.1, p.1–38, 1977.

DUCASSE, S.; POLLET, D. Software Architecture Reconstruction: a process-oriented taxonomy. **Journal of Transactions on Software Engineering**, [S.l.], v.35, n.4, p.573–591, 2009.

GANSNER, E. R.; NORTH, S. C. An Open Graph Visualization System and Its Applications to Software Engineering. **Journal of Software – Practice & Experience - Special issue on discrete algorithm engineering**, New York, NY, USA, v.30, n.11, p.1203–1233, Sept. 2000.

GARCIA, J. et al. Obtaining Ground-truth Software Architectures. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Piscataway, NJ, USA. **Proceedings. . .** IEEE Press, 2013. p.901–910. (ICSE '13).

GARCIA, J.; IVKOVIC, I.; MEDVIDOVIC, N. A Comparative Analysis of Software Architecture Recovery Techniques. In: INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING. **Proceedings. . .** IEEE, 2013. p.486–496.

GARLAN, D. Software architecture: a roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, New York, NY, USA. **Proceedings. . .** ACM, 2000. p.91–101. (ICSE '00).

GUYON, I.; ELISSEEFF, A. An Introduction to Variable and Feature Selection. **Journal of Machine Learning Research**, [S.l.], v.3, p.1157–1182, Mar. 2003.

HALL, M. et al. The WEKA data mining software: an update. **SIGKDD Explorations Newsletter**, [S.l.], v.11, n.1, p.10–18, Nov. 2009.

HARRIS, D. R.; REUBENSTEIN, H. B.; YEH, A. S. Reverse engineering to the architectural level. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York, NY, USA. **Proceedings...** ACM, 1995. p.186–195. (ICSE '95).

HAWKINS, D. The Problem of Overfitting. **Journal of Chemical Information and Computer Sciences**, [S.l.], v.44, n.1, p.1–12, 2004.

HENDERSON-SELLERS, B. **Object-oriented metrics**: measures of complexity. 1.ed. [S.l.]: Prentice-Hall, 1995.

HOCHSTEIN, L.; LINDVALL, M. Combating architectural degeneration: a survey. **Journal of Information and Software Technology**, [S.l.], v.47, n.10, p.643–656, July 2005.

KAZMAN, R.; CARRIERE, S. J. Playing Detective: reconstructing software architecture from available evidence. **Journal of Automated Software Engineering**, Hingham, MA, USA, v.6, n.2, p.107–138, Apr. 1999.

KAZMAN, R.; O'BRIEN, L.; VERHOEF, C. **Architecture reconstruction guidelines**. [S.l.]: Carnegie Mellon University, 2001.

KEIM, D. et al. Visual Analytics: scope and challenges. In: SIMOFF, S.; BöHLEN, M.; MAZEIKA, A. (Ed.). **Visual Data Mining**. [S.l.]: Springer Berlin Heidelberg, 2008. p.76–90. (Lecture Notes in Computer Science, v.4404).

KNODEL, J. et al. Static evaluation of software architectures. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings...** [S.l.: s.n.], 2006. p.10 pp.–294.

KOHAVI, R.; JOHN, G. H. Wrappers for feature subset selection. **Journal of Artificial Intelligence**, [S.l.], v.97, n.1, p.273–324, 1997.

KORF, R. E. Linear-space Best-first Search. **Journal of Artificial Intelligence**, [S.l.], v.62, n.1, p.41 – 78, 1993.

KUHN, A.; DUCASSE, S.; GÍRBA, T. Semantic clustering: identifying topics in source code. **Journal of Information and Software Technology**, [S.l.], v.49, n.3, p.230–243, Mar. 2007.

MACQUEEN, J. B. Some Methods for Classification and Analysis of MultiVariate Observations. In: BERKELEY SYMPOSIUM ON MATHEMATICAL STATISTICS AND PROBABILITY. **Proceedings...** University of California Press, 1967. v.1, p.281–297.

MANCORIDIS, S. et al. Using automatic clustering to produce high-level system organizations of source code. In: INTERNATIONAL WORKSHOP ON PROGRAM COMPREHENSION. **Proceedings...** [S.l.: s.n.], 1998. p.45–52.

MANCORIDIS, S. et al. Bunch: a clustering tool for the recovery and maintenance of software system structures. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE. **Proceedings...** [S.l.: s.n.], 1999. p.50–59.

MAQBOOL, O.; BABRI, H. Hierarchical Clustering for Software Architecture Recovery. **Journal of Transactions Software Engineering**, Piscataway, NJ, USA, v.33, n.11, p.759–780, Nov. 2007.

MCCABE, T. A Complexity Measure. **Journal of Transactions on Software Engineering**, [S.l.], v.SE-2, n.4, p.308–320, 1976.

MEDVIDOVIC, N.; JAKOBAC, V. Using software evolution to focus architectural recovery. **Journal of Automated Software Engineering**, [S.l.], 2006.

MEDVIDOVIC, N.; TAYLOR, R. N. Software Architecture: foundations, theory, and practice. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, New York, NY, USA. **Proceedings...** ACM, 2010. p.471–472. (ICSE '10).

MITCHELL, T. M. Machine learning and data mining. **Journal of ACM Communications**, [S.l.], v.42, n.11, p.30–36, Nov. 1999.

MLADENOVIC, N.; HANSEN, P. Variable neighborhood search. **Journal of Computers & Operations Research**, [S.l.], v.24, n.11, p.1097 – 1100, 1997.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. Software reflexion models: bridging the gap between source and high-level models. In: SIGSOFT SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 1995. v.20, n.4, p.18–28.

NUNES, I.; NUNES, C.; CIRILO, E.; KULESZA, U.; LUCENA, C. **Expert Committee Architecture**. Available at: `http://www.inf.ufrgs.br/~ingridnunes/maspl/index.php?base=casestudies&page=ecArch`. last accessed in March 25th, 2014.

OLIVEIRA, P. et al. Metrics-based Detection of Similar Software. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2013. p.447–450.

PARNAS, D. L. Software aging. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1994. p.279–287. (ICSE '94).

PASSOS, L. et al. Static Architecture-Conformance Checking: an illustrative overview. **Journal of IEEE Software**, [S.l.], v.27, n.5, p.82–89, 2010.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **SIGSOFT Software Engineering Notes Newsletter**, New York, NY, USA, v.17, n.4, p.40–52, Oct. 1992.

SHAW, M.; CLEMENTS, P. The Golden Age of Software Architecture. **Journal of IEEE Software**, Los Alamitos, CA, USA, v.23, n.2, p.31–39, Mar. 2006.

SHNEIDERMAN, B. Tree Visualization with Tree-maps: 2-d space-filling approach. **Journal of Transactions Graphics**, New York, NY, USA, v.11, n.1, p.92–99, Jan. 1992.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: a survey. **Journal of Systems and Software**, [S.l.], v.85, n.1, p.132–151, Jan. 2012.

SOKOLOVA, M.; LAPALME, G. A systematic analysis of performance measures for classification tasks. **Journal of Information Processing & Management**, [S.l.], v.45, n.4, p.427–437, July 2009.

STANDARD-1471. **Recommended practice for architectural description of software-intensive systems**. [S.l.]: IEEE, 2000. i–23p.

STOL, K.; AVGERIOU, P.; BABAR, M. A. Identifying architectural patterns used in open source software: approaches and challenges. In: INTERNATIONAL CONFERENCE ON EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2010. p.91–100. (EASE'10).

TZERPOS, V.; HOLT, R. ACDC: an algorithm for comprehension-driven clustering. In: WORKING CONFERENCE ON REVERSE ENGINEERING. **Proceedings...** [S.l.: s.n.], 2000. p.258–267.

WARD, J. Hierarchical grouping to optimize an objective function. **Journal of the American Statistical Association**, [S.l.], v.58, p.236–244, 1963.

XIAO, C.; TZERPOS, V. Software Clustering Based on Dynamic Dependencies. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING. **Proceedings. . .** [S.l.: s.n.], 2005. p.124–133.

APPENDIX A – COMPLEMENTARY RESULTS

This Appendix presents complementary results achieved by the three relevant subsets identified in Chapter 6. Then, we detail precision, recall and fMeasure of Bst1, all element dependencies and combined features subsets. Actually, the results presented in this Appendix follows the same conclusion derived based on the average accuracy discussed in Chapter 6.

We present, in Table 8.1, the metrics related to Bst1 subset. As occurs in the average accuracy, OLIS achieved the best results — precision, recall and fMeasure. However, Metrics achieved the only the worst precision. It occurs by the low number of modules in Metrics. Its recall is affected since it has only four modules to be predicted. On the other hand, the RecSys high number of modules affects their recall because some of its modules are not identified due to their low number of elements. One important thing to highlight is the average precision of 60.9%, which is better than six of the nine algorithms evaluated in the comparative analysis of Garcia, Ivkovic and Medvidovic (GARCIA; IVKOVIC; MEDVIDOVIC, 2013).

Table 8.1: Average precision, recall and fMeasure of the Bst1 usage

Case Study	***Precision***	***Recall***	***fMeasure***
OLIS	0.788	0.761	0.775
EC	0.670	0.701	0.685
Port	0.535	0.545	0.540
Metrics	0.501	0.467	0.483
RecSys	0.550	0.372	0.444
AVG	0.609	0.569	0.585
SD	0.119	0.161	0.140

Table 8.2 detail the measures of the subset containing all element dependencies features. These results are significantly worse than Bst1 with an average precision of 45.1%, 15% less than Bst1. Similarly to Bst1 results, the number of

modules affects the recall and fMeasure, because the case study with less modules performs better with these metrics. However, the precision still representing the same scenario from accuracy, when OLIS performs best and Metrics performs worst from all case studies evaluated.

Table 8.2: Average precision, recall and fMeasure of the subset with all dependencies usage

Case Study	***Precision***	***Recall***	***fMeasure***
Metrics	0.448	0.578	0.505
OLIS	0.507	0.490	0.498
RecSys	0.482	0.463	0.472
Port	0.458	0.381	0.416
EC	0.415	0.332	0.369
AVG	0.451	0.392	0.419
SD	0.034	0.066	0.052

Finally, in Table 8.3, we show the results achieved by the combined features subset. Actually, the results achieved are worse than Bst1 subset, but they are better than select all element dependencies. In fact, this subset is highly influenced by the Bst1, however with less precision. These data corroborate with the idea that appending element dependencies prediction to the Bst1 subset does not improve the results.

Table 8.3: Average precision, recall and fMeasure of the subset with combined features

Case Study	***Precision***	***Recall***	***fMeasure***
OLIS	0.664	0.864	0.751
EC	0.599	0.529	0.562
Port	0.541	0.426	0.476
RecSys	0.453	0.458	0.455
Metrics	0.464	0.360	0.406
AVG	0.544	0.527	0.530
SD	0.090	0.198	0.136

Again, Bst1 subset overcome the use of element dependencies and combined features subsets. Actually, the evaluation of precision, recall and fMeasure also show the relevance of source code metrics to identify architectural modules.

APPENDIX B – RESUMO ESTENDIDO

Avaliação de Informação de Código para Identificação de Módulos Arquiteturais

Arquitetura de software desempenha um importante papel no desenvolvimento de software, quando explicitamente documentada, ela melhora o entendimento sobre o sistema implementado e torna possível entender a forma com que requisitos não funcionais são tratados. Apesar da relevância da arquitetura de software, muitos sistemas não possuem uma arquitetura documentada, e nos casos em que a arquitetura existe, ela pode estar desatualizada por causa da evolução descontrolada do software. O processo de recuperação de arquitetura de um sistema depende principalmente do conhecimento que as pessoas envolvidas com o software tem. Isso acontece porque a recuperação de arquitetura é uma tarefa que demanda muita investigação manual do código fonte.

A pesquisa de recuperação de arquitetura objetiva auxiliar esse processo. A maioria dos métodos de recuperação existentes são baseados em dependência entre elementos da arquitetura, padrões arquiteturais ou similaridade semântica do código fonte. Embora as abordagem atuais ajudem na identificação de módulos arquiteturais, os resultados devem ser melhorados de forma significativa para serem considerados confiáveis. Sendo assim, listamos abaixo os principais problemas relativos a recuperação de arquitetura idenficados na litetura atual.

1. *Falta de confiabilidade nos métodos atuais de recuperação de arquitetura.* Apesar do esforço aplicado pelos atuais estudos, existe ainda uma falta de confiabilidade nos resultados obtidos através deles. Assim, muito ainda pode ser feito para reduzir o esforço demandado pelas abordagens atuais.

2. *Falta de informação relacionada a arquitetura de software.* Arquitetura de software documenta os principais conceitos aplicados durante o desenvolvimento do software, contudo não são conhecidas quais informações são de fato

relacionadas com a arquitetura de software. Usualmente, os trabalhos de recuperação de arquitetura avaliam padrões arquiteturais, regras de dependência entre elementos e semântica de código para predizer uma arquitetura.

3. *Falta de visualizações dedicadas a recuperação de arquitetura.* Existem várias formas de visualizar um software, pois um software envolve diversos tipos de interesses, tais como qualidade e funcionalidades, que são representados de formas diferentes no modelo arquitetural. Durante a recuperação de arquitetura, as avaliações são geralmente baseadas em dados para representação de uma arquitetura somente. Ou seja, sem a existência de uma visualização visando comparar uma arquitetura recuperada com uma arquitetura concreta.

Nesta dissertação, objetivamos melhorar o suporte a recuperação de arquitetura explorando diferentes fontes de informação e técnicas de aprendizado de máquina. Nosso trabalho consiste de uma análise, considerando cinco estudo de casos, da utilidade de usar um conjunto de características de código (*features*, no contexto de aprendizado de máquina) para agrupar elementos em módulos da arquitetura. Atualmente não são conhecidas as características que afetam a identificação de papéis na arquitetura de software. Por isso, nós avaliamos a relação entre diferentes conjuntos de características e a acurácia obtida por um algoritmo não supervisionado na identificação de módulos da arquitetura. Consequentemente, nós entendemos quais dessas características revelam informação sobre a organização de papéis do código fonte. Nossa abordagem usando características de elementos de software atingiu uma acurácia média significativa. Indicando a relevância das informações selecionadas para recuperar a arquitetura. Além disso, nós desenvolvemos uma ferramenta para auxílio ao processo de recuperação de arquitetura de software. Nossa ferramenta tem como principais funções a avaliação da recuperação de arquitetura e apresentação de diferentes visualizações arquiteturais. Para isso, apresentamos comparações entre a arquitetura concreta e a arquitetura sugerida. Seguindo a nossa abordagem, enumeramos nossas principais contribuições abaixo.

1. Um procediemento padronizado para avaliar a relação entre as informações extraídas do código fonte e módulos da arquitetura de software.

2. A indentificação de quais atributos dos elementos da arquitetura tem mais relevância para recuperação de arquitetura.

3. Diferentes visualizações da arquitetura de software visando a comparação entre arquitetura concreta e arquitetura predita.

4. Definição de diretrizes para construção de métodos para recuperação de arquietetura de software. Dentro dessas diretrizes, analisamos a importância de cada parte do processo de recuperação de arquitetura.

Para construção da nossa avaliação definimos um processo de características de elementos de arquitetura. O processo é dividido em quatro etapas: seleção de informação de software; preparação de dados; processo de aprendizado; e análise dos resultados. Cada um desses quatro passos tem grande importância no processo de recuperação de arquitetura: (i) a seleção de informação foca em entender quais características podem ser importante para o contexto de arquitetura de software; (ii) preparação dos dados visa de fato extrair as características do software de uma que seja computável; (iii) o processo de aprendizado é fundamental para a identificação dos módulos arquieturais, pois através dele que são descobertas as similaridades entre elementos da arquitetura; (iv) a análise de resultados faz a medição da relevância de cada característica do software para que se tenha uma quantificação do impacto de cada tipo de característica no processo de recuperação. Nosso processo avalia as dependências entre elementos arquiteturais e um conjunto de métricas (acurácia, precisão, recuperação e fMeasure) de projeto para recuperação de arquitetura.

As dependências entre elementos da arquitetura são comumente usadas nos métodos atuais de recuperação visto que elas definem as regras de comunicação entre módulos da arquitetura, por exemplo, uma arquitetura em camadas onde uma camada (módulo) só pode se comunicar com a camada diretamente abaixo dela. Contudo, analisar somenete as dependências não nos trás informações sobre o que os elementos da arquitetura efetivamente fazem. Caso um elemento viole as regras de comunicação da arquitetura (por exemplo, um elemento da camada de nível mais baixo de uma arquitetura em camada se comunica com um elemento da camada de nível mais alto), ele, segundo as regras de comunicação, não será classificado de acordo com seu papel de fato na arquitetura.

No processo de avaliação de cada tipo de característica, nós selecionamos cinco estudos de caso, definimos uma abordagem de seleção de variáveis (Wrapper exaustivo) e definimos um algoritimo de aprendizado (Expectation Maximization). Nós realizamos experimentos analisando o impacto das dependências entre elementos; métricas de projeto e nomes; e uma abordagem de união de ambas informações.

Para as dependências entre elementos, nós levamos em conta dependências diretas, inversas e diretas (representadas na Figura 4.4, onde os elementos *WeatherResponse* e *ForecastResponse* usam elementos internos do projeto e usam um elemento externo ao projeto). Os resultados obtidos através dos experimentos com dependências entre elements, apresentados na na Tabela 6.1, atingiram uma média de acurácia de 66.9% usando todos os tipos de dependências. Isso mostra a relevância do uso de todos os tipos de dependências na recuperação.

Nos experimentos usando métricas e nomes de elementos de software foram gerados todos subconjuntos possíveis de características dos 15 tipos de informações entre métrics e nomes selectionados. Isso gerou um grande número de subconjuntos a serem avaliados – 147 mil para cada estudo de caso. Nosso foco então é apresentar os conjuntos mais relevantes para entender a relevância dessas características para a arquitetura de software. Primeiro analisamos o resultado do conjunto com todas as características (sem seleção de variáveis). Esse conjunto atigiu uma acurácia média de 70.8%, presentado no conjunto ALL na Tabela 6.2. Contudo, esse conjunto é assimétrico, privilegia um estudo de caso com sua seleção de características, assim como muitos outros conjuntos analisados. Procurando analisar conjuntos relevantes, selecionamos todos os conjuntos que obtem melhores resultados em todos os estudos de caso individualmente. Essa seleção resultou nos conjuntos Bst1–23 apresentados na Tabela 6.2. O melhor resultado foi obtido através do conjunto Bst1. Ele atingiu uma acurácia média de 80.1% com um desvio padrão de 6.3%. Dados os resultados, ele é o conjunto que obtém os melhores resultados e que tem a menor assimetria.

Intuitivamente, combinar os dois tipos de informações analisados traria bons resultados visto que uma das informações é relevante para comunicação e a outra foca no papel desempenhado pelo elemento arquitetural. Contudo, nossos experimentos mostraram o contrário. A abordagem de combinações de informações que realizamos resultou em uma piora nos resultados, atingindo uma acurácia de 72.2% como apresentado na Tabela 6.6.

Os dados obtidos atráves dos experimentos são relevantes, porém os dados de acurácia não mostram diretamente os conceitos recuperados de uma arquitetura. Visando entender melhor a relação entre arquitetura concreta e predita, nós desenvolvemos uma ferramenta de suporte a pesquisa em recuperação de arquitetura – chamada de ArchViz [2]. Nossa ferramenta possuí funcionalidades básicas de gerência de arquitetura de software e três visualizações focadas na comparação entre arquitetura concreta e predita. As três visualizações implementadas são treemap (Figura 5.1), de módulos (Figura 5.3) e de elementos (Figura 5.4).

Descrito todo o processo de avalaição e experimentação desenvolvido durante a elaboração desta dissertação, abaixo são descritas as principais contribuições derivadas do trabalho realizado.

1. *Diretrizes para o processo de recuperação de arquitetura usando aprendizado de máquina.* Com os resultados obtidos através dos experimentos, nós extraí-

[2]`http:\\archviz.herokuapp.com`

mos diretrizes para a construção de métodos de recuperação de arquitetura baseados em aprendizado de máquina que são relevantes para serem aplicados em futuras abordagens.

2. *Comparação entre características de elementos da arquitetura.* A avaliação de quais características de elementos de software são mais relevantes auxilia na escolha de informações utilizadas na recuperação de arquitetura. A comparação de duas características através de um método padronizado mostrou a relevância de cada uma delas.

3. *Uso de métricas e nomes para identificar módulos da arquitetura.* Nosso estudo mostrou a relação entre métricas e nomes com arquitetura de software. Obtendo resultados superiores do que a comumente usada informação de dependência entre elementos.

4. *Ferramenta de suporte a recuperação de arquitetura.* Visando aumentar a padronização dos métodos de recuperação de arquitetura, nós desenvolvemos uma ferramenta que realiza a avaliação dos métodos de recuperação de forma padronizada. Além disso, nossa ferramenta fornece visualização focadas em comparar arquiteturas concretas e preditas.

Em resumo, nós avaliamos a relevância de diferentes fontes de informação utilizadas para recuperar a arquitetura de software. De fato, a recuperação de arquitetura de software continua sendo um assunto de pesquisa desafiador. Contudo, estabelecendo a relação entre as diferentes fontes de informação e a arquitetura de software, nós fornecemos evidências para futuros métodos de recuperação de arquitetura reduzirem a demanda de interação humana e melhorarem seus resultados.